Flame Engine — Game 2D

Transkrip referensi — folder sumber `5_flutter_flame`

12 bagian · dibuat otomatis

Daftar Isi

1_introduction

1. What is the Flame Engine?

All right.

Let's talk quickly about the flame engine.

So first of all, the flame engine, as you must have gathered from the course name, sits on top of the flutter framework.

The flutter framework is what isolates your code from the underlying platform and whether you want to run your game on iOS, Android or the Web.

You only have to worry about a single code base.

You do not worry about the underlying platform right on top of the flutter framework.

You have the flame engine.

The flame engine is basically a set of optimized components as well as a complete 2D game framework, which also includes a physics engine.

So it is a perfect vehicle for writing games, especially for mobile platforms such as iOS or Android.

So of course your game runs on top of the Flame Engine and benefits from Flutters Widget framework and

Darts object oriented paradigm.

So if you know, flutter and dart and you do not really have to know them that well, you're set for this course.

But if you don't know, Dart for example.

But you know, another object oriented language such as TypeScript or Java or Swift or C++ or Sharp or any of those languages, then you can still greatly benefit from this course as we will build everything on top of Flutter.

So you do not really have to know that much about the flutter framework and anything that you would need to know will be explained in the course.

Okay, guys, see you soon.

2. What The Course Covers

Hi, guys, and welcome.

Let me introduce you to the course contents.

This will tell you in a nutshell what this course will cover.

So first of all, we're going to go through the flame engine setup and we'll look into the fundamentals of the structure of a typical project in flame.

We will learn here about the concept of the game loop and we will explore the main methods in the game engines such as update, render and load, as well as how flame components are used to render all sorts of data onto the screen.

We will be using Visual Studio Code as our ID, so we will show you the best way of using Visual Studio for Flame projects.

Next, we will learn about the game design fundamentals.

We will look at how the flame engine is architected and how component trees can be used together to build a full game.

Think of how Lego pieces are used together and you will have an idea of how out of the box components in Flame Engine work together to create a full game.

We will cover such components as text components, Sprite and sprite sheets, images, sound shape, particle, parallax, joystick timer, sound and others.

And we will see how we can put all of them together to create a full game.

Next, we will look at the most important flame components used for 2D gaming and we will explore through

case studies how to use these components as building blocks for any 2D game.

We will also see how to create our own components that fit into the flame engine architecture.

After that, we'll look at how does one go around designing a game.

You do not want to just jump right away into coding a game.

You want to design it.

First.

We will start with a formal mission statement and create an architecture identifying the necessary components

and design how they will work together to make a working game.

And then of course we will code it and it will be fun.

The game we will be creating is a clone of the now classic Atari Asteroids game, which was very famous in the eighties and is a perfect example of a classic 2D space shooter in which we will use particle physics, parallax timers, joystick control, music, sound effects and well, you get the idea.

It is a perfect game to design and to code as your first game on the flame engine.

See you in the next lecture.

3. Concept vs. Code

All right, let's talk about concept versus code idea.

So we need to be able to separate the ideas and software engineering behind a game as opposed to just looking at the final product of the design, which is code.

A well-designed game is easier to write, debug and modify.

So in this course we will strive to teach you as much as possible to the problem and not the coding language.

This is what the founder of Tesla, Elon Musk, actually said that he's looking for.

When he's looking for engineers, he's looking for the people who think to the problem, not the tool.

So in this course, we're going to be teaching you as much as possible to the problem and not to the coding language.

What this means is that as we go through coding examples, we will also pay attention to the concepts behind many of the flame engine elements and explain them from a software architecture perspective.

So, for example, a concept is a clear idea of what will be done, and the emphasis is on what it basically means, proper breakdown of the idea into smaller independent elements.

And this allows for proper analysis which will save in the end coding time.

And this creates an intermediate documentation for the concept which makes the end result easier to communicate.

So what this means is that when you conceptualize what you want to do, it will be easier for you to create the documentation and also explain what it is that it's supposed to do.

Code, on the other hand, is an end product of a concept.

It shows how it is done and usually it is hard to understand and learn just from code.

Code can be hundreds of thousands of lines.

It will be impossible for you to go into the code and try to understand what is happening there.

You need a proper concept and a proper design so it is difficult to maintain code without an intermediate representation such as an architectural documentation, for example UML diagrams.

What you're seeing here is an example of very simple code.

What you're seeing here is some sort of an encapsulation of some sort of behavior for you to figure out what this behavior is.

Just looking at the code obviously would be relatively difficult.

It would take time.

There is a couple of things in this code.

There is what it does and also there is the different relationships between the classes and the elements that are being used.

So let's look at what this code actually does.

And as you can see here, what the code does is it basically just shows a bunch of balls on the screen.

And the things I would like you to note is that balls are generated when the user taps on the screen.

Each ball is spawned at the location of the tap.

Each ball disappears at the edge of the screen.

Each ball also travels at constant speed.

The purple outline on each ball is the hitbox area of the ball.

This is the area of collision detection.

Each ball also changes color when a collision is detected.

This is what that code does.

There's another way in which to conceptualize your game or any code for that matter, and it is through a diagram.

In this case we're using a simplified UML diagram.

What you see here is the essence of the code we are looking at.

We have two main classes circles, which extends the Flame Game class so that this is our main game class and my collider ball, which is a class that extends from position component, which as you will learn later, is a flame engine component for shapes and objects that can be located on the screen.

Apart from these two classes, we also see their basic relationship circles, aggregates the micro-level class, which means that it controls the lifecycle of my collider ball, which in our case represents each ball on the screen.

There can be zero or more such balls created at any point in time.

The most important thing here is that you can think of circles as being the controller for the creation of the balls that you will see on the screen.

What you see here now is a more complex version of the diagram, which adds more information about the

relationship between the classes used in the code.

Of the main importance would be the following.

We can see that Michael Edible uses canvas, which tells us that this is where the actual balls are drawn.

We can also see that circles or the main game class is where the user's taps are processed since it

uses the tap detector.

We can also see that Michael Edible has access to the main game class through the has game ref mixing so that it can, for example, be altered when the ball goes out of screen through the screen Collider class which is aggregated in the main game class.

What you really see here is more of an architectural diagram which shows a lot of information about the interaction of the classes in your code.

This is just a quick overview, but this is how in this course we will learn about the components and the code that will create.

We'll be looking at some architectural diagrams to help us with understanding sometimes very complex interactions.

So in a nutshell, the better you understand the concept of the application you're writing, the better your code will be.

In this course, we want to understand the building blocks of a game and how they fit together.

So we will spend some time to understand the architecture of the game engine and its fundamental components.

This is very crucial.

The game that we'll write in this course will be conceptually designed first and then mapped to the flame engine's elements in an architectural manner, and the code will flow out of that.

See you guys later in the next lecture.

4. Quick overview of Visual Studio IDE

Let's do a very quick overview of Visual Studio that we're going to be using.

If you're not familiar with Visual Studio, I have given you the links to download it.

Some people use Android.

Some people could be using Xcode, but we're going to at this point concentrate mostly on understanding the game engine, understanding how everything works.

So let's quickly have a look at Visual Studio.

So this is pretty standard.

What you have is you have your project structure here.

You can see, for example here that we have an Android port, we have an iOS port, we even have a web part lib is where we're going to be getting all of our code.

And of course, we're going to go over this in much greater detail.

This is obviously your code.

You're going to have different tabs for different collections of your code.

This is the code outline.

For example, here we have an enum, we have a class.

And you can see, for example, that it shows here that we have the enum shapes.

Right on the left hand side you have, for example, my shape component.

So and you have the main, which is the entry point into the application.

So you can see that here we have the code outline and finally here we have the debugger.

One other thing that is quite important is how do you run your code?

Usually you're going to be running it on an emulator.

So what happens here is if you look at the lower part of Visual Studio, you're going to see this tab, this large tab running at the bottom, and you will see that right here you have, for example, the emulator being used.

If you actually click on that, what will happen is you're then going to see in the upper right corner all the devices that can be used to run your game on.

So whether it's Chrome or Edge or you can actually start a flutter emulator, etc..

So this is just a very, very quick overview.

We're going to jump into actually coding things.

So you're going to see everything when we're actually coding it.

2_setup

1. Setting Up Your Environment

All right.

Let's go into the environment, set up that you will need to do in order to be able to run the code in this course.

So here's a checklist of the items for our setup.

First of all, we'll need the flutter environment.

So we will need to go to the download website, get the distribution.

We need to set up the environment, including any path or anything that will be necessary.

You will then run the Flutter doctor diagnostic, which is a tool that allows you to ensure that your environment has been set up properly.

Next, you're going to install Android Studio if you're on Windows or Linux or Xcode, if you're on a mac, we need this because that's our compilation environment for either Android or iOS.

And also you can use it if you want to have more extensive emulator configuration.

Next, you will need Visual Studio.

Now Visual Studio is going to be our IDE, which is integrated Development Environment.

We'll be using that throughout the course to write and test our code.

Note Going back to the previous mention of an emulator, note that Visual Studio actually comes with its own emulator that we can use to run our code.

We will need to though configure the ID to use dart and flutter extensions because natively or out of the box, Visual Studio does not support Dart or flutter.

Once you have done all that, we will be able to then test your setup with a little Hello world sample code that we have provided for you.

Here are the URLs for the items on the checklist.

So in order to get the flutter environment, you will go to docks that flutter dot dev for Android Studio or Xcode.

For Android, you will go into developer Android for Xcode, you're going to go into developer Apple.com for Visual Studio.

You're going to go to Visual Studio, Microsoft.com.

2. Windows Setup

All right.

Let's talk about how you would set up your Windows environment.

So for the flutter setup environment, you go into docs, dot flutter, dot dev, slash, get started slash install.

Once you get to this page, you've got a number of options for Windows.

You're just going to go in here, make sure that your system requirement is all right.

But basically this supports Windows seven all the way to Windows 11.

So the first thing that you're going to do is you're going to get the Flutter SDK.

That's a zip file.

So what you're going to do is you're going to basically extract it and you're going to unzip it and you're going to put it into a folder like something for development.

Just please make sure that you don't install it into a directory like C program files because that requires elevated privileges.

So I normally do something like I have a development folder and then I put all my development related elements and materials in that folder.

So once you have extracted the Flutter SDK, what you might need to do is you might need to update your path and that's basically you're going to set up your environment variables so that you can easily run things, for example, from a command line without having to go into the specific directory, which is the flutter directory.

You could run it anywhere.

It becomes global.

So go through this, make sure that you set it up properly.

Once you have done that, once you have extracted your flutter distribution and you have set up the environment, the path environment variable, you will need to run flutter doctor and Flutter doctor is a utility for diagnostics.

It will tell you basically whether you're missing anything you're going to see.

What you see here is that Android SDK is missing, assuming that you don't have Android SDK on your computer.

So the next step is going to be Android setup.

So what you're going to do is you're going to follow this link here we go here into Android studio and in here what you're going to do is you're just going to go down here and you will see that you have

all the different distributions here.

We're going to choose the executable.

That's the best one.

All you need to do is basically run it, follow any instructions on the screen, and once it's done, what you would like to do is you would like to go back to your flutter environment and you want to run flutter Doctor again.

You should run Flutter doctor with the Android licenses as shown here in order to have your environment be correctly set up for license agreements.

So once you have set up your flutter environment, once you have set up your Android environment, the next thing that you're going to do is you're going to download Visual Studio.

Now Visual Studio is basically an integrated development environment.

So what you're going to do is you're going to go into this version right here, you're going to click on the free download and you're going to go with Windows X 64.

Once you install this, the next thing that we'll need to do is we'll need to configure your Visual Studio for Dart and Flutter.

So here I have an empty Visual Studio interface, and as you can see on the left hand side, you've got a number of icons.

So this icon, for example, is your explorer.

This is where you're going to see all your code here.

You can do a search global search on on strings so that you can find maybe snippets of code based on maybe a comment or some sort of text.

Next, we have this horse control, which we won't have to worry about at this point.

Then we have the run and debug, which we're going to use.

What you're interested is this one extensions.

So what you want to do is you want to click here and then you're not going to see this installed because you're not going to have it.

I wouldn't worry about this.

Recommended.

I already have Dart flutter and material icons installed, but what you would do is you would go into right here and you would type dart.

What will happen is it will look for Dart.

Here it is.

You're going to click on it and then you will have an option.

Here I have an uninstall here, you will have an option to install, which is what you should do.

You should install it.

Once this is installed, the next thing you're going to do is you're going to go flutter, same thing.

You're going to click and you're going to install.

Once you're done with that, I suggest also that you load and install material icons.

You can go into the material icon theme, click here and you will again click on the install.

And this is the same for whether you're on Windows or on a mac or on Linux.

All right.

So this is basically the next step for you.

It's going to be to run a very simple hello world flutter application, just to make sure that everything that you have set up is working properly.

3. macOS Setup

All right.

Let's talk about how you will set up your Mac OS environment.

If you are a mac user and you need to install Flutter, what you're going to do is you're going to go here into the Mac OS.

I'm going to click on this and then just make sure that obviously you have enough disk space, etc.

and then what you're going to do is you're going to download this installation bundle right here, and that's going to give you the latest stable release.

You should then extract the file in the necessary location, like the desired location where you want.

I usually put it in some sort of a development folder and then make sure that you add the flutter tool to your path.

And then once you have done that run, Flutter, Dr. Flutter Doctor is going to tell you whether you're missing anything.

Look at what it's asking you to do.

But the next step that we're going to do, once you update your path, it's going to be Xcode.

You're going to either go into Web download or the Mac App Store.

You will need to install and configure Xcode.

Just follow these instructions here.

This is perfect.

We're going to talk about deployment once we're done with the game.

So now that you have installed Flutter and you have installed Xcode, the next step is going to be to install Visual Studio.

So what you're going to do is you're going to go into the Microsoft Visual Studio of Microsoft dot com downloads.

You're going to take this version right here and where it's free download, you just go and get the Mac Universal package.

I'm not going to download this because I happen to be on Windows.

But once you get that, once you have installed it, the next thing that you will need to do is you will need to configure it for dart and flutter.

So here I have an empty Visual Studio interface, and as you can see on the left hand side, you've got a number of icons.

So this icon, for example, is your explorer.

This is where you're going to see all your code and your dependency.

Here.

You can do a search global search on on strings so that you can find maybe snippets of code based on maybe a comment or some sort of text.

Next, we have the source control, which we won't have to worry about at this point.

Then we have the run and debug, which we're going to use.

What you're interested is this one extensions.

So what you want to do is you want to click here and then you're not going to see this installed because you're not going to have it.

And I wouldn't worry about this.

Recommended what you really want.

Like you see, I already have dart flutter and material icons installed, but what you would do is you would go into right here and you would type dart.

What will happen is it will look for Dart.

Here it is.

You're going to click on it and then you will have an option.

Here I have an uninstall here, you will have an option to install, which is what you should do.

You should install it.

Once this is installed, the next thing you're going to do is you're going to go flutter, same thing.

You're going to click and you're going to install.

Once you're done with that, I suggest also that you load and install material icons.

You can go into the material icon theme, click here and you will again click on the install.

And this is the same for whether you're on Windows or on a mac or on Linux.

All right.

So this is basically the next step for you.

It's going to be to run a very simple Hello world flutter application just to make sure that everything that you have set up is working properly.

4. Linux Setup

All right.

Let's talk about how you would set up your Linux environment.

All right.

If you're a Linux user, once you get to this page, what you're going to do is you're going to click on Linux.

Just make sure that your system requirements are good.

And then what you're going to do is you're going to get the Flutter SDK.

You can either use Snappy or you can basically just do manual installation.

Manual installation is basically you're just extracting an archive and then setting it up.

So for example, right here you have the necessary commands.

Please also make sure that you set up the path for Flutter so that you can use it from anywhere in your terminal.

And then once you have done that, make sure that you run the Flutter doctor diagnostic tool, whi

5. Testing Your Flutter Environment with Visual Studio

All right.

Welcome back.

What we're going to do right now is we're going to test our environment with Visual Studio.

But before we do that, let's have a look at how everything is laid out.

The flutter framework is what the Flame game engine is running on top of.

So we need to make sure that this part is working properly with Visual Studio, which is the tool we're going to use before we continue with any of the flame code.

As you see, flame is on top of the flutter framework.

First, we're going to make sure that your flutter environment is working properly after you have it set up.

We will create a standard flutter project in Visual Studio.

We will then replace the standard code that was generated with our own test code.

There are two ways to do it.

One is you simply paste the test code into the code that was auto generated by the flutter application.

So you open the file with our code, you copy the code, and then you paste it into the main dart.

Or you just replace the whole file with the new code, which is the simplest way.

You simply drag and drop and you overwrite it.

And that is what we're going to do.

All right.

So what we're going to do right now is we're going to test our environment for Flutter with Visual Studio.

We basically need to make sure that the flutter environment is working.

So what we're going to do is we're going to run a little bit of our own flutter code to make sure that it's working.

So the first thing that we're going to do is we're going to generate the project from scratch.

So we're going to go into view command palette.

And inside here you will type.

Flutter new project.

You have it right there.

You can click on that.

And here you have a number of different templates.

We're going to choose the one to generate a flutter application.

This is going to ask you right now where you would like to do that.

I am just choosing a dev directory, but you can choose any directory of your liking.

So I'm going to say let's create our project inside the dev directory.

Perfect.

Now this is going to be the name of your projects folder, which is basically going to encompass all of your source code and all the different artifacts that your project needs.

I'm going to call this one Hello World.

Perfect hit enter.

This is right now generating perfect.

We have the source code.

So what we're going to do first is we're going to remove this code inside main door dart and we're going to replace it with the code that we have provided you with the course.

And this is normally what you will be doing throughout the course when you're going to be trying the new code that we give you.

So what I'm going to do is I'm going to open the folder.

Inside the source code that we provided you.

Once you go in, you will see a number of different directories.

You're going to go into the setup one inside here, you're going to go one more time into setup.

And here is the Hello World project that we have provided you with.

So we're going to go in.

I'm going to go into the lab.

This is the source code name Dart, and this is pub spec YAML, which we don't need right now because the one that comes with flutter is good enough.

So we're going to go into the lab and all we're going to do is we're going to grab this and we're simply going to drag and drop it here.

It's going to ask you whether you want to replace it.

You will say yes.

And as you can see, this is the source code of our own little test application.

So now the question is, how do you run it?

What you will do is you can go right here at the bottom on the right hand side of the bottom.

If you click on that area, there's going to be a number of different devices here.

And what I will show you right now is how to actually create a new emulator.

So what you're going to do if you don't see it, you just scroll down a little bit and you'll see here that it says create Android emulator.

Or you could basically just type here, create Android emulator and it shows up on iOS.

You would go start iOS simulator when you click on this, this is going to now create a brand new emulator for you and you will be able to reuse this emulator later on.

I'm just showing you what will happen if you do it for the first time.

So it's creating the emulator there.

It is just going to move it up a little bit, make it a little bit larger.

We're just going to wait for it to connect.

So this emulator right now is connecting to Visual Studio.

Might take a little bit of time.

Whenever you're creating a new emulator and whenever you're doing it for the first time, it will take a little bit of time.

So have a little bit of patience.

I, of course, am speeding this up in the video so it will go a little faster.

All right, perfect.

So we have now an emulator running great.

So now what we're going to do is we're going to run our code inside this emulator.

So you're going to go into the upper right corner, you're going to click on that arrow.

You're going to go run without debugging.

It's going to take a little bit of time.

Notice how the debug console has opened.

It's going to give you some information.

It takes a little bit of patience, especially when you're running something for the first time.

All right.

And success.

Awesome.

So our code is running.

If you got to this point, everything is good.

You're ready to definitely use the source code from the course.

And I'm going to see you in the next lecture.

Congratulations.

Take care.

3_2d-game-basics-with-flame-engine

1. The Game Loop - Flame Engine's Secret Weapon

One of the most important concepts in any game engine, but especially in the flame Game Engine is the game loop concept.

So we can basically say that at the heart of the Flame Game engine is the game loop concept.

In a nutshell, what it is, a game loop is an infinite loop, which continuously runs through a set of very simple steps.

It gets the input data, it then updates the game state with the new data, and the input data could be a joystick input.

It could be, for example, you tapping on the screen and once it gets this input data, it then renders the game state, whether it's the screen update or whether it's audio.

It renders that back to the user.

So the simplest way that we can look at it is as follows You start the loop with getting the input data.

You then update the game state with the new data.

So you grab that data from the user or from a timer or from wherever your input is coming from.

You update the state of the game with that.

And finally, the state itself is rendered back to the user so the user can see what is happening.

Now, one question obviously that could be asked is how tightly, how quickly does this game loop run?

And typically for the Flame Game engine, you're running 30 PS up to 60 PS.

That's all for now.

See you again soon.

2. Game Instantiation - How does Flutter Run Flame Games?

Let's talk about the game instantiation.

What happens is that the system calls the main function.

This is like your typical flutter setup.

Then it calls the run up.

It then calls the game widget.

So these three elements, the system calls the main, which calls the run up, which calls the game widget that then instantiate and calls our game instance.

The game instance then calls unload, which happens basically one time.

But every time the game is reloaded, that's what happens.

But it's a one time call.

It's outside of the game loop.

And then what happens is we go into the gaming loop, which is the update and render.

And basically what happens is that the render updates information on, for example, the mobile device

and it shows the user what is happening.

So let's quickly recap it.

The system calls the main, which is your entry point to the application that calls the run app, which

holds basically the flame engine hook that calls the game widget because Flutter uses widgets.

Our game instance is a game widget, and then what happens is the unload method so that you can initialize

the application is called and then it goes into the loop, which goes quickly between update and render.

Of course, somewhere in between you also have the input, which is not shown here.

All right, see you guys in the next lecture.

3. Game Execution - How does Flutter execute and run Flame Games?

Let's talk about game execution, which is that element after unload.

So let's have a look here.

So what happens as stated before, is we have the unload call, which initializes the state of your

game.

And then we go into the loop, which calls the update and the render methods.

Now, the actual time that this happens is known as Delta time, so we can think of it this way.

The update method is called periodically by the game engine to request that the components update their state.

So what's going to happen is that every so often the system is going to call all the components in your

game to update themselves and it's all going to happen basically at the same time.

So every time the method is called, it provides a delta time since the last call.

So the Delta time is the time since the last call.

It's not a cumulative amount of time.

It's just how many seconds have passed since the last update call.

So the actual delta time is given in seconds, but it's with microsecond precision, which is pretty

incredible because if you think about milliseconds, which is 1,000th of a second microsecond is 1000

times more than that.

Here's an example of what it looks like.

So, for example, we have a very quick little update loop here that shows the time between the calls.

So here is where you have the Delta time and you can see that it's actually very, very quick in the order of 0.01 second.

So to recap this, the game execution is primarily concentrating on updating the state at a Delta time, and then it renders that state.

4. Generating Our First Flame Project

Hi, guys.

We're going to create our first very basic application, which will showcase two things.

One is how to set up your code for Flame Engine.

And we will also explore the game loop.

We'll start with a command line tool.

I'm just using Windows right now, but this is the same under Mac or Linux.

First, we'll check the version of Flutter to make sure that Flutter works on your system.

If your version is slightly different, it's probably newer than mine.

That should be fine.

Now that we checked that our flutter is working properly, we'll use the flutter.

Create command to create a new flutter project.

I will name mine Flame Game Loop example, but you can name yours something else if you want to.

So here we go.

As you can see, it is creating the whole project structure.

Isn't that great?

Let's quickly look at the folder structure that was created.

Note the web iOS and Android folders.

This is where the specific code will be created to run on those platforms.

Our source code will be in the lib directory, just like in a normal flutter project.

All right.

This project is ready for us to open in Visual Studio.

What we're going to do next is we're going to remove the sample code.

We're going to add the flame engine dependency, and we're going to add our very basic game loop code.

5. Running Our First Minimalist Game-Loop Code

All right.

So what we're going to do right now is we're going to look at the code that was generated with the create method.

I'm going to go into Visual Studio and I'm going to open the folder.

Wherever you have put your code, that is where you're going to go.

In my case, it's going to be Flame crash course, and here it is, Flame game loop example.

When I open it, you will see on the left hand side that we have the Android iOS and the web folders.

We have the lib folder with our source code.

Now this is the source code that was generated by Flutter.

We're going to remove all of this.

We don't need this class.

We're also going to remove this since we've just removed that class itself.

What are we going to do right now is I'm going to put in the minimalist flame engine code, and we're going to then add the flame engine distribution into our project.

So first we're going to add the game widget, and the game widget takes one parameter, which is the game, which is going to be our game class.

We're going to name this class my first game.

So now what we need to do is actually create this class, which is going to be the flame engine based class.

So what we have just done is we've created our class, which is super minimal.

As you can see, we have done really nothing more than extend the Flame Game class, which is a component

that incorporates most of the flame engine elements.

So what we need to do right now is fix these issues that we have.

So if I go here and I go quick fix, you'll see that the only thing, the only choice I have right now

is to create the class flame game.

But clearly this is a class that comes from Flame Engine.

And the same thing with this game widget.

So what we really need is to be able to import the flame engine libraries, but we have not incorporated the flame engine yet, so we're going to do that right now.

So we are going into the pub dot dev flutter repository and we're going to look for flame.

Here we have it and minimalist flutter game engine.

So we're going to go in.

What you'll see right now is that the current version is 1.10 and it was only published a couple of days ago.

This course is actually written and developed around Flame 1.0.

There aren't major differences between the two and eventually we're going to update it.

But at this point, we would like you to work with the code that we've created, which works very well with 1.0.

So what you're going to do if you see a version like 1.1, is that we're going to go into versions and in here we're going to look for 1.0, we're going here.

Then you basically just click right here or you can just copy this.

Or you could very much just remember that it's flame 1.00.0.

I'm just going to hover over here and click on it.

And that has just copied that string to the clipboard, which we're going to use right now in our code.

So we're going back to Visual Studio, we're going to go into the pub Spectre, YAML, and in here we're going to add two dependencies the flame engine.

Okay, perfect.

So we have that right now.

I'm going to save this file.

As you can see, there's there's two files that are unsaved right now.

One is the main dot, dot.

The other one is this one.

So we're going to save this.

And as you can see, Flutter is executing the Pop get command, which actually brings the flame engine dependency into your project.

Okay, now that we have this, let's go back to our source code.

We can now fix this up.

As you can see now here, if I go quick, fix it actually shows me the flame, dot, dot.

And this is perfectly fixed right now.

So what we have right now is a minimalist, the smallest code that you can have.

So the next question is OC, how can we run this code?

If you go to the bottom here and you click on this part right here, what I'm going to do is I'm going

to start the flutter emulator.

If you if you don't have this start flutter emulator, you can create an Android emulator, which is going to create that.

So I'm going to start it so as the emulator is.

Starting right now.

This is perfect.

We're just waiting for it to connect.

Might take a little bit of time for it to start the first time.

Perfect.

So we have it ready for us right now.

So what we're going to do is we're going to run this code and you do that by going into the upper right corner here and we're going to go run without debugging and see that it's running Gradle.

And we're just going to wait for it to run.

You can see in the debug console what is happening.

So it might take a little bit of time.

The first time when it's running, it's kind of like cold running.

So it takes a little bit of time for it to start.

All right.

So it's just about to run.

Perfect.

Our code is running, but obviously you're going to say, well, I mean, it's just a black window.

But actually what is happening right now is that our code is running through the game loop.

It's continuously updating the screen, which happens to be fully black right now.

So first, I want you to just like look at this and you will see that there's a couple areas in here.

First of all, we have the bar at the top, then we have the canvas and then we have the thing at the bottom.

Now the part at the bottom is more part of the emulator.

It's not really part of the phone itself, but this usually is rendered in the phone.

We'll figure out later on how to get rid of that so that our game can run full screen.

All right.

So now that we see this code running, we're going to tap into the three most important methods for any game.

6. Adding Explicit Game-Loop Code Hook Methods

So one of the things that you're going to notice right now is when we're looking at the emulator, nothing seems to be happening.

But in fact, there's a lot happening.

The game loop is running continuously right now.

So what we're going to do at this point is we're going to write a little bit of simple hook code to get our code into the methods of load update and render.

So let's do that.

So first we're going to add an override for on load.

We're simply going to remove this and we're just going to add a very simple print statement.

Basically what we have done here is we're overriding the on load method.

We're going to add two more methods.

We're going to add the update method.

We're going to remove the common.

And we're going to add also a print statement here.

What we're doing here is we're basically going to print the Delta time that this method gets called with.

So we're going to see basically how often this is called, but not only how often what the Delta time gives us is the amount of time in seconds that has passed since the last call to update.

And the last method that we're going to override is going to be the render method.

And we're going to do the same thing.

We're going to print.

All right.

As you can see, we haven't really done anything special.

All we have done is just written little hooks with the unload update and render methods.

And all we're doing is basically just printing some statements to the debug console.

So we're going to run this right now.

And as you can see, if you're looking at the bottom here, you can see actually that this is continuously printing.

So what is interesting here is that you're continuously seeing a print of our print statements.

So first, let me just filter that out on on load.

And of course, on load is not going to show here because we need to do the return statement after.

So I'm going to restart it.

Perfect.

What I'm going to do is filter the on load.

And as you can see, unload has been called only once, which is exactly how the unload method is called.

Next, let's have a look at the update.

And as you can see that the update method, you can see the kind of the scroll bar moving here because this is continuously printing.

And as you can see, that each time the update method is called, it gives us the delta.

So this this little part right here is the amount of time that has passed since this one here was called.

And then if we look at the render, you can see that the render is called.

But more importantly, if you look at what is happening here is you can see that there's an update render, update, render.

And this is how the game loop works.

What it does is it calls the update method so that the data can be updated and then it calls the render method so that you can render what has been updated.

All right.

We're going to do two things.

First, we're going to change the background color to red, and we're going to do that inside the render method.

We're going to use the canvas itself.

We're going to use the draw paint method, and we're going to instantiate that with paint object and we're going to initialize the paint objects color attribute to the color red.

And as you can see, we have all sorts of red colors here.

For example, we could choose this shade of red to 100, and the way you would do it is go here and there's the shade to 100 and you can even see the shade right here.

I'm going to rerun it.

And as you can see, each time that the render method is called, it renders the canvas with this color.

One last thing that I wanted to do here at this moment is we're going to remove this bar at the top, because usually when you're writing games, you don't have that bar at the top.

So what we're going to do is we're going to go to the top.

And just before we instantiate our game widget, we're going to instruct Flutter to remove the bar.

So we'll do that with the widgets, flutter binding class and we're going to use the static and sure

initialized method.

This way we'll be able to make changes to the main widget.

We're going to ask the flame engine to remove the bar.

And as you can see, it is not recognizing this class.

So we're going to look at what the problem is and we're going to go into the quick fix and we're simply going to have to add the flame dart.

This is done.

We can now go back here.

And as you see, the top bar has been removed.

So this right now is a very, very simple way of looking at how the game engine works and what you see here.

These particular methods are probably the most important methods and understanding not only the flame

engine, but the whole kind of workflow of any game engine, which is the game loop itself that works with getting data and then updating the state of that data and the game and then showing the update to the user via the render method.

One thing that I would just like you to note is that look how frequently this is called.

We're going to see how that translates into how many frames per second.

But the idea is basically that these updates are happening so quickly that, you know, this is this is happening many, many times per second.

The Flame Game Engine can run in 30 apps or 60 apps, and it's variable just like for most games.

So we're going to have a look at that as well as how you can actually see how many frames per second are being run.

So first, we're going to add a tab detector that will allow us to detect when the user is tapping on the screen of their mobile device.

I'm going to just comment out the Sprint statements so we don't have to see that happening.

And as you can see right now, nothing is happening here.

So let's add tab detection.

So we're going to do it as follows We're going to add tab detector mix.

And this comes with a method which is going to be on top up.

So in this case, we're going to have a look at when the user is tapping up.

And since we're not percolating that anywhere, we don't need to call the super.

We're just going to add a print statement here.

We're going to look at the location.

It's going to grab this and copy that for the Y location.

So let's run this and let's look.

Let me clear the console.

And let's have a look here.

So you can see that the location is 165 and 188.

We can we can kind of round it up.

But if I get to the left upper corner and click there, you see that it's very close to zero zero.

This this little thing right here would be close to about zero zero.

I mean, this is we're talking about fractions of pixels to to a great extent.

All right.

So this is now how we get tap information.

So as you see as I'm clicking around, this is telling me the information.

So this is how we can add the tap detector.

One more thing that we're going to do is I'm going to change the color to just a normal shade of red.

So as you see, this has changed to red, a slightly deeper red.

Perfect.

And now what we're going to do is we're going to do one more thing.

We're going to add an EPS counter mixing so that we can see how many frames per second are being rendered.

And the way you work with this is that this EPS counter gives us information from the engine about what the numbers of frames per second are.

So we're going to add some code into our render function to render that information to the screen.

And I'm just going to write a comment here so that we see what we're doing.

Okay.

So what are we going to do right now is we're going to get a formatter first.

All right.

So we need to get the number format right now.

If you go here, if you go into the quick fix, it doesn't really know where it is.

So we will have to put that package ourselves.

This is actually an internationalization package.

So we're going to import that.

And what I have just done is I have actually pulled that package.

I've asked Flutter to get this package for us.

This is just strictly a DART number formatting so that we can actually show something on the screen that looks relatively good.

So now that we have the formatter, what we're going to do is we're going to actually format our string.

But first we have to get the string.

We're going to create the full string that we're going to output to the screen and we're going to grab the string formatter string and we're going to format it.

And now we're going to grab the number of frames per second up to 60.

This part right here comes from the PHP counter and you can actually look here.

If I if I click on I'm holding right now the control key.

And if I hold the control key and kind of hover with my mouse over it, it shows you the information here.

And basically you can see that this PHP method, what it does is it returns the PHP based on the frame times from the callback.

And in our case, we're basically saying that we're looking for the average to be about 60 frames per second.

So this will now give us up to 60 frames per second.

Now that we have this string, what we're going to do is we're going to output it.

So what we need is we need to get a font, we're going to go up here and we're going to create a static font and color definition.

We're going to initialize it with the color itself and the font size.

We're just going to go with color black, and the font size is going to be 12.

This creates the definition of the text, font and color that we can use to render onto the screen.

We're going to go here and render it.

And what is interesting is that this particular text is a component that can be rendered.

So we're going to render it.

The text itself that we're going to output is our PHP string.

So what we're doing here to kind of quickly explain it is we're going to paint the canvas red, We're then going to create a string formatter which is going to basically show two digits and it's going to be in the kind of English us, for example, from left to right as opposed to right to left.

Next, we're going to actually create the string that we're going to show on the screen, which is going to be PHP plus the formatted string here.

And the formatted string is going to be the PHP information that we get from the PHP counter here from the mix in next, what we're going to do is we have defined a component which is a paint component.

It's specifically a text paint component, and we're going to render it onto our canvas.

We're going to render the string that we have defined here and we're going to render it at the location in the upper left corner with the coordinates of 1010.

So let's run this.

And as you can see now right here, it is showing us the frames per second.

So as you can see, for example, right now, it seems to be stuck at 30.

As you can see, it hovers.

It kind of jumps around, but it sticks around 30.

Okay, So this is it.

This is just a very, very simple way of showing you how we can work with the game engine.

Next, we're going to look at all the different components and we're going to learn more about how to actually put objects on the screen.

7. What are "Game Components" In the Flame Engine?

Let's talk about components in flame.

The central piece of the game engine is a component.

A component is a venerable and updatable entity, which acts as a building block for your game.

You can think of it as the Lego pieces for your game.

Consider a simple game such as Atari's asteroids.

All of the following are components the spaceship.

Each asteroid that you see, the bullets that the spaceship fires, explosions.

The timer used to periodically show a UFO flying for bonus points.

It's interesting because timer is not really a visual component, but it is still considered a component.

It ties into the updatable aspect.

Even the background parallax is a component, so you can think of your game in essence as a collection of components.

Now all of these, when put together, create a component tree.

So you construct your game by building a component collection, which can be viewed as a tree, which

at its root starts with the game class instance.

So for example, at the Delta time, every time you have an update, what happens is the update method of the game is called that in effect calls all the other update methods on the parallax background, on the spaceship, on the asteroid, on the UFO, which also goes even deeper.

If you, for example, have the parallax background made of two elements such as stars and galaxies, It will call that if you have bullets and explosions for the spaceship, it will call that.

So in effect, you can look at the update method as simply calling an update on all of the components at the same time, roughly at the same time.

We're going to get into that a little bit later when it comes to priorities.

But this, in essence, is what a component tree looks like.

We can take this paradigm of a tree component and the relationship between the components.

We can take it a little further.

It's not just the update method that is related to the other components.

This way, if you look, for example, at the way the render method works, imagine that the render method is called that some delta DT prime, which is a little bit after the actual update method is called, you will see that the render method of the game widget is called, which then percolates or goes down to the parallax background, the stars, the spaceship and everything.

And it does the same thing.

The render method calls all the render methods in the tree.

We can take this even further.

If you look at the way the unload method works, which is usually called only once in the lifecycle of the game.

So that's why here we show it as being called at start, the same thing happens, the game widget has an unload method, but then all the other components, all the other widgets also have an unload method.

It gets called at the very same time.

So this is a very important paradigm in the way the component tree works.

We will see more of this and how important it it is as we're writing more code and we go through the different case studies.

Okay, guys, we'll see you in the next lecture.

4_flame-engine-case-studies-overview

1. Case-Studies: Introduction

In this course, we're going to learn about the flame engine through the usage of case studies to understand

the flame engine.

We'll take a dual approach.

We're going to use the architectural approach, which will allow us to look at a more of a bird's eye

view at the engine's inner workings through UML diagrams.

We're also going to take a divide and conquer approach where we'll look at specific code snippets that

show us focused aspects of the game engine.

We're going to go one topic at a time.

We will master each snippet first before we move on to the next one.

And the idea is basically that in the case studies we will be looking at the main components and utilities

that flame engine has to offer, such as displaying and managing components, working with timers,

parallax, joystick, particle physics, collision detection, etc..

The idea is that we're going to concentrate at one manageable chunk of information at a time.

We're going to master it, and then we're going to move on to the next one.

2. How to Run The Provided Lecture Source Code

All right.

So what we would like to talk about now is how would you load up the source code provided with the course?

So as we were talking before, when you look at a typical flame game application, because it's built

on top of the flutter framework, you first have to make sure that your flutter environment is working

correctly.

Then your flame engine is actually sitting on top of the flutter framework.

So there's going to be a couple of things that we'll need to do to the flutter environment to make it

flame engine compliant.

And then of course, your game or our source code is going to be running on top of that.

So let's have a look at how it is that we're going to do that.

When it comes to running the flame source code.

Once your flutter environment is working well, we will run the flame sources that are provided with the course.

That's going to be like the bulk of what you're going to be doing in this course is you're going to be working alongside the source code that we provided you.

The workflow to do that is basically the same as with when we're testing the Flutter project.

You create a standard flutter project in Visual Studio.

You then replace that standard generated code with the provided source code, and the simplest way to do it is you remove the originally generated main dart or you don't even have to actually remove it.

You can simply overwrite it.

Then you just drag and drop all the source files into the lib directory of the project.

You can do that directly in Visual Studio as we'll show you.

And then you will also need to copy the perspective memo that was provided with the sources and overwrite the existing file of the project.

You can again just drag and drop the file if there are any asset files.

You also want to drag and drop the whole assets directory into the project as we'll show you.

All right.

So as usual, you're going to be starting with an empty project in Visual Studio.

So what are we going to do is we're going to go into view command palette.

We're going to create a new flutter project.

We're going to use the application template.

We're going to go right into wherever you want to create your project.

I'm doing that in my dev directory, so I'm going to choose this directory as the project folder.

So next what I'm going to do is I'm going to give it a name and we're just going to call it Joystick Project 001.

Once you've done that, you hit enter.

This is creating the full project with the source code.

Most importantly, everything that you see here is basically the environment that was created for the Flutter project.

We need that.

But what we're going to do is we're going to get rid of this and obviously put our own source code.

We're going to go and open our source code folder, we're going to go in, we're going to go into the case studies and we're going to choose, let's say, this joystick.

So we're going to go in.

Then let's say we're going to choose the second one.

We're going to go into the lib.

We're going to grab these three files, which is the source files, and we're just going to drop them right here into the lib, of course, because there is already a main Dodge Dart here.

It's going to ask us, hey, do you want to replace it?

Yes, we do.

Because what we're doing is we're replacing this code that you see on the screen.

Great.

Here's our main Dodge Dart.

But of course, everything is red because it is still a flutter project and we need it to be a flame project.

So what are we going to do is you're going to go back and go up one level.

And now what we're going to do is we're going to grab pub spec YAML.

And we're just going to drop it right on top and replace it.

And what this pub spec dot yaml has, it has the flame, as you can see, which it just popped out.

Get just got it.

And also we have defined our assets here.

So the last thing that we need, as you see, we do not have assets here at all.

So the last thing that we need to do.

If you want to grab this assets folder and we want to drop it right in here and it says add folder to the workspace or just copy folder.

We're just going to copy the folder.

And as you see, because you're not creating a new workspace, you want to make sure that step is important.

You want to make sure that you're copying the folder.

And as you see, what happened right now is it created the assets slash images folder.

We have our asset in here and now what we can do just to make sure that this is working, everything is good.

I'm going to get rid of this.

I'm going to get rid of this.

I'm going to go into the lab.

Just open that.

Here's our code.

We're going to run it.

Sometimes it takes a little bit of time to run it.

And here is our joystick.

And as you can see, the code is running fine.

Bullets are shooting.

It's all good.

So that way you are able to basically use the Flutter project scaffolding to put our source code in.

And if for whatever reason you did not have the emulator, let's say it didn't start with it, I'm going to get rid of it.

I'm just going to show you quickly how you can get the emulator.

All you would have to do is you go into the lower right corner right here.

You click on that.

And you'll have your emulators here.

This list right here is emulators that you have already used before or created.

As you've seen, I've used quite a few of them.

If you don't have one, you would just go here and say, Create Android emulator.

You just have to remember that this you have to scroll it.

Or if you are on a mac, you would be dealing with an iOS simulator.

But the idea is essentially the same.

I will just pick this one.

It launches this particular emulator.

Perfect.

And that's all you would have to do.

Then you can run it without debugging.

Always takes a little bit of time.

And there it is.

It's working.

So this is how I suggest that you bring in the source code from the source code that we have provided

for you so that you can go along with the lectures and understand what is happening, go through the source code, play around with it, etc. You basically create a flutter project.

You then replace the source code and the spectre YAML and add any assets that you need from the source code that we have provided you.

All right.

I'll see you in the next lectures.

Good luck.

5_case-study-1-flame-engine-essentials

1. Case Study #1 - Introduction

So let's look at what our first case study is going to be.

We're going to learn about the following flame engine elements.

We're going to learn about the component class, the position component class.

We're going to learn about a very important vector class that allows us to control all the aspects of the position and the motion of a component.

We're going to learn about object velocity, which is the vector that controls how fast it's moving and where it is going.

We're going to learn also about angular velocity, which is how fast and in which direction the object is rotating.

We're going to also learn about the anchor property of an object, which is the logical center around which we can, for example, rotate the object.

You could rotate it at the middle of the object, or you could rotate it, for example, around one of its corners if it happens to be like a rectangle.

We're also going to learn about the component tap detection.

If I have, for example, a square on the screen that is moving, how would I know that that particular square has been tapped as opposed to just anything else on the screen?

We're going to learn about that.

We're also going to learn how to pause the game engine.

We're going to learn about random number generation and we're going to learn about different shape components.

2. Position Component

All right.

So in our first case study, we're going to deal with a very important component, which is the parent class of many other components, the position component.

This is a base class for visual components with a few very important attributes.

First, each position component has a position which is the screen position of the component.

In other words, where is it on the screen?

It also has a size which is used mostly for cap and collision detection because it has to do with how large the component is.

It also comes with a property which represents the angle, which is the angle of the shape in radians.

So, for example, we could rotate it a number of radians.

Now radians and degrees are used interchangeably.

You can think that in a circle you have 360 degrees and a circle is two pi radians.

Pi is basically 3.14 radians and pi represents 180 degrees.

So for example, 45 degrees in radians is 180 divided by 445 or since pi is equivalent to 180 degrees, it would be 3.14 divided by four in radians.

We're also going to learn about anchor, which is used to represent the logical center of the shape.

In other words, if I want to rotate a shape, what is it rotating around?

Is it rotating around its own center?

Is it rotating around, for example, one of the corners?

You can specify that using the anchor.

We're also going to learn about how to generate different types of shapes, like a circle or an ellipse, a rectangle or a square, and also a much more versatile way of generating shapes, which is through a polygon, which is where you can define different vertices that are then joined together.

So for example, a triangle is a polygon with three vertices.

Rectangle, of course is a shape with four vertices, but something like a pentagon would be five vertices and you could basically define any shape by creating the vertices for that.

With the position component, we're going to learn how we can control the position and speed and the rotation of the object.

We're also going to learn later on during collision detection what a collision hitbox representation is.

But we'll learn a little bit more about that later.

All right.

So what will the actual case study be?

Well, in this case study, we're going to look at how to render shapes on the screen that move based on a velocity parameter and respond to a user tapping on the screen.

A tap on the screen will spawn a new shape.

In our case, it's going to be a square and tapping on the square, which will give us component type detection, will reverse its velocity.

So if it was going down when you tap on it, it will go back up.

Double tapping on the screen will pause and unpause to game engine.

We will then add the ability to have the shapes rotate at a certain speed as they are moving.

And we'll also keep score of how many shapes have been generated even when those shapes leave the screen.

We will also track how many objects are out of bounds, which is when they have moved out of the screen.

3. "Squares" Code Demo

All right.

Let's have a look at the first use case code.

What we have here is an empty screen.

When I click on it once, what it does is it generates a square child component of the game.

If I click on this square, it will reverse its action.

So basically what I'm doing here is I have the ability to detect where I'm clicking on the screen,

and if I happen to click on a moving square, it will reverse the squares action.

If I click outside of the square, it will generate new squares, and if I click on them again, it

will reverse their action.

If I double click now, it will pause the game engine.

If I double click again, it will unpause it.

One thing to notice here, though, is that all of these shapes are moving at the same speed.

And when we generate them, they're all moving downwards.

And when I click on it, it simply reverses the action to move the object in the opposite direction.

So I can click on this one again.

And as you can see, if I generate one here, it's going down.

If I click on it, it's going up.

If I click on it again, it's going down.

One other interesting aspect of this code is also what happens to these objects when they leave the

screen.

Are they still in memory?

Are they still alive, or is something happening where we're removing them?

We'll have a look at that as well.

So this is just a very simple demo of the code we will be looking at.

4. "Squares" - Architectural UML Diagram

All right.

What we're looking here, it's a UML diagram of our first case study code.

What you're seeing here basically is as follows In the middle right here, where it says main, dot, dot, we have two classes.

We have component examples, 001 and we have squared these are the actual two classes in our code.

Notice that the component example, it actually extends the flame game engine.

Notice though, that the flame game itself is actually a component.

You can see it here at the top.

Notice that the component right here has zero or more children that happen to be components as well.

In other words, Flame Game.

The basis of all our game code is a component, and the component itself contains subcomponents which are the children.

This is quite important because in the code you will see that whenever we add something it actually is added into the flame game engine as a component.

So our component example 001 is a flame gain extension.

Notice that component.

Example 001 has one method.

The main method that we worry about, which is on top up.

Notice also that component.

Example 001 uses a mixing, which is tap detector from tap detector.

It gets tap up info, which is the information about when a user taps and we also notice have a double tap detector.

So what you're seeing here on the left hand side is the information about what our class uses.

Also, notice this line at the bottom that shows that our component example 001 uses a vector.

We'll see that in a code in just a bit.

Now notice that component.

Example 001 has a number of children.

Actually, these children here are the same ones as here.

I'm just showing it here for clarity.

What component Example 001 does is whenever you add a new square, like when you tap on the screen, you're basically creating a new component, which is a square notice.

That square itself is a position component.

Position component has a position which is a vector.

We need a little bit more than just a position.

So for our square, what we're doing is we have added a number of attributes.

We've added the velocity, which is the vector that specifies in which direction it goes and at what speed it runs.

We have also added the size of our square and we have added an additional color.

Notice that we override the three methods on load, render and update.

If you go a little bit further to the right, you can see, for example, that the position component, which is the one that we're extending, it contains a position which is a vector that makes sense, that is the location of the component.

Notice also that square uses a canvas that happens in the render method.

Notice also that the square uses an anchor, which is how we define the anchor point for our component.

We're also using a basic palette, and that is to specify the color.

And we're also using a painting style, which is going to specify how we're going to draw the shape.

We're going to draw it as an outline instead of, for example, a filled in object.

So one important thing to note here when you're looking at this is that in reality you can see that there is a relationship between component and square, where the component example contains many instances,

zero or more instances of our square class.

Notice that square also has a velocity, which is a vector.

We see it right here and we're also seeing that component.

Example 001 It also uses a vector.

We'll see what it is using it for.

So this is just a very quick outline of our code.

Now let's have a look at the code.

5. Coding "Squares"

All right, So let's have a look at the code.

You can just follow the code that we're going to show here.

But if you want to actually play with the code yourself, you're going to find it in the case studies directory underneath the component dash 001 square shape directory.

And then there will be one more directory which is component underscore 001, underscore 001.

And that's when you're going to see the lib with the main dot dot, which is this particular code.

All right.

So first of all, let's have a look at at the outline of the code.

Whenever you want to know what kind of methods you have, if you just go at the bottom here and you click on this outline, it shows you your code in a visual structural manner.

So, for example, here we can see that we have Main, which is right here we have Square, which is this class right here.

And then we have component example 001, which is this class right here.

And you can also see that square, for example, has a number of attributes, velocity, square size and color, and it also has a number of methods.

Unload, update and render.

Same thing with component.

Example one It has an attribute called running on tap.

So this is on on the left hand side here.

If you want to have a quick look at all the methods and the attributes, this is how you would do it.

Of course, this particular area right here is simply your project and especially spec YAML, where we have the information about, for example, this is where we're specifying that we're using frame 1.0 and this is usually when we'll set up dependencies.

But the most important thing is inside lib, and that's the main dart right here.

So let's have a look at the code.

So first of all, when we're looking at the main method, the main method is the entry point into our code.

And as you can see, all we're doing here is we're creating and component example 001 object and then we're basically adding that to our game widget.

So this is what basically runs our flame engine code.

Now we have two classes here, as you remember from the architectural diagram, there are two main classes there.

So first of all, let's have a look at the square class.

So I'm just going to open it up a little bit.

I'm going to scroll up so you can see that Square extends position component.

And if I go here, you can see the basically position component as discussed in the architectural diagram

is an implementation that represents an object that can be freely moved around the screen, rotated and scaled.

Great.

So our square extends the position component and we have three attributes velocity, square size and color.

Notice that velocity is just a vector zero zero, which is normalized and it's given a size of 25.

Now, vector zero zero basically means that it's not going anywhere.

It has no real direction.

So this is really velocity of zero.

This is a default which will basically make the square just stand in place.

The default square size is 128 pixels, and then our color is going to be white paint and we're going to have it as a stroke.

So an outline and the stroke width is going to be to notice that these are the default values as specified right here.

So we're overwriting a number of methods.

First of all, we're overwriting the on load.

What happens is when this unload method is called for the square, what it's going to do is it's going to set the values.

Notice that this size comes from.

Position component, and we're setting the size of the position component based on the square size that we get from initialization.

Notice also that we set the anchor unload to be in the center.

In other words, our anchor point is going to be the center of the object of the square.

So this is pretty straightforward.

All we're doing here is we're calling the super method, and then we're setting the values as well as the anchor.

The next most important thing here is the update method.

Notice that what the update method does is it changes the position.

Again, position comes from position component.

It changes the position to be velocity times the delta time.

And what that does is it basically moves the position of the object by the amount of velocity multiplied by the delta time.

And this is a frame rate independent refresh, which means basically that whether you happen to have, for example, a mobile device that is running at 30 frames per second or 60 frames per second to both people using the two different devices, the movement of the object will appear to be the same.
In other words, it will not be dependent on frame rate.
So this is important.
And notice that all we're doing here is we're basically saying that the position is equal to previous position plus the velocity times the delta time.
And this is this is pretty standard for when you're trying to move an object.
So now that we have updated the position, next we're going to render it and the render method which we override, basically all it's going to do is it's going to use the canvas, it's going to use the draw rectangle method.
And all it's going to do is it's going to draw a rectangle, which is going to be the size of the rectangle at the specified color, which is all right here.
And notice that this size comes from the initialization that we have done with the values here.
Okay, perfect.
So now that we have checked the square, let's have a look at what the actual game class does.
So notice that component.
Example 001 extends Flame Game.
So this is our game class, but we also have a mixing of double detector and type detector.
We also have.
An attribute running equal to true.
We're going to use this attribute to keep track of whether the game engine has been paused or not.
So if it is running the game engine is not paused.
If it is not running, it will be paused.
And we're overwriting.
Basically, as you can see right here, we're overwriting two methods.
First is the on top up, which basically means that when the user taps and they lift their finger, this is when this particular method is going to be triggered.
We're getting the tap up info.
And what we're doing right here is we're basically getting the touchpoint, which is the position within the game or within the screen as to where the user has actually tapped.
As you can see here, this is a vector, so a touch point is a vector which is going to give us the

coordinates of where the user has tapped.

So the next thing that we're going to do is once we have the position where the user tapped, we're going to check remember this from the remember this from the architectural diagram, the flame game engine being a component, keeps track of all the children.

The children here are going to be the squares.

Every time a user taps on the screen, we're going to create a new square and we're going to make it a child of our flame game.

So what happens here is when the user taps, we're going to look at the set of children and we're going to run this particular function, the any function.

What it does is it checks whether any element of this iterable set, whether it satisfies some sort of a test.

In our case, what we're doing is we'll basically saying.

Look at all the children of this component, which is the flame game and check whether any of them OC any of the the component here stands for each one of the children here, whether any of the components satisfies this following criteria, this is the test.

So what the test does is it basically this children any what it does is iterates over every single child that has been added to the flame engine component.

And what it does is for each such component, it does this test.

The test is as follows if the component is a square been meaning this right here.

So this is a type check.

If the component is of the type square and the component contains this point.

In other words, if we have clicked on that particular square, we're going to reverse the components of velocity and we're going to return true meaning we found it, otherwise we're going to return false.

This value, true or false, will be assigned to this flag handled.

The value of the handled flag is going to be tested here.

Notice that this test, if it finds that a square has been clicked on, it will negate the velocity and return.

True.

But what happens if we click not on a component?

In other words, we click outside of any one of the squares.

Well, at that point the handle is going to be false.

And here this particular test is going to check if it has not been handled.

In other words, the user has not clicked on a square.

What we're going to do, we're going to create a new square as follows create a new square component, set the position to be the touchpoint.

So we're going to spawn or create the square in that position.

We're going to set the size to be 45.

We're also going to set the velocity to be vector zero one, which is basically pointing downward.

And as you're going to find out from oncoming lectures, you will see exactly how that works.

So vector zero one creates a downward vector.

We then normalize that vector to give it the length of one, and then we multiply it by 25 to give that whole vector the length of 25, which gives us the speed of 25.

So the velocity is downwards.

At the speed of 25, we're going to set the color of our square to be red and we're going to set the style to be stroke.

So we're going to outline it and the stroke width is going to be two pixels.

So what this does is this code basically checks user has tapped on the screen, go through all the components that have been added to our game.

And if the component is a square and the component contains the touchpoint, negate its velocity and return true meaning that it was handled.

If it was not handled, meaning it was tapped outside of the square, then return false.

If it was false, which is what this condition is, we add a new square.

Notice that this add method, what it actually does, the add method actually adds the square component to the set of children here.

The last piece of the code here is overriding on double tap, which comes from double tap detector.

All we're doing on Double Tap, which is basically when the user taps quickly two taps as we're checking is the engine running.

If it's running, then we pause it.

So if I double tap while the whole application is running, it will stop or pause the game engine.

Otherwise it will resume the game engine.

And of course, here all we're doing is we're reversing it.

All right.

So this is pretty this is pretty straightforward.

So let's run this code, start flutter emulator.

It's starting right here.

What I'm going to do is I'm just going to a large it a little bit.

All right.

So now we're going to run it.

To run without debugging.

So it's running.

So based on what we've seen here, if I click on it right now, it should create a red square that is just an outline of a square.

And as we can see, we have the square going.

It's moving roughly at about 25 pixels per second.

And if I click on it, it will reverse this.

And if I click on it again, it will reverse it again.

I'll click on it so that it goes up.

And all all that was happening here is we get the location where it was clicked.

It finds that there is actually a square component.

Finds that there is a square component, and if I click on it again, it negates it.

If I click not inside, but outside, this becomes false.

And because it wasn't handled, we're adding a new square and it is added at the position that I clicked.

So if I click on these two again now, if I double tap it pauses the engine.

If I double tap again, it resumes the engine.

So let's make let's make a small change.

Let's change the vector, Let's change the speed to, let's say 75.

That should make it go a little faster.

So I'm just going to restart it.

If I click on it right now, you can see that it's moving much faster to a point where I have a hard time actually catching it.

So let's go here.

Yeah.

And actually, this is an interesting point when I'm clicking on it, it's not detecting at this point because it's moving so fast.

It's not detecting at this point that I'm actually clicking.

So I'm going to click a little lower.

As you can see, it caught that.

I'm going to click a little higher.

So it's almost like in or when it goes at the speed in order for me to actually detect it, I have to click really low.

And there's a reason for that.

And I'll I'll ask you to actually try to figure it out.

Okay, let's do another change.

Let's change this to a 50.

But this time, what we're going to do is we're going to change this to a minus one.

So let's run this.

So now if I'm going to click.

If I click on it, you see, it's a little different.

I'm going to change this to, let's say, a one.

If I run it again, let's have a look at which direction this is going to move right now.

If I click right now, it's going to it's going down and to the right.

If I click on it again, it comes back.

Let's make this a negative one and a negative one.

Now, let's have a look what this is going to do.

If I click now, you can see that it's going the other direction.

Let's change this to something else.

Let's say that we're going to make this minus one and zero.

If I run this again.

And if I click right now, it's going to the right as opposed to.

So as you can see, by changing the directionality and then normalizing it and then making it go a certain direction, that gives us.

A completely different direction.

So we can control the direction through creating a vector, normalizing it, and then actually making it go at the particular speed.

This will be explained much better in the next lecture.

One other thing we can also do.

So notice one thing that if I click on it right now, watch carefully when out where I'm clicking.

So I'm going to click.

I'm going to click right on the edge.

As you can see.

So I'm going to click right here.

Notice that where I clicked.

The square seems to be generated with half of it outside of the frame.

You can see that.

And the reason for that is that we have set in the square class.

We have set the anchor to be center.

That means that whenever I click on it, the the point, the kind of 00. for the object is in the middle of the object, which is why when I when I click here.

The click itself happens to be interpreted as being in the middle of the object.

I'm going to change this instead of center.

Let's make it into.

Let's make it.

Top left and see what happens.

I'm going to do the same thing.

I'm going to click here.

And as you can see right now, the anchor point is.

Right in the top left corner.

So when I actually click on it, most of the square is hidden.

And the reason for that is that the anchor itself is being put.

Right here, which is at the top left.

And if I change this to, let's say let's put it into top right, and if I run it again, it's going to be interesting because if I click on it right now.

See that it.

When I click on it, you can see that the square is actually generated fully inside.

The screen.

That is because it's generated based on its upper right corner.

So this will become more evident and will explain the whole Angkor aspect in the next lecture.

So what I would like you to do is go back to the architectural diagram, have a look at what is happening in the code, and try to reconcile the code with the architecture.

We'll be talking about the vector normalization and anchor in the next set of slides.

See you in the next lecture.

6. Importance of Vectors in Game Development: Velocity, Rotation. and Anchor points

All right, let's talk about vectors.

It is important to understand how vectors work, and they are represented in flame engine by the vector two class.

This is basically a class that represents a 2D vector, which is an encapsulation of an entity with X and Y values.

A vector also has a length measure.

So what is the vector useful for?

It is useful for representing a position, for example, as an x and y position, which would be a scalar.

We could also use it to represent a velocity, which would be a vector with direction and magnitude.

In other words, where does it point to and how large it is?

Or we could also use a vector and the x and y to represent a rectangle.

So for example, let's say we have a vector which is 25, 15 x is 25, the Y is 15.

We could use the vector for a number of things.

First of all, we could use it as just a point in space at a location of 2515.

That's one way of using the vector.

Another way is that we could use it to represent size.

For example, we could have a rectangle which is 15.

By 25 it's 15 high and 25 wide.

So it's a rectangle with the size given by 25 by 15.

But there is one more way in which we can use a vector which is actually very important for us, and that is a vector with a magnitude.

In this case it's a vector with a magnitude of 2915, which could, for example, represent the speed of 29 and 15 units at the specific direction, which in this case happens to be down and to the right from the position which is shown as a blue point of origin.

So what are some of the useful methods of the vector class apart from being able to instantiate it?

Some useful methods that you need to be aware of are negate which reverses the direction of the vector.

What that means that if the vector, for example, was moving up, when you negate the vector, it would

be moving down.

If it was moving right, you would, by negation, make it move to the left.

The next very important method is the length which gives us the size or the magnitude of the vector.

Could be, for example, the speed with which a particular object is moving.

Another method that is important is the normalized method, which normalizes the vector, which basically makes the vector specifically of magnitude one.

One other method that is of interest to us is the random method which generates a random vector and pay attention to this specifically in the range of zero 0 to 1 one.

What is zero 0 to 1 one mean when it comes to directions?

There are a number of ways in which we can look at the vector as a way of giving a direction.

So in terms of directionality, we can think of vectors as designating directions.

Think of very simple square, which shows a number of quadrants for quadrants.

And imagine that we have a center here which would be zero zero.

And now if we move up or down or left or right, that would be the direction from that center, from the red dot.

So for example, if we have a vector moving up, we could represent that as simply a vector with x and y of being zero and minus one.

We can think of it this way the x is zero.

We're not moving away from the red dot to the left or to the right.

We're moving away from it by going up, which in this case would be negative one, which is the Y.

If we wanted to specify the direction of going to the upper right corner, it would be given by the direction of one minus one.

If we then look at the one going to the right, it would be one zero because we're moving one to the right, which is the X, and then we're not moving the Y direction at all.

We're not moving up or down.

So that's why it's zero.

The lower right corner would be one one.

Then all the way down would be zero one.

All the way to the left corner would be minus one, one minus one zero, and then minus one minus one.

This is very useful to think of this as a way of describing the direction of the vector.

This will be useful when we work with joystick later in our case studies.

One thing for you to note, though, is that these directions do not follow the normal kind of mathematical

Cartesian coordinate convention.

Usually in that convention, if you go from zero zero up, it would be zero one because the direction

in Cartesian coordinates positive is up and negative is down in the flame game engine in most game engines,

just remember that it's the opposite.

Keep this particular picture in mind when we're thinking about directions as this will be important

later on.

Okay, so let's talk about it a little bit more in detail.

So for example, normalization, why would you want to normalize a vector?

Well, normalizing a vector allows you to standardize its magnitude.

Consider a very simple example how to create a vector that is moving at 45 degrees southeast, which.

Just down right corner.

And exactly with a speed of 25 units.

If you think about it, 45 degrees south east, which means down right corner.

Let's go back to this picture.

Down right corner is one one.

Well, the way we would do it is we would create a vector as one one.

Then we would normalize it to give it a unit vector in the same direction.

And then we would multiply it by 25, which would give us the required magnitude.

So we first choose the direction, we then normalize it, then we multiply it by the exact amount of

our speed encode.

It would look something like this.

The vector to the direction is one one.

By normalized, it makes the vector exactly one unit and then we multiply it by 25, which gives us

a vector moving at 45 degrees southeast, which is going at the speed of 25 units.

Now, one quick note, the reason why you would want to normalize a11 vector, because if you think

about it, using the Pythagorean theorem, a vector that has the x is one and the Y is one, the magnitude

of that vector would be one squared plus one squared.

And then you take the square root of that, which would give you a square root of two.

Square root of two is more than one, which is why we need to normalize this vector.

Okay, Now what about random generation?

Let's say we would like to generate a random vector, but what exactly do we mean by random?

It could mean a few things, if you remember, because we could use the vector to represent a number of things like velocity or position or even a rectangle size.

So it could mean a number of things.

It could mean, for example, a random placement.

In other words, if we wanted to randomly place an object on the screen, we could use a randomly generated vector to get the position.

We could also generate a random magnitude.

Let's say we want it to create an object that moves with a random speed.

We could then use the random generation of a vector as well if we wanted a random direction, meaning where is the vector going?

If we wanted that to be random, we could also use the same method.

So as you can see, depending on our creativity, we could use random generation to create either a random placement where the object starts from.

We could create a random magnitude, which is how fast the object is moving.

We could also create a random direction, which is where is the object moving?

Any one of those could be created randomly.

And how do we do that?

We use the function which generates a random vector and pay attention to this in the range of 00211 now 00211 is a very specific area of when we look at the picture that we had, again, if we go back here, if you look at this shape 00211 goes from the red dot to the lower right corner, which means that it only generates vectors in that quadrant, in that one subsection of that full square that you see there.

This will become more apparent later on.

But just remember that.

So going back, if we now try to generate a vector in that random range, we use this function vector to plot random that will create an actual random vector.

At that point, if we want to, for example, have a random direction, but we want it to have that move at a very specific speed like 25, for example, to control the magnitude.

We could do this.

We could say vector to random, which generates the random vector.

Then we normalize that to create a unit vector, and then we multiply it by 25 to create a vector that

represents the speed of 25.

So what was the random used here for?

It was used for random direction.

Let's look at another example.

If we wanted to control the position, which is the placement, then we can set the position of the randomly generated vector as follows.

So let's say we have generated the vectors before.

It has a random direction.

But the question is where do we place it on the screen?

So we could do the following thing.

First we get the random vector and set its velocity, so velocity is equal to vector to random times normalized times 25, which gives us a random vector going at the speed of 25 units.

And then if we wanted to set the position, we would do this specifically, we could say we want the position to be vector to 8100.

So what that would do is whatever object we just generated the velocity for, we are setting its position to start at the location 8100.

So you can see that when we think of a shape, we can use the random generation to randomly generate the velocity and then specifically put the location of that shape at position of 8100.

We could also generate the position randomly.

But in this example, the only thing that we generated randomly was the velocity OC.

The next thing that we're going to talk about is the angle of a vector.

Now what does that mean?

When you think of a vector, you can think of it as being some sort of a line like an arrow.

And when you think about the angle, the question is what is the angle of the vector?

Sometimes we really need to find that out.

So in some cases you will want to know the angle in radians of the vector.

Now radians measure the angle in terms of pi, pi represents 180 degrees.

So two times pi is 360 degrees.

Radians themselves are not degrees.

PI is roughly 3.14 units.

So the equivalency is that 180 degrees equals roughly 3.14 gradients.

So what we want to do is sometimes we would like to know how far a vector has been rotated.

In other words, what is the angle of that vector?

Let's call the angle that we're interested in angle A So imagine that we have a reference line.

This line itself just represents zero degrees.

If we have a vector v one.

What we would like to find out is the angle.

A How do we do that?

Well, the way to do this is we first get assigned angle from a representative vector with zero degrees and then we take the angle between them.

So let me show you what that would look like we have.

Our representative line here, which represents zero degrees.

So what we do is we create a vector, let's call it v two, and that will be our representative vector.

This vector has zero degrees.

We can represent this vector as follows It's simply vector of zero minus one.

If you go back to the picture of the different directionality of a vector, you will see that this particular vector which is pointing up is zero because it's not moving either to the left or right, and it's minus one because it's moving up.

So we have V two, we then have the V one, which is the vector that we're interested in.

We would like to know what the angle of the v one vector is.

This is the angle.

So what we're really doing is we're basically taking the angle between v two, which we know and V one.

The angle between them is what we're looking for in code.

This is what it would be like.

Angle A is equal to V one using the method angle to sign to V to what really angle to assign gives us is the angle between v one and v two because we have set v two to be itself angle zero.

This gives us the exact angle of v one.

This will be important later on, especially when we're dealing with shooting from our spaceship.

So if we know the angle of a vector, how useful is that?

Well, let's talk about angular momentum.

Angular momentum has to do with being able to rotate a shape at a certain speed, at a certain angle.

So, for example, if I wanted to rotate a shape 90 degrees, I would have to be able to figure out how fast to animate that, how fast to show it so we can represent the angle of rotation for the component using angular momentum.

In general, by default, any shape that you create, it's angle is zero, so zero radians and the angle is always in reference to an anchor point.

So the idea is that we can rotate the shape through its angle.

Property, for example.

Here is how we would rotate the shape by 90 degrees.

The angle itself would be pi divided by two, since PI represents 180 degrees 3.14 radians.

When you divide it by two, that would give us 90.

So if you want it to represent the angle of 90, it would be pi divided by two.

Okay, great.

So we know how to create an angle, but how do you, for example, rotate a shape at a certain speed?

Well, you can rotate a shape at a certain speed with reference to the delta time in the update method, because this is how we render things in flame engine.

We first change any necessary property for animation in the update method using the delta time and the formula that you would use is as follows.

The angle delta, which is the change in angle, is going to be the delta time given to us by the update method times some sort of a rotation speed, which we would use to specify how quickly we want something to rotate, which would be given in radians.

So let's say that we have some sort of a speed of 0.3.

So as the Delta time is changing, what we want to do is we want to find what the angle was depending on the delta time.

Simply it is the Delta time times and rotation speed.

And then if you want to actually calculate the angle, the new angle of the shape, you have to add the angle delta to the angle as follows Angle is equal to the previous angle value plus the new delta value.

And of course, since there's only 360 degrees, because once you go past 360 degrees, you're basically rotating again from zero.

We're using the modulus of two times pi, which is basically 360 degrees.

So the angle is always kept between zero and two pi.

OC So since we're talking about rotation, we need to be very specific about what are we rotating the shape around and this is where the anchor point comes in.

So the anchor point is actually a very important concept.

The anchor is a property which is used to represent the logical center of the shape.

It doesn't have to be the physical center, just the logical center.

And the idea basically is that you can think of the anchor as the point where flame grabs the component.

Any kind of transforms, like, for example, rotation would occur around this point.

This is the position where the anchor point will end up after the component is translated.

So, for example, the rotation and scaling would happen around this anchor point.

We'll demonstrate that as an example in just a second.

But when you are specifying the anchor point, here are the possible anchor points.

So if we have a shape like a rectangle, the anchor points are as follows By default, the center is the default anchor point, but you can designate any one of the other eight anchor points as being the anchor point.

So for example, I could have the anchor point for my object.

The upper right corner, the upper left corner, or in the center of the side or in the center of the bottom or the top.

But how does that actually affect any kind of transformation?

For example, let's say that if we make the anchor point to be the left top right, so the left top corner and then let's say we wanted to do a counterclockwise, so kind of moving up counterclockwise, 45 degree rotation, what would that look like?

So we start with a shape.

We designate the anchor point to be the upper left corner.

And then what we would like to do is we would like to rotate this 45 degrees up.

The way that would be done is exactly this way.

The rotation would be around that particular point, and this is what we would end up with this kind of object.

Let's think of a different example.

What if the anchor point is actually in the dead center, right?

And then what would the counterclockwise 45 degree rotation look like?

Well, it would look as follows.

Here's our shape now.

The anchor point is in the middle.

So now when we do the counterclockwise 45 degree rotation, we're actually rotating it down but around the center.

So we end up with this.

So you see that it really affects the rotation, the scope of the rotation, where the anchor point

is.

Typically though, by default, this is where the anchor point is in the middle.

7. Adding Vector Dynamics to "Squares" code

All right.

Now that we have looked at some of the more theoretical aspects like angular momentum, rotational speed,

etc., let's look at how we would implement some of these things in code.

So what we're going to do is we're going to go back to our previous example of the square object falling,

basically moving either up or down, and we're going to add rotation to it.

We're also going to add here the ability to count how many objects we have created.

So let's have a look here at the outline and we can see that we have added a rotation speed.

That's one attribute.

Notice also that inside the Flame Game Engine class, we have added something called the debug mode.

And the debug mode is the ability for you to see more information about your visual objects for debugging

purposes.

So you can see, for example, the hitbox outline, you can see where the anchor point is and you can

also see the exact coordinates of your objects.

We'll see that in just a second.

Right now, it's set to false meaning that it's not going to be running.

Notice that we're overriding the debug mode that is actually a property that we have inherited from

playing game engine.

We're also adding a text paint.

Now a text paint is basically a text rendering object.

And what we're doing is we're creating a text paint that we're going to use to paint something on the

screen.

In this case, text, the way you set up text paint is it takes a style property and we're setting that

to a text style with the values of font size 14, which is going to be how large the font is and the

type of font we're going to have as awesome font.

All right, so let's run this first and then we're going to look deeper into the code.

All right.

So we have the code running.

Notice that objects active is zero at this point.

Let's have a quick look at how we're rendering that.

So inside the render method, which is of the component example right here, the render method, what we're doing is we're using the text paint, which is this right here.

This is the variable that we're using.

We're going to render it onto the canvas.

We're basically rendering this string at this location.

So vector 1020, which is roughly about here.

And this is the static part of the string that we're rendering objects active.

And then this is the dynamic part.

And in both flutter and in flame, the way you inject an element inside a string that you're, for example, creating is you can use the dollar sign with the two parentheses on the side and inside you can put the actual value that will be dynamically evaluated and placed into the string.

Notice also that what we're putting there is the number of children.

If you remember from the architectural diagram, children are the components that have been added to our flame game engine.

And for example, right here you can see that we have the ADD method whenever this method is called.

What it does is it adds a square to the children.

Okay, So let's let's actually give it a spin.

So if I'm going to click on it right now, what you see is that we have an object that is falling just like before, but it's rotating clockwise.

If I click on it again, it's going to continue going up.

So reverses the direction and if I click a few more, it behaves exactly the same as before, except that now all these shapes are rotating as well as falling or moving.

So if I double click on this just as before, it pauses, I can unpause it.

You can see here that the number of objects active is five, and notice that as the objects are going out of the screen, they're no longer like this one is no longer visible, this one is no longer visible, but it still shows that the objects are active, meaning that actually these objects are still in memory, they're still part of the children list.

So one of the things that we're going to do right now is let's see how we can modify this to spin the

other way.

So if I click here and you can see there's six, you can see that it's moving clockwise.

So let's see how the actual code for rotation works.

We have rotation speed, which we said here to 0.3, which is relatively slow.

This rotation speed is part of our square because that's what is rotating.

Now, one interesting thing here that we have in the update method is that apart from updating the position,

which is like this is why this is actually moving downwards and when I click on this moving upwards

because the velocity, the position is being updated based on the velocity and the delta time, but

the rotational speed itself is calculated as follows We have angle.

Delta is equal to rotation speed times the actual delta time.

In other words, the time that has passed since the last time the update method was called.

So in this case, what's going to happen?

Angle Delta simply gives us the arc or the number of radians that would have rotated the object in that time.

So once we have the angle delta, the number of radians that it should have rotated, what we do next

is we're basically taking the previous value of the angle, we're adding the angle delta, and then

to keep the actual number of radians within the 360 degree confine of a circle.

We're basically doing a modulus of two times pi.

So this is pretty straightforward and this is why you can see it's rotating.

And if I change, for example, this from 0.3 to let's say 0.7 and I rerun it, you will see that it's

rotating faster right now.

So this is pretty just going to revert it back.

This is pretty obvious.

Now, the question that you should ask yourself is, okay, this is right now rotating clockwise, right?

How do we make it rotate counterclockwise?

You can pause the video if you want to figure it out yourself.

But the idea basically is that when you think of radians, when they specify the length of the arc on

an actual circle, we can think of the circle as basically going from kind of like the 12:00 if you're

looking at a clock and then going clockwise all around in a positive direction.

So the positive direction is clockwise.

It's kind of like as if, let's say the rotation was moving from 12:00 then to 3:00, then to 6:00,

then to 9:00, 10 to 12:00 again.

And it just repeats.

This is.

60 degrees.

If we want it to rotate the other way, then it's actually simply the negative value you're actually subtracting.

So you're going from 360 back or zero minus one actually becomes with our modulus becomes 359.

So all we need to do is just change this.

We just make it into a minus.

And if I'm going to run it right now, before it was going clockwise, which is in this kind of direction, but now it's going to go the other way.

It's going counterclockwise.

So right now it's going against, so to speak, the flow of a typical clock.

So this is this is pretty this is pretty straightforward.

So let's have a look right now at what this debug mode does.

So I'm going to change the debug mode to true and I'm going to rerun it.

Okay.

So if I click on it now, watch what happens.

What happens now is in debug mode, I'm going to double click on this so you can see it.

I paused the engine in debug mode.

Extra information is put on the screen.

First of all, you can see here that we have the x and Y coordinates of this corner.

We also have x and Y coordinates of this corner.

This little shape in the middle is your anchor point.

And this outline is not actually the outline of our square per se, it's the outline of the hitbox.

Hitbox is something we're going to learn a little bit more about later.

But Hitbox is you can think of it as the area that defines the boundaries that are necessary for calculating when two objects collide together.

So, for example, if I resume this and I, for example, start this object and I click on it at this point when these two objects touched, that would be considered a collision because they're head boxes attached.

I'm going to just put a couple more boxes on on the screen for you to see us.

For example, the X and Y is changing.

I'm going to do one more thing as an illustration so you can see that when I click, you can see that

exactly where I clicked is where the anchor point is.

You can see that little, almost like a little X or a little cross.

What I'm going to do right now is I'm going to completely change our anchor point when we actually create

the square.

The anchor point is defined here in a center.

So what I'm going to do right now, so right now, if I click, you can see it's in the center.

Let's look at some of the values.

And what I'm going to do right now is I'm going to, for example, change it to bottom center, which

basically means that the point like right now, if I click, it's here.

It would actually be right here at the bottom.

So if I'm going to do bottom.

I'm going to do bottom center if I rerun this right now.

If I click on it, watch carefully where that little cross will be.

Sometimes it's difficult to see, but you can see that the cross is right here and you can actually

see that the rotation now is around this point.

Watch carefully when I click on it.

When I click on it, it started here and then it's rotating around this point.

Okay.

So this is pretty this is pretty straightforward.

You should use this debug mode whenever you want to work, especially with hit boxes or you want to

see what are the exact coordinates, where is your anchor point, etc..

Do you do that by simply adding the debug mode?

So I'm going to set that to false just as a quick demonstration.

I'm going to now remove a component.

So instead of making the square go back up or change its direction, I'm going to remove the component

instead of negating the actual velocity.

So let's let's run this.

So if I click now, you can see that it's it's falling.

One object is active.

If I click on it right now, you can see that the number of objects active is zero.

Why?

Because we remove the component.

What I can do now is I'm going to click a couple of times.

And notice.

As the objects are falling off, they're not becoming.

This this doesn't go down because we don't have any code here that removes the object when it goes out of the boundary.

That's going to be actually one of your exercises.

So again, if I click here, this simply removes it.

If I click here again, I can remove the seven active objects.

I can remove these objects.

And now I still have the three that have gone out of bounds.

Okay, so this is it for this particular example.

See you in the next lecture.

8. The concept of Component Composability

All right, let's talk about component COMPOSABILITY.

It's a very important concept.

Composability is the ability to add a child component to a parent component and have it behave as if it were an integral part of the parent object.

For example, we can add a small square to a larger square for a composite square and square shape.

So we start with a shape like just a normal square.

We add a little square in it.

And the idea now is that we can look at this as two separate squares, but in reality we would like to treat it as if it was one composite object.

So, for example, if we rotate or move the parent object, the larger square, the child object will automatically rotate and or move with it.

If the two objects have been composited together, this is what it would look like.

Let's say we have our two squares in that kind of position and we would then like to move and rotate it, but we would like to only have to do that with the parent object.

We want the child object to follow exactly what the parent is doing.

So we would like to end up with this.

This is very powerful and has many interesting applications in game creation.

In the above example, we have moved the object to the right and rotated it 45 degrees.

So we started with the object which you see the outline of, and we ended up with the object on the right hand side which has been moved and rotated.

But the idea is that we don't have to think about moving and rotating two objects.

Composability means that we add the objects together, we compose a larger object out of them, and then we can move everything with single operation.

A very simple, powerful example would be a health bar over a game character or an object that always moves over, travels with the given object.

This is something that you see in a lot of games.

Your character might be running around with a little health bar at the top so that when something happens

to the character, whether they're being attacked or they have been injured, you can see for their individual character, their health.

So let's think about it.

Consider a simple game object.

Again, we just have a square.

The object has its own update and render methods which control its appearance and behavior.

This is the kind of like main aspect of the game engine.

Every single component has an update and render method.

Adding a health bar would look as follows We would now have our square object with a little rectangle above it that represents its health.

Here it's in perfect health.

So how could we achieve this?

We could create a health bar as a combination of a number of rectangular objects.

We start with the outline of our health bar.

Then we have another rectangle which contains the health percentage and will be colored either green for when you're healthy or red when you want to alert the player that their character is almost dead.

We also would like to have an opacity mask to show the missing space from the health bar.

Usually it's a subtle shade with an opacity of about 50%.

These are the three elements.

They're almost like Lego pieces that we put together to create the actual health bar.

So what do we do?

We combine them, we put one on top of the other.

This is what we end up with.

And this is what, for example, when you're dangerously close to dying, what it would look like.

We start with the outline.

We put the opacity mask on top and then we put the amount of, for example, red.

That's the concept behind our health bar.

Now, when you think about how an object is composed, you have to be able to place the child object in some sort of a relationship with the parent object.

And what you must think about is that the coordinates for the composite child object are always relative to the parent object.

Let's see what that means.

So let's say that we have a parent object and for example, it has an anchor in the middle and our parent object has a certain height and a certain width.

The actual anchor is sitting at some location X and Y.

The location of the anchor of X and Y is relative to our screen coordinates.

We can think of it as the upper left corner of our screen is zero zero and then X is moving in the positive direction to the right and why it's moving in the positive direction down.

So whenever we put a child component inside the parent component, the left upper corner.

Of the parent becomes the 00. of reference for the child.

This is quite important.

The child's point of reference is not the 00. of the screen.

It is always of the parent upper left corner.

So let's consider right now that we're going to add a child component, which is going to be our health bar.

Our child component is going to have its little anchor at the top left corner and it's going to have a certain width and a certain height.

Of course, our health bar its upper left corner.

We're going to designate it right now as the coordinates are us being X prime and Y prime.

What we need to figure out right now is how the X prime and Y prime, how it must be related to the zero zero of the parent component in order for the composite object to achieve what we want.

So what is it that we want?

If we put our child component at the location of zero zero, that zero zero location is relative to the zero zero of the parent component.

So when we add a child to the parent component, if we add it at location zero zero, that's what it would look like.

It will align itself exactly with its upper left corner at the location, zero zero of the parent.

But what we really want is we would like to put our bar above our parent and a little bit to the right.

So we need to do a little bit of math here.

What we need to do is we need to figure out the X prime and Y prime so that the child object is placed relative to the parent as it is in the picture.

The math is relatively straightforward.

If we want to make sure that our little red coordinate point on our health bar is aligned exactly as shown in the picture, the formula is relatively straightforward.

The x coordinate for our health bar will simply be the width of the parent object minus the width of the child object of our health bar.

That difference is where we want to place our object in reference to the parent.

We want to offset it by the difference between the width of the child object and the parent object.

What about the y coordinates?

Y coordinates are a little different.

Notice that what we're trying to do, since y is positive when you go down and we want to go up, which is away from that, we need to have actually a negative coordinate.

And the negative coordinate is pretty straightforward.

We're taking the height of our child, which is H.

And if we simply just move the y h units up, which is negative H, we would be exactly on the border, the two borders would touch.

So we will add another two units.

In this case, we will be subtracting, we will be subtracting two units from that so that we can place it just a little bit above the parent.

So the formula for Y prime would be negative H minus two.

It's pretty conceivable, let's say, that the height of our health bar is ten pixels.

So what we want to do is we want to move it up ten pixels so that it will be exactly on top of the parent.

But then we also want to give a little bit of a gap between the parent and the child.

So we move it up to more.

And again, because the Y coordinates are negative going up and we want to go up, we basically subtract the height and then we subtract the two pixels to create that gap.

And that's basically it.

I suggest that you kind of go over this a few times to understand fully what is happening.

But the important thing now is that once we have related the child component to the parent component in the exact position that we want, any manipulation of the parent component, whether we move it or whether we rotate it, it will automatically move and rotate the child component with it.

We're going to see that in the code that we're going to look at next.

9. Architecting Composability: UML Diagram for Adding a Health Bar to "Squares"

All right.

Let's have a look at the modified UML class diagram for our component examples.

001 You can see that the square class has been expanded apart from the rotation speed that we have from the second batch of code, you can see that we have something called life bar elements, which is an array of three rectangle components.

You can also see that we have a method called Create Life Bar, which basically will create those elements.

And what is interesting is that these life bar elements are going to be added to the square components children.

And what I'm saying to the square components children, if you look at the square component, you see that it extends position component and you can see that position component extends component and you

can see that component has a list of children.

So every component can have a number of children.

So what we're going to do is end the create life bar method.

We're going to actually add the rectangle component array as a number of children.

The reason why we can do that is if you look at life bar elements, you can see that it's actually a collection or an array of rectangle components.

You can also see that rectangle component extends shape component and shape component extends component.

And since the children of component are components, it's almost like a recursive definition.

We can definitely inside our square add the rectangle component array.

As children.

This is important because when any component has component children, when that component is being rendered

by the flame engine, that flame engine is going to then render the children of that component.

So in our case, for example, we have a collection of square components.

These are the children of the flame game, but each square has its own children.

When the flame game engine renders our square, it also automatically will render all of the children components of our square.

And this is important for Composability because that means that whatever action our square component takes like rotation, the children components of that square will also rotate with it.

So this is quite important to to realize that Composability works because we're using this architectural aspect of the flame game engine where all components to a great extent are related.

And this whole idea of this tree of components, this tree of children components is very important because that's how we can take huge advantage of composability and that's what we're going to be looking

into next when we look at the actual code implementation.

Okay, See what the code session.

10. Composability: Coding a Health Bar for Game characters

All right.

Let's have a look at the code for the life bar, which showcases composability of components.

So in this case, let's have a look how the code runs.

So in this case, if I click here, as you can clearly see, the life bar, which is made of three rectangles, is actually perfectly mimicking the way the square is moving.

So if I click it here, you can see that it rotates properly.

And that's really what we want.

We want a way in which we can define the motion of one parent component, and we can then associate certain other components like visual components with that component.

But we don't have to write code that tracks where exactly is this component moving.

We can simply compose all of them together.

And then as long as there we have the full compatibility, what happens is as the parent moves, as the parent rotates, so will the children without having to add any more code, which is great as long as you make it basically any component that you would like to follow in the same path as the parent component, you simply make it a child.

One other aspect that I would like to demonstrate.

Have a look at these two shapes right here.

You can see that basically one is on top of the other.

But what is interesting is that when you look at this square here, you can actually see it through the life bar.

And the reason for that is that we made the inside of the life bar that has not been filled with any information about the life, which is basically the kind of empty part.

We've made it transparent or at least partially transparent.

The way to control transparency is through opacity.

For example, 50% or 0.5 opacity means that your object or your color is semi-transparent, which means that you can see things that are behind it.

And this is very useful, especially for something like a life bar, which gives you secondary information.

It might be important for you to see exactly where your character is moving, so you might not want, for example, the light bar to fully obstruct your view.

All right, so let's have a look at the code.

So the first thing that we notice in the code is right here, and that is the list of rectangle components which make up our life bar elements.

As you remember from the lecture that we did, that our life bar is made up of three rectangles.

So what we're basically saying is that our life bar elements is a list of rectangle components.

And what we're doing is we're filling this list.

We're initializing it with three elements where each element is exactly the same one by one vector.

We basically just want to make sure the reason why we do this is we want to make sure this is good coding practice that we initialize our data structure before we use it so that we don't have dangling pointers.

And also one other thing to note is that flame and in general Dart has become much more null value averse, meaning that you have to carefully code so that you don't have null values floating around.

So what we're doing here is we're basically creating a data structure that is filled with three elements.

These elements are initialized.

These are these are default values that we're not going to use, but at least we're guaranteed that they're not going to be null.

We're not going to have a null pointer exception.

And we're also telling that life bar elements, the global aspect, is false.

What that means is basically that we do not expect this array to grow or shrink.

We're not going to ask for more elements than three.

So how do we do this in our on load method?

What we're doing is we're setting up all the values for the actual square, which is the red square here.

But then and we're setting its anchor to center.

But then what we're doing is we're create we're going to create the life path as part of the load.

So let's have a look at the life bar method.

And the life path method is pretty straightforward.

First, we define the size of our life bar to be 40 pixels by ten pixels, so it's 40 wide and ten pixels high.

And then we're creating three different colors.

We have the background fill color, which is that kind of grayish looking color.

Inside.

We have the outline color, which is the white outline, and we also have the live danger color to just show what it would look like if you filled it actually with like a red color.

Notice that our background color, the gray one, has an opacity set to 35%.

So now that we have defined the colors, here are the actual three rectangles.

So the first rectangle is just going to be actually, we can change the code.

To be this as per what we have in our slides where we're calculating the position, which is this the position of the bar, we're calculating it relative to the parent.

So this is the size of this is the width of our square, the red square, and this is the width of our life bar.

And then this is the height.

You can go back and see how we did it in the lecture, but basically this is calculating the position of.

The outline relative to the zero zero of this.

So the next part is so this is the outline, which is the the white rectangle and this is the fill portion.

The fill portion is basically just going to be the gray color that goes inside here.

And.

The last thing that we're doing is we're actually adding this is hardcoded, but we're adding a ten by ten, which is this little red square here to show the live danger color, which is the red color.

So what we're doing here is we're basically creating these three rectangles.

We're putting them into an array.

And then what we're doing is add all is a method that comes from within component class.

And all we're saying is basically take all these elements and add them as children.

This is all you need to do.

Now, these children are going to move whenever.

See this update method.

This update method only affects the parent, but because these rectangle components have been added as children.

In other words, we have composed a much larger object.

As you can see, if I double click, all of them move nicely together and that is, that is a very powerful concept.

Composability is a very powerful concept and there is really nothing else here.

All the code is basically the same as before.

So the main main thing here was this this create life bar method, which gets called this is important.

You have to basically add the children elements before anything is rendered and you have to make sure that you place them relative to the parent exactly where you want it, because at that point it will move or it will be affected by the way the parent is moving.

But everything else stays exactly the same.

Okay, so this is pretty much all for this particular code.

There's nothing else here.

So see you in the next lecture.

11. Case Study #1 - Exercises

All right.

Let's talk right now about some exercises that you can do that will allow you to go back to the materials that we have learned.

Look at the code again, maybe with new eyes.

But basically this will allow you to practice a little bit of what we have gone over.

All right.

So exercise one.

What I would like you to do is add random generation to the shapes based on a random position and a

random velocity vector.

So when a user taps on the screen, you will generate a new shape in a random position with a random velocity.

By new shape, I mean a square, at least initially.

Make sure to put some reasonable limit to the position and the initial speed.

In other words, you don't want necessarily for the random position and the random velocity to be right at the edge of the screen, for example, because then it will show up and it will disappear.

For example, if it happens to be if the direction happens to be out of the screen.

So what you might want to do is maybe think of the screen as an X and Y rectangle and then put some margins like this is just in your calculations, put a margin of maybe 200 pixels on the top, 200 pixels at the bottom, 100 pixels on the right and 100 pixels on the left, and keep that as your boundary for where your random generation is going to happen.

So at least you will be able to see what is happening in exercise.

Number two, what I would like you to do is when the object goes out of bounds, remove it from memory.

So the idea is that when an object is leaving the screen, we don't see it anymore.

I would like you to remove that object from memory.

And the hint I'm going to give you is you will need to detect the boundary of the screen Flame game class already comes with, for that matter, any kind of component position component comes with a size that's the size of the object for the flame game.

The actual size refers to the screen size.

So you will be able to write some sort of an algorithm that is going to check whether we have reached the boundary, whether at the top or bottom or left or right, the boundary of the screen, And you would check that in the update method.

Let's look at exercise number three.

In exercise number three, I would like you to explore the shape component classes in Flame.

So first of all, Shape Component class is a parent class for specific shapes like rectangle, polygon or circle.

So I would like you to look at this documentation and then think about rewriting the square class in our code using the rectangle component.

And here is a URL to where you can read a little bit more about it.

I would also like you to look at the other derived classes such as Circle Component and Polygon.

Polygon is especially interesting because the way you draw shapes other than let's say a circle ellipse

or a rectangle or square is that you could actually define the vertices and the lines.

You connect the lines to create a shape.

So for example, a triangle is a polygon with three sides.

You could specify three vectors, like three positions of the points, and then you could link all of these points together to create a rectangle.

So do have a look at that.

And that's something that we will also explore in the game when we're creating the asteroids.

OC And now for the last text size exercise number four.

This one is a little bit more ambitious.

Think of creating a new class called Life Bar, which would have the following input attributes the current life, which is What is the life right now?

It would be in the range of 0 to 100.

100 obviously is your fully alive.

Zero means your your dead red zone, which would be a threshold number such as say, 25, which would tell the life path to turn the red color on when, for example, it's less than or equal to 25 or green otherwise.

So you could specify the threshold so that visually life bar when it updates itself and it sees that, oh, my life is less than or equal to 25, I should turn on the danger color, which would be, for example, the red color.

We also would like to have the ability to customize the color like the danger color, but leave it by default as a red.

So it's an optional input.

And also we would like to have the ability to define the healthy color, which by default is going to be green, and it's also an optional input.

The next step, once you've create this class, the next thing would be how would you integrate this class into another component, like a shape or a sprite so as to show the current life and health of that shape.

That shape would be aware of what its own life is, and it would then delegate the visuals of that to the life bar.

So have a look at that.

Think about it, draw a couple of diagrams and then of course, I'll give you the solution to that.

Well, I hope you're going to enjoy these exercises.

12. Case Study #1 - Solution to Exercise #1

Welcome back.

Let's have a quick look at solution to case study number one, exercise number one.

Let's first run the code so that you can see what it's doing.

When I click, as you can see, the square has seemed to appear in a random place.

I'm going to click a few more times.

Watch what is happening.

As you can see, what is happening in this scenario is that every time I click, regardless of where I click, a square appears to be randomly placed.

And also it seems to be going in a random direction.

And also you can see that the speed itself is random because sometimes it goes a little slower, sometimes it goes a little faster.

That was the essence of what we wanted to do in the exercise.

So let's have a quick look at how this was achieved.

First of all, what we did was we have encapsulated and abstracted all the necessary functionality into a separate class, which is called it Utils class.

And in this class we have a number of static methods such as generate random position.

We also have generated a random velocity.

And if you go through the code, you will see exactly how this was achieved.

And it is also explained to some extent in the codes comments.

But there is a better source for you to understand how this was designed and developed.

We have also provided you with a PDF document which gives you a solution in writing to the problem.

And as you can see, for example, in the case study, number one, exercise number one, we start with assumptions.

We give you information about how we're going to develop the code.

If you read through it, you will see that we specify the solution as well.

And this goes on for exercise two as well.

So everything that you need is in this document.

There's also exercise three and exercise four, which we cover in other videos.

If we go back right now to the code here, you can quickly see, for example, that to generate a random position, what we have decided is we have as input, we accept screen size, which is how large the screen is, and also margins, which is how far away from the edge of the screen we're going to generate

our random position.

It wouldn't make much sense to generate the random position right on the edge.

So the margins kind of confine the random positioning into a smaller area.

And as you can see in the document that we have for you, it is explained right here, this is what

the margin is.

Please go through the document first.

Every single exercise solution has a PDF document that we provided you.

I suggest that you read through the document first and then go into the code to understand how the code

was written.

All right.

We'll see you in the next lecture or exercise happy coding.

13. Case Study #1 - Solution to Exercises #2 and #3

All right.

Let's have a look right now at the solution to the case.

Study number one exercise is two and three.

Let's have a look at the PDF document that was provided for you with the solutions.

I would like you to look at exercise number three.

The exercise asks you to look at different types of shapes and especially the polygon.

So what we have done is we have actually created an asteroid shape and we explain how you would actually

build it.

If you look at this shape here and this is just a reference shape, the important thing for you to note

is that, first of all, if you look at the vertices, which is the v one, v2v3v4, notice that we're

actually showing them in a counter clockwise manner.

This is usually how you're going to list your vertices in a polygon.

The other important thing to notice about when you're creating a polygon is that there is a reference

point of the polygon, which is the middle part of the polygon.

The other very important thing to realize is that when you're going to give the polygon vertices, all

of their lengths have to be less or equal to one.

And the reason for that is you can see that on the right hand side is that the total length and height

of the polygon that we're describing is two units.

Units here are, of course, relative.

But the idea is that you're going to specify all the vertices as being in the range of 0 to 1 or minus one.

For example, in this image that you see here, what we have is what looks like an actual asteroid.

You can read more through this Full explanation is here.

And as you can see, for example, all vertices are in the range of -1 to 1.

So basically when you go from the zero, you can go either up left here or down, left and down right, which means that basically you're going to be confined, as we've seen before with unit vectors.

You're going to be confined into the -1 to 1 range.

Let's have a look how we could describe very much an asteroid like this.

So what we have here is an actual code for an asteroid with 11 vertices.

We've added an extra one.

The most important thing for you to see that first of all, all the vertices are given in counterclockwise direction.

And you can see that all the vertices, all of them are between minus one and one.

So for example, the first one is 0.2 and 0.8, which basically if you look back here, kind of would point roughly where V one is.

You can go through these and look at the code.

But the idea is basically that what you're doing with when you're describing your polygon that way is you're almost creating like a unit representation of your polygon that doesn't specify the size of the polygon.

You do that later on.

So as you can see, we've created the vertices here and then we create an asteroid, which is a polygon component.

We're putting the normalized C, it says normalized normalized vertices are basically vertices even in the range of -1 to 1.

We put the vertices and then we specify that we would like the size of our asteroid to be in the 100 by 100 box.

So the underlying flame engine is going to calculate the necessary sizes so that your asteroid fits within the 101 hundred box.

This is great.

If we want to create exactly the same type of an asteroid but a little smaller, then all we do is specify

just 50 by 50.

And you can also specify here the position of the component.

And we specify this as being 500 by 500, like in the point of 505 hundred.

And of course, everything else here is defaulted.

So for example, the anchor point would be the middle exercise.

Number two was basically dealing with how do we know that our shape has hit the screen boundary?

And the solution is pretty straightforward.

We're basically just checking whether the position X and the position Y of the shape is greater than the bounding rectangle, which is the size given here.

So let's quickly have a look at the code.

So we've combined both code elements into one piece of code.

So this takes care of two and three.

So as you can see here, we have the vertices for the asteroid.

We create the asteroid here, our polygon, we're going to painted red just like we did with the square.

We have simply replaced the code for square that we had before with the code for this asteroid.

So if I if I run it right now, what you're going to see is the asteroid, which is basically a polygon shape.

It's going to behave exactly the same way that our square did because we're just reusing the same code.

Except all we have changed is the component that we're rendering.

As you can see, asteroid extended position component, exactly the same way that's squared.

So let's have a look at what it looks like.

All right, There it is.

So if I click here now.

As you can see here is our asteroid.

We have actually put the debug flag to be true.

So you can see we have this asteroid.

And if I.

See how it goes out of bounds.

Boom.

As you can see, objects active is zero.

So our ability to see whether the object has left the screen boundary is working well.

So let's have a look here.

As you can see here, we have created a utility class and we basically check if the position that we put in is out of bounds, which is the position of our asteroid.

Then we just remove it and you can look, we have abstracted everything into a utility class, which we can then reuse.

And you can see here that we have, for example.

This is position out of bounds and you can read the documentation here as well.

So as you can see, if I click it a few times, we've got all these nice little asteroids twirling around.

And as you can see at the top, objects active is going down as the anchor.

You can see the anchor hits the boundary.

That's when it's going to be removed.

So you have to be kind of careful as to where your anchor is because in reality, a better solution would be to remove the object the moment it leaves the boundary or maybe the moment it hits the boundary.

But this is a good enough solution to our exercise.

All right.

Thanks so much for watching.

And I'm going to see you in the next exercise or lecture.

Keep encoding.

14. Case Study #1 - Solution to Exercise #4

All right.

Welcome back.

Let's have a look right now at the solution to exercise number four in the case, study number one.

So what that asked you for was basically to build a class that would encapsulate everything about the life bar or the health bar that we were looking at.

So what you see here is a UML diagram that shows what we have done.

So let's first make this a little simpler.

I'm going to make it a little larger for you to see.

Okay, Let's first look at the life bar class itself and you can see that we have a number of private elements that keep track of things.

This is all stipulated in the actual document that you also got the PDF document that shows the solutions.

But as you can see, what we have done is we have basically encapsulated everything into the life bar class.

We have also created a little life bar placement enumeration, which allows you to decide whether you want to place the life bar to the left in the center or to the right of the upper left corner of the particular object that you're attaching the life bar to.

We don't have the ability right now to attach the life bar to, for example, the bottom or to the side.

It's always at the top of your actual object.

The way this is used, as you can see here, most of the elements and the methods are private.

So for example, if you look at this calculate bar position, that is an internal helper method.

The only public methods that we see is increment current life by and decrement current life by.

What these methods do is they allow you to increment or decrement the current life value of the life bar by some sort of a custom number of, let's say, life points.

And the reason why we decided to do it that way is so that you as the user could easily you know, if, for example, there is a large explosion and you got hit with 25 points, rather than decrement something by 125 times, you simply specify remove 25 life points from this life bar.

The only other thing that we have exposed here publicly is the get current life, which tells you how much current light you have.

And we also have a set current life.

So if you don't want to decrement current life by a certain amount or if you don't want to increment the current life by by a certain amount, you can actually set the current life to be exactly what you want it to be.

So rather than going from 100 to 75 by decrement by 25, you could specifically go in and say, Hey, I would like this life bar right now to be 75.

Notice also that we have two getters for the colors.

We're actually returning just the color portion.

So we have the warning color and we have the healthy color.

We are returning, not the paint, which affects more than just the color we're just giving you as the user of this class, we're just giving you the color.

So perhaps you could maybe put it somewhere, match it with something else in your game.

That way you could just query what is the current color of the life bar for the warning, which by default is red and the healthy, which by default is green.

So how do you use the life path class?

As you can see, looking at the square, you can see that we have basically aggregated one life bar element.

And notice that the way Square works with it is that we have a public method called process hit.
Now what process hit does is it tells the square that it has been hit.
So if this was, for example, an asteroid or if this was maybe some sort of a spaceship, when you get hit, maybe by a bullet or something, one way of letting that object know that it's been hit, you can expose a public method that tells it that you have been hit with such and such, for example, value.
This is not the only way to do this.
Sometimes you might want to have that done internally so that, for example, a spaceship is aware that it was hit because it has collision detection.
But in this case, to really illustrate how the life bar works, we have simply created a public method process hit in the square.
So what happens here is that when you tap on the actual square, the component example is 001, which is our flame game class.
It becomes aware of the fact that there was a tap on the screen.
It then processes the tap, and if there happens to be a square at the location of the tap, it lets the square know that it's been hit.
And what it does is it calls the process hit method.
Then the process hit method.
It tells the life bar to decrement.
In our case, we decrement by ten life points.
So in the code when you run it, you're going to see that the life will start going down as you're actually hitting the square.
All right.
So this is basically it.
You can look at this UML diagram a little bit more, but the basic idea was to abstract the life bar class as much as possible away from the way we had it done before in the square.
So now you could, for example, take this life bar and.
Use it with any other object.
All right.
So let's have a look at how this would look in code.
Before we go into code, let me show you a different way of starting an emulator.
So what I'm doing here is I'm under windows, but it's exactly the same command that you're going to

use in any other operating system.

What we're doing is if you use the emulators command with Flutter, and if I hit it now, what happens is that it's going to give you a list of all the emulators that are available.

And usually this is from within Android Studio.

Obviously, if you're on iOS, you're going to see the iOS simulators, unless of course you have installed Android Studio as well.

Then you would see emulators as well.

So what we have here is we have a number of different emulators.

What I am going to do is I'm going to start the pixel two API 28.

So you're basically doing the same thing.

You're still using the Flutter emulator command, but this time you're going to use the dash dash launch, which is going to launch a particular emulator, and then you're going to give the ID of the emulator that you would like to start.

And in our case, as you can see, the ID is simply pixel two API 28 with underscores, which is the first column.

So if I'm going to run it right now, what this is going to do is it is going to start the emulator.

I'm going to bring the emulator in.

Going to expand it a little bit so you can see a bit more.

Even a little bit more.

Okay.

What we're going to do right now is we're going to run this code and see what it does.

Just a quick explanation here.

The square hasn't changed, as you've seen from the WML diagram.

As you can see, what we do here in the on load is we create the life path that is attached to this.

And the create Life bar method if you simply go here.

What it does is it creates the life bar using a named constructor.

You can see we have information here.

So for example, parent size initializes the component with the size of the parent.

If you want to change the threshold, you can read through that.

This is the parent size.

So basically this is the size of the square to which we're going to be attaching this.

So this is basically the size of this component right here.

Next.

What we have is the size that we would like the bar to be.

And we're defining here as basically we would like it to be ten pixels smaller than the size of our component, and we would like it to be five pixels in height.

We're going to specify that the placement of the bar should be in the center, and then we just add the live bar to our square.

So using Composability, both of these right now are together.

And then all that we do here is notice that in the on tap method of the game.

Plus what we're doing is we're calling the component, which happens to be a square.

We're going to call it process hit method if you go here.

You can see that the process hit method.

All it does is it delegates to the live bar object and tells it to decrement the life by ten.

So let's run.

This always takes a little bit of time to run.

And don't forget that in this code, what we have is the square is still rotating.

It's basically the same code as you've seen in the study.

Number one, comparability.

But we have just added the life bar as a separate object, and we're going to reuse this life bar later on in some of the exercises when we deal with collision detection.

And I believe it's in the timer case study where we're going to actually add the life bar to those balls that are moving and bouncing around the screen so that we can see as the balls hit each other how their life is going down.

So this will be great because all we have to do is just add the actual life bar class to that project.

All right.

There it is.

So if I just click anywhere right now.

It spawns a square just like before.

But now what happens and you can see the life bar is fully green.

Now what happens is if I click here, notice that the actual life has gone down by ten.

If I hit here again, it goes down by ten, again and again by ten, again by ten, again by ten, etc., etc..

But you'll see now that when I hit now, because our threshold is specified to be 25, if I hit now,

this colour will become red.

And as you can see, the threshold has been met where right now at 20 this is a ten and now the bar is empty, it's zero.

And if we we can do multiple objects like this and as you can see, all of them behave exactly the same way.

All right.

So this is pretty this is pretty straightforward.

I suggest that you study the code.

Look at the actual code.

Now, one other thing that we have included for you is we have actually added a PDF document for you to look at.

If you go into the Exercise Solutions folder, you're going to see that with this code there is also a PDF document and I'm going to show you the PDF document right here.

So you can see, for example, that right here in Exercise four, you have all this information that defines how we worked with the actual class.

This is an exercise for.

So I suggest that you actually look into reading the PDF document first, then going through the WML diagram.

If we have provided one for you in this case we have and then simply have a look at how the code is working.

Run the code, go through the code.

The code has a lot of comments.

If you see here, for example, you can see that if I go into the life bar, we have fully commented everything in the live bar component.

So you can go through it and see there's some spelling mistakes, of course, because everybody knows that programmers are horrible spellers and I'm always guilty of that.

So please have a look through this and just keep on coding.

All right.

We'll see you in the next exercise or lecture.

See you then.

6_case-study-2-joystick-component

1. The anatomy of the Flame Joystick Component

All right.

In this case study, we will look at how to create a joystick component.

We'll also look at how to use the joystick to control a specific element on the screen.

We will code the ability to move the object on the screen in all directions, as well as spin the object, 360 degrees in place.

We'll add the ability to shoot bullets from the object will basically have a spaceship and it will shoot bullets.

The object itself is going to be a sprite image.

So we'll show you how to load a Sprite, an image.

It will be a Sprite image of a spaceship.

We could also draw a polygon, for example, shape to look like a triangle, which is what was done in the asteroids game.

But we're going to use an actual image at this point.

The shooting action will process a user tap anywhere on the screen as being the shooting command.

So in other words, we will not need a dedicated button anywhere the user taps on the screen, apart from the joystick will tell our game that the user is trying to shoot.

So what is the joystick component?

The joystick component is a special type of HUD component.

HUD stands for heads up display and a heads up display component.

They are basically used for control and status display within the game.

The joystick itself is relatively simple.

We have a background and we have the knob here.

You can see that it's circular, both the knob and the background, but they could really be any shape.

But we're going to go with the kind of intuitive background being a larger circle and knob being the smaller circle.

The knob is what the user moves and background is the boundary as to how far the knob can go.

The knob can come out about halfway and usually no further.

Here are some extreme positions for the knob as an example.

So this is our background.

The knob starts in the middle and it can move into any one of these extremes.

So what you can see here is that it kind of moves a little bit like almost halfway out, but you will actually be prevented from moving any further, just like you would with a physical joystick.

Two very important elements of the joystick need to be considered.

One is the speed of the movement of the knob, and the other one is the angle of the knob, the direction.

So how fast you move the knob would control the speed where you're moving.

The knob should control the direction.

So basically the joystick knob movement provides a velocity vector.

These two parameters, the speed and the angle allow us to control any object with its speed and angle of rotation.

Let's have a look at what it looks like.

So here is our joystick and if we move the knob, as you've just seen it, we're basically creating a vector.

So how quickly the user moved the knob gives us the speed of the movement.

And this is the angle, the angle in radians.

This is important.

It's in radians.

The angle of the knob movement is relative to the y axis.

So for example, 180 degrees would be 3.14 radians, which is basically pi radians.

And as you can see, that line going through the middle is considered to be zero degrees.

So everything is relative to that line.

All right.

So the next thing we're going to look at is the classes that make up the joystick component inflame season.

2. Joystick Component - Architectural UML Diagram

All right, let's have a look at the, um, diagram of our joystick component.

So you will see that even though there are a lot of classes here, in reality, there are only two classes that we write.

So you can see there is the joystick example, which is in main part.

And then there's also a joystick player which is in joystick underscore player.

These two classes are our classes.

These are the classes that control the joystick.

Let's go with the joystick player first.

Our joystick player is going to be a Sprite.

We're going to load an image that will represent our spaceship, which is why we're extending this sprite component.

But you can also see that the sprite component itself extends position component, and we already know quite a bit about the position component.

And the two main elements that we're going to be interested in are going to be the angle and position that already come with the position component.

Notice also that joystick player has access to our joystick example, which extends Flame Game through the Hess game.

Ref Mixon.

The reason why we're using that for now is because we need the flame game component to load the sprite

for us.

So this is the main reason why we're using it right now.

And altogether, notice that the joystick player actually aggregates or contains a joystick component,

which is the class that comes with Flame Game Engine.

And the reason why we need to have the player holding on to the joystick component is because the player

needs to be notified by the joystick component whenever there are changes that the user makes.

So for example, if the user moves the knob or if the user changes, let's say the angle by moving the knob in a circular manner, our joystick player needs to know that because the joystick player will

be the one that will render itself.

Okay.

So that's pretty straightforward.

And you can see also when you look at the joystick component, which is in the joystick under store component dot dart package, you will see that it is a HUD margin component.

And what that means is basically that it's a HUD component, so it's a heads up display component.

So it will be on top of everything.

It will not be affected by what is happening in the visuals that we change.

For example, on the screen, let's have a look at what are the elements.

So we have a knob which is a position component.

That's how the user is going to control the joystick.

We're going to have a background which is also a position component that specifies how far your joystick can go.

We also have something called margin.

Now margin is simply in reference to the screen.

In other words, how far from the edge of the screen would you like this HUD component to be?

That's basically what it is.

Edge Inserts is a class that basically takes the X margin, which is the one that goes from left to right and it takes the Y margin, which is the one that controls the top and bottom.

Then we also have Delta.

Delta is the change in the position of the knob, and we'll see that in the actual code.

That's what alerts us where we're checking every time there is an update.

We're checking whether the delta has changed and there's also relative delta, which gives us the angle at which the actual knob has changed.

So there's two pieces of information that we need.

We're monitoring the delta to see if there was a change.

And then we get the relative delta, which is going to be the angle of the change itself.

Given all this information, let's look at how our main class relates to everything.

So notice that joystick example is simply a frame game, which is like every other frame engine game extends that.

And notice that inside we have two things.

We have the player, which is our joystick player, and we also have the joystick itself.

The joystick component, it's exactly the same joystick component as is actually aggregated inside the joystick player.

The reason why we have that is because obviously the joystick example class being the main class creates the player creates the joystick, but it creates the player with the joystick.

In other words, the joystick player does not create the joystick itself That's actually created by our main.

One other thing that I would like you to quickly look at is that the joystick example uses a mixing, which is called has drag balls, and you have to do that in order for the joystick component to work

properly.

You can see that the joystick component itself uses a dragon ball mixing.

So the joystick example flame game has to tell the system that it is using something that is Dragon Ball.

That's basically what we're doing here.

All right.

So let's have a look at the actual code.

3. Coding the Joystick Experience

All right.

Let's have a look at what this code does.

So I have it running at this point.

And you can see that we have basically two elements here.

We have what appears to be like a spaceship, and then we have this joystick shape.

And if I actually manipulate it with my mouse, you can see that the joystick is working properly and it translates pretty well into how the component is working.

Notice that if I move the joystick further, the speed increases.

So for example, if I to make it move slow, I can just move it a little bit and I can also rotate the shape.

Perfect.

So let's talk first about this shape.

So this shape is a Sprite.

A sprite is an asset.

So first, let's have a look here.

This is what it looks like.

As you can see, this is just an asteroids underscore ship, PNG.

It's a PNG with a transparency mask.

This is what it looks like.

If I, for example, click on it like this, you can see that I can enlarge it.

If I hold the control key and click, then I can minimize it.

Okay, perfect.

But how did we actually include that in our code?

So the first step is if you look at the pub spectre YAML, you will see that we have this entry here which is assets and then underneath we have dash assets slash images slash.

And the idea behind is, is as you can see right here, this is exactly where we have it.

So this is basically exactly in the same place where our code is.

But we have created a separate directory called assets and we have a subdirectory for that called images.

And this is where we put it.

So in the pub spec YAML, what you're doing is you're actually just telling the flame engine where to find the assets and these are the assets that are images perfect, so flame engine will know where that is.

Now how do we load it?

Let's, let's quickly have a look at the code that actually loads this.

So we're actually loading that in a joystick underscore player.

It makes sense because that's how our player, that's the class.

And you can see that what we're doing basically in the on load method, what we're doing is we're using the game ref.

See how here we have joystick player extents sprite component with has game ref.

This gives us access to the actual main flame engine class and the flame engine class actually has a load sprite method which we're using.

So this game ref refers to the instance of the Flame Game Engine, which is basically our main dart right here.

This is the class, the joystick example.

That's our this is what extends Flame Game.

So this is what we're referring to.

So basically what we're saying in the Unload method is we're saying using the main class, let's load our sprite, given that we have told flame engine where to find the sprite, it will load it.

This is of course a future method.

So we have a wait.

This is an asynchronous call, but we're going to wait for it to finish.

And once the sprite has been loaded, we're going to actually set the initial position of our player, which is this shape right here as the size of the screen divided by two.

Notice that the size of the screen is actually this is actually a vector.

As you can see.

This is a vector.

So when we divide it by two, we're dividing both X and Y, and this position is a vector as well.

Now notice that both Sprite and position actually come directly from the sprite component.

So we don't have to define these.

These are automatically done for us.

Of course we have the position because a sprite component is a position component and the actual sprite comes from the fact that it's an extension of position component and it gives us the sprite entry so that we can load an actual sprite image.

Okay, perfect.

So let's right now look at the two classes that we have.

So we have our joystick example class.

And we have our joystick player class.

Let's first look in more detail at the joystick player.

So the joystick player, what it does was we're creating right here, you can see there's a max speed, we're creating the max speed value so that we control how quickly.

The sprite is going to move when we hit the max of the joystick.

So this is our Max.

This is our max speed.

Notice also that the joystick player has an instance of an actual joystick component inside it.

And the reason is that we need an interaction between this component right here and our actual spaceship.

So whenever I move this, for example, whenever I do something with this, the data that it generates, we need to know that because we need to update the actual position and velocity of our ship.

This is why the joystick component is in here.

The joystick component will actually be inside the joystick player and will notify the joystick player of any changes to the joystick component.

So that makes sense.

So now what we're doing here is first of all, we're specifying here in this constructor that you need to initialize the joystick player with a joystick instance.

And also we're setting the size, which is the size of the sprite to be 50 by 50.

This is a vector when you use the method.

All what it does is it basically takes this one argument and puts it into X and y.

This is the same as if I actually wrote something as follows.

And if I run this right now.

You will see that basically the size is exactly the same.

So this is just a shorthand.

So I'm reverting it back.

So this is just a quicker way of writing the same thing.

Okay.

So once this has been initialized, the unload method is going to get called.

The unload method simply just loads the sprite image and sets the position to be in the middle.

So you'll see that if I rerun this again, you will see that this starts right in the middle.

This is what the initial position is.

And then the update method is relatively straightforward.

What the update method does is every time the update ticks, it checks whether there is been any kind of change in the joystick position.

The delta here represents a change in in any aspect of the joystick.

And if the delta is zero, that means that there's been no change.

So basically what we're saying is if there has been some change, let's execute this code.

And this code is pretty straightforward.

What we're doing is we're taking the position vector and we're adding to it the relative delta of the joystick.

So the joystick generates whenever I move this, the joystick generates relative delta.

And as you can see here, this is the percentage presented as a vector and direction that the knob is currently pulled from its base position to the edge of the joystick.

So this tells us basically, like if I move it here a little bit, this relative Delta would have given us that information.

So all we're doing here is we're basically taking the relative delta, we're multiplying it by the maximum speed, and we are multiplying it also, of course, by the Delta time.

Now, because this is a percentage if this is pulled all the way here, this would be one.

So one times 300 would give us the maximum speed, which is why you can see that if I if I'm here, for example, in the center, if I just move it a little bit, you can see that it's pretty slow.

If I move it up a little bit, it's pretty slow.

But if I go all the way, it starts speeding up.

And the main reason for that is that because this is a percentage, obviously 0.5 times 300 is only half of the speed.

The other thing that we're doing here is so this is the position, This is basically determines.

So the position is somewhere here right now.

And if I move it, it just change the position to somewhere here.

But we are also notice changing the angle so we're not only calculating which way it's pointing this, this is like a vector that is pointing, but we're also calculating based on on this subtle movement here, what is the angle that we should represent?

And this is the angle.

So what we're basically saying is make the angle of the ship to simply be the screen angle, which is the angle of the actual knob as it's being pushed.

We're basically saying let it be the delta of the screen angle.

In other words, we're not multiplying anything.

There's no need for that.

We're simply adjusting the angle to the delta of the knob.

And let's actually do a simple experiment.

I'm just going to comment this out.

If I comment this out, what happens is that the angle is just going to be zero always.

So if I run this right now.

So as expected, our spaceship is in the middle.

But notice not what happens.

As you can see, it's not rotating anymore because the actual rotation, the actual angle is not being updated.

So this would be perfect for some sort of maybe a space shooter when all you're doing is moving like left and right and up and down.

But you don't need to rotate.

In our case, we need the rotation.

So we actually keep the angle right here.

Okay, perfect.

So this is the joystick player, basically the spaceship.

But let's have a look at the main right now.

So what the main does is basically it aggregates the player and it aggregates the joystick.

So both joystick and the player are initialized inside our main class.

So let's look at the unload method, basically the most important method here.

So what we're doing is right here, we're basically most of this code is basically to create the joystick.

So what we're doing is we're saying the following We're defining the color of the knob, and the color of the knob is basically going to be green with an alpha of 200, which just saturates the color more and more.

And then the background paint, which is the larger circle, is going to be also green, but with Alpha a little less saturated.

So you can see the difference between the colors.

So these are the two colors and now we're defining the actual joysticks.

So we're using the joystick component class and this comes with a number of attributes.

The first one is KNOB.

We're basically defining the shape and size of the knob, and in our case, it's just going to be a circle component with a radius of 15 pixels and the paint will be this very paint from here, which is the knob paint.

The background which defines the larger circle is going to be a circle component with a radius of 50.

And the paint, of course, will be that paint.

Now, the margin itself is an interesting element because what the margin is, it's something called edge insets.

And Edge Insets is a class that basically defines just margins.

And the margin is really this this little area right here, this space between here, that's the kind of like x margin.

And then this area here would be the Y margin.

So if I change this, for example, to 40, you will see that when I run it, it will move about 20 pixels to the right.

So I'm just going to keep this mouse for reference.

You can see that it has moved 20 pixels, so I'm going to revert it back.

And the other thing that we can also do is this doesn't have to be a circle component.

Let's change this right now to a rectangle component.

So what I'm going to do is I'm just going to comment this out.

So what I've done here is instead of the background being a circle, I actually made it a rectangle.

The rectangle itself is going to be the size of of this rectangle is going to be 100 by 100 vector.

So it's really not a rectangle per se.

It's going to be a square.

And as you can see, I'm also using shorthand notation of all and we'll leave the background as the same.

So if I run this right now, you will see that this will change to a square.

And there it is.

It looks a little weird, but it basically acts the same way.

And you could do the very same thing by changing, for example, this component here.

What is even more interesting is that the knob could actually be a sprite and so could the background.

So you can experiment with that a little bit.

But the main idea is pretty straightforward.

I'm just going to put the code back where it was.

All right, perfect.

So what we do in the download is we have created the joystick and we're also initializing the player by giving it as input the joystick.

So now that we have created these two components, we add them to our Flame game engine.

So these two right now will be part of the game loop and that's how it starts.

Now what is interesting is that our update method doesn't really do anything except I'm printing in the console here the angle of the player.

And let me just.

Clear this.

And you can see, for example, that the current angle I mean, because obviously this is an update method so it constantly showing us.

But you can see that the angle right now is 2.5 radians, which as you can see, it's pointing down.

It makes sense because since the actual rotation goes this way, if it went all the way here, it would be 3.14, which is 180 degrees, which is pi.

So if I let me just restart this so that I don't have to look at a square here.

So you can see right now that the angle is zero if I'm going to completely reverse it, you can see now that the angle is minus, well, I can move it.

Let's, let's go this way.

So for example, right here, you can see that the angle is 1.57, which is about half of pi, and that's about 90 degrees.

And if I, let's say, pointed downward like this, I've it's difficult to know.

It's not very easy to control, but that's roughly close to.

There you go.

I almost got the pi it's three point almost like 3.7.

This is almost 180 degrees.

So the idea basically is that I'm outputting simply the angle of the player.

Let's have a look if there's anything else here.

But that is basically it.

This is pretty straightforward.

You can experiment with a little bit.

What we're going to do in the next version of this code is that we're going to add the ability to tap

on the screen so that whenever you move this and you tap it will shoot bullets.

So let's get to that.

See you in the next lecture.

4. Shooting Bullets - The Issue of Angle of Rotation

All right.

Welcome back.

One very interesting problem that you have with the joystick is as follows When the object being controlled

by the knob is, for example, shooting bullets, we expect the bullets to move at the angle of the

rotation of the spaceship.

In other words, if the ship is rotated 45 degrees clockwise, for example, then the bullets should

also move at 45 degree angle.

Note that the bullets are not being rotated.

They simply just move along the line of rotation.

Here's an example.

The vertical line is always used here as a zero degrees reference.

So let's say we have an angle.

We have, for example, our spaceship is sitting at that particular angle.

And now what happens is I would like to shoot some bullets.

The expectation is that when I shoot the bullets with the spaceship being oriented the way it is, I

would expect the bullets to go along that red line vector.

Here's another example.

Let's say we rotate the object like our spaceship a little bit more.

Let's call this angle A So here is where the spaceship is pointing.

And when I shoot, the expectation is that the bullets are going to travel at the angle that was created by the spaceship.

If you have an angle A in radiance in the range from minus pi to pi, then to convert it to a vector, you would do it as follows.

So if we know the angle, but we would like to convert it to the vector because vector gives us the direction, we could use this formula.

The x coordinate is going to be cosine of the angle and y of that vector will be the sign of a angle.

So that's, that's one reasonable way of obtaining the directionality vector from an angle.

But the flame engine makes it even easier.

What we do is we can simply rotate a directional vector, so we create a vector zero minus one.

And as you remember from previous lectures, vector zero minus one is a directionality vector that is simply pointing up.

It's just a vector that is exactly at that line that we have in the middle, the one with the dashes.

So all we have to do is create a directionality vector and then rotate that vector by the angle by which the ship has been rotated.

And that is actually going to give us that red vector, which we're looking for.

All right.

So let's have a look at what that would look like in code.

5. Shooting Bullets - UML Diagram

All right.

So let's have a look right now at this improvement on our previous joystick component code.

You will notice right here that we have added a bullet class.

It's a very simple class.

It holds speed, how fast it's moving and velocity, where is it moving?

Notice also that the bullet itself at this point with this architectural diagram is controlled by the joystick example class, which is the flame game class.

The reason for that is that both player, which is what the joystick controls and the joystick itself are part of the joystick example class.

So it makes sense that having those two components there, the bullets would be also there.

And it's also in the joystick example that our handler for tapping is there.

In other words, whenever a user taps on the screen that tap information goes to the joystick example class, it then finds out where the player exactly is, the position of the player, it finds the angle of the player and then creates the bullet and fires it with the particular speed and velocity as per what you will see in the code.

That's really about it.

But it would be a nice idea for you to kind of like look at this architecturally and understand why is it that the bullet is controlled in the joystick example and for example, not inside the joystick player?

You could you could work both ways, but I just want you to think about it.

All right.

So let's head over to the actual code and have a look how it works in code.

6. Shooting Bullets - Coding the Solution for correct Bullets angle

Let's have a look right now at the code where we're actually shooting some bullets.

This will also look into how is it that we can align the angle at which the bullets are shot with the angle of the spaceship.

Okay.

So let me just run it first so that we can see what we're talking about.

All right.

Almost there.

Okay.

So notice a couple different things here.

First of all, we're outputting the exact degree at which the player Sprite is oriented.

So right now, for example, is zero radians.

If I move it this way, you can see that it's 1.49 radians.

This is negative because we're going the other way.

So this is just a quick reference as to what the actual angle is.

This is not really necessary for a game, but it just gives you a visual element that right on the screen you can see what's going on.

Okay, perfect.

So look at this.

So I'm moving it.

And if I tap on the screen, you can see that we have little bullets shooting and you can see that the bullets are perfectly oriented with the angle of the spaceship.

So if I move the angle that way, it will shoot that way, etc..

So this is pretty straightforward.

So first, as you can see right here in our lab, we have main dart, we have the joystick player like before, but we now have a bullet dart.

It's a separate class.

So let's have a look at the bullet dart and you can see that the bullet class has three attributes and the unload render and update method.

So let's have a look at it.

So bullet first of all extends position component makes sense.

We're specifying the paint to just be white.

We're specifying the speed to be 150 pixels per second or 150 pixels.

And we also have a velocity.

The reason why I use the underscore, which I haven't used in the other ones, is that I just wanted to make a differentiation here between, for example, when the bullet enters the velocity here, I want it to very quickly have a simple differentiation between the two of them.

So these are the three elements, and the bullet constructor is pretty straightforward.

We're initializing the bullet with the initial position and the initial velocity.

Anybody who calls this will basically say, okay, I would like to create the bullet at such and such position.

So for example, in our case it would be the position of the tip of the spaceship and we're also initializing it with velocity which defines the direction.

So where and where to, so to speak.

And notice also that we're defining the size here as just being two by two.

So it's a tiny little bullet.

As you can see, when I click, it's pretty small.

If I make it a little larger, let's say four, you will see that it will be a little easier to see it.

I mean, now it's a slightly bigger bullet, right?

So there you go.

All right.

Now this is just an initialization.

And then in the unload, what we're doing is we're taking the velocity that was given and we're going to scale it to speed.

In other words, when we're given the actual velocity, that could be anything.

It could be something that really is large.

All we want from velocity is the angle.

We want the vector to be defining the direction.

We don't want that vector to be defining the speed.

So the reason why we're doing that is what we're taking is we're taking the velocity that you're initializing the bullet with and we're scaling it to our speed to our 150 pixels.

So this is just a simple scale.

So what do we do in the update method?

In the update method, all we're doing is we're basically taking the position and we're adding to it velocity times the delta time, and this is basically equivalent to position equals to position plus velocity times delta time.

But this is just a nice little handy vector method, so this just adds it.

And then all that we do here in the render method is we're basically just on the canvas, drawing a little rectangle with the size defined right here and with the paint defined right here.

And this is basically it.

So the bullet class itself is pretty straightforward.

It's no different than the square class that we've done before is just a simple little position component.

But how is it being utilized?

If you look at the joystick player, you can see that the joystick player, the code hasn't changed at all.

The player, the spaceship is not aware in code that there's any bullets coming out of it.

The actual control of this is in the main and you can even see here that we're actually importing bullet dart and of course the joystick underscore player.

Okay, so let's have a look.

This is us.

Before we have our joystick player, we have our joystick.

And this right here is just our little handy text that will be used to display the radius.

In on tap.

So when I click right here.

And on tap, we're basically doing the following.

We're defining the velocity initially to be vector zero minus one, which is just a vector pointing straight up.

Then what we're doing is we're taking the player angle, which is the angle of the spaceship, and we rotate this velocity to be that particular angle.

This velocity then is what we add as a velocity to the bullet.

So this code right here is all you need to properly direct the bullets so that they align with the direction of the player.

So we just get the player angle, we rotate the velocity by it and we just push the velocity into the bullet and that's basically it.

And then this simply is just a call to Super OC.

So now, now that we have done this, how do we display the gradients?

We're pretty straightforward in our render method.

All we're doing is we're taking the player angle, which is exactly what we've done here, and we convert that to a string which gives us five decimal places.

So you can see right here, one, two, three, four, five.

So we're basically kind of it's not so much that we're rounding, we're basically just displaying five digits after OC.

So all we're doing here is, as you remember this, this dollar sign open bracket player, that angle to string is fixed five close bracket or close parentheses.

That is a dynamic way of putting something into your static string and then it will say radians.

So what it does is this part here is the dynamic part.

This is what happens here.

This is then injected into the string, and the string is simply shown here.

And notice that we're outputting the string at location 20 and then size y size y is basically the full height -24.

So all we're basically doing is we're taking the full height of it, subtract 24 to give it that space.

So it's basically 20 this way and then 24 up.

So this is, this is pretty straightforward.

So now.

If you look at the rest of the code, all that, all the magic is really happening right in the on tap method.

And it's pretty straightforward.

What we all we're doing is every time a user taps, we create a bullet and we add this bullet to the list of children for the flame game and this bullet.

Then, because this has its own update method, it then updates itself and renders itself here.

If we removed these two methods, obviously nothing would be happening.

So the actual rendering and adaptation of the bullet class is obviously delegated to the bullet class because it knows best how to update itself.

And this is really it.

The biggest kind of magic that is happening is right here in the main dart is just this part, which is basically just figuring out what the angle is and then properly shooting the bullets at that particular angle.

All right.

So please have a look at the code, go through the code and through the architecture diagram to kind of have a better picture.

But otherwise, I hope you enjoyed this and I'll see you in the next lecture.

7. Case Study #2 - Exercises

Let's have a look at some exercises for case study two, Exercise one.

What I would like you to do is currently when bullets are added to the game instance and the user shoots, they are never removed once they leave the screen.

It's pretty similar to what we did before.

But this is even more important because you could generate a lot of bullets.

This means that they are getting updated and rendered but are not visible on the screen.

So your task is to write code that will remove the bullet the moment it goes out of the boundary.

Exercise number two is going to be currently the ship can be moved out of bounds and it disappears at the code to have the ship's position wrap around the screen so that when it moves to the top of the screen, it reappears at the bottom and vice versa.

And when it moves to the right of the screen, it reappears on the left and vice versa.

So basically, if I move the ship to the right and it reaches the end, if you will, or the edge of

the screen, I would want it to kind of disappear and then come from the left.

And the same thing for when you go up or down.

All right, well, have fun with those.

8. Case Study #2 - Solution to Exercises #1 and #2

All right.

Let's have a look right now at the solution to case study.

Number two, exercises one and two.

We have combined everything into one code base.

One of the things that we wanted to do is we wanted to make sure that whenever we shot the bullets and the bullets reached the boundary of our screen, that we remove the bullets because there is no point for them to be traveling any further.

So you can see right now I have the code running here and it shows that there are two children in the component tree and the two children are the joystick and the spaceship.

Those are the two children.

So notice now when I shoot the bullets, how the number of children is going to go up.

So I'm going to shoot three bullets.

One, two, three.

And as you can see, it has gone all the way to five and then back to two.

And you can even see that whenever I click and the bullet shoots, just when the bullet hits the boundary, you can see in the debug console how the number of children goes down.

The other thing that we were going to do in the exercise was being able to wrap the spaceship around the boundaries.

What that means is basically that the spaceship doesn't just fly off and we never see it.

So let's have a look at that.

So if I'm going up, you can see it comes at the bottom.

If I go here, it comes from the right.

So this is pretty this is pretty straightforward.

That's what we meant by the wrapping.

And as mentioned in the other exercise solutions, we have provided you with a PDF document that shows how we arrived at the solution.

So you can see here, for example, that we have exercise one, we make some assumptions, we have some

information about what the solution is, some information about how good the solution is, and maybe what could be made to make the solution better.

You can go through this.

This document was provided so that you can see the reasoning, the kind of architectural reasoning, and then you can look at the code.

So here we had solution to exercise number one, and here at the bottom we have the solution to exercise.

Number two, both solutions have been put inside one code base.

If we go right now into the utilities, you can see that we have two methods.

We have wrap position, which wraps the position.

There's also an API documentation here and we also have is position out of bounds.

Notice that both methods are static so that we could easily reuse them in other circumstances.

And in our game we're going to use a utility class just like this one with a lot of helper methods that are going to help us with some of the functionality.

So this is basically it.

Please have a look at the document.

Also go through the code and we'll see you in the next lecture or exercise.

And as always, keep on coding.

7_case-study-3-sound-in-games

1. Using Sound in flame Engine

All right.

Let's talk about the very important aspect of sound in our games.

Sound is very important in video games, and we can break it up into two categories.

There are sound effects.

This could be explosions, beeps and clicks, little audio clips of short music themes or audio clips

of speech in general.

This creates the tapestry of the sonic experience in a game by adding many sonic elements to the gameplay,

usually on demand.

Think of an example of when you have a spaceship and you're firing bullets.

If you don't fire bullets, there's not going to be the sound for the bullets.

So obviously the sound of actually firing a bullet or an explosion is an on demand element because it

is only played when a particular event happens.

The next category very important is music.

This could be background music that plays while the game is being executed, or it could be some atmospheric

sound in the background and it will usually be looped.

The idea, of course, of using sound in games is to enhance the game experience through sound.

Okay, so how do we actually achieve this?

We need to add the sound module as a dependency in our perspective YAML file.

In other words, audio does not come natively with flame engine.

This is how you would do it.

You go into the dependencies section, you add the flame, underscore audio, and you put the version

which in our case is 1.0.0.

Then what you need to do is you need to add the specific assets to the assets directory.

This is how you would typically do it.

You can say flutter and then you have assets just like what we did with paying for our spaceship.

You then go dash assets, slash audio slash and you can put the name of the asset and you can actually

list all of the assets like that.

Another possibility would be to simply just go assets slash audio slash, and that's enough.

You can then directly reference the name of the file and this is how we're going to do it in our game.

Lastly, before you can start using anything, since you have added flame, underscore audio to your dependencies, you then need to actually import the package.

Once you have done that, what is the code that you're going to use?

We can use the following methods.

So for example, as you see, there is a few of them.

But one common element of course is that we're using a static class reference of flame audio.

So for example, if you want to use short clips like a beep or firing of a bullet, you will use the flame audio dart, audio cache play, and then you're going to give the name of the actual file.

You could also loop using very similar method where you can say flame audio loop.

Then you can also have a longer audio which would not necessarily be looped, and then you could loop a longer audio file with the loop long audio.

A long file would be something like maybe 30 minutes or 20 minutes or 40 minutes.

You can experiment with that.

And if you want to be able to ensure that your background music can be paused.

For example, when you're playing a game and let's say somebody asks you a question and you want to pause your game.

So if you pause your game, you don't really expect the music to be playing, because sometimes the music is set to the specific maybe difficulty or the specific area in which you're playing.

So you would want to pause the music and if you want to do that, then you will use the Flame Audio BGM for background music and then don't play music.

MP three Notice also that most of the sounds here in these examples that you see are amp threes, but there's many other formats that can be played.

For example, Wavefile as you will see in code.

So now that we know how to play sounds, the question is what is the best way to play the sound?

And we're really talking here architecturally as an example, consider this in our code we shoot bullets when the user taps on the screen.

We seem to have two simple choices.

The first one is we play the sound in the main class when the user taps on the screen, just as we add the bullet to the children of the flame engine class, that's one way.

The other way is to play the sound inside the bullet class when it is actually created.

I'm going to show you both ways of doing that.

We will explore both options in code and discuss the caveats and benefits of either.

One.

Last thing you need to consider is the efficiency of loading sound files.

To avoid any delay in loading, we would normally load and cache the sounds for future uses as follows.

You can use load assets.

It's an asynchronous call and we would say await flame audio, dot, audio cache load all and then you would load all of the different amp three or WAV files that you have.

Once you do that, those would actually be loaded ahead of time.

So this would be good if you have some sort of a loading screen or something like that, because it might take a little bit of time to actually load them.

So why would.

We want to do that.

Caching is a pretty efficient way of doing this since the system loads the files into memory where they can be accessed almost instantly.

So compared to IO operations that are needed each time files are physically loaded, this is indeed very fast and here's how it would work.

You have your files.

These files are then loaded and then they're being played.

This is the non caching.

The other way would be to same thing.

You have the files, you're putting those files, let's say on your disk, and then you're actually writing the files into a cache in memory and then you play the sound.

So the most important aspect here is that in the first example, we're playing the sound directly from the disk drive.

So it has to load at each time.

Whereas in the second way what we're doing is we're loading or pre loading all of the audio files into memory and then we play them from memory, which is much faster.

Of course, that would then increase the footprint of your application in memory.

So if you happen to have, for example, files that altogether are like 100 megabytes, then you will need 100 megabytes of memory.

So you have to be relatively careful about it.

And one other aspect you have to think about is that MP three files are the better way of working with audio because they're compressed and therefore they're smaller.

So you have to consider the nature of your game.

If music is very, very important, then you might load WAV files, but most of the time high quality amp three files are going to be good enough.

All right.

So we're going to be looking now at some code in action.

See you there.

2. Coding Sound Fxs and Music

Let's have a look at how we could add an audio asset into our project.

So if you look right here at the top where we have assets, images, what we're going to do is we're going to add an audio asset in there.

So first let's look at the actual audio we're going to add.

So here is a number of different sound effects.

And for example, here we have one which is like a laser.

So we're going to use this one.

What I'm going to do is I'm first going to create space for it.

So inside right here, assets, I'm going to go create a new folder, I'm going to call it audio.

So now we have an assets, audio and assets images.

So right here inside the audio, I'm going to add our Sound effect.

And what I'm going to do is I'm simply just going to drag this into my audio.

So we have now a laser 004 loaded into our project.

Next, what we're going to do is we're going to add that asset into our assets definition.

So we're going to go into pop specked YAML.

And right here where we have adding the assets, we're going to add a generic audio assets path.

And this simply matches what we have created here.

We're not putting the name of the file here because we can actually specify that directly in our code before our project can play any sounds.

We need to add an actual dependency on flame audio, which is the extension to Flame engine that actually plays sounds for us.

So what we're going to do is we're going to go into our.

Pub spec on Yamal.

And in here we're going to add the dependency.

And when I'm going to save it.

Right now, this is going to initiate a pub, get command, which is going to bring the flame audio dependency.

So this is perfect.

And now what you can do before we actually start putting any sounds in right here at the bottom, you can see that we have a dependency just like we had an outline.

So, for example, outline, as you remember, is used for when you're looking at the source code.

But we also have a little tab here that allows us to see the dependencies.

And you can see that there's a number of different dependencies.

Here is one called Def Dependencies, and if you go into the pub of YAML, you can see that there are dependencies in dev dependencies.

So if we look into the direct dependencies, which is this, this dependencies entry here is direct dependencies and you can see that we have the Cupertino icons here, which is, which is this line right here.

You can see that we have flame, which is this right here.

And you can also see that we have flame audio, which we have just added.

One interesting thing that you should know in order for you to explore this a little bit more, is that if I just make this a little larger here so it's easier for you to see if you actually go inside the flame audio, you can see that there is not only a read me and a pub spec dot YAML, but there's actually an example here.

If we go into the example, you can see that in their assets they have some audio assets that you can use.

So for example, we could grab this background music that OGG and we could add it, for example into our assets and notice that they also have some sound effects where they have some fire amp three and fire two, amp three.

Now notice that what they have done in the example is they have assets, audio and then they actually make a further breakdown into music and SFX, which is sound effects.

We could do the same thing.

This is just a matter of organization.

It's not a very specific reason to do that, apart from just how you want to organize it.

As long as inside here in your assets, you organize it one way or another and then in your perspective YAML and this entry for dependencies, as long as you specify in your own assets some coherent way of defining your structure, you could really do it any way you want because what ultimately matters is when you're loading it, flame engine just has to be able to find it.

And the way it finds it is because you've created it somewhere underneath the project assets, let's say audio, and then you specify in your code out of these assets which one do you specifically want.

So let's have a look at how this would actually be coded.

Once we have added our dependency and we have defined our assets and we have specified the path to our assets, what we need to do now is we need to think about where we're going to place the code for the actual sound.

Before we can do that, we need to import the necessary package.

We're importing the package plane, underscore audio, and then it's flame, underscore audio start.

Once we have this, what we're going to do is we're going to use.

Some code to actually shoot it right here where we have the ad bullet right after or before the bullet.

What we're going to do is we're going to type in the actual code, which goes as follows.

And in our case, it is just going to be the laser for since this is audio already, it already knows that it's going into the audio assets.

So all we need to do is just put the name of the file.

Let's run this right now.

And now if we shoot, we should be getting a sound.

Oh.

And as you can hear, it's working perfectly.

What we have done is we have basically added sound generation.

We've added it inside our own tap.

So whenever I tap and the bullet is created, we add the actual sound of the bullet.

We could actually delegate the sound itself to the bullet so we can basically get this code.

Go into the bullet.

And right here.

And the on load.

We can just add this.

Of course, what we need is we need to grab this import package, put it here at the top.

Going to rerun it right now.

And now what's going to happen is that we're going to be playing the actual audio inside the Unload method, which is when the actual bullet is loaded.

So now if I move this around.

Let me just make sure that it's restarting.

We would like to see this go into the middle.

Perfect.

Now, if I'm moving this around.

You can see that the effect is the same.

So now that we have the sound.

So we're getting a little closer to what the game is going to look like.

But the important thing to realize is that sometimes there's more than one way of achieving the same results.

So you could think about it in terms of.

Is actually playing the sound.

Is that the prerogative of the bullet or is it the prerogative of the thing that shoots the bullet?

This is truly an architectural decision.

There is a number of ways in which we can decide that one is better than the other, and it really depends on what your aim is.

Do you want to make the sound customizable?

Is the sound baked in with the bullet?

So those are the things that you will need to think about.

In our case, it could go into either.

I would prefer to put it, for example, with the bullet, just because it makes it a little easier.

In real world, bullets make the sound in a sense that when you're actually firing a bullet, it's the exploding bullet that makes the sound, not the gun.

So so, you know, there's there's a way of thinking about it.

All right.

So the next thing that we're going to do is we're going to look at how to actually play some background music.

The best place to add the background music is going to be in the on load method of the actual flame

game class.

So what are we going to do is we're going to create a simple background music start routine.

But first what we're going to do is we're going to grab this import here and we'll need it into the main darts.

So let's have it here.

And next, what we're going to do is we're going to create a little.

Helper method here, which is just going to be.

Background music starter.

And what we're going to do in here is we're going to initialize the background music player.

And we're going to load our asset.

But before we can load our asset, we have to add it into the list of our assets right here into audio.

So what I'm going to do is I have a short piece of music right here.

It's called Race to Mars.

What I'm going to do is I'm just going to copy it and drop it in here.

Notice that it's an MP three.

Now, one thing that is important is you can't just drop any file.

This has to be named properly.

It can't have any spaces.

So we're going to rename this.

And we're just going to make everything lowercase and we're going to make sure there are no spaces.

So to make it readable, we're going to simply just put underscores.

So this is good.

So now we can now that we have that asset, we can go.

And we're simply going to play it.

This is done.

So now that we have created this routine, what we're going to do is we're going to actually call it and we're going to call it in our load method for the game.

So I'm just going to comment this out.

Now that we have this, we can save it.

And we can run it.

So when the unload method is done, our background music will start playing and it will loop throughout the game.

We'll see you in the next lecture.

3. Case Study #3 - Exercises

All right.

Let's talk about a few exercises for you to do.

So exercise one is going to be as follows When the bullet is fired, we make a firing sound.

What about adding an explosion?

Sound when the bullet hits the screen boundary?

Note that eventually we're going to add a sound like that, like an explosion for when an asteroid is hit by our bullet.

We have at it an explosion asset in the resources directory with your case study code, so you can use that.

So go into the actual source code that we provided you for this case study and you will have resources directory.

That's an extra directory that has some extra resources for you to play with for the exercises.

Exercise number two.

In this one, we would like you to experiment with layering sounds.

For example, after we fire the bullet, why don't you add another sound which could make like a swoosh sound as the bullet is traveling.

So the bullet comes out of, let's say, the spaceship, and then it kind of like moves.

You can hear that sound of it moving.

A similar idea would be maybe to add some kind of a sound like that when the rocket is accelerating.

So there are a number of possibilities, but it would be nice for you to experiment with that.

So I hope that you're going to have fun with it.

I'll see you in the next lecture.

4. Case Study #3 - Solution to Exercises #1 and #2

Welcome back.

This is going to be a case study number three exercises one and two combined together.

We're going to go over the solution here.

As usual, you will find the solution in the PDF that has been provided.

Let's have a look quickly at exercise number one.

What about adding an explosion sound when the bullet hits the screen boundary?

So what we have actually done is we have replaced the original bullet firing sound to be something different.

But we've actually done an interesting thing here.

We're going to add a new effect, a visual effect, which is the camera shake, and we're going to use the flame engine's camera shake method to achieve this effect.

The effect comes with two parameters.

One is how long you want the shake of the camera to be happening.

The second one is the intensity, and you control the second element, which is the intensity of the shake by specifying the displacement distance in pixels.

In our case, we're going to use five pixels as the displacement, and we're going to have the duration of the shake to be only 250 milliseconds, which is quarter of a second.

So 0.2 5 seconds.

We have provided you here with the actual URLs where you can get the sounds.

And I suggest that you actually look at that whole resource, which is the free sound, dawg.

There's much more there that you can use for your own games.

The solution is pretty straightforward.

Whenever the bullet goes out of bounds, we create an explosion sound and that's basically it.

Now, for exercise to what you have been asked to do is to layer another sound, kind of like maybe add some sort of a swoosh sound as the bullet is traveling.

So we've added that too.

And what we basically do, as you can see here, if I scroll down a little bit, all we have done is just layered an extra sound on top of the original explosion sound.

And because the second sound that we're layering is longer, it will give you that impression that the bullet is flying.

So let's quickly just look at the code actually in action.

So I'm going to run it.

And what I would like you to do is when the code is running, just listen to what is happening with the sound.

You're going to hear the sound of the firing of the bullet.

Then you're going to hear that swish sound.

And then the explosion.

So as you've seen that in action, it's pretty straight forward.

Please do have a look at the code.

But basically, if we look at the bullet, for example, all that we do here is you can see that we're just layering two sounds.

One is a missile shot, the other one is a missile fly by.

And that is basically all that is happening here as far as the camera shake is concerned.

You can see it right here.

When we're updating the component, we check whether it is out of balance by giving it our own position.

And if it is, we remove the bullet from the component tree.

But then we also play the missile head, which is the explosion sound.

And then you can see that we're using the camera shake functionality, which gives us the duration of quarter of a second and the intensity of five pixels.

One thing that you can also notice when you're looking back at the footage is that what is shaking is the kind of like a camera point of view.

But what you don't see shaking is the joystick and you don't see also what we have written, like the text for Radiance is not shaking either.

That's because the camera itself does not encompass those two elements.

The joystick itself is part of HUD, so the HUD is not influenced by the camera shake.

And this makes complete sense because you don't want your HUD elements or what you have written on the screen to be shaking around.

All right.

This is everything for this particular exercise solution.

See you in the next exercise or lecture.

And as always, happy coding.

8_case-study-4-collision-detection

1. Collsion Detection in games

All right.

We're starting to get to some exciting things in this case study.

We will look at collision detection.

How do we know that objects on the screen have collided?

How to complex objects like sprites or polygons collide?

And obviously, this is important.

If a spaceship, for example, is shooting bullets and they're going to hit asteroids, we need to know that.

Consider the following simple square shape.

Collision detection is simply the overlap of intersection points between shapes.

So here we have our simple square shape, and here's another simple square shape.

And the two have obviously collided.

Collision detection works on figuring out the intersection points as shown in this diagram.

These two shapes have come in contact and these two red dots show the intersection points.

Now, collision detection works for an object.

So if we have three, then it would work, for example, for three objects.

And what is important to understand is that we get a notification for each object that collides with another.

So for example, if object A collides with object B, then automatically that means that object B also collides with object A This is important because as you get a lot of objects, that is a lot of interactions.

Now, the way collision detection works is through a concept of hitbox.

In hitbox is the shape that is added to the actual object to act as the collision boundary.

In other words, the shape of the object itself is what we see.

Hitbox is how we detect collision.

The two are not necessarily the same thing.

For example, if we have a simple object like a circle or a rectangle, then the collision hitbox can be the shape itself.

That makes sense.

So if I have a square, the hitbox could be just the shape of the square.

And then the two are combined.

We have the shape, which is what we visually see, but we also have the hitbox, which is the area that is responsible for deciding when there is a collision.

But what if the shape is much more complex?

We could create an exact hitbox for that shape.

So let's say we have this kind of strange looking polygon.

We can then create a head box that looks exactly the same way and we overlap that onto our shape.

So the hitbox is exactly the same as the shape.

But to make things simple, we could consider any of the following to be the head boxes for the above shape.

Here's the shape the hitbox could be, for example, an ellipse going around it.

We could also have a square kind of approximating roughly where it is, or we could have a circle.

Each one of the head boxes that you've seen is an approximation, and if we use an approximation, then of course we will have false positives as well as false negatives.

What that means is that if I have a shape and I put a circle around it to be the hitbox, since the circle is not exactly that shape, it's very possible that when the circle which is the hitbox comes in contact with another shape, that doesn't actually mean that our shape has come in contact with that other shape.

It's an approximation.

So why do we do that?

Having the hitbox being defined separately from the actual shape gives you the freedom to define the hitbox however you need it.

This is actually important.

Here is why.

The question basically is why do we need a head box in the first place?

Why not just use the shape?

Exactly.

Consider a very simple sprite.

This is a little spaceship.

Now usually sprites are rectangular or square.

They're not the exact shape of the actual spaceship, so sprites are always rectangles, but most often they represent shapes different from rectangles.

In the example case of the spaceship, the following would be better Hitbox shapes in red if we wanted accuracy.

So here's our sprite.

The whole square is the sprite.

That would actually be by default the hitbox.

But we could define the hitbox to be a little square inside, which means that it's a much better approximation of the actual visual shape that we see.

Here is another approximation.

This time it's an ellipse that just goes around.

And of course, as you can see, there are areas that are not really part of the spaceship that would give us a detection in a collision detection algorithm, but it's much better than the outlined square.

And here is even a better approximation.

Now, the best approximation would be the exact shape of the spaceship.

It just takes more effort to define that because you would need to use a relatively complex polygon.

So the point here, though, the reason why we defined the shape itself and then the hitbox being a separate entity is that it gives us a lot of freedom when we're defining the hitbox in terms of collision.

Okay, So now that we understand the concept of the hitbox, how do we do collision detection?

So in flame Engine, we do collision detection by making our objects use the collider ball mixing and by adding a hitbox mixing.

In other words, if we want to detect collision between objects, we add a collectable and hitbox mixing to each such object's class definition.

So for example, if we have a bullet class and an asteroid class, how would we define these classes so that we could detect the collision, for example, between a bullet and an asteroid?

So first we add a collider mix into the bullet class and a box.

We then add the collider mixed in to the asteroid class and a hitbox.

Now, this is the first step in flame engine.

Objects are added to the Flame Game class hierarchy componentry so that they can be part of the game loop cycle in the same manner as the update and render methods.

This means that we will need to tell the flame engine that we would like to add the check for collider

balls to the game loop.

Now this is important.

Checking for collision detection, especially when you have many objects, is very expensive, so we do not want to wire the game loop to constantly be checking for collisions unless we want it.

So this makes sense.

Since collision detection is very costly, we would like to only hook this functionality when we need it to hook collision detection to the game loop.

We add the mixing has collider balls to the flame game, and the final thing that we need is we need to override the on collision method in each of the collider objects that we want to detect the collision for.

To recap the whole process, each object that we would like to check for collisions has to add the collider mixed in and define a hitbox.

The flame engine has to add the Hess collider balls mix.

Then we then override the on collision method for each object that we wanted to monitor for collisions.

Let's look at this in more detail.

Consider three Collider objects a bullet, an asteroid and a spaceship.

We have a number of possible interactions.

A bullet can collide with a bullet.

A bullet can collide with an asteroid, and vice versa.

An asteroid can collide with a bullet.

And of course, we can have an asteroid collide with an asteroid.

We also can have bullet colliding with a spaceship and vice versa.

And we can have an asteroid collide with a spaceship and vice versa.

So as you see, it's a lot of interactions.

So when you ask the flame engine to notify you of any collisions, the engine tracks all possible collisions and notifies you when any collider or hitbox objects intersect.

The notification is sent to any collider object that is interested in knowing what it collided with.

Now that's important that it's interested here.

We're only interested in the bullet and asteroid and asteroid and spaceship interaction, so we only really need for the asteroid to be notified of the collision.

Let's look at this scenario from the perspective of the flame engine.

This is our game loop.

The game loop starts with getting the input data from the different sources.

Then the update method is being called so that the game state can update itself with that data.

Then the game loop is going to detect collisions.

And this is new.

If we have wired our game to detect collisions and we have instructed the Flame Game class to check for collisions, this will be added to the game loop.

The collisions is going to send notification to all collectable objects that have registered to receive collision notifications.

After that, the render method is going to be called.

So let's look at the collision detection in a little bit more detail for each pair of colliding objects that intersect.

In other words, that have collided.

A call will be made to the on collision method, and the on collision method call will provide the intersection points with the other object.

So basically imagine we have object one and object two.

When the two objects intersect, the collision method is going to be called for each one of the objects that on collision method will provide the intersection points between the objects.

It will also tell you which is the other object that you have collided with.

So if object one collides with object to an object, one's own collision method is called that on collision method will be called with the intersection points where object one and object two has intersected and also it will provide object two as the object with which object one has collided.

This is what the method signature looks like.

It's an on collision method.

There is a set of vectors which is a set of intersection points, and then the other is the collide object that this object has collided with.

All right.

So now let's have a look at something a little more visual.

We're going to look at the diagram of the code that we're going to write for collision detection.

2. Screen Boundary Detection and HitBox Rendering

All right.

Let's talk about a special case.

One special case that we will see a lot is the detection of the screen boundaries.
And I have given you a number of exercises where you were asked to think about it.
How would you detect, for example, that you have reached the edge of the screen?
This can be treated actually, and interestingly enough, as a collision detection problem.
And flame has a nice solution for this.
Flame has a special collider class called Screen Collider.
This class can be used to detect if any other collider object has collided with the screen boundary.
If so, then we know that our object has reached the boundaries of the current screen and we can, for example, bounce back from it or we could remove it from memory, etc. This is how you would test it.
And you will see this in our code if when there is a collision, if the other object that we have collided with.
So I have a bullet for example, and there's a collision detection and they collide method is called the other object in that call.
If it is screen colliding, well then we know that we have hit the boundary of the screen.
That is basically all you have to do.
One other problem that we have to think about when you're debugging your code, and especially when your hitbox shape is different from the visual shape like we've seen in the example of the little spaceship Sprite, how do we render the hit boxes so that we can, for example, debug our code?
There are two ways to do it.
One is we use the special method, render hit boxes on the canvas and flame to render the hit boxes.
The second method is one that we have seen already.
We set up the debug mode variable to true in our flame game class.
Flame Engine will usually use a specific color to render the hit boxes and when it is in debug mode you will also see coordinates displayed on the screen.
So for example, as you will see in our code, the shape inside the filled in ball is our shape.
The circle outside is our hitbox.
So if we're going to render hit boxes, this is what you would see.
So the hitbox drawn around the shape is what you would see if you were using render hit boxes.
On the other hand, this is what the debug mode would render.
It's a little bit interesting.
So you will see that the hitbox drawn around the shape using debug mode is still the circle.
But notice that we're also have what looks like a square.

That's square is just a delineation of the shape boundary in general.

To give you the x and Y coordinates of the upper left corner and the lower right corner.

Also the little cross in the center is the anchor point for the shape and the outlining box slash square is for reference only.

The head box is the circle.

All right, So that is it for collision detection.

We'll see you in the next lecture.

3. Collision Detection - UML Diagram

All right.

Let's have a look at what our UML diagram reveals about collision detection.

So let's start first with our main class, which is circles.

As you can see, it extends flame game class, which means basically this is our main class.

This is where the game loop is happening.

Notice there has collided balls mixing as was stated before.

This makes sense is necessary when we want our flame game class as part of the game loop to check for collision detection.

Without that makes sense.

No collision detection will be done.

The has collider balls mixed in basically tell circles please do collision detection.

The next important thing is the my collider class.

Now this class is basically just a bunch of circles.

It's going to actually draw a circle on the screen and that circle is going to have a hitbox, which will be a larger circle for illustrative purposes.

What we're going to do is we're going to actually detect collisions between the same objects.

So it's not like between a bullet and an asteroid, which are different classes here.

We would like to detect the collision between two of the same classes.

So in this case, my collided.

So let's have a look at what my collectible does in order to be part of collision detection.

So the most important two pieces are that it has a collider mixing and it has the has head boxes mix in any class which declares collide will and has head boxes makes sense is basically saying I would like to participate in collision detection.

The next step is have a look at the on collision method.

The collision method is where each one of those objects of my collectible objects will be notified when it has collided with another object.

And as you can see, the collision method will give you a set of intersection points.

In other words, where did the two objects intersect?

What are the points?

And then it will also provide other which is the other object that this object has collided with.

Everything else about this diagram is pretty straightforward.

You can see that, for example, my collider ball has a velocity, it has an anchor, it has access to canvas.

One interesting aspect is you can see that my collider aggregates hit boxes and in this case it aggregates specifically one which is a hit box circle.

This hit box circle is going to be our hit box shape.

This is where we define what the actual boundary of our object is that we would like to detect as part of collision.

It could have been any other hit box.

It could be a polygon, it could be something else.

But in this case, we're using a circle.

So we just made exaggerated a little bit by creating a larger circle.

But my collectible object itself is pretty straightforward.

We define speed, we define the velocity.

We also define three different colors.

Just for illustrative purposes.

We have a collision color, we have a default color, and we have the current color.

Collision color, as you will see in the code, is we visually will show when two objects collide, the color of their hitbox is going to change.

The default color is just the default color of the hitbox, and the current color is going to be the actual color, either the default or the collision.

One one of those colors is going to be the current color.

In other words, when two objects collide, you will see that the hitbox color will change to show you that they have actually we have detected a collision and when the two are not colliding, it will have a default color.

We also have a Boolean flag, which basically tells us that, yes, we have detected a collision and we also have and this is a bit of an extra, we will also show you how to detect a collision with the screen boundary.

And this is why is wall hit flag is there, too.

Now Michael edible also has a mixon which is the hash game ref that gives it access to the circles class.

One other aspect that I would like you to have a look at is Michael Edible is a position component.

And when our class defines the head box, which is the shape for collision detection, notice that we actually specify that it's going to be aggregated as head boxes.

But you can see in the position component at the top, which is just what it is actually in flame engine, you can see that the position component has the usual, the angle, the position, the size, but it also has a hitbox shape array underscore hit boxes.

When in our code we're going to define a hitbox for Michael editable class, we're going to actually add our hitbox shape to the height boxes because that's what is being used by flame engine.

It will actually detect collision based on your hitbox shape, not.

Your actual component shape.

One other aspect that I would like you to have a look at is if you look at the circles class, you can see that it has as its first child, a screen collider plus a screen collider.

All is nothing more than a representation of the screen boundaries as a collider object.

So what Circles does is it adds screen collider all as one of its children, the first one.

And then whenever we create more microwaveable objects which are then added to circles, because that's all going to be part of the game loop.

The interesting trick here is that because Screen Collider ball is a child of circles and so are my collider objects, when my collider will actually hits the screen collider which is the screen boundary, my collider will is going to get a notification in on collision that will tell it you have just collided with something, some sort of a child object of the game.

And when we look in the other, we will see that, oh, we have collided with a screen collider will object, which will tell us basically that we have just reached the boundary of the screen and you will see that in our code.

So this is pretty much it for the architectural aspect of this.

Let's have a look at the code.

4. Coding Collision Detection

All right.

Let's have a look at the code sample for collision detection that shows you how collision detection works.

First, let's have a look at how the actual code performs.

So if I click on the screen, we generate a circle.

Basically, there's a ball inside and then there's an outline of the outside.

The ball inside is the actual shape.

The outline is our hit box.

So the purpose of the hitbox is to delineate or to outline the area which allows the flame engine to detect collision between objects.

So, for example, if I create a shape right now by clicking, you can see that it's moving up.

If I put one in the opposite direction, you just saw that when they touched, they changed color.

That was how we're showing collision detection here.

And if I actually click it a few times, you can see that this works for multiple components.

You can have many of them kind of going through each other and lighting up like that.

You will notice that if I click, for example, let's click here in the corner, you can see that this is moving towards the center.

If I click in this corner, this is moving towards the center.

If I move here, it's moving towards the center.

The code is pretty straightforward when it comes to the way these shapes travel.

We basically move it towards the center at a constant speed.

Now, one interesting thing to note here is if you carefully look just at the moment when they come into contact, almost close, going to try to get one that is like very, very close.

Like, look, Right.

They're almost going to try to get one that is very, very, very go.

You could see how that moved that all they have to do is pretty much touch now.

Okay.

So let's have a look at the actual code.

Here's the outline.

We have basically two main classes.

We have collider class and we have the Circles class.

This is of course our typical main.

Our imports are here.

There's really nothing new here components, extensions, material, DART, etc. So in our main we just instantiated.

Nothing has changed here.

So first let's look at the actual circles class, which is our flame game based class.

So you will see first that this class uses a hash collider mixed in and a top detector.

Let's have a look at the Hess collider balls.

The Hess collider was mixed in keeps track of all the collectable objects in the component tree and initiates collision detection, everything.

So basically all that this does is when you declare that your class uses a hash collider was mixing, what you're basically saying is I would like to do collision detection.

Top detector, of course, is pretty straightforward.

We have on top down all that we do here is we create a new microlight object.

We'll just look at that in a second and we initialize it with the position where the user tapped.

Notice that we're adding it to the list of the games.

Children.

All that we do in the on load we're adding as a child a screen collider object, which is a special collectable shape that actually defines the screen.

If I make this debug mode true, and I just rerun it right now, watch what happens.

First of all, you can see that we have this outline, this pinkish outline.

This is actually the head box that screen collider will uses.

In other words, screen collider ball is simply a shape that is the size of the screen.

So any object that collides with screen collider ball collides with the boundary of the screen.

And if I, for example, click now to generate our shapes, you will see that these also have a number of hip boxes.

Now, the hit box here is the circle as explained in the slides.

Let's revert this back for now.

So in reality, you can see that there isn't really much happening in the Flame Game class in our circles class.

We're enabling collision detection.

We're adding a special screen collectable object, just one object into the component tree that defines

the screen boundaries.

And then we have an on top down that every time I click, as you can see here, we're just adding my collider ball object with the position where I just tapped.

So if I tap here, this will initialize the object with this position and if I tap here it will initialize it with that position.

Okay.

So now the actual star of this particular code is the my collider class.

And in my collider class, the first thing that we're going to see is it extends a position component that makes sense.

It's no different than the square class that we had.

It has a game reference to circles class, right?

It does.

And the reason why it does is because we are figuring out where the center of the screen is.

So we need we need the game ref reference.

So that's why we have this game ref circles and these two right here, the collider ball and has head boxes is what makes this class ready for collision detection.

In other words, this class becomes able to receive notifications about collisions.

So what is in this class?

Well, it's pretty straightforward.

We have the velocity.

We have a little flag that basically tells us whether a collision has happened.

We also have another flag that tells us whether we have actually collided with the wall, which is the screen boundary.

We specify the speed to be just 50 pixels per second.

And we have also defined three different colors.

We defined a collision color, as you can see, if the two collide.

There you go.

That's the amber.

So whenever there is a collision that was close, we get the amber colour and then the default colour, which is like when there is no collision, that's what it starts with is the cyan.

And the current colour initially is the cyan.

Now, this the actual color that you see around there, that's the default color of the hitbox.

Every hitbox comes with that color.

Okay, So now let's have a look at what it is that we're doing.

So first of all, in our constructor, we're accepting the position where we spawn the object.

So this is the position and all that we're doing is we're simply calling the super with the position.

We specify a 42 pixel vector.

So basically this is this becomes a square of 42 by 42, and we specify that our object is going to have its anchor in the center.

What is interesting here is that we right away add a hit box to our collider because this class extends position component.

It already comes wired to hold one or more hitbox elements.

In our case, all we're going to do is we're going to add a hitbox circle, and that circle is going to be the size of 42 by 42 pixels.

So when I click here, you can see that the size here is basically you can imagine there's a bounding square here that is 42 by 42.

Let me demonstrate that I'm going to make this much larger.

I'm going to make this, let's say, 64 note what happens right now and it only happens to the hitbox.

So right now you can see that the hitbox has become very large.

So watch what happens.

See how the collision detection.

So in other words, this program right now thinks that these two objects have collided, even though if I, for example, let's say, do something like this, you can see that these objects actually did not really collide.

What collides here is the hip boxes and the size of the head box was 64.

I'm going to bring it back to 32 when we start it.

All right.

So this is this is a much smaller little hit box, kind of approximating the class itself.

Okay.

So we've created the object, initialize it.

We've added a hitbox circle to it, and we then go into the load method.

Now, all that the load method does is it finds where the center is and it creates velocity vector, which is center minus position.

So if I click here and the center was here.

The vector is going to point towards the center, and then we scale this vector to the speed, which is 50.

So basically we're creating a unit vector and then multiplying it by 50.

So this is pretty straightforward.

All that we did here is we specified the directionality vector for this.

So now let's have a look at what happens in the update method.

Well, the update method is interesting.

In the update method, we basically check if we have had a wall, which is if the wall hit flag is true, we remove this object from parent.

This this method comes with the flame engine.

And what it does is it removes itself from the parent and it returns.

Otherwise, if the wall was not hit, we're basically changing the color.

If there was a collision, then we set the collision color.

Otherwise we set the default color.

This is the ternary operator.

So it's like a like an F and else in one in one line.

And then all that we're doing here is like what we're always doing is when objects are moving, we're basically saying, let the new position be velocity times delta time, which gives us the velocity delta, the little amount of path that the ball should have traveled.

And we add that to the current position.

So in other words, this this calculates the change in our position because we're moving and we're just adding that change that delta to the position.

And then of course we re initialize the collision to be false.

OC This is pretty straightforward, but it doesn't really explain much.

And in render, what do we do in render?

Well, we first render the hit boxes, so when you click on that it renders the actual outline.

And then next what we're doing is we're defining the local center, which is the scaled size and divided by two to offset.

I would suggest that you try to figure out what this actually does.

But clearly what is happening here is that when I click here, that little ball filled in in the in the middle there is actually our shape.

And this is this line right here is what paints that.

So what we're doing is basically we're drawing the circle at this location with radius eight and the color is going to be the current color which was set here.

If there's a collision, the color is going to be a collision color.

And if there was no collision, it's going to be the default color.

So in here, all we're doing when we're rendering is we're figuring out where the center of our ball is and then we draw it.

So the star of this code is this method right here on collision.

And what collision does is basically every time that an object collides, this method gets called.

So if I have two objects colliding, so for example, right here, these two objects when they have collided on collision method, will be called for each one of those elements that collided.

You're going to get the intersection points.

This is a set and then it's going to give you the other object that you've collided with.

Now, all that we're doing here is we're really interested in two things when it comes to the collision.

First, we're interested to know whether we actually had a collision and of course we did.

So we set the collision to be true.

The reason why we have this collision true is because we use it right here to set the color.

So we set the collision to true.

And then if collision is true, then we set the collision color.

The other part here is this and this is very interesting.

If the other if the other object that we have collided with is a screen collectable, which is basically the screen boundary, then we set the flag that tells us that the wall was hit to true.

And what do we do with that?

Well, if you go back here, you can see that in the update method.

If the wall was hit, if it was true, then we'll remove it from the parent.

And this is pretty much what this code does.

So I suggest that you go through this a few times, especially try to figure out what is happening here.

This right here, I mean here, this gives you pretty good information as to what is happening.

Okay.

One last thing before we leave here.

What I'm going to do is I'm going to actually show you a different hitbox here.

Let's try a head box rectangle.

Okay.

Now, this hitbox rectangle is going to create a 32.

Let's make it a 48.

Okay, let's run this.

Now you can see that our hitbox is a rectangle.

And notice that when boom, when they touched, when these two hit boxes, not the shapes.

If you look carefully, you can see.

So bold.

But if I go like this.

Let me try this like that.

Look at these two right here.

Very close.

Basically whenever they touch, you get the collision.

So we can specify a hitbox rectangle.

But there's another thing we can do as well.

We could actually create our own shape.

So normally you wouldn't do it here, but let me just do it.

I'm going to add a new shape, so I'm just going to create a quick little shape.

We're going to use a hitbox polygon.

And what are we going to do inside is we're going to define an actual set of polygon points using I'm just going to create a diamond shape.

So one zero, this is going to be zero minus one, and this one is going to be minus one zero.

So I've just created a little shape.

And what I'm going to do is I'm going to take this shape and actually put it right in here and I'm going to rerun this.

And now instead of having a rectangle, we're going to have a diamond.

Well, it's still kind of like a rectangle.

So this is a slightly different way of defining the actual hitbox.

But the most important lesson from all of this is that the hitbox is what actually participates in the collision detection.

It's not the shape, it's the shape of the hitbox.

All right.

So I think it would be great for you to play around with this code and try a few different things,

you know, try slightly different head boxes.

And of course, one other thing that I'm going to do, I'm going to kind of like exaggerate this a little bit.

I'm going to make this much larger.

I'm going to make it 64.

And so right now you can see that it's a it's a much larger hitbox.

But what I'm going to do is because we're actually drawing the hitbox right now, Right?

So we're rendering it.

I'm not going to render it.

Rendering head boxes is great when you're debugging, but ultimately when the user is playing, for example, the game, they don't want to see the hit boxes, they just want to have the collision detection.

So let's let's run this right now and see how our code is going to behave when we're not rendering the hit boxes.

So you can see now I'm going to see you can see that these these balls, they're basically they're color is lighting up much sooner than they touch.

And that is because the head boxes are pretty large.

So if I go back and re render this, that makes sense because when you look at the hit boxes, well, they're pretty large.

But what we can do is we can revert this back.

So I'm going to remove this polygon and I'm just going to go back to to what we had before, which was our hitbox circle.

Okay.

So again, if I render this right now.

You'll see pretty large circles, Right?

But what we can do, since we know that our circle is a radius of eight, which is really 16.

What we can do is I'm going to make this 16.

And now when you look at it, you will see that our hitbox, you can you can see there is an outline right on the edge.

Now, our hitbox is perfectly the shape.

So you can see that they only now collide when they're very, very close.

It's not like before.

And of course now, if I remove the render, you will not be able to see the hitbox.

And at that point, what happens?

These balls will only light up when they're actually physically the shape you see is what collides.

All right.

So I think that's about it.

I hope you have some fun with this code.

Definitely.

Try the exercises and I'll see you in the next lecture.

Take care.

5. Case Study #4 - Exercises

For collision detection.

We would like you to go through the following exercises.

Exercise one when the ball in our code hits the screen boundary reverse slash reflect its direction.

In other words, bounce it off the wall so that it basically bounces around the screen.

So rather than disappearing from the screen and being removed from the memory when the ball is traveling

right now, it should reflect off any of the boundaries.

And think of it this way.

Here's an example of what the expectation is.

Here is your ball.

This is the wall boundary.

This is the vector representing the velocity of your ball.

The expectation is what you see in the red vector.

The expectation is that it's going to reflect or bounce off the wall.

And one way of looking at it or thinking about it is that you can look at that normal line, so to speak,

and you can look at that theta eye and then you can look at that theta or so that's basically the incidence

and the reflection angles.

Those angles are exactly the same.

The idea is basically that you reflect the ball exactly at the same angle as it hits the wall.

With respect to that blue dotted line.

Exercise two is a little bit more interesting, but it's also a little bit more intricate.

Make the balls in our code bounce off of each other as well.

So in other words, as the balls are moving around when they hit each other, you should see the effect

like you have with billiard balls.

They will bounce off each other as well as off the wall.

Have fun with it.

6. Case Study #4 - Exercise Solutions - Solution to Exercise #1

All right.

Welcome back.

Let's talk about case study number four.

Exercise number one.

We're going to go into the solution.

What the exercise asked you to do was to create some sort of a reverse logic so that when the ball in our code hits the screen boundary, we then reverse or reflect its direction.

Now, we could have used the screen collider ball, but we decided to do a slightly different take on this.

There is a reason why we didn't want to use the screen Collider Well, and the main reason is that sometimes,

especially if you have a lot of objects on the screen and if you happen to have objects that are very small and your speed is pretty fast, there's always a chance that in between the update tics, your object might actually go out of bounds but not trigger the actual collision.

So we decided to create something here that is always going to work.

Probably the best first thing is to just demonstrate the code in action.

So I'm going to click a few times.

And as you can see, we have a number of balls right now bouncing around and it seems to be working pretty well.

So as usual, we invite you to look into the PDF documents that go with all the solutions so that you have a better idea of the kind of logic that went into creating the solution.

So here we have case study number four, exercise number one.

We put a number of assumptions here, and as you can see, we have a note here that tells you something about the collision detection and its limitations.

The solution is pretty straightforward.

What we're basically going to do is we're going to get the bounding rectangle of our ball because it's just easier mathematically.

And then we're going to basically test if rect left.

So the left side of our bounding rectangle is less than or equal to zero, which means that we are outside of bounds and the direction itself of our object is to the left.

Or if the right side of our bounding rectangle is greater than or equal to the screen size, which is the width of the screen and the direction is to the right.

So in other words, if the left side of our bounding rectangle is outside and our direction is to the left, or if the right side of our bounding rectangle is outside of the bounds and moving to the right, then we know that we need to reverse it.

So this is why the x direction is then reversed for the top and the bottom of the screen.

We do exactly the same, except that we're just using direct, top and bottom.

The idea here and the reason why we did it this way is that we need to test whenever we head a boundary, whether it's top or bottom left or right.

We need to do the test based on the specific boundary.

In other words, the logic is going to be different if we're going up or if we're going down, the reversal is going to be different.

You could do the same thing with Screen Collider, which would be a little bit more difficult.

But the biggest problem with using the screen collider in this solution here is that sometimes the screen collider might not detect the actual collision.

This is foolproof.

So let's have a look at the code itself.

First, let's have a look at the X direction and the Y direction, and notice that we have split the direction vector into an X and Y component and we are initializing it with one and one, which basically means that our vector is going to be pointing down and to the right.

And as you can see, if I, for example, create another ball, you can see that it's going down and again down into the right, down and to the right.

So then what happens is inside our update method, what we're doing is we're calculating the new position because the X and the Y, if I actually hover over the X, you can see that x is actually the x position of this component anchor on the screen.

So this x and Y already comes as part of our circle component.

The circle component is a position component.

So our microlight class is actually based on a position component.

The position component gives us automatically the X and Y, which is the location of our object.

We don't have to necessarily go dot position x, dot position Y, we could directly go X and Y.

So what we're doing here is we're basically updating our position by taking the direction, multiplying it by speed and then of course using the delta time.

Then what we do is we get the bounding rectangle again.

We do that for the ball because it's just easier for our calculation.

And then what we do is we have this test.

If the left is less than or greater to zero and the direction is to the left, or if the rectangle dot right is greater than or equal to the size of the screen and the direction is to the right, then we know that we're not only out of the boundary, but we're traveling away from our screen.

This is where we reverse the x direction.

We do exactly the same thing for the Y direction here.

And this is basically it.

It works pretty well.

One thing that I would like to do, and it will be a good idea for you to experiment with this code, play around with it a little bit.

What I'm going to do is I'm going to change the size of the ball.

I'm going to make it five pixels.

Now let's just create a number of them.

All right.

And as you can see, it's working pretty well.

The updates are happening at pretty much real time.

Mission accomplished, so to speak.

So please do look through the code and we're going to see you in the next exercise or lecture.

And as always, happy coding.

7. Case Study #4 - Exercise Solutions - Solution to Exercise #2

Welcome back to another exercise solution session.

In this video, we're going to briefly discuss solution to case study number four, Exercise two.

Now, this might seem easy to do, which is to have the balls bounce off each other, but it is actually a relatively difficult problem, especially when using collision detection.

I'm going to just demonstrate the code itself so that you have an idea of what it looks like.

So I'm just going to click a few times and I'm going to basically start the simulation.

So here is one ball.

It's bouncing off the.

In our side of our wall.

Here is another one that I'm going to generate one.

And as you've just seen, they bounced off each other.

I'm going to generate a few more.

And as you can see, the balls are not only bouncing off the wall, but they're also bouncing off each other.

Now, why is this a relatively difficult problem?

You might think that the moment the two balls come into collision, all you have to do is basically just reverse their direction.

To some extent, this is true, but it's not really, unfortunately, that simple.

Let's have a look again.

As usual, we encourage you to look at the PDF document.

First, let's have a look at the PDF document.

The PDF document which has the solution to exercise.

Number two basically starts with a very simple assumption that all we need to do to bounce the two balls of each other is to reverse their directions.

Now, you do remember that when we have collision detection, what actually happens is the collision detection is going to happen twice.

When object A hits object B, you're going to get a collision detection that will tell you that object A just hit object B, but you'll also have the other collision detection, which is going to tell you that Object B has actually hit object A, So you're going to have two pieces of code where you're going to reverse this, but that is not a problem.

That itself is relatively straightforward.

The problem is as follows If you have two balls colliding and you see this in this diagram, what can happen is that when they're colliding because of the time that it takes to actually detect the collision, these two balls might already be a little bit inside each other.

And that's where you're seeing in that second part of the image where it shows the intersection points.

But what can happen is that while we are actually handling that, the two balls are still moving because we haven't updated it, they're still moving towards each other, which means that we're going to get another collision.

And at that point, if the first collision, if you reverse them, and then when you get the second

collision, which is still the same collision and you reverse it again, the two balls are going to actually go back towards each other again.

And unfortunately, we can create something of a loop.

This can create a problem.

So now that we see that we have to have some sort of a solution that deals with breaking this occurrence into three different types of events, we can look at this logically as we have a start of the collision, which is the very first time the two balls intersect, then we have a collision in progress, which is the kind of keep on going towards each other.

And we still get the collision notification.

But at some point, whether they just pass through each other or we bounce them off, there's going to be the collision end, which is when the two balls are not touching each other.

They do not have any more intersection points.

How do we deal with this?

Well, we do have a solution here that we specify that I would like you to go over.

And we have a very interesting, I think, solution which deals with simulating those events by basically keeping track of all the collisions that happened.

You know, that every object in Flutter and therefore in Flame has a hashcode which uniquely identifies that object.

So we can use the hashCode of any object as an ID And what we do is we keep track of all the objects that a particular object has collided with.

So how does that help us?

Let's say we have three objects collide together, so that's going to give a lot of different combinations.

The way we treat it is that we only worry about the perspective of the object that has been collided with.

In other words, we have object A and then object B collides with it or objects C collides with it.

The way we deal with this is that what we do is we keep a map where we keep track of all the different objects that this object has collided with, and that is going to tell us when the object first collided with us.

In other words, we look into the map and we see how we collided with this object.

No.

Okay.

Let's add it into our list of collisions.

Then the next collision that we get, we will check.

Oh, have we collided with this object already?

If the answer is yes, then it means that this is a collision in progress.

But how do we know when the collision has ended?

Well, in our case, because we're reversing the direction the moment we have the first collision notification that this is before we have any of the other object in our map.

Before that happens, we check whether we have that particular object in our collision map.

If we don't, we reverse the direction and we put that object into the map of objects.

If that object is still in collision with us, the next time we get a collision notification, we simply check, Do we have this object in our map?

If.

Yes, That means that this object is still on a collision, which is in progress.

But how do we know when a collision has ended?

What we do is in our update method, we simply check whether this object, which obviously we already reversed its direction, whether this object is far enough from us that we don't have any intersection points with it.

And in our case, because these are just balls then or circles, all we need to do is basically use the radius as our length.

So let's have a look at the code itself and if we go in my collider.

All right, here, let's go into collision first.

So in collision, this is what we do.

We say if the collision contains the key of the other object, we do nothing because that means we already have collided with it.

Otherwise we add that object hashCode to our collisions map and then we reverse our direction.

Then in the update method, what do we do?

Is we go through all the entries in our collisions map and we basically check whether any one of those objects that we have collided with, whether it's distance is far enough that the two don't touch each other anymore.

In other words, is the distance of that object far enough away from me that it cannot have a collision with me anymore?

That would tell us, since we know that we're moving them apart, that would tell us that the collision has ended.

And what we do is we simply remove that object from the map.

This is basically it.

This makes sure that we have relatively nice and simple, clean collision detection routine that also removes any kind of memory, fingerprint or any kind of memory footprint away from our code so that as we have many collisions, we very quickly resolve them and free the memory that normally the map itself would hold, even though notice that we're storing mostly the keys and the objects which are just references.

So so the memory footprint is going to be relatively small even for hundreds of collisions.

So I'm going to quickly just change the size just to make it five.

So we have a little bit more and I'm going to start quite a few of them so we can just kind of have a look at them.

All right.

So I'm just going to start a lot of them and you can see how they're bouncing off each other.

Yeah, that's a lot of balls right here.

And you can see that's quite interesting.

I mean, there's a lot of stuff going on and it's pretty fun to coach something like that.

All right.

So I hope you've enjoyed this.

Please go through the PDF document.

And of course, as always, we'll see you in the next lecture or exercise solution and happy coding.

9_case-study-5-using-timers-in-games

1. What are Timers and how do we utilize them?

In this case study, we will look at how to use a timer to control repetitive tasks.

We will look at how to periodically invoke callback methods at specific intervals.

We'll look at two ways to create timers and flame.

We're going to look at the timer utility class and we're going to look at the timer component.

Now, the timer component is just a thin wrapper around the timer utility class.

Any timer, we'll need to offer the following basics.

We'll have to have the ability to set the repeat interval.

We'll have to have the ability to provide a callback function for timer tasks.

Basically, at any given interval, we would like the timer to do something that would be the callback function.

We also need to have the ability to determine when a timer is finished.

And of course we need a timer that has a reasonable interval accuracy.

So what is a timer?

A timer can be thought of as a trigger which executes a tick at a specific interval.

You get a notification from the system whenever this tick happens so that you can do something at that specific moment.

There are two main types of timers.

We have an infinite loop timer which runs in an infinite loop until it's stopped and we have a counter timer which runs for a specific time interval and then it finishes.

Please note that timers are not threads and usually run on the same thread as the timer creator.

In our case, in terms of flame engine timers are actually executed by the game.

Loop timers are best used for short, simple, repetitive tasks usually of low complexity.

All right, let's have a look at a more visual representation of a timer.

So we start the timer with a call to the start method.

After a period of time, there is going to be a notification.

So basically you will get a timer notification at each interval.

The timer will call your listener your callback function to notify it that a new tick has arrived.

At this point, the listener will execute its trigger code.

In other words, at every step of the interval, your callback function is going to be notified that it should do whatever it does and we can look at it as intervals pass again, you will get the timer notification, another interval passes, you will get another notification and this repeats itself at some point.

Of course, if the timer is finished, you will be able to check a finished attribute because it will be set to true when the timer is done.

Note A very important aspect of the timer.

A timer will only get the tick notification like the timer tick if it is explicitly part of an update method of the flame engine.

In other words, it will only get the notification if it is part of the game loop.

And as you remember, how do we add any kind of component to a game loop?

Well, in the flame engine derived class, what you do is you simply add it to be a child component.

So now let's look at the different types of timers that we have.

So we have a counter timer and a counter timer code is pretty straightforward.

The way you create counter timer is by initializing it with the time interval in which it will run and finish.

You can start the timer by calling the start method, and in order for the timer to actually execute, you need to put it into the flame game engine's update method so that it can update the timer.

And then if we want it to actually render some information about it, here is an example.

What we're doing here is we're basically just rendering the current value of the timer as the timer is running, as the timer is being updated, what happens is it has a certain value that would be the value of the current cumulative tick interval.

So for example, a timer that is supposed to run for 5 seconds, if you peek at the value of current at let's say two and one half seconds, what you would get is that it has so far run two and one half seconds.

So this is what you should know about the counter timer.

It's initialized with an expiry and seconds, so the timer will finish within 5 seconds.

It could go just a little bit over 5 seconds, but it will usually sit within that interval timers or a start by default.

But if you want to control the starting of the timer, you need to set auto start parameter to false because it's true by default.

So here's an example of how you would create a timer, but you would not allow it to start until you're ready.

And as you can see, we put the interval which is five, and we put the auto start to false.

And when it comes to the actual tick.

There will only be a single tick, which is after the 5 seconds have run out.

So when the timer is finished, that's when you would get the tick volume.

Basically, what a counter timer does is it notifies you when it's done.

All right.

So now let's talk about the interval timer.

This is a timer that repeats at a very specifically set interval.

The previous one, the counter timer, simply counts the number of seconds.

So let's look at the interval timer.

So we use the same class timer.

This is just something we keep track of.

This is just part of the illustration of this example.

So here is how we create the timer called interval.

We specify that it should have its tick every second.

This is a very, very simple on tick callback method.

We have to set the repeat to be true, which means that it will repeat forever.

And once you have done that, you can now use the start method to start it.

And of course, similar as you've seen in the other timer example, in order for this timer to work, you have to add it to the update method.

So notice that this timer is initialized with a tick interval in seconds.

The example uses one, so basically this will trigger every second.

If you want to create a tick an interval smaller than a second, then you use fractions.

For example, 0.25 would be a tick that happens every 0.25 of a second.

So every 250 milliseconds if you wanted to, for example, trigger this every 10th of a second, then you would put 0.10.

Now, if you want to create an interval timer, you have to make sure that you set the repeat attribute to true because it's false by default.

And again, the same thing here.

Timers auto start by default.

To control the starting, you will need to set auto start parameter to false.

And definitely in this case you need to implement a callback on tick method to get a notification each time the timer gets a tick.

Now, if you want to know when the timer is finished, you can check the finished property of the timer every time you get a tick, because once the timer is finished, it will then set the finished property to true.

Now what about using actually a component which you've seen there where an example of the utility timer class?

What if we wanted to use a time component?

Well, the thing about the time component is that it's basically a component, so we don't have to explicitly add it into the update method like you've seen with the timer utility classes.

So here's a very simple example of how you would create a component.

So notice that we're adding as a child the timer component and the timer component has a period of ten.

Same idea.

This is in seconds, so this will repeat every 10 seconds.

The repeat has to be true.

Here is an example of an on tick callback method, which in this case just prints.

The 10 seconds have elapsed and that is it.

That is how simple it is to create a timer component.

So the thing about the timer component is that it's just a wrapper around the utility class.

So once you understand how the utility classes work, this is just a wrapper with the convenience that once you're created, you can simply add it to the game loop by adding it to the Flame Game Children list by using the ADD method.

So as just stated, note that with the timer component you do not need to explicitly add the timer to any update method.

All you need to do is just add it to the flame game engine.

This is achieved automatically through using the ADD method, which adds the component to the game loop.

Okay, Let's go into the next video.

2. Coding with Timer Utility Classes

All right.

Welcome back.

Let's have a look at a pretty simple timer example code.

What we're doing here is we're working with the utility timer class.

So what we have, as you can see, is we have just a single class, which is the timer example.

If I open it up, you can see that we have we have two timers, we have a countdown timer and we have an interval timer.

These are just examples.

We also keep track of how many ticks have elapsed for the interval timer.

And this is just a flame game extension with a tap detector.

What we're going to do is when we tap on the screen, we're going to restart our counter timer.

So let's actually run this and let's have a look at what it does and then we're going to explain the code.

It's taking a little bit of time.

All right.

So our code is running.

All right.

So as you can see right away, what happens is we have a countdown where it says elapsed number of ticks and this is the interval timer that is running.

We have set the interval to be one second.

And basically at every second a tick notification is sent and we get that.

So now if I tap on this, notice that the countdown timer starts and it will run till about five and it stops at as you can see, it's 5.02.

And even though we set the timer to go five, it goes a little bit beyond.

Okay, let's have a look at the actual code.

So first of all, as you can see, our class is pretty straightforward.

We extend the flame game and we have a top detector.

What we do here is, as you can see, we have.

White text here.

This part right here, the final text paint Text config is just a quick configuration of font size 20 and a white color for all font.

And then what we have is we have two timers, we have the countdown timer, which we call Countdown, and we have the interval timer, which we call interval.

Our Flame game class also keeps track of the elapsed ticks for the interval timer.

So let's have a look at what we do in the on load.

So in the on load method, we initialize each one of the timers.

So as you can see, the countdown timer is initialized with an interval of five.

And we have also injected an on tick notification method, which all it will do is it will print when

there is a tick, when we get that notification that the interval has passed.

And notice also that we have a condition which is going to check whether our countdown timer has finished

and it will print.

Also, countdown timer is done.

Now, the way a countdown timer works is when you create an interval like 5 seconds, then there will

only be one tick and it will be at the end of that interval.

So the print countdown timer tick and the print countdown timer is done.

They should happen basically right after each other one time.

Please notice also that we set the countdown timers auto start to false.

And the reason for that is that we want to start the timer ourselves when we click on the screen.

Now the interval timer, very simple, very similar initialization.

What we're doing here is we're saying basically that we would like our timer to have the tick notification

happening every one second in the on tick method.

So in the notification, what we're doing here is we're just counting the elapsed ticks.

We're simply keep on adding one for each tick that actually happened.

And notice that we set the repeat to be true.

So that means that this interval timer is going to be repeating.

So this is an infinite timer, as you can see in the tap down, we start the countdown timer and the

update method we put both timers in because if we don't put the timers in the update method calling

their own update methods, then what would happen is we would not get any kind of notification.

And then in the render, all we do is we simply render the text that you see on the screen.

There you can see that we specify a countdown.

We're looking at Countdown's current value.

So basically that will give you the current elapsed time.

So basically it will show three significant digits and we're outputting that at location 3100.

And then for the interval, we're going to render the elapsed number of ticks.

And all we're doing basically is just looking at what is the current value of the elapsed ticks which have been updated right here in the interval timer.

That's basically all that it does.

So for example, if I click now, you will see that it restarts.

And then it stops.

And if I click again, it restarts the timer and it stops.

So it's a nice way of being able to reuse the timer, whereas as you can see, the other timer continuously keeps on updating.

So this just keeps on going forward.

Now when do you use the one or the other timer you can really think of?

The countdown timer as nothing more than an interval timer with just one tick.

So.

So you just have to ask yourself a question of what it is that you want to do.

If you want to periodically, for example, show something on the screen, like maybe generate asteroids, like in the asteroids game where maybe every five or 6 seconds an asteroid is created, then you would just use an interval timer that keeps them going.

On the other hand, if you wanted to get some sort of a countdown, like let's say the user has only 30 seconds to play something, then you can set up a counter timer that at the end of the 30 seconds it will tell you your time is done.

All right.

So one of the things that we can do here is, for example, notice that I can change when you look at the interval timer, if I change this into a smaller interval, like, let's say 0.25, which is quarter of a second, you will see that this right here is going to be moving much faster.

So as you can see, it's kind of rushing right now and we can even go further.

We can even go like every 10th of a second.

So you will see that it's going to run much quicker.

And we can even go to an extreme of every 100th of a second.

As you can see, this is running pretty quickly.

Of course, most of the time you will not need a timer to be doing something so quickly.

You have to give the task that it's actually trying to accomplish.

You have to give it time.

So usually your interval is going to be a second or more than a second.

All right.

So in the next code version, we're going to look at how to accomplish the same thing.

But using a timer component instead of just using a timer utility class.

In order for you to use the timer, you have to include the flame package.

Timer, dot dart, that's where the timer utility class and component class is.

3. Coding with Timer Components

All right.

What we're going to do in this lesson is we're going to convert our utility timer example to a timer component.

The first thing that we're going to do, we can remove the timer here.

And what we can do is we can add the components dot dart, which is where the timer component is.

And what we're going to do next is we're going to change this timer here to a timer component.

So now we have two timer components.

We're going to leave this as is.

Note that both of these are late because we're going to be instantiating them later.

They're currently not initialized.

And what we're going to do here is we're going to initialize them as actual timer components.

So I'm just going to grab the timer component.

And in here what we're going to do is in the component, this is called period.

And now one of the things to note is that if you look into the timer component code, actually if I go right here and I open the actual source code, you can see that what the timer component uses is it actually uses timer.

So we can reuse some of the code from before.

So in here what we're going to do is we're going to still check the finished, but we need to actually access the inner timer.

So this is good and that's basically it.

Everything else is exactly the same.

Now let's fix up this.

So this was the countdown component that we fixed.

So now what we're going to do is we're going to fix the interval one and the same idea we're going to put period.

So we have created these two timer components right now.

So one thing that we can do is we can remove the stuff from update here that will not be necessary.

We're going to right away take this interval and we're going to add it inside the on load method.

So we're going to go here and we're just going to say add interval.

What this does is this automatically adds and starts the interval timer.

Now what I'm going to do is I'm just going to create a little helper method to help us create the actual timer component for Countdown Timer.

And I'm just going to call it create countdown timer.

It will recreate the timer each time we're done with it.

So I'm just going to grab this and replicate it down and we're going to initialize it right now.

So we're going to actually grab this whole thing.

And we're going to return this.

Now, what we're going to do is, as you can see, the auto start right now is false like we had before, and that's because we were controlling the actual start when we tapped on the screen.

We're going to still do the same thing, but we're going to do a little trick here.

We're going to actually set the auto start to be true.

So it will start the moment we add it to the component tree.

So we're going to be still controlling this by actually when we're going to add it to the game loop.

So what we're going to do right now is we're going to actually initialize our countdown component.

We're going to initialize it.

So we're going to keep this here, but we're going to actually initialize instance of our countdown timer.

And what we're going to do is we're simply going to set Countdown to the Method, which is going to create the timer for us.

And as you can see, the name of the actual method is creating Countdown Timer.

So this will now create an instance.

So what we're doing basically here is that in the on load, we're going to create the countdown timer and we're going to create the interval timer, the countdown timer, because we wanted to be able to restart each time.

What we would like to do is basically be able to control that by recreating the timer each time we need it.

And then in the top down, instead of doing this, we're going to again create an instance of the timer.

So basically going to repeat this and we're going to add it to the game loop.

Now, one thing about the render is that because our countdown right now is a component, we need to access the inner timer if we want to repeat exactly the same functionality.

What's going to happen right now is that when we start this, we're going to have the interval timer start running and our interval timer is set here, zero one.

So I'm just going to make it, let's say, every 10th of a second.

So the interval timer is going to start.

And then when we click, the countdown timer is going to start.

But this will create some problems.

And we're going to show you right now what exactly it is.

So what I'm going to do in the actual render method, I'm going to output the number of children.

Okay.

So let's let's run this right now.

It always takes a little bit of time to get it running.

The reason why we're outputting the number of children is because we're adding the timer components as children of our flame game class.

And it would be nice to see as we're tapping on the screen what happens, how many children, whether the actual component tree of children is growing.

So we're going to kind of this is not really debugging.

So as you can see, the code started.

We have the elapsed ticks going at the rate that we specified, which was this one right here, zero one So basically every 10 seconds.

And now if I click, you can see that the other timer starts.

And you can also see right now that the moment I did that, the number of children is two and it ended right now.

So this is good.

So now what I'm going to do is I'm going to click one more time.

So it's running again.

But now you can see that we have three children, which means that the old timer, the first one that was running this is still sitting in memory.

So now I'm going to click one more time and watch what happens again.

The timer is running and now we have four children.

So this is a problem because if for whatever reason I would like to have a number of timers, it would be nice that once the timer is finished we can easily get rid of it.

And one of the ways that we could do this, so we could go into the children component tree and actually remove that timer as long as we can identify it.

But there is a much simpler way of doing this because the flame engine architects have already anticipated that.

So what are we going to do is when we're creating the countdown timer component, there is a special flag that we can use, which is remove on finish.

So what we're going to do is I'm going to basically set this parameter.

We're going to set it to true.

Okay, so now that I have this, I'm going to rerun this.

So as you can see, the number of children is one, because in the load method we have added the interval, which is what you see is running right now.

So now what I'm going to do is I'm going to click and as you can see, the other time is running and now we have two children.

But as it ended, you just saw that it became one child.

So this flag basically specified to the game engine that the moment this timer is done, remove it from memory.

So this is a nice kind of way of safekeeping, our memory.

So this is a pretty straightforward lesson.

We're basically just looking at how can we write the same thing that we did before using the utility timer class by actually using a component.

And as you can see, the logic is just a little bit different.

But the nice thing about the timer component is that it's so much more object oriented.

We can actually start adding things inside the timer.

Whereas with the utility class, obviously there's not much that you could do with it unless you wrap it yourself.

All right.

So have a look at this code, play around with it, and I'm going to see you in the next lecture.

4. Case Study #5 - Exercises

All right, So what kind of exercises would we like you to do?

Exercise one code, a timer to create a ball on the screen as follows.

And you can reuse the code from collision detection lecture, create a ball in a random position with a random velocity every second for 100 seconds.

Each ball has a lifespan of ten collisions.

So basically what this means is that each time it hits another ball or the wall, it loses one life.

So the ball loses one life and rebounds in the opposite direction at each collision.

So if the ball starts and it goes up and hits the wall at the top, you will subtract one from its life and you will reflect it.

You will bounce it so that it basically bounces away from the wall.

Once the ball has ten collisions, it should disappear, basically be removed from the screen.

Here is exercise two.

We would like you to improve on Exercise one as follows Add a text component above each ball with the number of lives left.

It should start with ten.

So this is almost like the life bar that we have created a couple of lectures before.

The idea is that you would like for the ball and the text above it to travel together, so you basically would have to use compatibility.

But the idea is basically that you could be looking at all the balls bouncing around and you will be able to see how much life is left for each ball.

Also, add a bounce sound for the ball collision.

It should be subtle, nothing too loud or too harsh.

You can look for sounds at this website.

There are three.

All right.

So have fun with this and I'll see you at the next lecture.

5. Case Study #5 - Exercise Solutions - Solution to Exercise #1

Good to have you back.

Welcome back to Solution to Case study number five, Exercise number one.

This particular exercise is really where you take pieces from the other exercises and you put it together.

Specifically, we're using the solution from case study number four.

The idea here was that you were supposed to code a timer to create a ball on the screen so that the

ball would show up in a random position with random velocity every second for 100 seconds.

And then you're going to simulate a lifespan of ten collisions for each ball.

So in other words, if a ball hits either another ball or if it hits a wall.

So basically every time that it bounces back, you reduce its lifespan by 10%.

And once the ball loses all its life, you remove it completely from the screen.

Let me first just run the sample that we have here.

Notice that the number of simulation objects that we have is ten instead of 100.

I've made it just a little shorter so that we can quickly see how it works.

So I'm going to run it.

All right.

So now we start.

We have two, three, four bowls.

As you can see, everything is starting.

But notice that some bowls are disappearing because they have already hit something on their way ten times.

And you can see that we have the number of ticks, we have number of objects alive.

Now we're just left with two.

And these will also die off when they have bounced around ten times.

There you go.

This is it.

So now let's just quickly have a look at what we did here.

The main thing really is that we've created a timer.

It's an interval timer.

And then in the on load method, we create the timer and then we create the on tick method, which is going to be notified every one second.

You can see that the interval that we have is one second and then our on tick method is pretty straightforward.

We generate a random position for the ball, we get the random velocity and we get the random speed.

Then what we do is we generate the ball with that random position and the random velocity and then we add the ball to the game.

We also up the number of ticks and then once the number of ticks has gone past, in our case ten, we stop the timer.

So this is pretty straightforward.

And the update method, we add the timer and then we just render some information onto the screen.

That's basically it.

Our collectable class has changed just a little bit.

Notice that we have the object life value which keeps track of how many times the object can be hit before it goes out of commission.

Notice also that we have created a constructor where we initialize it with position, velocity and speed, and then all that we do is basically, as we have done before in exercise Number four.

Notice one important thing that we have inside the update method right here at the bottom, we basically say if the object life value is less than or equal to zero, we remove this object from the parent.

This is quite important because we're not removing something from this object.

We're removing this object from the parent.

The parent is where if you go back here, the parent is where we've added the ball, which is right here where we have the odd ball.

The parent is actually our flame game, which is the timer example 001.

So in this example right here, what we're doing is we're removing this from the parent, and this is basically it.

And of course, as usual, we have provided you with a solution in a PDF.

You can look at that.

It's pretty straightforward.

So please do have a look at that and have fun with this code.

And I'm going to see you in exercise number two, which is a little bit more ambitious.

All right.

As always, keep on coding.

6. Case Study #5 - Exercise Solutions - Solution to Exercise #2

Welcome back.

Here we're going to look at solution two, case study number five, exercise.

Number two.

The exercise asks you to improve on exercise number one as follows It asks you to add a text component above each bowl with the number of lives left, and it should start with ten, meaning ten lives.

It also asks you to add a sound for the bowl collision, and we wanted it to be subtle so that it's

not overpowering.

So we're going to run the code.

There it is.

And as you can see, not only do we have the sound of bouncing, but we also have the text that goes above each of the balls.

We're going to look at this a little bit more in detail in a bit.

And so what I'm going to do then is I'm going to slow it down.

So it's a little bit easier.

But you can see, for example, this ball at the bottom right here.

That just disappeared.

You could see the kind of text that it had.

So let's have a look at always the handy and helpful PDF document.

All right.

So let's have a look at exercise number two in exercise number two.

We have made a number of assumptions.

We've decided to create a separate class extending the text component.

That's the part that you see above the ball.

And we also decided that the data that we're going to put in a text will have two elements.

It will have the ordinal number of the ball.

It will tell you the order of the balls as they're being created.

So, for example, the first ball will have one, the second ball will have two, etc. That's the ordinal number.

And of course the second part of the text is going to be the help of the ball in this exercise.

We'll also use the timer component instead of the timer utility that we had in exercise number one.

And there is one subtle issue when you're dealing with being able to do a sound when two balls collide.

As you remember, whenever you have a collision between two objects, you get two notifications because these two notifications happen back to back.

If you actually produce the sound for each of the notifications, you would actually create two collision sounds.

So we have to take care of that problem as well.

So let's have a look here.

So first of all, as you can see here, we're going to have a specially formatted text and we will take

two parts.

As mentioned, the first one is going to be the ordinal number.

Then we're going to have a dash and then we're going to have the percentage of life left In order to properly format the data.

We're going to need a package that will have to put into the pub spectral YAML, which is going to be the international or the intel with the latest value shown up there in the document.

This is how we format the numbers.

And then here is the full code for the actual text component, which we call simply life bar text.

So the full solution is given here.

It also explains how you're going to ensure that there is only one bounce sound for the two notifications.

The way we actually do it is very simple.

What we do is we combine the two hash IDs of each one of the bowls that are colliding.

We sort them so that the key itself is always the same, whether it's object one colliding with object two or whether it's object two colliding with object one.

That way these two notifications will produce exactly the same key.

So all that we do is when we have the first notification of the collision, we combine the two hash codes of the two objects, which are the balls, and we put this into a set which is like a map, but a set can only have one item for the particular key.

So what happens is that when the second notification happens, we create the key and then we check whether we have it already in the set.

If we do, we don't create the sound because we know by the virtue of the fact that we already have that key in the set, that that notification, the first notification already happened.

So we obviously already had the bounce sound.

So that way we don't execute the balanced sound the second time around.

All right.

So let's have a look at this a little bit more.

So first of all, our life bar has been put into its own class.

And as you can see, it extends the text component.

And then all that we do here is you can see at the top that we are importing the international package, which is a DART package.

And you can see right here in pub spec dot YAML that we have it right there in total 0.17.0.

So once you have added that to your pub spec ammo at that point in your life bar, you simply import it.

And in here all that we do is we create a text paint where we define the colour of our text and the size of the font.

That's the style.

And then what we do is we create two formats.

They're basically the same.

We have the ordinal formatter which is going to format the ordinal number and we have the health data formatter, which is going to format the health data.

And you can see that the number format is 000, and that's simply means that it will ensure that even though we have, for example, a small number like one, it will still put the two zeros in front of it.

So we'll always have the same length text and we've provided you with the URL to where the API is so you can have a look at it.

So the.

The two data elements that are inside the text component is going to be ordinal number and health data.

We have a default constructor which is going to take just the ordinal number because the ordinal number will not change once you initialize the text with it because the bowl is either the first or second or third.

But what's going to change is going to be health data, which is why health data is public and ordinal number is private.

Once you initialize the ordinal number through the constructor, you can't access it anymore.

Notice that in the on load method, what we do is we create a text render, which is in our case simply our text paint.

And then everything really happens in the update method.

What the update method does is it recreates the text by formatting the ordinal number using the ordinal formatter, and then it also formats the health data using the health data formatter.

And that's basically it.

You can see that what we're formatting here is exactly what we have specified.

Right here in the document.

This is this is the format itself right here.

All right.

So now that we have our life bar text in Michael readable, all that we really do here that is a little different than what we did before is we make the object live value to be 100 instead of a ten, because we're going to treat it as a percentage.

But notice also that we're adding the life bar text as a composite element of the collider class, which is the ball.

So what we're doing is we're using flame engine's composite ability by adding the text to the actual microlight object, which means that automatically with the ball, the text is going to travel.

We just have to obviously place it properly.

So notice that inside the constructor, what we're doing is we're creating the life bar component.

We initialize it with the ordinal number and we also specify it's X and Y and the x and Y is simply with reference to the upper left corner of the rectangle that bounds our microlight class.

So this will put it nicely above the ball itself.

And then in the unload method we pre load the ball bounce of ball sound effect so that way it will be available relatively quickly whenever we need it.

So this is, this is what you see here is we're pre loading it and of course notice that we're adding the life bar to our microlight table components.

So this is where the composition happens and then everything else here is exactly the same.

The only difference, of course, is that in on collision, when you look at it, we're doing something a little different than what we're doing before.

Notice that when the collision happens, obviously we see that we're colliding with my collider ball.

So the other is my collider.

So that's good.

Then this part right here from 141 down, this is what we've seen already before.

This is exactly the same as it was before.

This is where we're dealing with just the collision itself.

But when it comes to the sound, what we do is the following.

Notice that we get the two hash codes, we have the hash code and the other hash code, and we're basically

this is just a quick, simple way of sorting it.

Since we have only two values, then it's either the first hash code is greater than the other hash code or vice versa.

And then what we do is we create a collision key by combining the two hash codes.

And then what we do is we have a set which is observed collisions, and we basically check whether this observed collision set already contains this key.

If it already contains the key, this means that the collision has already happened before, so we can remove it from the set.

But if the collision key is not inside the set, then what we do is we add it to our set and then we generate the sound.

And this actually should be our set, not our map.

And if we go back to where the definition of this is, this is, as you can see, it's game ref.

So this is actually inside our main dart.

So we'll go back here and you can see that right here we have the set.

This is the observed collision set.

It's a global set and this is to a great extent all the code that you need to take care of the whole exercise.

One other thing, though, that we've decided to do as well is we have in Exercise one, we're using a timer utility here.

We have converted our code to a timer component.

So as you can see here, we have a timer component and then inside the on load method, what we do is we initialize the timer component with a period of one second, so it will basically tick every second we have the removal finish.

True, which really doesn't do anything.

Unfortunately, we'll have to ourselves manually remove the component when we're done.

We say auto start true, meaning that the moment we add the component to our parent, it will start ticking.

And then on tick method is basically exactly as what we had before with one slight difference.

And the difference is that once the number of ticks is greater than the number of simulation ticks.

So for example, if we want to generate only ten objects, so obviously the moment we have 11, what we do is we stop the timer and then we remove the timer from the component tree.

This is important.

Otherwise it will still sit in memory and this is basically it.

This is all that there is to it.

So what I'm going to do right now is I'm going to run this code one more time, but I'm going to change the number of simulation objects to ten.

So it's a little bit less, but also.

What I'm going to do is right here.

When we set the min and max range for speed, this is a little too fast, so I'm going to go from a slow like a 20 to 100.

That way we'll be able to see a little bit better how the balls are moving.

Otherwise it's a little too fast and I think this is pretty good.

We're going to run it right now.

And there you have it.

You can see that it's pretty much slower right now.

It's a little easier to see

there bouncing around.

And you can see that when you look at the string,

when you look at the actual string, you can see clearly what the data is.

For example, this one going into the corner is ten and it's only 30%.

It's going up right now and want it bounce.

It's 20%.

It's going to bounce again.

It's going to be 10% and now it's going to disappear.

And then when you look at the 0.08, it's a 30%.

It's going to bounce off.

Right now, that becomes 20%.

It's going to go up and hit the top of our screen.

It's going to become 10%.

And now the moment it hits, it will disappear because it has run out of its life.

So this is pretty, pretty straightforward.

And we're done.

All right.

Please go through the code, Try to understand exactly what was done here.

And I'm going to see you in the next lecture or in the next exercise.

And as always, keep on coding.

10_case-study-6-parallax

1. What is Parralax?

All right.

Welcome back.

In this case study, we will look at what parallax effect is and how to create it in the flame engine.

The purpose of parallax is to create a looping that's usually not always, but usually a looping background motion made up of two or more parallax layers.

The idea in Parallax is to move the background images past the camera more slowly than foreground images.

In this picture, you have an example of the parallax effect.

The above parallax is made up of three separate images, each moving at a different speed to create the illusion of relative motion.

Notice that the closer images are moving faster.

As an example, consider the following three simple and separate images.

There are separate layers.

We have these little clouds in the sky.

We have some sort of foreground little plants that look like cacti or cactuses.

And we also have a foreground image, which would be maybe like a path with a little bush.

The corresponding parallax animation would look as follows.

As you can see, the sky is moving very slowly.

You can see that it's moving because the clouds are moving.

Then these plants, let's call them that, that look like silos.

They're moving a little faster.

But the foreground, which is like the path with a little bit of grass in that bush is moving much faster.

And that creates a very interesting cartoon like motion effect.

Notice that each layer is moved at a different rate and we can control this rate in the flame engine.

To adjust the parallax effect, you would obviously have to have some sort of control as to how fast the different layers are moving.

And the way that we do that is by create something called a velocity multiplier for each layer.

Each layer is going to be some sort of a multiple in terms of speed of the layer behind it.

So how do we code this in flame engine?

First, we need to provide the layers for the parallax component.

These are simply images that we would like to have in our background.

Second, the order of those images determines how they are stacked.

So image and location index zero would be the furthest background and images with index greater than zero would be closer to the observer.

So then we provide the initial background speed of the furthest image.

So the one at location index zero, we then provide a velocity multiplier which would be applied to each of the next layers, the subsequent layers.

So in other words, if the layer in the background is moving at some speed, maybe like 20 pixels a second, then we would say that the images in front of it, the layers in front of it could be moving maybe 1.5 faster or twice as fast, and then we would apply that multiplier to each of the layers.

Please note, though, that the images must be large enough to at least cover the visible screen.

It would be difficult to create that kind of parallax illusion if the image itself is not large enough so that as it passes through, in order for us to create the loop, it would have to be at least as large parallax effect will usually repeat the parallax effect in an infinite loop.

So in other words, when we start the parallax effect and we have the motion of the different layers, the idea is that the parallax effect in game energies usually repeats itself in a loop.

That's why it's important to create the images in such a way so that there is no sharp cut between when the loop is coming back.

All right.

So let's look at another example.

In this case, we'll have a number of images.

First, we're going to have an image of what appears to be like a sky with the moon or perhaps the sun.

That's our background image.

That would be index zero.

Next, we have a singular mountain.

Then we have some other mountains which would be put in front of it.

Then we have some sort of low, almost like a forest.

And then we have a number of trees which appear obviously to be a little bigger.

They're going to be closer to us.

So this example is relatively sophisticated because we have five images from which we will create the

parallax.

So of course, the image in the background, which is the one with the sun moon that will be moving slowest.

And then each one of the images after that will be moving a little faster.

So what we have is we have five layers from the most background to the most foreground elements.

So let's say that we set the base velocity to be 20 pixels, 20 pixels per second in the x direction and zero in the Y direction.

This is another way of saying that we would like this to only move horizontally either left to right or right to left.

And the reason why we put zero in the Y direction is because we don't want it to be moving up and down.

We then, for example.

Set the multiplier to be 1.8 pixels in the X direction and the Y direction multiplier is irrelevant since the base velocity is zero.

And we do all this because we want the parallax to be only horizontal.

Imagine you're in a car or you're in a train and you're looking out the window.

The motion of everything will be from left to right or right to left, depending on which direction you're going.

So here is what it would look like.

This is the actual parallax effect created from these five images.

So you can see that the image in the background is moving slowly and then the image in the most foreground,

which is the larger trees, it is moving much faster.

So let's have a look at what that would look like in code.

First, we create an array of the images.

In this case, each one of the elements is something called parallax image data, which is basically an image.

And you can see that our array, the zero index is the BG, PAG, which is the big background with the sun that we saw in that parallax.

And then of course the closest to us, which is index four is the foreground trees PNG.

So in other words, we specify how to stack the images by simply creating an array with the index zero being the furthest image and with the index closest to us being the largest index.

So in this case our first image, it's index zero is the furthest image and our last image at index

four is the closest image.

Once we have created an array of parallax image data elements, all we do is we're using a load parallax, component method and we are plugging in the array.

We then specify the base velocity to be our favorite class vector two We're specifying the x direction of 20 and the Y direction is zero.

Basically, it's irrelevant.

Next, we specify the velocity multiplier delta to be simply a vector of 1.8 and one, but notice that the one is irrelevant because it will be multiplied by zero.

So there will be no movement in the vertical direction.

And that's it.

We have basically right now created a parallax component which is loaded.

The reason why we have the weight is because this is asynchronous call.

We would like the system to wait for this call and then all we do is we add this component to our game loop and at that point it will automatically start working.

All right.

So this is pretty straightforward.

The code itself is pretty straightforward.

Let's look at an example that creates the parallax horizontally, which would be something good for like maybe a space shooter.

So imagine that we have a single image what appears to be very tiny and faint stars.

And then we have another image of slightly larger stars and notice that the two are like different spots.

What we would like to do is we would like to combine these two images to create a parallax that scrolls down.

Imagine that our spaceship is moving up so we have only two layers, which is the minimum really that you could have.

What we would like to do is we would like to set the base velocity to be minus three pixels in the Y direction and zero in the X direction.

In other words, we don't want it to be moving left to right.

We simply want it to be moving down.

We then set the multiplier, for example, to be 1.8 pixels just like the one before in the Y direction and the x direction multiplier is irrelevant since the base is zero.

This is a very similar case like what we saw in the other example.

Why do we do this?

We do this because we want the parallax to be only vertical and we are moving it top to bottom, which is why we have negative velocity and this is what it would look like.

Here is a simple parallax that has been created from these two images.

And as you can see, it's a pretty subtle, simple parallax effect.

All right.

So hopefully you've enjoyed that.

That's a pretty simple and quick lecture.

We'll see you in the next lecture.

2. Coding Parallax

All right.

Welcome back.

Let's have a look at the first parallax example from the two examples that we're going to look at.

The first thing that we're going to do.

Let's have a look at the outline of this.

And you can see that the code is pretty simple.

We have a main like usual and our basic parallax example 001 extends Flame Game.

Nothing strange about that.

What we have here is first we create an array of images that we're going to stack together in our parallax.

And basically this is like a background mountain for mountain mountains, trees, and then foreground trees.

The thing is, where is this coming from?

If we go into our code here and you look at the pub spec dot YAML, you will see that right here.

We have set up our assets, assets slash images slash, which is right here.

This is where our assets are and you can see that I have all of these.

All right here there's the background.

You can expand it by clicking on it.

And if you hold the control key, you can then minimize it.

So we can have a look at that.

We have the foreground.

And one important thing to notice is that this image, those kind of little squares that you see here,

this image is actually built with a transparency mask.

And that makes sense because when you're stacking these images together, the images that you put on top of the furthest background image, they have to have a transparency mask.

Otherwise they're going to obstruct or they will make the other image invisible.

So, for example, the mountain far, you can see it's the same thing.

All of those little squares here, that's actually a transparency mask.

Same thing here and here.

Our trees, same thing here.

It's the same thing here.

These are the trees closest to us.

And as you can see, it's a pretty long image.

Actually.

Our screen would normally be maybe about half of that.

So this this gives us a nice parallax when it's looping.

Okay.

So this is pretty straightforward.

Let me get rid of these.

So what we're doing here is we're simply loading all of these images in the actual order of the parallax.

So this is index zero.

That will be the first image.

That will be the most background image.

And then this one is index four.

This will be the image closest to the viewer.

Then what we do is in the on load method, well, we just call the super.

And then what we do is we use the load parallax component method, which actually comes as part of frame

game.

So you're basically get that for free, so to speak.

So we're creating this parallax component.

And you can see that if I click here and I go in here, you can see that what it returns is a parallax component.

So we get the parallax component, we initialize it with image names, which is all the images here.

We're also creating a base velocity, which is going to be a vector.

It's a 20 and a zero.

We're only interested in the horizontal movement.

That's why we have the X, and then we specify the velocity multiplier delta to be just 1.8, and the 1.0 here is irrelevant.

You could just as well make it zero.

And then what we do is we add this component to the frame game's game loop.

So the moment this application starts, the parallax is going to start.

Now notice also that this is an asynchronous method.

That's why we have the future.

And the reason why we put a weight here is that we want the parallax to load first before anything else is done.

In other words, this method is going to kind of stop and wait until all of the parallax component elements have been loaded and only then it will be added.

So let's run this and have a look what it will do.

And as usual, takes a little bit of time.

Given all the code that you see right now, what do you think is going to be the outcome?

What is it going to look like?

Here we go.

Okay, so this is very much what we expected.

But as you've seen, when you look, for example here, the background or let's look at the foreground, you can see that this is a pretty long image.

So that means that the image extends out.

So this is actually much better viewed in a landscape mode, not a portrait, but we just get rid of this.

But if I just flip the phone, it doesn't do anything right.

All we did is just like, you're moving the phone actually.

So I have it upside down.

Let me just.

There you go.

It's still upside down.

There you go.

So what do we need to do right now?

Let me just stop this.

What do we need to do right now is we need to actually tell our game that it should be displaying itself as a landscape.

And also, one other thing that we're going to do right now is we're just going to remove this bar at the top because it doesn't serve any good purpose.

So what are we going to do is we're going to go here into Main and we're going to get the widgets flutter binding static class, and we're looking for the ensure initialized method.

There it is.

What this does, as you remember probably from the introduction, is that it initializes the binding of the widgets for Flutter and see right here.

Okay, so now that we have bound that what we're going to do is we're going to now instruct Flame to first of all, go full screen and then set landscape.

So we're going to go here.

We're going to now use the flame utilities.

But as you can see, flame is not recognized.

Well, we'll remedy that right now.

A quick fix.

We're just going to add the necessary library.

So this is done.

And then we're also going to now say that we would like to have the device be in landscape.

Okay with this being done, Let's run it so this will lock our game into landscape mode.

As usual, we have to wait a little bit for it to run.

And as you can see, it's running in landscape right now.

There you go.

So now we have the actual landscape mode parallax.

And you could imagine that this could be some sort of a flying game where you have to, like, shoot things from the sky.

So this would be like a nice little background when you think about it.

The pretty famous Google Chrome game with the dinosaur jumping over the cacti or cactuses in the desert

basically has something pretty much like this running.

All right.

So this is pretty much it for this example to stop this.

Now, one thing that I would like you to do is just play around with it.

Just see what it would be like if we actually changed this vector to, let's say, let's make it just for the heck of it.

15.

And we're going to actually multiply this one one and let's have a look.

I mean, this will be just bizarre because it will actually try to move the parallax effect both in landscape, which is left to right as well as up and down.

So this will actually completely probably tear apart that image.

But let's have just the look.

So you have to kind of like you have an understanding of what's going on.

All right.

Almost done.

Brace yourself.

This will be strange.

So as you can see, this is complete madness right now because what is happening is the images are just being moved and torn apart up.

And so this is one of the reasons why we don't do that.

So I'm just going to revert that back.

This will just be zero.

And this this could just as well be zero.

I mean, it doesn't really matter.

So we're just going to restart it.

And now.

Now everything is normal with the world.

Okay, so this is it for this quick little lesson.

Play around with it a little bit and I'm going to see you in the next lecture.

Take care.

3. Coding more Parallax

All right.

Welcome back.

This time we're going to look at the second example of parallax.

This is even a simpler piece of code.

We only have two images, small stars and what I call big stars.

If we go into assets and we have a look at the big stars, you can see that we have an image that has a transparency mask.

We have small stars, which is again, also an image that has a transparency mask.

And as you can see, there isn't really a lot of stars that have been put in.

This is really just a very individual choice.

You can play around with it and see what you like better.

We've also included two more star images for you so that you can practice a little bit.

Now, one more thing that I wanted to look at is you can see that the size itself here seems to be close to the size that you would expect a phone to be.

And this was done on purpose because we want to demonstrate something here.

So let's go back into the code.

What we're doing here in the Unload method is we're using the load parallax component function.

We're putting the image names.

Notice that base velocity is zero in the x direction, which means that we're not going to be moving left or right and it's negative five in the right direction, which means that we're going to be scrolling down, we're going to be going away from the top and multiplier data.

This 1.0 is not really important because given that zero in the base velocity nullifies this, we've also included the multiplier to be 1.8 so that the layer in front of the background layer is going to be moving almost twice the speed.

All right.

And then we added to the game loop.

So let's run this and let's have a look of what's going to happen.

Okay?

So here it is running.

So as you can see right now, we have the parallax happening.

Almost looks like falling snow, but something curious is happening.

Notice that as it's coming down at the top here, it's not repeating the loop doesn't seem to be happening.

So what we're really getting is an effect of the two images just slowly sliding down, but they're not

being repeated.

Now, the way that you deal with this is that there is one more attribute to load parallax component that we should learn about, and that attribute is repeat and image.

Repeat gives us a number of capabilities.

We're just going to basically do a repeat here.

I'm going to rerun this.

What this does is it tells the engine explicitly to keep on looping the image itself.

So you can see now that as this is falling, it continues to be almost as if it was like on some sort of a conveyor belt.

It keeps on coming around.

So this would give us, again, kind of like an infinite parallax.

And this is really about it for this one.

This is just even a simpler example than the one we did at the beginning.

And I suggest that you experiment with the other images that we have provided and experiment with Speed of the Parallax.

For example, make it even slower if we go like to negative three, for example.

All right, So have a look at the exercises and I'm going to see you in the next lecture.

4. Case Study #6 - Exercises

Here's a couple exercises for you.

So, first of all, how would you reverse the Mountain Parallax from the use case examples to move in the opposite direction.

Right to left.

Next, we would like you to do a little bit of research to see how could you create a parallax effect where the velocity parameters for each layer are absolute and not based on a velocity multiplier?

In other words, where each layer has its own specific velocity, that gives you much more flexibility because you could possibly slow down some of the stuff happening in the background.

So that might be an interesting technique to have under your belt.

And finally, the most exciting one is add the parallax effect with the stars to the code we used in the joystick case study number two, so that the spaceship that we have in that case study is moving with the parallax effect in the background.

It will look probably a little bit more exciting.

All right.

Well, give it a try and we'll see you in the next lecture.

5. Case Study #6 - Exercise Solutions - Solution to Exercises #1 and #2

Welcome back.

Let's have a look at case study number six, specifically, exercise one and exercise two.

Exercise three is going to be shown in a separate video.

What we want it to do in exercise one was just to reverse the mountain parallax so that it goes in the opposite direction.

And as you can see here, the solution is pretty trivial.

All we do is we switch the x portion of our vector from 20 to -20.

So basically from going to the left to going to the right.

And as you can see here, I have the parallax running with the original 20.

So what I'm going to do right now is I'm going to simply reverse this by going -20.

That is all that it takes.

As you can see, it is going right now to the left if I rerun this.

Now it's going to the right.

That's all that basically we needed to do.

But now let's look at exercise number two, which asks you to get a little bit more into the guts of the parallax effect by being able to control the multiplier delta on its own for each specific layer in this code, as you see, what we do is we load all of the images and then we apply exactly the same velocity multiplier, which means that when you look here, the background, which is the clouds and the sun, they're moving at a certain speed, they're moving basically at the base velocity.

Then the next mountain that is closest to that background moves at 1.8 that speed than the mountains closer to us move 1.8 times the previous layer than the trees just before those mountains move at 1.8 multiplied by that particular layer.

And then finally the trees at the very front, the ones that go fastest, are moving at 1.8 times the previous layer.

So this works pretty well.

But what if you wanted to really control it a little bit more?

Let's have a look at that.

All right.

So let's have a look at this.

It seems to be working pretty much the same as before.

The only difference seems to be that the trees in front are moving much faster than before.

If you actually look at this more carefully, you will see that compared to the previous run that we had, the different layers are moving at different speeds rather than a multiple of 1.8 as we had before.

And there is a reason for that in this particular piece of code, which is a solution to case study number six, exercise number two, we actually control all the layers on their own.

So as you see first here in the layers meta, we have a map where we put all the names of our assets, which is the bag dropping, for example, which is the background, the mountain dash for the PAG, which is the furthest mountain, the lighter one, etc. This is a map.

So the actual key to the map is the name of the asset, which is the name of the image that we use for the layer.

The actual velocity multiplier is the value in this map.

So for example, for BG, PNG, we specify the multiplier to be 1.0, basically no change from the base value and then the mountain that is furthest we specify that we would like the multiplier to be 1.5 before it was 1.8.

Then for the mountains dot pag, we decided that rather than having a multiple of 1.8 or some other value, we want specifically 2.3.

Then the next trees, we put a 3.8 and then we make the foreground trees move much faster.

But look at what happens.

Now.

Let's say that I do something like a 50.

Watch what happens.

As you can see, they're moving very quickly by compared to everything else.

So obviously in the previous code that we had where there was just one velocity multiplier, we couldn't really do this.

So this simply gives you just more flexibility, more power over what we had before.

So to go back to the code, what you're seeing here is we set up the map of the data and then what we do is in the on load method, we're actually creating a map which is made of load parallax layers.

Each element in that map is a layer and notice that the image data is loaded based on the key from our map and the velocity multiplier is based on the value from our amount, and that is basically it.

At that point, we simply just create the parallax effect itself by loading the parallax component,

and that's all there is to it.

Let's just do one more little change here.

I'm going to change this back to 24, but I'm going to make this, for example, 5.0.

Let's see how that affects it.

I'm just going to run it, going to have a look here.

And as you can see, it's a little different now.

So it really gives you great, great capabilities.

All right.

That's basically it.

Have a look, as usual, through the document that we have provided you in the document, we tell you everything about size one that you need, which is pretty straightforward and trivial.

And then we also talk about exercise two and give you the specific code snippet that you would need.

All right.

As always, thank you for listening and watching this video and we'll see you in the next lecture or exercise.

And as always, keep on coding.

Take care.

6. Case Study #6 - Exercise Solutions - Solution to Exercise #3

Here we're looking at solution to case study number six, exercise number three.

So what you're seeing here is very simple star parallax that is in the background of our spaceship.

The stars are moving.

I can I can shoot things.

Everything works exactly the same as in the joystick example.

But now we have a parallax.

So what have we done here?

Pretty straightforward.

We simply merged the code from case study number two.

And the code from case study number six.

As you can see here in the assets right here, we have added the necessary images.

We're using just the small stars and the big stars.

And right here we have added them to the image names and then we specify the parallax component.

And then all that we do is in the on load, we simply load the parallax.

We specify a speed which is 25.

We put the negative of that so that it goes in the opposite direction.

So let's say if I'm moving up, it goes in the opposite direction of the spaceship.

So this is pretty straightforward.

The velocity multiplier delta is 1.0 and 1.8.

The 1.0 is for the x element, which as you can see in the base velocity, it's zero.

So basically we're just moving the stars in the vertical direction and in our case we're moving it downwards

and the multiplier is 1.8.

So the stars that are closer to us are flying 1.8, the speed of the stars that are behind.

Now, as you have seen, hopefully in the PDF, if we go to exercise number three, notice right here

in 0.2, we say one variation that is worth considering is to change the orientation of the parallax

based on the orientation of the spaceship.

So what that basically means is that if the spaceship is going up, the parallax is going to go in the

opposite direction.

If the spaceship is moving to the right, it will go to the left, etc. So that will give you that illusion

that as you're moving, the stars are kind of moving against the background, which is what you would

expect if you were looking out the window of, let's say, a spaceship and you're moving on all different

directions.

So let's have a look how we would do that.

So all we really have to do here.

Is we already have the code here.

So what we're going to do is we're going to in our update method, we're going to have a velocity vector.

And we're going to use this velocity vector as just a simple vector pointing straight up.

Then we're going to take the angle of our player, which is the spaceship, and we're going to rotate

it to obtain the actual directional vector.

So what we're going to do is we're going to do that.

I'm just going to change this.

And all that we're going to do here is I'm just going to grab this velocity, I'm going to put it right

here and I'm going to run it right now.

So as you can see right now, everything is still.

All that we're doing is we're just using the directional vector.

Notice that because the vector is basically a unit vector, this is very, very slow.

And as you can see, there's very, very slow movement.

So what we really need to do is we need to multiply this by some sort of speed magnification.

So what are we going to do is we're going to grab this parallax speed.

And right in here.

We're just going to multiply this by the parallax speed.

Now, when we run this, as you can see, the stars are moving faster.

That's pretty reasonable.

And now if I move down, notice that the stars are moving up.

If I go right, the stars are moving left.

If I go left, the stars are moving right.

And if I go at a diagonal, the stars are moving in the opposite direction, in diagonal.

So this is pretty nice.

These stars are always moving with the same speed.

So there's one more variation that we mentioned in the PDF.

And as you can see right here, in addition, this is 0.3.

In addition, another variation would be to tie in the speed of the parallax to the speed of the spaceship.

So let's have a look how we could achieve that.

So what we could do here is rather than doing all this, we're just going to go back to the code I had before.

We can simply use player current velocity.

But the question is what is current velocity?

If we go into the player, we have added a variable which is a public variable and it's called current velocity.

Initially we set it to zero because there is no current velocity, there's nothing happening.

But then in the update method of the joystick player, which is our spaceship.

What we do is we put the current velocity to be equal to the joystick dot relative delta.

And as you can see, if I hover here, relative delta is the percentage presented as a vector and direction that the knob is currently pulled from its base position to the edge of the joystick.

Well, translation, what that means is that this relative delta gives us a vector which gives us the percentage.

For example, if the joystick is pulled all the way up, it's going to give us two pieces of information.
It's going to give us 100% of the size of the vector, plus it's going to give us the directionality.
In other words, where is the joystick pointing?
What we're doing is we're basically multiplying this vector by our max speed.
So in other words, if the joystick is pulled all the way to the top, for example, if I, if I basically do this, if I pulled it all the way to the top, that means that our relative delta will be 100%, which would give us basically a unit vector in whatever direction it was moving.
If we were, for example, going about halfway like this, then it would give us not a unit vector, but about half of the unit vector.
So this is really a percentage based on a unit vector.
That's what relative delta is.
And then when we multiply it by max speed, our max speed happens to be 300.
You can see it at the top.
What we're really doing.
This current velocity is going to give us a velocity between zero and 300.
This will be the speed and it will also give us the directionality vector.
So when we go back here to our main dart, all that we're doing here is we're basically saying let the base velocity of our parallax be the current velocity of the joystick.
So basically that's what we wanted.
Now it's a curious case when you think about it, as to why this actually doesn't have to be reversed, because we don't have to reverse it.
Notice that if I run this right now, so at the beginning, nothing is happening.
There's absolutely no movement.
And the reason why there is absolutely no movement is because we have set the current velocity inside the joystick player.
As you can see, we have set it to zero.
So initially nothing is moving.
This is perfect.
But if I move it right now, notice that the stars are going in the opposite direction, which is what we wanted.
We don't have to actually reverse it.
I'll leave that for you to kind of figure out based on the code as to why that is happening.

But as you can see, this works pretty well because if I go slowly, the parallax is slow.

If I move faster, the parallax goes faster.

It's it's a very interesting effect.

All right.

So hopefully you enjoyed this and I'm going to see you in the next lecture or exercise.

And as always, keep on coding.

11_case-study-7-particle-system

1. Introduction to Particle System in Flame

All right.

Welcome back.

Today, we're going to be looking at the particle system, which is basically particle physics.

Now, this is a pretty complex topic, so we're going to break it down into a number of sub lectures so that you can digest each piece of information on its own before you move to the next one.

Flame Engine provides a robust and extendable particle system.

The purpose of particle systems is to simulate different types of behaviors with many particles acting together.

Each particle would typically have a limited and predetermined lifespan, usually in seconds have a velocity vector determining the direction and distance that the particle moves.

With each frame.

Update you could potentially use particle systems to simulate moving liquids, smoke clouds, flames, magic spells, explosions, fireworks, and a host of other effects where particles are acted upon by such forces as wind or gravity.

So if you look at those two points, having a limited and predetermined lifespan and then having a velocity vector, what you're basically saying is that when you define a particle, you're going to say, I would like this particle to live maybe 2 seconds or 5 seconds or 20 seconds.

Think of, for example, fireworks or an explosion.

Usually an explosion.

The particles disappear after a while.

So this would be pretty good for that.

And of course, with an explosion, you would need to have a well defined path that the particle is going to take, which would be, for example, a velocity vector.

Here's an example of a very simple particle simulation that looks like fireworks.

At the very heart of the particle system is a particle.

What is it?

A particle in this context is any entity that can be assigned a velocity vector and then subjected to

different forces such as acceleration or deceleration.

So the important distinction about this is that a particle is something where we can define where it's going.

Like think of throwing a ball into the air.

When you throw a ball into the air like above your head, it has a certain speed that you're starting with.

But then the ball itself is pulled down by Earth's gravity, so it changes its actual velocity at some point.

A particle in our context is exactly like that ball that we throw in the air.

We have a particle that has some sort of a velocity vector.

In other words, it's going somewhere and then we might put forces to act on it.

That will change its, let's say, trajectory.

When we say any entity, what do we mean?

Well, for example, we could assign to a particle, a sprite image, and then manipulate the velocity and forces acting on those sprite image particles, which in fact would act on those sprite images.

As an example, consider the following explosion, which is a set of sprite images animated using the particle system.

As you can see, it looks like a pretty much real explosion.

So how do we use the particle system?

A particle has a number of main features.

One is how it renders.

In other words, what is it that you see on the screen?

It could be a red circle.

Next we have its position.

Where is it at a given point in time?

Where is it right now?

It also has its speed.

How fast is it moving?

It also has acceleration, which is the vector that changes its speed and position.

And of course, acceleration could be positive or negative.

So if it's a negative acceleration, it would mean deceleration.

Something is slowing down.

Also, we have its lifespan, which is how long does the particle actually exist in the system?

A particle system is a collection of such particles.

So usually we would specify how many particles we want to be part of the simulation.

So we start with an individual particle.

We can define it as a little red circle with a two pixel radius, and then we could specify where it starts.

We could specify how fast it moves.

We can specify with the acceleration it's going to be.

We could give it a life span of 4 seconds and then we could combine many such particles and perhaps randomly assign the acceleration and the speed to create an interesting simulation of many particles acting together, which is what we're going to see in a lot of the code that we're going to write.

But the idea is that when we're talking about a particle system, we're talking about a collection of particles.

So what is even more interesting is that you can create a composition of different collections of particles under one simulation to create very organic looking particle visuals.

If you go back a couple of slides, you will see that there is an example of what looks like fireworks.

That is actually two different types of particles.

Once that define the initial explosion of the firework or the initial start of the firework where all the pieces are kind of moving, and then at the end where we have that kind of flare up, it almost looks like it's dying out.

That's a different set of particles.

But we keep them together to create one simulation.

So as we stated, all particles accept a lifespan argument.

The idea is that once such a particle has reached the end of its life, this value makes the particle component remove itself from the game loop.

So this is a very tidy architecture.

Here are some examples of how we could create a lifespan.

So we're creating a particle with a lifespan of 0.2 seconds, which is 200 milliseconds.

And the other example is we're showing lifespan of four, which is 4 seconds during its lifetime.

A particle tracks the time it was alive and exposes it through a progress getter, which returns a value between zero and one.

When you think of the value of why is it between zero and one?

If a particle, for example, lives for 4 seconds, the idea of the zero and one is that it is percentage zero is 0%, one is 100%.

So if you're going to see a value like 0.8, that means that the particle has moved through 80% of its lifespan.

So here's an example code for sampling the lifecycle of a single particle through its progress getter by sampling.

Here, what we mean is being able to peek into where exactly is the particle when it comes to its age.

So we create a lifespan of 2 seconds.

We create a particle with that lifespan, and then we add that particle wrapped inside a particle component into the game loop.

And we'll discuss in a second why does it have to be particle component?

But if you remember, the only thing that can be added to a game loop is a component.

So a particle component is a wrapper around the notion of a particle so that we can add it to the game loop.

So we have added now that particle to the game loop.

So what do we do next is we create a timer which is going to periodically wake up at the interval of life time 0.1 since life is two times 0.1 would be 0.2.

So that would be basically every 200 milliseconds the timer is going to wake up and it's going to print the progress of the particle through the getter.

This is pretty useful if you want to, for example, figure out how much time is left for this particle.

So at the heart of the flame engine particle system is the abstract particle class and its basic implementation,

the particle component class, which is used as a wrapper to more complex particles.

Particle component also is what you can add to the game loop.

So it kind of has a dual effect here.

A very important method of the particle class is the generate method, which allows us to plug in a generator to produce one or more particles that we want to simulate.

So this is important because ultimately a particle system is a collection of particles.

So the particle class actually has a generate method which we can override to specify how we would like to generate the particles, and then also how many we would like to generate.

So this means that our particle class can be viewed as the parent of more particles that comprise the simulation.

So we start with a particle abstract class.

Notice this is very important that the particle class has as its children more particle instances.

This actually makes sense since the particle class has a generate method, so it would have to hold on to those particles.

We then have a particle component which extends the particle class and we have other components that also extended.

But what is interesting is that the particle class acts at once as a singular particle as well as a particle factory.

Here is a list of the main and most important particle implementations.

We have a translated particle which translates each child by a given vector.

Translation is simply a movement.

So this is a type of particle that affects the movement of a particle.

We also have a moving particle which moves its child between two pre-defined vector instances.

So we basically have a start and an end which defines a path.

We could also be more specific in exactly what this path is.

It could be a straight line, it could be a curve.

But the idea is that a moving particle specializes in being able to move the particle between Points A and B.

We also have accelerated particle, which is very similar, but it allows for basic physics based effects like gravitation or speed dampening to be applied to the particle.

Notice that each one of these three particles, one, two and three, are particles that actually define some sort of movement, some sort of motion, some sort of a transformation.

The other class that we have is a circle particle, which renders circles of all shapes and sizes.

What is interesting about this particle is that this particle does not act on the actual particle itself.

It doesn't define where it moves or how it moves.

It only defines what it looks like.

Same thing goes with the sprite particle.

It renders a flame sprite within a particle effect.

Again, it doesn't affect anything about the motion.

It just affects how it appears on the screen.

We also have an image particle which renders dart UI image within a particle effect.

And we also have a component particle which renders a flame component within a particle effect.

And we have flair particle, which renders flare animations, which is rife within a particle effect.

Flare particle is a little more special.

There's a whole application out there called Drive that you can create animations and then import them into the flame engine.

We're not going to cover this in this course because that is a more advanced topic, but one day we'll cover that as well.

But I just wanted you to be aware of that.

One important thing to see is that the first three classes actually affect the movement or motion of the particle and the other four or five, six, seven and eight classes, they don't affect anything.

They're simply holders for the actual visualization of the particles.

So, for example, a circle particle shows a circle with a component particle.

I could put some other shape into it, like a square and for example, render a square.

All right, so this is it.

For this part of the lecture, I would like you to go over the slides a few times, kind of internalize it, think about what it is that we're dealing with here in terms of particle particle system, the fact that we generate particles, the fact that a particle can aggregate more particles, which gives us the actual particle system.

In other words, study this, go over this a few times, and I'm going to see you in the next lecture where we're going to be looking at some examples of code.

All right.

See you then.

2. Creating Simple Particle Simulations

All right.

Welcome back.

What we're going to do here is we're going to look at two examples of creating more complex particles for our particle system.

So the main concept in particle system in flame is that we can stack particles.

Consider this coding example.

We have a moving particle, which, as you remember, is a particle that controls movement.

It doesn't display anything.

It just controls movement.

So we're creating a moving particle and we're going to create a vector zero, which is going to be the starting point.

Vector zero is the upper left corner of the canvas.

It's basically the zero zero coordinate.

Two is going to have the size of the canvas, which is going to be the lower right corner of our screen.

We are going to define the lifespan, which is how long this particle will live to be, 4 seconds.

And we're going to specify the curve, which is the path that this particle is going to follow.

In our case, we're going to use a deceleration curve, which means that it's a straight line and it will start from the left corner at a certain speed, and then it's going to start slowing down.

Now, how do we render this?

This so far has defined mostly what this particle will do, not what it looks like.

So we're going to add to the moving particle child, which is going to be a circle particle, which will be responsible for creating the actual visual definition of our particle.

We are going to have a radius of two.

We're going to have the color as red.

And that is basically it.

One question you could ask is, okay, but what is the speed here?

Well, the speed of the moving particle is basically, at least by definition, defined by the path.

In other words, if, for example, the length of the path is, let's say, 1000 pixels and the lifespan of the particle is 4 seconds, then moving particle will compute what should be the speed of our particle so that it starts in the left upper corner, it ends in the right lower corner and exactly 4 seconds by default it calculates that.

So just to recap it, we are creating here a moving particle which controls the specific motion via curve property.

And as a child we're adding an actual circle particle which will render a circle with a radius of two pixels in a red color.

Notice that what we're doing here is we're compositing a behavior particle with a rendered particle.

So a moving particle is the behavior particle circle particle is the rendered particle.

And this is a very powerful mechanism from a reusability perspective, because I could quickly swap in and out different visuals for this particle without affecting the way it moves.

And we're not specifying speed.

This is indirectly tied to the lifespan 4 seconds of the particle.

All right.

Let's have a look at a different example here.

We're going to start with an accelerated particle, which is a different type of particle, similar to moving particle, but it is based on an acceleration.

So we're going to have some forces acting on this particle.

So this article is going to have a position which we're going to initialize to some input position.

And what we would like to do is create a downward shooting particle.

So the way we're going to do it is we're going to get a random value multiplied by 200 and subtract 100 from it.

So next double is something between zero and one.

At the extreme case, it will be one and one times 200 is 200.

-100 would be 100.

So the x value is going to be at most 100.

But how do we make a downward?

Well, notice that we have a random next double.

So a similar idea, something between zero and one.

And we're actually saying that we would like to get the maximum of random next double and 0.1.

In other words, what we're doing is we're forcing this to be 0.1 at the most.

We leave it positive so that it's pointing downwards and we're going to multiply it by 100.

So that is going to give us the X and Y speed.

Now that we have defined how this moves, next we decide what the particle is going to look like.

We're going to use the usual suspect.

For now, the Circle particle and the Circle particle will be small.

It will have a radius of one and we're going to actually create a random color between yellow and red.

And the way we're going to do that is we're going to use the CHIRP function, which I'll explain in just a bit.

And we're going to basically get a value that goes from yellow to red and is defined by a random value from 0 to 1.

This is going to give us a random color somewhere in the spectrum of yellow to red.

So an orange, for example, or something like that as well.

And that is basically it.

So this is another example.

So what we have here is an accelerated particle which controls the specific motion via speed property.

And as a child we're adding a circle particle which will render a circle with radius of one pixels in a random interpolation of colors between yellow and red.

And just as a recap, next double returns a value between zero and one.

It's uniformly distributed.

So it's a it's a it's a true randomness lookup, which is basically linear interpolation.

What it does is it linearly interpolate between zero and one.

So when we plug in random next double, which is going to be a value between zero and one that is going to choose color between yellow and red, it will interpolate between those colors.

Now one thing you could ask is, okay, great, but we did not specify what the lifespan of this particle is.

And by default in flame, lifespan of a particle is half a second if you don't specify it.

All right.

So this was a quick little example that kind of shows you how we stack particles together.

Next, we're going to see how we can actually create an explosion.

See you in the next lecture.

3. Coding a Particle Stream Simulation

Welcome back.

Let's have a look at some code for case study number seven particle system.

We're going to be looking at the code that we partially looked at.

An example, number one.

Let's first look at what exactly we have here.

You can see that we have looking at the outline that we have two classes.

We have a particle system.

Example 001, which as you can see is basically our main game class because it extends frame gain.

Also, this class uses a mixing pattern detector and detector makes sense.

It allows you to get notification whenever a user is dragging a mouse or their finger on the surface of the screen.

In this case, what we're doing is we're overwriting the on pen update method and the on pen update method gives us info.

In our case, what we're looking for is just the position of where the user's finger is as they're dragging

their finger on the screen.

So we'll come back to this.

The main class here that we're interested in is this particle generator, which is our own custom made particle generator.

And as you can see, we have a static method here which returns a particle component.

So it returns a single particle.

What it does is it accepts a position which is a required input.

It builds a particle based on two sub particles.

We're using accelerated particle, and we're using circle particle.

The accelerator particle defines the behavior of the particle and the child of that particle circle.

Particle, in this case defines what the particle looks like on the screen.

So let's have a look at what we're doing here.

So we're returning a particle component which wraps around accelerated particle.

The lifespan of this particle is 0.5, which happens to be a default lifespan.

So we could remove it, but I'm putting it here for clarity.

We're also setting the position of the accelerator particle to be the input position.

And then what we're doing is we're defining the speed, the movement of the particle.

And this is relatively interesting.

What we have here is a vector.

If you look at the x value, the random next double, what it does is it returns something in the range of zero and one.

When you multiply that by 200, it's going to give you something in the range of zero and 200.

If you subtract 100 from it, that means that you're going to just shift that.

So the range of X is -100 to 100.

So what it does, basically it defines the particles that they could be to the right of the position or they could be to the left of the position.

So it's balanced.

Let's look at what the Y does.

Now, the Y is a little interesting here.

What we have is again, we have random X double, which is going to return something between zero and one.

But because we're using the max function and if you see here, this is intuitively what you would think,

it returns the larger of the two numbers.

It returns max of 0.1 and a number that is in the range of 0 to 1 since the smallest number defined here is 0.1.

This guarantees that your value coming out of max will be most one.

That's because of an X double, but it will be at least 0.1.

So in other words, we don't allow a zero.

So once you have the range of 0.1 to 1 and you multiply it by 100, what you will have is really 0.1 times 100 is ten.

Obviously one times 100 is 100.

The range that you're going to get here is going to be 10 to 100.

So because this is a vector, we can think of it as the speed, the velocity.

And what that's going to do is it's going to define something that is pointing down, because notice that, Max, of the random next level that's going to be positive, it's 10 to 100.

So it will always be positive, which means that it will always be pointing down.

So basically the particle is going to be moving left or right and down.

So we've defined that this is going to create a downward shooting particle.

Now let's look at the child.

It's a circle particle.

It's going to have a radius of one.

So very, very small.

And the color is going to be a function which is lookup.

And the function, what it does is it linearly interpolate between zero and one.

The first two inputs specify a range of colors, so our range is yellow to red.

The third number defines basically the kind of mixing the interpolation of these two colors based on that number.

So it's almost like a like a mixing of the two colors and that's basically all it does.

So we're, we're generating based on a random value, some sort of particle that is going to be in the color somewhere between yellow and red.

This is a single particle, this function all it.

Does is it simply generates a single particle.

So if we go back up right now and we look at what on Penn update does, what on Penn update does is it takes the position of where the user's finger is and it creates a single particle based on that.

So makes you wonder if this is just a single particle.

I mean, we're not going to see a lot of particles if you're dragging your finger across the screen because your finger is in contact with the screen.

All this time, I could say secondary to this on panel update will be called very, very frequently.

Think of the game loop and how often it updates.

This is almost that kind of updating.

So you will actually be generating tens, if not hundreds of particles per second because this will be called many, many times.

Every single time you move your finger as you're dragging it, it will generate the particle.

So let's quickly run this and let's have a look at what it does.

So this is this is pretty interesting in a sense that we're only actually creating a single particle.

Always takes a little bit of time to get it running.

All right.

Almost there.

So now when I drag my finger, well, I'm actually dragging the mouse.

Notice that if I just click, nothing happens.

Because this mixing here pain detector, what it does is it only is affected when you drag the mouse.

So I'm going to drag it right now and look what happens.

Look how many particles we're generating.

That is a lot of particles that are being generated that's easily more than 100 per second.

And that's because even though we're generating one particle per the update method, because this method is updated so often we're generating so many particles.

So it creates quite an interesting effect.

We could change a few things.

These.

I could, for example, play around with this, make this 100 and make this 50 so that this becomes more narrow.

Let's see what this will look like so you can see this is now it looks almost like like we could have like a little spaceship running and I could narrow it even more.

I could, for example, go, let's say, 70 and subtract 35, which is going to narrow it even more.

These are just arbitrary numbers.

Sometimes you just want to have a look at that.

But basically I'm defining right now -35 to 35.

And if I run this, you can see that this could, for example, be like an exhaust of a spaceship.

Because it looks looks reasonable enough.

Yeah, that's pretty cool.

So you can you can play around with this.

And one other thing that we could do is we can change the interpolation as well.

Let's go from instead of yellow, let's make this blue.

And instead of red, let's make it green.

Let's run this.

This will be looking different.

There it is.

That looks a little strange.

Like toothpaste.

Okay, so I definitely like the the yellow and red much better.

Okay, So I suggest that you play around with this, but this is an example of wrapping a particle component

around accelerated particle, which defines the behavior.

And then as a child of that wrapping a circle particle, which defines the visual aspect of the actual

particle.

All right.

So I hope you enjoyed this.

We're going to be looking at further examples of the code.

See you in the next lecture.

4. Designing an "explosion" Particle Simulation

Let's look at another example of how to use particle system.

How would we simulate a particle explosion?

Let's look at the anatomy of a simple particle explosion.

First, it originates at a center.

The explosion has a center, the origin point.

The epicenter breaks away into n separate particles.

By n, we mean just many.

It could be 20, 30, 40, 50.

It depends on, obviously, how large the explosion is.

The end particles then move away from the center in all directions with accelerated speed.

That makes sense.

And explosion accelerates the particles.

The acceleration vector of each particle is random.

In other words, it should be happening in all directions.

So to create such a particle simulation, we will need to do the following steps.

We will need to create a particle generator for GN particles, say like 20 or 40.

Each particle will be an accelerated particle with an acceleration defined by a random vector.

This will define the path of the particle in a straight line and one that speeds up.

As it travels away.

Each particle will be rendered using circle particle.

So basically we're going to simulate the pieces of the explosion as little circles.

Okay.

So as we have said, we will use a particle generator to create all the particles since we will be using accelerated particle as the path definition.

So that will be our definition of how the particle is going to be moving.

And we're going to use the circle particle as the rendered particle.

We will create random vectors for velocity away from center, some sort of an origin point.

This is what it will look like.

You can imagine that the very center of that is where the explosion starts and all these particles are going to move in different directions and each one of these arrows is a vector.

So let's have a look at possible code that we could use.

So we will use the generate method of the particle class.

We're going to set the count of our particles to be 40.

We're going to override the generator for each of the 40 particles.

Our generator is going to create an accelerated particle.

The acceleration is just going to be a random vector.

The position of our explosion arbitrarily.

Right now, we're just setting it to be at 100, 100.

This is something that normally you would input for the illustration.

We just put it at 100 and hundred.

The child of the accelerated particle or the particle that will be rendered will be a circle particle.

We're going to make the color of each particle to be red.

We're going to make it radius two, and we're going to define each particle to have a lifespan of four.

And this is basically it for the code.

So if you actually run the code right now, this is what it will look like.

As you can see, there is a little dot in the middle that seems to be expanding a little bit.

Let's have a look at why this is happening.

The problem is with vector to random.

The random function of the vector to class generates a random vector in the range of zero 0 to 1 one.

That is a very, very small vector.

In other words, in order to actually create a much larger vector, if you look at this picture that we have of the explosion, if each one of those vectors was just in the range of 0 to 1, that's literally one pixel length.

So what we really need is we need to scale that acceleration.

And the way we do that is by scaling it by 200.

Scaling is just a shorthand for making the vector normalized and then multiplying it by 200.

So if we do that, let's have a look right now what the code itself, if you run it, what the code would look like and if we run it right now.

This is what it looks like.

So you can see that the explosion looks pretty good right now.

But there seems to be one problem.

Even though it's supposed to be a random vector, it seems to be generating things only in one quadrant and it's in a specific direction.

Everything seems to be going down and.

Right.

Why is that?

We need to look a little more carefully at how they vector to random method works.

So let's have a look at that.

So the problem is that vector to random generates a random vector in the range of zero 0 to 1 one.

We can visualize it.

As you remember from previous lectures, we could visualize all the different directionality vectors

with this handy little diagram.

And you can see that right there.

That green arrow is where the vector is going to be generated.

In other words, this function, the vector to the random function, what it does, it only generates

numbers that fall within that gray area.

This means that if we want to cover the whole area of the above box, we need to modify the random function

to return a vector in the total range.

And what is the total range?

One way to do this is to generate a random X and Y values in the range of minus one, -1 to 1 one,

which would cover that whole box.

And we could use the random class.

Here is how we would write it.

We can create a function called random vector, which returns a vector.

We start with a random class.

We set our minimum to be minus one and our maximum to be one.

Next, we're going to create the x component of the random vector.

The formula is pretty straightforward.

We're going to basically take the minimum, which is our min minus one.

We're going to add to it the difference between max and negative min.

So in this case it would be max.

Minus min would basically be one minus minus one, which is two.

That's the range basically.

And then we multiply it by the next double random number.

That number is going to be between zero and one.

So what we're doing here is we're basically saying that the minimum for our X is going to be minus one.

And then the range that we would like to create will be two.

The range is going to be two and then we multiply it by the random number and we add to that for Y,

we can do exactly the same thing.

We're going to basically just generate a number in a similar fashion.

So this will generate a random vector that is going to fall within that box, and then we simply return

a vector with those two initialized x and Y values.

So let's have a look at what that will look like when we actually run the code with that.

And now our explosion looks much more organic and it seems to be going in all the different directions.

All right.

So this was pretty short and sweet little lecture.

Please go over the code, try to understand what is happening here.

And I'm going to see you in the next example for particle system.

5. Coding a Particle Explosion Simulation

Welcome back.

Today, we're going to do things a little bit differently.

We are going to start our project basically from scratch.

So what are we going to do right now is we're going to look at the code for case study number seven particle system.

We're going to be looking at example number two from the case studies, and we're going to be looking at the code that simulates an explosion.

So what we're going to do is you're going to go here into view, into the command palette.

You're going to look for Flutter New project.

And we're going to create an application.

I'm just going to give it an arbitrary name.

I'm just going to call it component 007002.

But you can call your project whatever you want.

So this has now generated a brand new project.

And if you look here at Main Dot Dart, this is a flutter project, so we do not really need this.

So what I'm going to do is I'm going to copy the code from the previous example that we were working with, which was that stream of particles that looked like an exhaust from a spaceship.

So basically we're going to grab that code and we're going to rewrite it.

So I'm going to get rid of this code completely, remove it, and I'm going to paste the previous code.

And as you can see, this is our previous code.

So obviously right now, because this is purely a flutter project, we cannot compile it.

What we need to do is we need to go into Pub Spectre YAML, and we need to add the flame dependency.

We then save it.

Och, this is done.

Get rid of that.

So now, as you see, everything is comparable.

But what are we going to do is, first of all I'm going to name this particle system.

Example 002.

I'm going to grab this, change it here at the bottom.

So now we have basically the code from before.

First of all, we're not going to be using a pattern detector.

What we're going to do is we're going to be using a tap detector.

So I'm going to change this.

And of course, for tap detector, we're going to be using a different method here.

If you, for example, don't remember what the tap detector function to override is, you can just go in here on Windows.

I'm holding the control key and I'm just going to click in This now gives me the code and you can see that the mixing tap detector has all these methods.

What we're going to be looking for is on on top up.

So I'm just going to copy this.

You can get rid of this.

I'm going back right now, and I'm just going to change this.

And now we have a flame game extension with tap detector.

Now what we're going to do is we're going to change this to have our explosion code.

So the first thing I'm going to do is just give this a slightly better documentation.

Okay.

What we're going to do here is we're going to remove all this and we're going to type in the other code.

I'm just going to paste it.

We can now paste in our code.

This will currently not compile just because we are returning a particle component, whereas this creates a particle.

So we can do two things.

We could either change this to be particle.

Right.

But then, of course, this doesn't work because we want to put political component here.

Or we can go back here and we can.

Return.

A particle component.

Initialized with this.

Okay, so this full work.

So now what we're going to do is we also need this random vector function.

So which we covered in in class.

I'm going to add it into the same class.

Here it is.

The main problem that we have right now, the reason why this is showing in red is if you hover over this, it shows you that it can be accessed from a static method.

So what I'm just going to do is I'm just going to make this static.

That way it's available.

So let's run this code and then we're going to talk about it.

Okay.

As usual.

Takes a little bit of time.

All right.

So now, at least in theory, if I click, we should be getting an explosion.

And we are.

But we're getting it in the wrong position in the code that we get from the case study, we arbitrarily have put the position as being 100 hundred.

What we want to do is we want to use the position of the click.

So I'm going to grab that and I'm just going to plug in here.

I'm going to restart it.

And now we have an explosion wherever we click.

Okay.

So let's quickly have a look at this code.

Okay.

So basically what we're doing here is we're using a particle generator.

So we're using the particle generate method.

We're creating 30 particles in the generator.

Each particle is going to have a lifespan of five.

Basically, the lifespan of this particle is five.

So that automatically translates to the children.

Our generator is going to be using accelerated particles for the behavior.

So we're setting that.

The acceleration is a random vector.

We scale it to 100.

Next, we set the position to be the position of the click and our child, which is the particle that is responsible for visualization.

We're using the circle particle we painted red and the radius is two.

That's pretty straightforward.

We have this code right here for the random vector.

We have covered it in the case study, but basically what it does is it randomize the vector so that it truly goes into the different directions.

In this case, what we're doing is we're just making sure that we're covering all of the quadrants that a vector could point to.

So this is truly a random vector function.

We don't need this piece right here.

So this looks pretty good.

Now, one thing that I would like you to notice is that this lifespan right here for the particle is five.

But what I could do is I could remove this lifespan and put it here for each one of the circle particles.

And you would expect that this would still be the same.

But as you will see right now, when I compile this, so notice this is 5 seconds.

When I compile this.

Notice now that it only lasts half a second.

And if you remember, particle component by default has a lifespan of half a second.

When we specify that the child here, the circle particle, has a lifespan of 5 seconds, that is irrelevant to the parent.

It doesn't flow up.

It only flows down.

In other words, setting the lifespan of five for the child particle doesn't do anything.

We need to take this lifespan and we need to put it to the main particle, the topmost particle, the

parent.

And that's why we have the lifespan here.

And for example, if I make it 3 seconds, our explosion is going to be we'll take a little shorter time to expire.

There it is.

Perfect.

And these numbers that you see here, in case you're curious.

I am just outputting this random vector here to show you how this randomization of the vector, how well it actually works.

All right.

So I suggest that you play around with this code.

You know, change the lifespan, change the count, and just see what you can do with it.

If you check out the exercises, you're going to see that in the exercises, we ask you to actually go back to the joystick code where we have the spaceship that you can move and shoot.

We would like you to modify the code where you're shooting the objects that are flying around.

We had those circles, those balls, so that each time when you shoot one of them, they actually explodes.

That's something that you can do in the exercises.

All right.

So I'm going to see you in the next lecture.

Take care.

6. Designing A "fireworks" Particle Simulation

All right.

We'll come back.

Let's do something a little bit more complex.

We will try to define a particle system to simulate fireworks.

This is basically what we're trying to do.

This is the effect.

It looks like a bunch of fireworks have been shot into the sky.

They change the color and at the end they flare up.

Notice, though, that we have an acceleration and then we have a deceleration.

It comes up by the force of the explosion and then it's brought back down by the force of gravity.

This is what we're going to simulate.

So what we need is to have the ability to define how gravity works so that when the fireworks initially explode and they start falling down, we can slow them down and draw them back together for that fireworks effect.

We will create a randomized color renderer for each particle so that as the particle falls, it can change its colors.

We will also draw a larger flare at the end of the lifespan of a particle so that it would simulate fireworks exploding at the end.

What we need to use here is another type of particle implementation in flame, which is the computed particle.

The computed particle allows us to fully control the life cycle of a particle by being able to monitor its progress as it progresses or moves through its lifespan.

Here is how we will structure our particle system.

We start with the particle.

We are then going to use an accelerated particle within our system to define how the particle moves.

And to render our particle, we're going to use the computed particle.

It's interesting to note that the computed particle is actually a descendant of the particle.

We will use this system as follows.

Particle class will be used as a generator of all our particles.

Accelerated particle will be used for simulation of initial explosion and the gravity kicking in to bring the particles back down.

Computed particle will be used to fully render the accelerated particle aesthetic as follows.

We will draw a circle.

We will choose a random color and dynamic opacity for the accelerated particle, which is the circle we're going to draw.

We will also create a flare effect at the end of the lifespan of the particle.

How does a computer particle work?

It takes one main input to work the renderer, which comprises of the following the canvas, which it will use to render the visual of the particle effect to the particle.

To visualise it will be given a notification of the life cycle progress of that particle, so that the computed particle can monitor the lifespan progress of the input particle.

So basically these two points mean that we will need a canvas to which we're going to draw the visualisation

and we also need the particle that we're working with that we're going to be visualizing as input so

that we can use the progress getter of that particle to find out what it's doing.

Now.

What is the purpose of doing all this?

The main purpose is a complete control of customisation.

It gives us great flexibility of creating all sorts of particle effects.

If you recall from the other lectures, particle class provides a getter for progress, which is a value

in the range of 0 to 1, which we can think of as from not started to done.

In other words, zero means our particle has not started doing anything yet and one would mean the particle

is done, it's finished, and this value is basically a percentage of the actual lifespan progress of

the particle.

What we will use in our simulation here is we will use the progress to control the particle behavior

as it reaches, for example, 60% of its lifetime.

In other words, that flare that you've seen at the end of the fireworks, once we reach about 60% of

the lifetime, that's when we're going to do the flare.

All right.

So let's have a look at the first piece of code we're going to define here, a palette to paint over.

We put this in quotes, the sky, which is basically what the colors of the fireworks are going to be.

So this is just an array.

We also have a random function that allows us to pick one of the elements from that list.

And here is the visualization of the colors that we're actually going to be using.

So what this is first, we define an array of colors that we will use in the fireworks simulation.

So as you can see, we have amber, amber X and red red accent, yellow, yellow accent, and then we

have a blue for like a lensing effect.

This array or list is then mapped to a list of paint objects with the corresponding colour initialized

to the color value from the array.

In other words, initially this paint array just holds colours elements, but we need it to be paint

elements.

So we're just remapping our array to be an actual list of paint objects where the paint dot colour attribute

of that object is set to our colour.

We also have defined a random element function which will randomly select an element from that list.
In other words, when we call the random element and we provide the paints list as input, this function will return a random element of this list.
So we will use this to randomly select an element from our paint list.
So once we have defined our colours and our random method, let's next move to actually defining our particle system.
So we're going to have a generate method.
We're going to have ten elements.
In other words, we'll have ten particles.
Our generator will look as follows We'll define an initial speed, will define our deceleration.
We're also going to define gravity.
Here is our accelerated particle.
We define the lifespan of each particle to be 2 seconds.
The position we're going to use is just going to be an arbitrary 100 hundred.
On the screen, we're going to define the speed of our accelerated particle as the initial speed that was defined at the beginning of the generator, which is a random speed.
Our acceleration will be defined as deceleration plus gravity, and the child of the accelerated particle is going to be a computer particle.
We're going to look at that code in the next slide.
And this is basically it.
So to do a quick recap, each particle will have an initial random vector speed.
And we also define a deceleration, which is the slowdown to be a vector opposite of speed.
We also define gravity to be a vector pointing down with a magnitude of 40 with an initialize the particle speed to be the initial speed and the acceleration to be the sum of deceleration and gravity that makes sense deceleration.
We don't know what direction it's going to be, but we also know that our gravity is supposed to pull all the particles down, so to speak, down to the ground.
The accelerated particle will then slow down the particle as the particle progresses.
Now, I'm going to do a little digression here.
I'm going to tell you a little bit more about the accelerated particles so you understand how the particle actually does the acceleration deceleration.
If you're curious as to how the accelerated particle actually defines the speed and acceleration, here

is the implementation of its update method.

As you can see, it's relatively simple.

Speed is simply an accumulation of acceleration force and position is simply a motion displacement formula

and see that the way the accelerated particle defines speed is simply speed is equal to current speed

plus acceleration vector times the delta time, and that is added basically to our speed.

In other words, if the acceleration is positive, then our speed will keep on increasing.

Now the position itself is going to simply be the displacement based on the acceleration.

So position is going to be equal to speed times d, t minus, acceleration times D squared divided by

two.

Now this might all seem a little mathematical, so I'm going to show you where you can have more information

about it.

Here is a website.

The find right here.

You can have a look at it where you can find out a little bit more information if you're curious about

the math.

But you can basically see that this right here is the formula.

So you can look a little bit more on this website if you want to find out.

But just to let you know, basically that this is what is happening.

Okay.

So we can go back to our code right now.

Let's have a look at the computer particle.

So the computer protocol is going to take a render as input and it will take two attributes.

Canvas and particle particle is the accelerated particle, and canvas is just the canvas from the flame

engine.

So we're going to choose a random paint.

We're going to also change the opacity of this paint where they're going to draw a circle.

The offset zero here means that we're not changing the position of the input particle.

So offset zero means basically we're drawing a circle in the position that the particle already has.

We're not offsetting it.

We're not changing it.

And here is the actual radius.

This is quite interesting.

The actual radius is simply the progress of the particle times, a randomized number between zero and one.

If that the random number times the particle progress is greater than 60%, then we take the first option.

We create a particle with the radius of 50 times particle progress, times a random number, which will usually be a larger, almost like a flare.

If, on the other hand, the particle progress times the random value is less than or equal to 60%.

We do the second option, which is simply two plus three times particle progress, which simply will be most of the time much smaller than the first value.

The first value could very well be if, for example, next double brings us one.

And particle progress, for example, is 60%, and that value will be the radius will be quite large.

And then the circle is painted with the paint that we got from the random element, and that is it.

So to recap, the computed particle creates the visualization of the particle as follows We randomly choose a color, we change the colors opacity.

So as the particle is closer to the end of its lifespan, the opacity drops.

In other words, it becomes less visible.

We then draw the circle at the position of the original accelerated particle.

That's why we have offset zero.

There is no offset.

We compute the radius as follows.

If the particle progress times the random number is greater than 60%, then we make the radius larger else.

On the other hand, we just set it as a two plus three times progress.

And this is basically it.

Just to show you again what it would look like.

This is basically what this code simulates.

We're going to actually go through this code in an editor, but I would like you to study this without actually trying to run it, because that way it forces you to think a little bit more about why this code works the way it does.

All right.

I'm going to see you in the next lecture.

7. Coding a "fireworks" Particle Simulation

All right.

Welcome back.

This code is going to go through case study number seven particle system.

Example number three specifically.

We're going to go over the code responsible for that fireworks simulation that we have done.

So let's have a look first at what it looks like on the screen.

When you look at what is happening before you see the flares, you can see that the particles, while they're traveling, they're changing color.

So we can think of this system as a number of different things happening depending on the time during the fireworks.

So initially when the fireworks explode, they go up and away.

Then gravity kind of brings them down.

But as they're going up, they're kind of like flaring a little bit, changing color.

But then as they go up to the very top of their display, when they start coming down, you can see there's a larger flare.

And as was mentioned in the case study number seven particle system, example number three, what we're using here is a special type of particle, which is the computed particle.

The idea behind the computed particle is that a computed particle gives you more flexibility in terms of being able to see what is happening with a particle.

A computer particle has extra hooks into the life cycle of a particle.

When I click, you can see that we're making changes to the particles as they're moving.

We do need some sort of a way of hooking into the lifecycle.

So let's quickly look at what is happening with the computer particle.

We're kind of going a little backwards here.

The computer particle, as you can see, relies on a renderer.

So in other words, we're giving computer particle complete control over rendering.

The render takes two parameters.

It takes canvas onto which we will be rendering the particle, and it takes the actual particle that we're simulating.

So what do we do here with this computer particle?

Notice that the first line where we say paint is equal to get random list element.

What we're doing is we're randomly picking a color.

Next, we're changing the opacity of that color dependent on where in the lifecycle the particle is because we go one minus particle progress, Particle progress goes from 0 to 1.

It's a percentage.

So if the particle is completely done, that would be one minus one, which is zero.

Opacity of zero means that it's completely invisible.

The basic thing that you can take away from this is that as the particle progresses through its lifecycle, it becomes less and less visible because opacity is going to start dropping.

We're subtracting more from one.

So once we have gotten the the paint, the next thing that we're doing is we're going to draw a circle on the canvas.

So again, we're using a particle that is going to look like a like a ball.

The offset zero is pretty straightforward because this particle is being drawn depending on another particle, the one that was input.

We could offset this particle depending on the position of the parent particle.

In our case, what we're saying is we would like to use the exact position of the input particle as our position when we're rendering this.

So that's what offset zero means.

The next line is interesting because what we're doing is we have a ternary operator, so it's like an F and L statement.

And the condition is that we're taking a random number between zero and one.

We multiply it by the particle progress, which is also going to be between zero and one.

And we test whether this is greater than 0.6, which is basically is it greater than 60%?

If it is greater than 60%, then we're going to render a particle with a radius of 50 times particle progress, times some random variable.

Most of the time that line is going to tend towards a much larger circle, meaning it will give you that effect of kind of flaring up.

If, on the other hand, the particle progress times next double is less than 60%, then we're just rendering arbitrarily two plus three times particle progress, which is basically a smaller particle.

All of this is the radius and of course the paint is the one that we have assigned just at the beginning of the render.

So this is what the computer particle does.

The next question is OC, what is the parent particle doing?

Well, the parent particle is the accelerated particle, and that particle basically takes position, which is where are we starting from or where is it?

It takes speed, which in our case is going to be some initial speed, which is like, what is the speed that we're starting with?

And then we have another.

Tribute, which is acceleration.

Acceleration is going to be deceleration plus gravity.

What we're doing here is we're basically computing the acceleration, which initially is going to be pretty small.

So it will not counteract the initial speed, but eventually acceleration is going to become less and less.

So what we will have is negative acceleration and the particle is going to start slowing down.

And not only that, it will reverse its trajectory and start going down.

So that's pretty straight forward.

If we go even higher than the accelerated particle, we can see that we have a particle generator.

The lifespan of each particle is 2 seconds.

The number of particles we're going to generate is ten and our generator is going to, for each particle, create the initial speed, which is going to be a random vector scaled to 200, which is the random vector function that we have used already a few times that distributes evenly the chance that the vector is in any direction.

So just a proper random vector, the deceleration is just the reversal of the initial speed, and we define gravity as a vector with a force of 40, where the 40, which is in the y direction, is pointing down, it's pointing towards the ground.

You probably have a question which is let me click one more time.

And I mean, how do we account for the fact that the color is changing?

Well, we have to understand one thing about the computer particle.

The computer particle is not called only once per generation.

In other words, think of it this way When the particle degenerate is called, it creates these particles, and then these particles go through their life cycle as the computed particle is going through their life cycle.

All of this code from render all the way down to paint gets repeated over and over multiple times as

the particle is moving through its life span.

Since each time the renderer is going to be called as the progress of the particle is moving.

Each time that the render is called, it will get a random paint value, so the color of that vertical will be changing.

And again, if you look at it carefully, any particle that we're looking at, you can see that it's the same particle, but it's changing color while it's moving.

Well, that's because the renderer for computed particle is called multiple times during the lifetime.

And therefore values such as paint and obviously particle progress and everything is going to change.

So this is basically it for this code, this value right here for random vector.

It's the same one that we've used before.

Let me just quickly talk about how we have created the colors.

So when you look at the paints array, what we have here is just an array of colors.

Instead of just using color, it might be easier for you to use colors.

So I'm going to define it right now in terms of colors.

So this one, for example, is blue.

This color here is going to be yellow accent.

This color here is going to be just yellow.

This color here is going to be red accent.

This was just going to be read.

This one is going to be Amber accent.

And this one is just going to be Amber.

Okay.

So what we have done basically is we have just made it a little more user friendly here.

We're using words which might be a little bit easier.

So we have created an array of colors that we're going to use.

And then what we're doing is we're remapping these values so that they become actually a list of paint color.

If I go into amber, you can see that that's a material color.

But if you go down here and look at this paint, you can see that that's actually a paint color.

So we need this kind of value.

So basically what we're doing here, we're just we're just mapping something that is a little easier and user friendly in terms of just colors because you can name them into paint color.

That's what this line does.

And then it makes it back into a list.

So that paints is actually an array of paint dot color values.

It's not material color, it's paint color values.

And the last thing that I would like to show you is this function right here.

Get random list element.

What this function does is it's simply given a list, gives you a random element from that list.

And we're using this right here to get a random paint so that when you click here, you're going to get random colors.

All right.

This is basically it.

Have a look through this code, play around with it.

And I'm going to see you in the next lecture.

Thank you for listening.

8. Working with SpriteSheet "explosion" Particle Simulation

In this lesson, we're going to talk about Sprite Animation in the context of particle system.

So we can also use particle system to animate sprite images.

It's not always just little circles and balls flying around, but what are sprites?

They are images that usually are part of the gameplay, such as game characters or enemies.

What is Sprite Animation?

This is a technique where you use multiple sprites to create a frame by frame animation of a character or a game event such as an explosion.

There are two parts to animating sprites.

First, you have to load all the images that make up the animation.

Secondly, you then have to execute the animation in a frame by frame manner so that it looks like the character is moving or doing something.

Because loading sprite images takes time.

We usually combine all the frames of the animation into a single sprite sheet image and load that into memory that only then takes one load.

This, in effect, is what sprite sheet animation is.

But why use particle system to animate sprites?

The main reason is that it gives you a different abstraction of working with sprites.

You can control the duration of the animation through particles, lifespan, property.

So for example, if you have 20 images in your sprite sheet and you display them with a lifespan of one second, then you're getting a 20 piece animation.

There is a more involved way to work with sprites and sprite animation and Flame, but will not be covering it here.

For example, character movement would be best controlled in ways other than particle animation for sprites.

But if you want to simulate, for example, an explosion, then you can easily use the sprite animation particle to achieve just that.

So let's talk now about the structure of a spreadsheet.

What is this sprite sheet?

A sprite sheet is usually a grid of images in some sort of an end times m configuration.

You have n rows and you have m columns.

So you can treat a sprite sheet as a two dimensional array.

Here is a simple example of an explosion spreadsheet, which is an eight by eight grid of images.

So as you can see, we have all of these images where we start from a very, very tiny beginning, like a seed of an explosion all the way where you see on the bottom where the explosion has gone through all of its cycle and it's dissipating.

You can look at this sprite sheet as basically a collection of n rows and m columns.

In fact, what is happening is that's exactly how we're loading these images.

I want you to also note that we could have any combination of rows and columns.

They don't necessarily have to be an eight by eight or a five by five.

Here's another example.

This time it's a one times 20 sprite sheet.

We have one row and we have 20 columns.

This happens to be sprite sheet of a simulated thunder strike or lightning.

If you have individual images and you would like to combine them into a spreadsheet, there are tools that will allow you to convert a number of individual sprites into a single spreadsheet.

Simply do a search on convert PNG files into spreadsheet using your favorite browser.

All right.

So this was just a very, very quick introduction.

Now let's have a look at some code.

All right.

So first, we'll quickly just demonstrate what is happening here.

So when I click, we have an explosion.

If I click here again, we have an explosion.

I can click a couple of times and we have all these explosions happening.

All right.

This is done in a couple of stages.

So first of all, in our Flame game class, in the on load method, we are using an images dot load function, which is part of frame gain to load our image, which happens to be a spreadsheet.

The system doesn't know this at this point.

It's simply just loads in one image.

The image is name is boom up pag.

And you can see right here that inside the assets we have boom dot PNG right here.

All right.

So what this does is it loads the image into a cache.

It doesn't display or do anything yet.

Let's have a look at what happens when we tap on tap.

What we're doing is we're getting the position of the tap.

We then subtract the size of the image itself to adjust for how the sprite animation is rendered.

The size of our image is 128 pixels by 128 pixels that usually offsets the location.

So we're just adjusting that here.

Then what we do is we create a particle component because that's the only thing that we can add to our engine.

A component and our particle component is going to wrap around an accelerated particle.

We're going to set the life span to 3 seconds and we're going to set the position to this position that you've seen just above.

And the child is going to be a Sprite animation particle, and the animation is going to be the sprite animation that we're loading.

The size of our animation is going to be 128 pixels by 128 pixels.

And then we add the particle at the bottom, and this is what puts the particle into the game loop.

And of course it gets rendered.

The star of this code is to get boom animation.

Let's have a look at that and notice that it returns Sprite animation.

In here we specify that we have eight columns and eight rows, so we are telling the system how our sprite sheet is divided.

We will specify how many frames there are.

Of course there is eight times eight or 64 frames.

Then we're going to load the whole spreadsheet from the cache.

As you remember, we have loaded it initially in our load method and then what we're going to do is once we have that image, we're going to create a sprite sheet out of that image by using the sprite sheet class and the from columns and rows method.

And this simply takes us input.

The sprite image takes the number of columns and the number of rows.

So now that we have loaded the image into our sprite sheet, of course to do the animation, we need the individual sprite.

So the next line that you see, which is the final Sprite is equal to list.

What we're doing is from the spreadsheet, given the number of frames we generate, the actual frames of the sprite.

And then what we do is we add that list of frames to the sprite list of the Sprite animation class and the step time that we specify.

If I just hover over this, we're basically saying that the step time is going to be 0.1.

All that this means is that all of these are going to be going one after another with the same amount of time in between.

In fact, this part will not be controlling how fast our animation is moving.

But what this does is it creates the sprite animation.

The Sprite animation then is added to our Sprite animation particle, and that starts the animation.

So when I click here, we have this lasting 3 seconds.

Now what I can do is I can change this to 2 seconds and you will see that it's going to explode much faster, which is a little more realistic.

And I can make this a little larger.

I could, for example, make this 200 by 200.

Of course, it's going to get a little pixel sized.

And as you can see, it is larger.

Now you can see that it's a little offset where I'm clicking.

See how it's below it.

That's because we need to make that adjustment here of 200.

And if I do that now.

There you go.

We have our explosions.

All right, so this is basically it.

I have also included with the code, this lightning sprite sheet.

You can play around with it and see if you can load that up.

All right.

See you in the next lecture.

9. Case Study #7 - Exercises

Let's have a look at some exercises for you to do after the case Study for particle systems.

Exercise one, Create a ground level explosion.

What do we mean by this is create an explosion effect as if the explosion happened on the ground.

So what that means is the explosion should shoot above the ground.

All the particles should be moving up and away from the ground.

And also think about how would you control the splash area.

This is the area of how far the particles would travel away from the epicenter of the explosion.

So think about a scenario where you might have a spaceship or something else falling to the ground and then on heading the ground it explodes.

Exercise two is a little more ambitious.

Modify the joystick code with the spaceship and the ball's bouncing around that the spaceship was shooting

at from your earlier exercises so that each time the ball is hit, you render the particle explosion

effect.

Then for added bonus work, modify the above code that you just created so that each time the ball is

hit, you render the sprite particle explosion that we covered in the code.

Have fun with this and if you have any questions, do not hesitate to ask.

See you in the next lecture.

10. Case Study #7 - Solution to Exercise #1

All right.

Welcome back.

Let's have a look at solution to case study number seven, exercise number one.

The exercise asks you to create a ground level explosion.

And the idea was to just create an explosion that shoots above the ground and has a certain splash area.

But the key difficulty in this exercise is just controlling the particles to not go below the explosion epicenter.

Let's have a look at our PDF document.

The idea is to create an explosion that starts with an epicenter and then goes in random direction, whether it's left or right and it falls within this splash area.

In other words, you can see those two red areas.

Those are the two quadrants.

We want to confine our random vector generation to those quadrants.

We also want to make sure that the lifecycle of each particle is such that once it goes up and then comes down, it does not go below that black line, which is where the origin point lies.

For example, if you're just looking at a single particle, that would be the idea.

The particle starts somewhere at the epicenter, it then goes up and then it has gravity and maybe like wind or air friction bring it down.

So we can kind of think of this problem as being simply the problem of generating a random directionality vector, which would be shown here with the red arrow and then having a deceleration vector that works against it, which is the green arrow that kind of brings it down to the ground.

So that's one element.

And the other element is to make sure that once you come to that black bar there, that the particle disappears because the idea is that the explosion should only go above the ground.

It should not go below the ground in this solution.

What we're going to do is we're going to use the accelerated particle that makes sense.

The accelerated particle generates that kind of behavior.

But in order to control H particles position, we need to have access to position.

So let's have a look at it from a perspective of a UML diagram.

I'm going to switch right now to a better view in the middle, which is the main part that you see here, which is the particles, the dart.

What we have is the QML diagram of an accelerated particle.

Have a look at what the relationship is so we can see that accelerated particle.

It extends curved particle and it also works with a mixing which is single child particles.

So those are the two dependencies.

If we go a little further, you can see that curved particle extends from particle.

And if you look at the particle abstract class itself, there is no position there.

All that we have there is the generate function and we have the progress which basically gives us more of a life span.

There is no position there.

If you look at the curved particle, you can also see that the curved particle does not have a position, neither does curve.

So the only place where we can see that there is a position is an accelerated particle, because the accelerated particle computes the path of each particle by controlling its position.

So in order for us to be able to control when the particle is not visible anymore, we have to be able to look at the particles position.

And then if the particle happens to be just hitting the ground, we need to turn that particle off or remove it.

So the way that we do that, actually, since we can't do it directly through accelerated particle because we cannot tell the accelerated particle to turn itself off, what we do is we extend it, we create a custom accelerated particle.

All that the custom accelerator particle does is it extends the accelerator particle which gives us the position we automatically inherit that.

What we also do in the custom accelerator particle is we add a special variable, which is the origin position.

That would be the origin of the explosion.

And then one other thing that we have is we have a function which is a test position, threshold function.

That's simply just a definition of a function that we can then add into the custom accelerator particle that can tell our custom accelerated particle class when a certain threshold in the position has been met.

And as you can see, the function returns a boolean.

So the idea is that we're going to inject into the custom accelerator particle our own test for the position, which is going to check if the position is below the origin position in the Y direction,

then in the update and render, we basically turn the particle off.

So this is the WML diagram.

Everything that you've seen in the WML diagram has been explained inside this document.

So you can follow this for more explanation.

But now let's have a look at the actual code, what we did for.

First with the code is we first played around with a single particle.

So let's look right here.

What we have right now in our particle component generator, what we have is we have simply one particle.

We give it a life span of 10 seconds.

Notice that we generate the random vector, which is going to give us those two quadrants that we're looking for.

As you can see, the outline of those two red rectangles.

This is where we want to generate our vector.

So this random vector up our quads gives us exactly that.

We then scale each one to the same velocity of 300.

Then we also have something which is the deceleration, which is that green vector that we have seen in right here.

If you look here, the deceleration is that green vector, it's the vector that is acting against the vector that is going up the initial velocity.

So what we have is we have the initial speed being that red vector.

Then we have our deceleration is the green arrow.

And then what we do is we simply create our rendering particle to just be a simple circle component with radius four and a red paint.

And then all that we do is we return our custom accelerated particle.

We initialize it with this remove particle test function, which is right here.

Remove particle test is pretty straightforward.

It basically checks the threshold position, which in our case is the origin and it checks the current position.

And if the current position in the Y direction is greater than the threshold position in the Y direction, which is basically in the vertical axis, if it's greater, that means it's below.

Then we return true by saying that the particle should be removed.

So let's have a look at this in action.

So if I click here right now, watch what happens with the particle.

It goes up and it disappears.

Another one goes up and disappears.

Another one goes up and disappears.

And if I click it many, many times, you can see that the all disappear when they come to the ground level.

Let's generate 25 and I'm going to make the radius a little smaller.

Let's make the radius two and we're going to rerun it.

And here it is.

And now if I click anywhere, we have what looks like an explosion that is only moving up.

And then as you can see, all of the particles disappear just at the ground level.

So this is obviously a relatively simple way of working with this.

As you can see, we can control certain aspects of this.

So, for example, if you look at the random vector upper quad, what we do here is we generate the X and we divide it by four.

That basically confines the X to a much closer range.

Then we also have our Y, we multiply it by 0.75 so it doesn't go as high.

We have another random generation here.

We basically decide whether the vector should be going left or right, because these next double that we see there are generating it in just one quadrant.

So by randomly selecting whether it should be going left or right gives us both quadrants, the left and the right quadrant.

This is not the only way to do this, but this is just a nice, simple, quick way of doing this.

One quick mention of the remove particle test that we have plugged in right here, you can see that the custom accelerated particle, it takes a function.

If we go into the custom particle right here, you can see that we have defined the function here at the top, it's test position threshold, and it's basically a function that returns a Boolean and takes two vectors.

As you can see, our custom accelerator particle simply extends accelerated particle.

We have the origin position, which is the position of the origin of the explosion.

And as you can see, all that we do here is we require a child.

We require also this function.

Once we initialize this, which is mostly we're delegating everything to just accelerated particle.

What we really wanted to do was just control the position.

And what we have here is a very simple update and render.

What we do is we use the function that was passed in to check whether we should remove the particle.

And if this is true, then we do nothing, meaning we don't allow any more updates.

So the particle doesn't show up anymore because we're not processing it.

You can look at it as follows If we don't have to remove the particle yet, then we update and in render.

If we don't have to remove the particle, then we render.

Otherwise, if the particle should be removed, we choke off the processing of both update and render.

So the particle disappears.

So this is a pretty simple solution.

It's a bit of a quick solution.

There are better ways of doing this, a little more architecturally sound, but for a simple exercise

to understand how particle classes can be extended or worked with, I think this is quite good exposition.

So I hope you enjoyed this.

And as always, keep on coding.

11. Case Study #7 - Solution to Exercise #2 - Segment #1

We'll come back.

We're going to look at case study number seven, exercise solutions specifically.

We're going to look at exercise number two.

We're going to break down this particular solution into a number of segments.

The reason being that this is a relatively involved solution that takes a lot of different pieces from

what we have learned in the course and puts them together.

And by pieces we mean a lot of the pieces of the code.

This segment is going to be an introduction to the solution and a demo of this solution.

All right.

So let's get to it.

As always, we have provided a PDF solution for you to look at first to exercise number two.

The first thing that you will see is that in our assumptions, what we're saying is that we're going

to reuse pieces of code that we already have.

They are in different files, they do slightly different things, so we might have to modify them.

But we do tell you in the solution here what the modifications are going to be.

You will see here that we list a number of code bases.

You can see that we have case study five, which was timers.

We're going to use that for the light bar and we're going to use that for the timers so that we can generate the balls.

You also can see that we're going to use X size 003001, which is basically case study three.

We're going to use that for the space ship, the bullets and the joystick control as well as sound effects in the video and in the source code.

We do not actually use C, which is exercise solutions component 006 Parallax.

That's something you can add as a mini exercise for yourself.

That's pretty straightforward to just add the parallax, but we do use D and E, as you will see for both the particle explosion which we use for the asteroids or the balls being destroyed, and also the sprite based animation, which is more like a cinematic animation of the explosion that we use or the spaceship explosion.

So let's have a look first at what the actual code looks like when it's running.

Here's a little demo of the x size itself.

You will notice that everything is a little smaller.

We have actually shrunk the sizes of the spaceship as well as some of the other things.

That was just a very quick demonstration of the exercise.

What we're going to do right now is we're going to take all the different exercises and the different code samples that we've used throughout the course, and we're going to put the solution for this exercise using all of those.

Let's go step by step by modifying some of the code first and then going through the rest of it.

12. Case Study #7 - Solution to Exercise #2 - Segment #2

Welcome back.

This is going to be segment two of exercise number two, solution to the case, study number seven.

What we're going to do here is we're going to modify the solution code to case study number five, exercise

number two.

What are we going to do is we're going to remove all the ball bouncing logic and we're going to provide ball wrapping logic.

So basically we're preparing case study number five, exercise number two, to be more like what the asteroid's game is going to be, which basically had the asteroids are just moving around the screen. They're kind of passing through each other and they also wrap around the screen, meaning that if an asteroid goes to the left edge of the screen, it reappears on the right edge of the screen.

So that's exactly what we're going to do here.

We're going to take the solution to case study number five, Exercise two, which basically just has a number of balls bouncing around the screen.

If I run this.

You can see that we have the usual behavior as we had before.

The balls are bouncing around.

And what we want to do right now is modify this code so that they're not bouncing off each other.

So I'm going to stop this.

We're going to go into my collider ball and we're going to go into the update method, which is where we're actually controlling all the bouncing.

As you can see here, we have the X and Y being calculated and that is based on the direction.

So what we did before is we basically check whether we pass the left or right screen or whether we pass the top or bottom screen edge, and we basically reversed the direction.

This is what gave this code the ability to bounce the balls away from the screen's edge.

What we want to do is we want the balls to wrap around the screen.

So I'm just going to add a little bit of comment here.

And we're just going to wrap that.

So all that I'm doing here is basically I'm checking if we have moved to the left of the left edge of our screen.

What we want to do is we want to move it then to the extreme right.

So what I'm going to do here is simply change the X to be equal to that, and there's no need to change the direction.

The direction will be the same.

In other words, if it was going to the left, we just wrap it around and it's still going to the left.

It's just going to come from the right edge.

We're going to remove all of this and we're simply going to say X is equal to zero, which means that we're wrapping the ball to the left side.

We're going to do exactly the same thing for the bottom.

And we're going to do exactly the same thing here.

And here we're simply going to change the way.

The next thing that we're going to do is we're going to remove the code that deals with when two balls are colliding.

So we're simply going to remove all of this.

And we can leave the rest of it as is.

So let's run this and see what's going to happen.

So the expectation is that now when the balls are generated, they will not be bouncing off each other and they will simply wrap around the edges of the screen.

So they will just constantly be flying, but they're going to be flying through each other.

They're not going to be bouncing at this point almost there.

So here's our first ball.

And as you can see, the wrapping around.

But for some reason we still have them bouncing off each other.

So I'm going to stop it and we're going to see why that is happening.

So let's go back into my collectible.

And of course, we have to go in on collision and we're going to remove all of this.

So basically everything from my collider ball down, we can remove that.

We don't need any of this right now.

So I'm simply going to remove all of this.

At this point.

We're not going to have any collision code.

All right.

Let's run this one more time.

Let's have a look.

So this will generate a few balls right now and they should all be wrapping around.

Perfect.

They're not bouncing off each other.

So this is good.

There's one relatively annoying thing, though, when you're looking at how they are wrapping around.

You can see that the moment they touch the edge of the screen, they automatically get wrapped around when it comes to the aesthetics of this.

It doesn't really look that great.

For example, if you look at this 003 100%, it's just about to hit the edge.

When it hits the edge, it automatically wraps around.

What we would like to do is we would like to have it wrap around after it leaves the screen with the rightmost edge of its body, so to speak.

This, of course, depends on where the anchor is.

So if you want to make the calculation, but the necessary calculation is going to be relatively simple.

What we're going to do is we're going to check whether it has moved out of the boundaries by the length of its body.

So, for example, if it's going to the left, we're going to wait until it is the length itself to the left.

And when it's on the right, it's going to be the length of itself to the right, which is on the outside.

So let's modify that code.

So right here, what we would like to do is not when direct left is less than or equal to zero, we want to make sure that it's less than or equal to negative size of itself.

And in here.

We're going to say.

Plus.

And we're just going to say.

Right.

Okay, let's see if this is going to work by running and just checking the left and right.

All right, let's have a look.

So we can see that when the ball is moving exactly to the left or right, that seems to be working relatively well.

All right.

So what we're going to do is we're going to do exactly the same.

For the wide direction.

So let's make that change.

All right, let's run this.

There it is.

Let's have a look how it's going to behave right now so we can see that some of them seem to be working pretty well.

It seems that the ones on going to the left are not working well.

So, for example, when you're looking at the one going up, you can see that it disappears the moment it touches the edge.

But on the other hand, if we have something that is going down, it seems to be going very nicely.

So look at the one on the left right now.

See the moment it touches the left edge, it disappears.

And the fact that we only have a couple of objects right now means that we have lost a few objects somewhere.

Some objects are not showing back up.

Okay, So let's have a look at our code again.

Going to change all that to width here.

And here we're going to change that to height.

Okay.

Let's have a look at how this is going to run.

And the idea in an exercise like this is that one of the most beautiful things about computers is that you can try a couple of different ways of doing things and see how they're working out and then modify the code to do what you want it to do.

So let's have a look how this is going to work.

Let's look at the 0404 went very nicely.

This looks much better, much better now.

And we're not losing objects.

They seem to be taking a little bit of time, but this seems this is much more pleasing to the eye because there's no abrupt change.

So now the objects, like when they move to the left or to the right or to the top or to the bottom, they don't just suddenly disappear.

They disappear slowly, and then they show up on the other side.

So we're just making sure right now we still have all of our objects.

This is pretty good.

Okay, so this is what we wanted to accomplish.

So this is the first step in creating this exercise.

Now that we have these objects, what we're going to do is we're going to bring in the code that we had for our joystick.

So we will do that in the next segment.

13. Case Study #7 - Solution to Exercise #2 - Segment #3

Welcome back.

In this segment, what we're going to do is we're going to merge some code bases.

We're going to take the case study number three, exercise number one and two code, which is the joystick with the sound and the spaceship.

We're going to use that as our basis, our source for the whole exercise.

And then we're going to take the previous code that we remove the bouncing logic from and to which we added the wrapping logic to, and we're going to merge it with this one.

So we have a larger code base.

Let's get to it.

What we're looking at is the source code for exercise, for case study three, which takes care of exercises one and two.

As you remember, this is the code where we have the joystick player that is able to shoot bullets.

When the bullets hit the boundary, they disappear.

And we also have some sound effects.

We're going to use this code as our base code.

We're going to bring all the other pieces of code in here.

We're going to take the code that we have just created in the previous segment where we have modified the ball's behavior so that they don't bounce off each other and they wrap around the edges of the screen.

So we're going to bring that in.

We're simply going to first drag and drop those particular pieces of code that are completely independent.

So, for example, Michael Edible is a completely independent piece of code, so we can just bring it right into our lab.

And there it is.

Of course it's not going to compile right now because it is dependent on the life bar.

So we're going to add that as well.

Now, before we fix all of this, let's look at the live part.

The reason why we can't compile it is because we need to bring in the dependency, which is the international

DART package.

So what we're going to do is we're going to go into pub spectra and we're going to add the international dependency.

I'm just going to drop it right here just to make sure that it's properly indented.

Okay, perfect.

So now if I save this, you will see that flutter is going to execute the pup get.

And as you see, life bar text is properly compiling now because this package is recognized.

So the last thing that we need to do is in my collider ball because this obviously we took it from a completely different project.

We need to change what game it refers to, which is the has game ref.

So we're going to go into our main dart.

We're just going to grab this name.

And we're going to drop that in here.

And that is basically it.

So right now what we have is we have the code from the previous segment where we took case study number

five.

We modified it and we put it into our case study number three, exercise one and two.

Of course, what is missing is that let me just get rid of this pub spec.

What is missing right now is in here.

We had our own utilities.

So let's have a look at what the utilities are here and we can basically just do the outline here.

So we can have a look at what are the utilities we have generate random position.

Random velocity is position out of bounds and position.

These are the functions that are used in here.

If we look at the utilities from before, I'm going to bring in another one, which is generate random speed is going to drop it here.

So we have random velocity.

I'm just going to drop this one, which is a random speed.

Okay, So now that we have this, the last thing that we're going to do is we are going to take the timer that we had.

So in the previous segment that we're looking at in our main dart, we actually had a timer.

So what we're going to do is we're going to add this timer here.

So we're just going to copy a few pieces.

So this is a number of simulations.

Here we have the interval timer.

We're also going to output the same pieces of information as we did before.

And we're going to import that.

We're also going to get the elapsed number of ticks.

Put it underneath here.

And then what we need is we need to get the actual timer initialization and this is going to be happening in our own load.

So we're going to go into the unload method right here, and I'm going to put that as the first thing that we do.

Of course, we need to get the utils, so we're just going to import that.

Looks like we're missing the random direction, so I'm going to have a look for it.

Just going to add it into.

Utils.

So we have here we have the, we have the position velocity and I'm going to put it right after velocity.

It's going to added right here.

I'm just copying it from the previous one.

So here it is.

Save it.

And if we go now here.

Let's have a look here again.

This all looks good.

For some reason it's unable to find it.

We have it right here.

Generate random direction.

That's what we're looking for.

So this seems to be a problem.

Okay.

What I'm going to do is I'm just going to restart.

Sometimes Visual Studio seems to have a problem just to restart it.

And as you can see, the only problem that we have right now is with my collider.

If I go here quick fix, we'll go import the library and we're good.

We have everything.

I'm just going to remove this.

We don't need that.

Okay.

So we have everything seems to be compiling.

One other thing that we need to do is that once we go back into main dart, once we have created the interval component right here, what we need to do is we need to add it here.

So I'm just going to say.

So they're simply going to do this.

And now we should have a pretty much working example.

So let's save this and let's run it.

Okay, so this is about to run.

So what we should be seeing right now is that balls are being generated.

Okay, so there's some sort of a problem.

Balls are not being generated right now.

Okay.

So we'll need to check why that is.

First, I'm going to get rid of the music.

Just for now.

And what do we need to do here is we need to add the collectables.

One other thing also is that we need to from our collider ball.

Because we have removed that capability of making sounds.

We need to make sure that we remove anything that actually loads.

So, for example, right here we have this part which is loading.

We don't have that asset and we don't need it.

So we're just going to remove that here and then.

We're going to look if there is any point.

There's nothing here.

This is good.

This is pretty good.

Okay.

So we can rerun this again.

Okay.

So as you can see, balls are being generated right now.

The next thing is going to be to avoid hitting them.

And.

Basically shutting them out.

So that whenever whenever we shoot a particular ball, that ball, for example, gets destroyed or whenever the ball actually goes over our.

Ship, it destroys the ship.

So the idea would be to try to navigate as much as possible a way while you're shooting.

And obviously we have you know, the density of the balls is way too much.

We can, for example, increase the interval.

So let's have a look at how we would increase the interval so that we don't have so many.

If you go into the main part, you can see that the number of objects is ten, and if we go into where we use the interval, what we're doing here is we specify the period as being one.

So for example, we can make the period to be four, which is going to be 4 seconds.

So if I rerun this right now, what will happen is that every 4 seconds you should start seeing a ball.

So there's one, two, three, four, another one just popped up and then we should have another one.

So this will be a little more manageable.

Every 4 seconds they're going to show up so you can kind of navigate and not get destroyed and you can then shoot them.

We're going to add into this collision detection so that when the ship is hit by any one of these balls, the ship explodes.

Or if the ship hits with a bullet, any one of these balls, we, for example, will either explode the ball or just start removing the life from the ball.

Okay.

We will do that in the next segment.

14. Case Study #7 - Solution to Exercise #2 - Segment #4

All right.

Welcome back.

In this segment, number four to case study.

Number seven, exercise number two solution.

What we're going to do is we're going to add some collision detection.

Specifically, what we're going to do is we're going to detect when a bullet and a ball collide.

And then we'll also detect when a ball and a spaceship collide so that if a ball or an asteroid hits the spaceship, the spaceship should disappear or explode.

All right.

Let's get to it.

So what we have here is the code is running right now.

We have our spaceship right here in the middle.

We have the other objects, the balls being generated, randomly generated.

And of course, I can do.

I can shoot what I'm going to do right now just to avoid distractions.

I'm going to do two things.

I'm going to stop this.

So first of all, what I'm going to do is I'm going to make the debug mode to be true so that we can see all the head boxes.

I'm also going to.

Remove the sounds themselves.

For now, I'm going to remove this and same thing here.

This is the.

Sound of the explosion.

We're going to reuse it a little bit later.

So what I'm going to do right now, let's go back to our main dart and we're going to run it again.

This time we should be able to see all the hit boxes.

All right.

Here it is.

So as you can see and you will see, balls are being generated right now.

You can see that there is a number of hit boxes all around them.

So, for example, our spaceship has a hit box and then the ball has a hit box.

And the text above it also has a head box.

And if I shoot right now, you can see that the bullets also have head boxes.

We have all of the necessary elements together in one code base.

Great.

But what do we need to do right now is when I shoot and the bullet, for example, hit something like, for example, right here.

Well, I missed I guess I'm not so good at playing this.

If I shoot the if I shoot the bullets and they hit something like, for example, that one right there.

Well, once the ball is hit, we're going to remove it.

But of course, what we need is collision detection, which right now we do not have.

So let's stop this code for now.

What do we need to do?

We have basically three main objects.

We have the ball, as you have seen.

We have the spaceship and we have the bullets.

There is a relationship when it comes to collisions in a sense that a ball can collide with another ball.

We're not interested in that.

A ball can collide with a bullet or a bullet can collide with a ball.

We want that.

And also a spaceship can collide with a ball.

We also want that.

But first, let's do the following simple addition.

First, we need to wire the bullet to be collectible.

As you can see, the bullet right now is not going to be part of the collectible contract.

So what are we going to do is we're going to add two more mixes.

One is going to be has head boxes and the other one is going to be collectible.

Perfect.

What this tells our code is that this class, which is the Bullitt class, is going to be part of the collision detection within the flame engine.

But before any collision detection can actually happen, we still need to add a hitbox to our bullet.

So what we're going to do is when we're loading the bullet right here, we're going to add a hitbox.

And the way we do that, we just use a simple method, which is add hitbox and the add hitbox method.

What it does is it takes the shape that we want to overlay as a hitbox for our component.

In our case, we're going to simply add the hitbox rectangle, which is going to grab the size of the component.

And basically that's going to be our hitbox.

Now that we have wired our bullet component and we have specified the hitbox, which is basically just this 4x4 bullet.

What we're going to do next is we're going to go into Michael Ball.

And at the very bottom here, we have our own collision.

So what we're going to do here is we're going to basically check if the other what we have collided with.

If it's a bullet, then we should remove this ball.

Michael, table is our ball.

We have to remove this ball from the game.

So what we're going to do here is we're just going to say if if the other is of the type bullet.

So in other words, if we were hit by a bullet, what are we going to do is we're going to remove this from the parent.

There you go.

We should also remove the bullet.

So let's go to copy this.

And we're simply going to say remove other.

So what's going to happen now is when I run this, what you would expect is that when a bullet hits any ball, that ball should be removed from the screen and it also will remove the bullet.

So let's run this.

There you have it.

Now we're just going to bring the ship down and we're going to wait for some of the boats to show up.

And I'm going to let's see if I can missed it.

I'm just going to go to the bottom here.

Boom.

As you've seen, it was removed.

Let's see, I'm doing this on the emulator, so it's not the easiest.

Again, as you see, it was removed.

I'm going to get this one now removed.

I'm going to get that one removed.

So as you see, the code seems to be working pretty well.

One of the things that I'm going to also do is I'm just going to remove that camera shake that camera shake happens in the bullet.

We're going to use this camera shake a little bit later.

It's going to remove it right now.

So let's just do a very quick recap of what we've done.

So what we want to do is we want to make sure that whenever we fire a bullet, if that bullet collides with our ball object, we want the ball object and the bullet to be removed.

So what we did first is in order for a component to be part of the collision detection framework, you have to add the collider ball.

Nixon And you also have to specify the has head boxes mixed in.

Once you do that, it is still not fully wired.

What you need to do is you need to specify the hitbox without specifying the hitbox.

There will be no collision detection.

If I remove this and you do the shooting, what you're going to see that inside my collectible.

This will never be triggered because there is no collision detection if you don't have a hitbox.

So we're adding a hitbox here.

And when you use this class hitbox rectangle, the hitbox rectangle is just a simple class that allows you to create a hitbox.

That is exactly the rectangular shape that would go around whatever this particular position component is.

So in other words, your position component has a size right here.

And whenever you say hitbox rectangle, what the hitbox rectangle does is it goes into the parent class, which in this case is bullet checks what the size is, and then creates a rectangle that goes around that size in order for our spaceship to be destroyed whenever it's hit by, for example, one of the balls are like an asteroid, We have to do the same thing.

That collision has to be detected.

So we're going to do exactly the same thing here.

We're going to add the has head boxes mixed in and we're going to add the collider ball mixing.

As you remember, this is not enough in the on load.

What we need to do, we need to now add the head box.

So we're going to say add right here, head box.

And if I simply say right now, hitbox, rectangle, all that this will do is it's going to basically look at the size of the joystick, which is this one right here.

You can see the size is 50 by 50 and it's just going to create a 50 by 50 rectangle.

That's all it does.

So now that we have wired our joystick player, which is our spaceship, now that we have wired it for collisions, what we're going to do is we're going to go back here.

And we're going to basically just repeat this code.

But this time we're just going to do a very simple variation.

What basically say if the other one is the joystick player?

Of course, we have to import it.

There you go.

As you can see, the import happened right here.

The import joystick player, the dart.

So once we have done this, what we're going to do is we're going to remove the joystick player and we're going to remove the asteroid.

But actually, we could say that the asteroid is not destroyed when the ball hits it.

So I'm just going to say, let's not do that.

When the ball hits the ship, the ship gets destroyed, but the asteroid or our ball does not.

So let's run it right now.

There you have it.

So here is what is going to leave this spaceship here and wait for a ball to hit it.

So I'm going to.

Put it right in the path and boom, the spaceship is gone and the and the ball keeps on going.

Okay, so this is it for this particular piece of code.

What we're going to do next is we're going to wire this so that when the bullet, for example, hits the ball, we're going to put an explosion instead of that.

So we're going to actually create an explosion part to this.

And additionally, when the ball hits our player, our spaceship, we're going to do the same thing.

We're going to explode the spaceship.

That's what we're going to do next.

15. Case Study #7 - Solution to Exercise #2 - Segment #5

Welcome back.

This is segment five of the solution to exercise number two for case study number seven.

What we're going to do in this segment is we're going to be adding explosions.

We're going to reduce the particle explosion code from case study number seven, lecture number two,

And we will use this effect for both the ball explosion when a ball is hit by a bullet and for the spaceship explosion when the spaceship is hit by the ball.

All right.

Let's get to it.

So the next thing that we're going to do is we're going to add our explosion code from particle case study to show that the bull when hit explodes.

The first thing that we're going to do is we're going to remove the debug mode so we don't see all of those head boxes.

So let's.

Restart it.

Perfect.

So now we have a slightly more normal.

And.

And we can shoot.

Okay.

So this is good.

So now what we're going to do is we're going to add the code from the particle system case study, which was the case study number two, where we have created that kind of particle explosion.

First, what I'm going to do is I'm going to actually create a new class.

So we're going to put our particle generator as a separate class.

So let's go here.

New file.

I'm just going to call it particle generator.

And I'm going to paste our code here.

Here it is.

Now, we will have to, of course, import a number of packages so we can just hover and then go quick fix and we're using the components.

Same for particle.

We're doing the same thing.

We're going to it's not showing for some reason quick fix.

So I'm just going to copy the import itself this and then the colors that's going to be in the dart.

Actually, we're going to go from material dart and then the utility is this one right here, the utils,

the dart.

So we're just going to.

And there it is.

And we're missing this method.

So what I'm going to do is I'm going to grab that method from that very same place.

I'm just going to drop it into our utility class.

This is the method that was used in the particle generator.

So when we go back here right now, I just need to.

This is good.

I'm just going to save this.

And then when we go here, this is a little strange.

Methodists clearly here.

Okay.

What I'm just going to do is I'm just going to restart.

Sometimes you have to, unfortunately.

Restart Visual Studio, because sometimes Visual Studio is not a smart.

So now, as you see by simply restarting it, it properly understood that we needed that.

So everything seems to be compiling properly right now.

So what we have done here is we have created a particle generator, which in this case creates our explosion.

We're using this function which we have also gotten from the same source code.

But what we did to make it a little easier, we have simply put it inside our utilities class.

So now that we have everything, what we need to do right now is we need to go into my collider ball,

which is the ball.

And in here, when the bullet hits our ball, what do we need to do is we simply need to add the following

code, put it right here.

What we're doing basically is we're going to add to the parent.

From the particle generator we're going to call the Create Particle engine method, which is going to basically create this explosion.

Of course, what we need to do, the reason why it's complaining is because we have not imported the particle generator.

So if I just hover over here, quick fix, as you can see, it recognizes that we should import that this is done.

So right now what's going to happen is that when the bullet hits our collider class, which is the ball, we're going to create the particle explosion and we're going to remove this, which is the ball.

And also we're going to remove other which is the bullet.

Okay.

So let's give this a try.

Let's run it and let's see what it looks like.

Okay, so now when the balls are generated, you just have to wait a little bit.

Perfect.

I'm going to shoot right now, and boom, we have an explosion and let's wait.

Boom.

We have an explosion.

Okay, so this works.

Pretty well.

Of course we can.

We can play around with this a little bit more.

We just got destroyed.

Och, this seems to be working pretty well, so the next step is exactly the same thing.

We can basically add.

This very same code and just put it right below, which is going to create an explosion when our spaceship is hit.

So if I rerun this right now.

If I put myself, I'm just going to wait for.

And.

There you go.

It exploded the ship.

We could put slightly different code for the different explosions, but the idea is that we have all

the pieces right now to create the game.

One interesting thing that you should notice that notice that when I tap it still is shooting.

This is obviously something of a bug.

We should only have the shooting when the spaceship is up.

So obviously this game, or rather this exercise still has a few bugs.

But the main point of this is to illustrate how to put all the different pieces together so that we can kind of start thinking about having having a game.

All right.

In the next segment, we're going to add a slightly different explosion for when the spaceship is destroyed.

16. Case Study #7 - Solution to Exercise #2 - Segment #6

Welcome back to Exercise number two Solution Segment six.

This is for the case study.

Number seven.

What we're going to do in this segment is we're going to add more explosions.

Specifically, we're going to reduce the sprite particle explosion code from case study number seven, lecture four.

We will use this for the spaceship explosion when it gets hit by the ball.

All right.

Let's get to it.

What are we going to do right now is we're going to modify the explosion visualization for when our spaceship explodes.

So as you remember, we have just put this piece of code to create a normal particle explosion for our spaceship.

The first thing that I'm going to do is I'm going to remove this because we're going to put different code in here.

I'm going to rename this method.

Create particle engine is not a very good method name.

I'm going to actually call it create particle explosion.

So I'm going to go into particle generator.

I'm just going to change it here and then we'll go back here and just change this.

So all I've done is basically just made a name change, so it's a little easier to understand what's

going on.

The next thing that I'm going to do is we're going to use use Case seven collision detection, lecture number four.

That was where we have used a sprite explosion that looks very photorealistic explosion.

So we're going to add that I'm just going to grab the image that we will need for this particle explosion so you can look right here.

I'm just going to expand a little bit.

This is the actual image that is inside that particular project.

So what I'm going to do is I'm simply going to drag it right into images right here.

So now that we have this asset, the next thing that we're going to do is we're going to move that whole code from case study number seven, lecture number four into particle generator.

The first thing that I'm going to do is I'm going to grab that get boom animation helper method in that particular case study.

I'm going to drop it in here.

This is a helper method that is going to be used in the animation component.

First of all, we need this spreadsheet, so we're going to get it.

This is from the sprite.

That's good.

Now this images, unfortunately, that comes from the Flame Game class.

This piece is right in here in main part.

So we will need to pass this to this particle generator.

So what I'm going to do is I'm just going to basically just go images.

That's what we need to do.

Now.

This is going to be coming from the assets and we're done.

Now, what I'm going to do is I'm going to actually create a method which we can then call for the explosion,

and I'm just going to rename that method.

So here is our method.

I'm just renaming it to create Sprite particle explosion as opposed to what it was before.

And as you can see, my get boom animation already has the images.

This will be called from within classes that have access to the game ref mixed in so they will pass

the images that we can then pass to the get boom animation.

We'll need to make sure that this image right here.

This boomed out Pag, which is this one right here.

I'm going to go into our main dart and as you can see in our load, we don't have any image loading at this point.

So I'm just going to basically say that we're going to load or pre load the image which is used by our animation.

The next thing that we will need to do is we'll need to now use this animation within our ball.

So all that we need to do right now is make a very similar call like what you have here.

But instead of particle generator, create particle explosion.

I'm just going to grab this.

Instead of create particle explosion, we're just going to go create the sprite particle explosion.

So as you see, it is complaining right now about this particular method.

So the biggest problem that you see right now is that we have simply just copied the other function signature, the Create Sprite particle explosion signature.

If you go back here, it actually requires that we provide the images.

So how do we do that in my collider?

Well, what happens is if you see here, you can see that Michael table class it has game ref so we can use this to get access to the main class.

So what we're going to do here is we're going to pass, we're going to say images, which is what we're we're passing.

That's the named attribute.

And then we're just going to say game ref dot images right there.

And we're done.

So now what we're going to do is we're going to save it.

And I'm going to go back into Main Dot and we're going to run it.

And let's have a look what it looks like.

At this point, What you should see happening is that when the spaceship explodes, we're going to have a different type of explosion than what you would expect from the ball exploding.

The ball exploding uses particle explosion.

This one uses a Sprite particle explosion.

There you go.

A little cleaner now.

We're just going to wait for.

There you go.

And as I crashed in, you could see that we had the explosion.

So that's it for this segment.

We have a number of different sound elements in this exercise.

We have the background music, We have the bullet firing.

We have also overlaid a sound for when the bullet was flying.

And we also have the sound of the explosion.

We're going to assign slightly different volume values to each one of those sounds.

So they're not all fighting each other.

We're going to do that in the next little segment.

17. Case Study #7 - Solution to Exercise #2 - Segment #7

Welcome back to Case study number seven, Exercise Solutions.

Specifically, we're looking at exercise number two, and this is the solution, Segment seven.

What we're going to do here is we're going to see how we can tweak sound levels.

Specifically, we're going to put sound back into the code because we've taken it out in the previous segments.

But what we're going to do is we will micro-manage the volume levels before all the volume levels were exactly the same.

So it didn't really give you the best kind of feel for what was happening with the different sounds.

Also, what we're going to do is we're going to scale down the game elements because right now, just for clarity, the ship is relatively big.

The balls that are flying around are relatively big.

But of course, in an actual game, all of these elements are going to be a little bit smaller so that you can have a little bit more real estate.

All right.

Let's get to it.

We are basically using four different sounds in this exercise.

We have background music, which is playing in a loop.

We then have the missile shot, which is when the bullets get shot out of the spaceship.

Then we also have a missile flyby, which is the sound that accompanies the bullet as it's traveling.
And then we have a missile hit, which is when either the asteroid or the ball in our case is hit or when the spaceship is hit.
Now, what we have done before is we have used the same level of sound for everything.
Everything was at the maximum level, but we can actually control the sounds in terms of the volume.
So let's do that.
So the first thing that we're going to do is let's turn the music back on.
So in our main dart, we have commented out this piece of code right here, which is start background music.
So I'm going to uncomment that out and I'm going to run this right now.
And when this is running, what is going to happen is it will run at the maximum volume.
So you could hear the music.
It was playing.
What we're going to do right now is we're going to lower the volume of this music by simply going and changing this.
The play method, if you hover over it, you can actually see that it says that we have string file name, which is what we're using, but we also have an optional named variable called volume.
And that by default is equal to one.
So what I'm going to do is I'm going to make it half.
So right now, if I play this, the music should play about half the volume.
So probably as you could hear, it was lower in volume.
What are we going to do right now is we're going to mix all the other sounds together so that we have the music playing and then we have all the other explosions and missile shots happening in levels that are slightly different than, for example, the music.
So the music is playing right now at 0.5, which is roughly about half the volume.
We're going to make the missile shot to be also at the same level, but the actual flyby is going to be much, much lower.
We're going to put it at 0.2, so 20%.
So let's go into where we have our shots happening and it's right here and the unload for the bullet.
So I'm going to uncomment these two for the missile shot.
We're going to make it exactly the same as the Sound of Music.
But the flyby.

We're going to make it about half of that.

Additionally, we're going to put the missile hit, which is when the spaceship or the ball or the object has been destroyed.

So we're going to go into my collider and right in here we're going to put that code.

So I'm just going to copy it.

If we have the bullet colliding with our object, which is the ball, then we're just going to play the sound and we're going to put it at 0.7 and we're going to do exactly the same thing for our spaceship.

We're going to also put it at 0.7.

It's going to be exactly the same sound.

One of the other things that we're going to do is we're going to bring back the camera shake, but only when our spaceship is destroyed.

Since this is our spaceship, we're also going to do a camera shake and we're going to change the duration.

Before it was 0.25.

This is what it was before.

We're going to change it into 0.5.

Okay.

Let's run this right now.

All right.

So as you have heard, that was probably a little more balanced.

And the nice thing when you think about the volume is because we can control the volume at a level of zero, which is silent all the way to one, which is the maximum volume, You could even in a game give users the ability to mix their sounds by allowing them to set maybe the music to be a little louder, or maybe the shots to be a little louder to their liking.

And all you would have to do is basically just play around with these values right here for volume.

One other thing quickly that I would like to just mention is that everything right now on the screen is relatively large.

This game, for example, that we're going to be playing will be much better played on a tablet.

So let's just quickly have a look at what this would look like on a much larger screen.

So I'm actually rendering it right now on the Chrome Web browser, but we're going to see a much larger screen and you will see that obviously at that point you'll have a little more space to play around with.

It's not as easy to see it on the phone because our objects, the balls and the ship are pretty large for the screen.

So here we have our.

So this is this is much this is much nicer, much easier to play around with, of course.

There you go.

When it comes to the browser itself, there is actually an issue with playing background music, some sort of a bug.

But again, I wouldn't worry about it because this library that we're using works quite well for mobile devices.

One thing definitely that that you could do is you could, for example, come here and make the actual spaceship much smaller.

You could, for example, make it 25.

And then in my collider, Bill, we could also make it smaller by, for example, setting the radius to be only ten.

So I'm basically made everything half the size.

So what I'm going to do right now is I'm just going to switch to our pixel and I'm going to run it and you will be able to see how everything will obviously look a little bit smaller on the screen.

But this almost gives you like a little bit more real estate.

And of course, we should also make the bullets smaller.

But you do get the point.

Well, that was pretty short.

Got myself destroyed right away.

Anyways, you do get the point when we get into the final case study, which is the game, we're going to look at an architectural aspect, but you can basically look at all the code that we have here, all of these elements, and this will give you an idea of what the game itself is going to be.

I'll see you in the next segment.

And as always, of course, keep on coding.

18. Case Study #7 - Solution to Exercise #2 - Segment #8

Welcome back again to case study number seven, Exercise Solutions.

This is going to be exercise number two, solution segment eight.

And I'll let you know now that this is the last segment for this long exercise.

What we're going to do here is we're going to add a scoreboard.

Specifically, we're going to add a simple scoreboard to track how many elements we have destroyed.

We will also add some code to spawn the ship back after it has been destroyed.

Typically, in a game like asteroids, we would have three lives, so we will add code that will spawn the ship three or four however many times we want it to spawn.

And of course, this being the last segment in this long exercise, the next thing that we're going to do after all this is we're going to actually go through the game.

All right, Let's get to it.

All right.

So in this last segment, we're going to modify a few things.

First of all, we're going to change the information about this exercise.

This is really case study number seven, and it's exercise number two.

We're also going to change the name of the class.

So it reflects that.

And of course, we need to change the actual class name.

And then obviously you can see that my collider ball doesn't quite like it.

You can see it right here as well.

So if I go here, that makes sense because we're using the generic right here.

Okay, So this was the first thing that we did.

The next thing that we're going to do before we get into the bulk of things is we're going to remove this top bar.

So the way we're going to do that is we're just going to go right here.

So we will ensure that our flutter binding is initialized.

And then we're just going to use the full screen flame device.

Should be capital.

And we need to import this.

Okay, so this is good.

So if I run this right now, you will see that the top bar is going to be missing.

Excellent.

So this is working fine.

The next thing that we're going to do is we're going to add a couple of variables that will allow us to track things like the score as well as how many bullets have been shot, as well as how many lives

are left for our player.

But before we do that, we're going to create a new timer that will be used to respond our player, our spaceship, every time it gets destroyed.

So we have the timer, and before we actually write the code for the timer, we're going to create a couple of variables.

First, we're going to define a constant that is going to specify how many extra lives our player can have.

So we're just going to define it as being three.

There's going to be three extra lives.

Now we're going to define also the variable that is going to keep track of how many lives are actually left.

And we will initialize it with the number of extra lives.

Next, we're going to define the variable that is going to hold the value of how many bullets have been shot.

And we'll initialize it to zero.

And we're going to rename this text paint instead of text config.

We're going to just rename it into dashboard.

This is going to be used to show information about how many bullets have been shot, how many lives are left, etc. And we're going to reduce the font size to, let's say 15.

Okay, So now let's create a helper function that is going to tell us whether our player is alive.

In other words, before we can write the timer, we will need to know whether we need to spawn the player.

So we need some sort of a mechanism of knowing when the player is alive or not alive.

So let's go to the bottom and create a helper method.

I'm just going to call it this player alive.

And what we're going to do.

One way of doing this is that we're going to see whether the player, the joystick player has been removed from the children's list for our game.

In other words, when our spaceship is destroyed, it's removed from the children's list.

So that's how we're going to check.

This is not the most efficient way.

We're going to find a better way later on.

But this is a pretty straightforward way of doing this.

So we're going to use the any function to basically look for any element that is a joystick player.

We're just going to click on that.

It will create a little stub for us and we're getting the element and we're going to test whether this element satisfies our condition.

And the condition is pretty simple.

We're basically checking whether any element inside our children's list is a joystick player of the type joystick player because we would only have one spaceship at a time.

So we'll basically just say element is and we're just use the joystick player and if we have found it, then we can say that the result is true.

The result is false.

Of course, we could have just gone around not using the result, just simply returning true or false directly.

But I prefer to usually write it this way so that I can later on debug it a little bit easier.

Or if I have to expand the function, I don't have to worry about changing the logic.

So we'll leave it like this.

So now that we have this function, let's write the actual code for the timer.

So we're going to go up here where we had our other timer.

So here you can see we have the interval timer component.

What I'm going to do is I'm just going to simply copy this.

And I'm just going to remove some stuff from it.

So.

So as you can see now, we have two of them.

So what I'm going to do with this one, we're going to be using the spot timer.

So I'm just going to copy that.

And we can leave the period of 4 seconds.

That's pretty good.

What we're going to do is we're just going to inside our own take.

We're going to remove all this.

And we're going to add new code.

We're going to make a simple check.

If the player is not alive, then that means that they have been removed.

So we're going to remove the number of lives and then we're going to test if the player has any lives

left.

Then we're going to recreate the player and just add them to the game.

That's pretty much it.

Just make sure that I spelt this properly.

Let's go downstairs.

Yeah.

Just rename that to be.

Uppercase.

So if the player is not alive, then the first thing we're going to do is we're going to remove the life from the player.

So we have just removed the life.

And what we're going to do right now is we're going to test whether the player has run out of lives.

So if the number of lives left is still greater than or equal to zero, then we're good.

So we might just as well add the new player.

And once we have done this, we will simply add the player to the game.

Okay, so this logic right now should work in such a way that if a player gets destroyed within 4 seconds, the player is going to be recreated.

So the last thing before we can test this that we need to do is of course our spawn timer is a component, so it has to be added to the game just like anything else.

So right here we're going to add our spawn timer and this should be good enough.

I think this is fine.

Let's run this.

When the game starts, we should have three extra lives.

So in other words, we can get destroyed four times.

Here we go.

So I'm going to destroy it right now, a few times.

So we have some problem.

Let's have a look at what the problem is.

Of course I'm going to.

This clearly cannot be final.

All right, let's run it one more time, because we had the player as final.

We could not re initialize it.

A final element once initialized cannot be changed.

Sometimes you have to just run it and fix any issues that come around.

All right.

Let's have a look at this.

Perfect.

That's how our second one.

That's our third one.

And that's our fourth one.

And after this, we shouldn't get any more.

Okay, so let's have a look.

Perfect.

Now we still have a problem.

Notice this, if I actually should.

Right now.

You can hear that the bullets are still being shot.

So we need to fix that.

And when you think about it, the shooting actually happens right here.

And this on tap method, let me just lower this a little bit.

And the on tap method.

You can see that every time we tap, we add a bullet.

So what we need to do is just use this method right here is player, live and test.

If the player is alive and you tap, then you let the bullet be created.

Otherwise you don't.

So basically we're going to put this inside and if statement.

So if the player is alive, we just grab this and this makes complete sense.

If the player is alive and you tap, then you add the bullet, otherwise you don't.

So that should take care of that problem.

Now, the next thing that we're going to do is we're going to add some information that tells us how many lives we have left, how many bullets we have shot and what our score is.

So what we'll need to do right now is we'll need to create a few more variables.

What we have at this point is we have how many lives are left.

That's good.

And we have the number of bullets shot.

That's good.

But what we also need is the score.

So I'm just going to add another variable and we start with the score equal to zero.

So now what we're going to do is we're going to actually create the rendering and obviously that is happening in our render method.

As you can see here, we have this player angle that we're putting out.

So what we're going to do is we're going to change this.

First, we're going to render the number of lives that are left.

If there is extra lives left, we're going to show the number of lives.

Otherwise we're going to just say game over.

So the first thing that we're going to do is we're going to test how many lives are left.

So the number of lives left for the player is greater than or equal to zero.

Then we're going to render the extra lives else we're going to render.

Game over.

So let's write that code.

We're just going to grab this.

And as you remember.

We're not using ship angle, text, paint.

We're using this text dashboard.

So we're going to go like this.

And right now we're basically going to say here, we're going to say extra lives.

And we're going to output the number of extra lives that are still left.

So we're simply going to output.

Yes.

Now this is an integer, so we need to output it as a string, but we don't need it to be fixed string.

We're just going to output it as a string and that's it.

And we can remove the radius and this will output.

The number of lives.

Now we're going to change this as well.

So what we're going to do is we're just going to render it at location 2020.

We can always adjust that.

And then for the game over, it's going to be exactly the same, just different texts.

So I'm going to grab all this.

And instead of this, we're just going to render here, game over.

So this is good.

This is the number of extra lives or the game over.

But there's a couple other things that we're going to render.

So the next thing that we're going to render is the number of shots that you have so far fired.

So we're basically just going to grab this again.

I'm just going to.

And we're simply going to say shots.

And.

We already have this variable.

I just don't remember what it's called.

So let's look for it.

A number of bullets shot.

Just click here.

It takes us to where the problem is.

Number of bullets shot.

Of course, we can't put it in 2020.

So what we're going to do is we're going to output it just below the other text.

And we know that the font size if you go back here.

The font size is 15.

So we're just going to create a one pixel gap, which is going to add 16.

So this was 2020 right here.

We're going to make it 2036.

So this is good.

And then the last thing that we're going to output is your score.

And we're going to basically do the same thing.

We're going to grab this.

And we're just going to say.

Score.

And we're going to output simply the score variable.

Now we're going to put it on the opposite side.

So on the right hand side, so we can just take the width of the screen, subtract 100.

This is arbitrary at this point.

This is really we can format it a little bit better later and we're going to make this 20.

So it aligns with the number of lives.

Everything is pretty good.

One thing that we need to do is obviously we will have to update the shots and the score.

Let's run this right now just to see what it looks like.

And we're going to update that next.

Okay, Here we go.

So we have three extra lives.

We have fired zero shots and our score is zero.

So let's let's fire a few bullets.

And as you see, shots now are not being updated and neither is our score.

So now let's look at the lives.

Okay, That's good.

We have one life left.

Let's crash this again.

We have this our last live.

We have no extra lives.

Let's crash it.

It hasn't.

You can see the game over.

So what do we need to do now is we're missing the score and we're missing the number of shots fired.

So where do we check the number of shots fired?

Well, the actual shots are fired right here.

So all we really need to do is in the on top up, simply add one to the number of bullets being shot.

Now for the score, it gets a little more complicated because the score has to do with destroying the balls, which is my collider ball, which means that inside my collider ball, I have to be able to somehow access this variable right here, which is the score.

This is not the greatest way of doing this, but this will suffice since this is a public variable.

We can actually access this since we have this has game ref.

All we need to do is when there is a collision and specifically when there is a collision when a bullet actually destroys it.

So right here what we're going to do is update the score and the way we do that.

The game ref, which refers to our main class and right here you're going to see score and we're just going to do that.

So now if we run this, we should have the score and the number of bullets being updated.

All right.

So let's shoot a few things.

So our score is one.

We have fired three shots.

Four shots.

Okay, so our lives are too.

All right.

It's quite difficult to play this on on an emulator.

Okay.

So far, my score is two.

I kind of suck at this game.

Okay, so I have only one leg left.

Which is this one?

Okay, so my score is five and I got 44 shots.

So that's pretty good.

Now, one thing that we could do see right here, we have a number of simulation objects.

We could, for example, set it to 100, which means that it would actually generate 100 balls.

So that way you could play this a little longer.

All right.

So this is basically it for this exercise.

We're done.

One thing that you might be asking yourself is that when I stop the game, I don't know what my score was or like my top score for that, we would have to be able to write to some sort of a storage.

And we're going to explore that possibility in our actual game.

But the most important thing about the game itself is the logic and the architecture.

We're not going to be really concerned with figuring out the top score or writing things down to storage,

because that's really more of a generalized idea of just writing data to the data store, which is something that you normally learn pretty early on.

So anyways, we're going to explore now the full game in the next main and final case study, which is going to be the asteroids game itself.

All right.

So I'll see you there.

And as always, keep on coding.

12_architecting-and-coding-the-asteroids-game

1. Introduction To Our Approach

All right.

Welcome back.

This is going to be the crowning jewel of this course.

We're going to go over the asteroids game.

Your whole course was a preparation for this.

All right.

Let's start with the architectural and just the general overview of what we're going to create.

So this is the introduction.

So in this final case study, what we're going to do is we will design, architect and develop the asteroids game.

We will go through a number of steps which will culminate with a clean and robust game code on which you will be able to build further if you wish to do so.

The main aim of this final case study is the following.

We want to document the game assumptions and main objects and the expected behavior.

This is important in a real life project.

You do not want to kind of go in blind.

You really want to have an iteration of ideas.

Next, we're going to design the workflow of the objects in the game.

In other words, how do the objects interact with each other and what is it that they do in the game?

We're then going to architect the solution.

We're going to develop the solution, we're going to test it.

And of course we're going to deploy the game to our target platforms.

So let's talk right now about our assumptions.

But these are draft assumptions.

This is almost like the first kind of concept.

The game workflow assumptions that we can make are as follows We will have a single ship, so one ship at a time, which can fire a single bullet at a time.

In other words, you can't just tap on the screen and it will create some sort of an almost like a machine gun effect.

You have to keep on tapping one tap for one bullet.

The ship can move freely all around the screen.

We can also rotate it.

It can go left, right, up and down, basically in all the directions.

The user can fire bullets in the direction of where the ship points.

In other words, if you rotate the ship, then the aim of the ship is going to be where the ship is pointing to.

The game produces asteroids which will move around the screen.

Those are your obstacles.

If an asteroid hits the ship, it gets destroyed.

What I mean by this is that an asteroid will get destroyed and so will the ship.

Of course, we can have a variation where the asteroids is not destroyed.

So the main point here is that the ship definitely gets destroyed.

If a bullet hits an asteroid, then this is what happens.

Big and medium asteroids will split into two smaller asteroids.

If the asteroid is small, it will be destroyed.

Otherwise it will split.

Players will get three extra lives.

So we start with one life and then we have three extra ones.

So there are four lives in total.

The game will provide a bonus life in form of a UFO that randomly shows up during the gameplay.

Bullets never run out.

Both the ship and asteroids will wrap around the screen.

Bullets will not.

So you can't kind of shoot up so that it leaves the upper portion of the screen and then somehow magically reappears at the bottom.

That only works for the ship and the asteroid, not the bullets.

Asteroids will also rotate in flight.

This is a little different from the original asteroids game where the asteroids were not rotating.

We're doing the rotation as an added feature so that your understanding of the Flame game engine is a little better.

Asteroids will fly with different velocities and different rotation speeds.

Some are going to go fast, some are going to go slow.

When an asteroid is split, its speed will increase.

So if I have an asteroid like a large one and I headed with a bullet, it will split into two smaller in this case medium asteroids.

And each one of those asteroids will go faster than the original asteroid.

So that adds a little bit of a difficulty to the game.

We will track the following for the player.

The number of lives left, the number of shots that you have fired your score.

And we're also going to give you shots to kill ratio, which is a nice little indication of how efficient you are with the bullets.

We will remember the top score on the device.

So this is quite important.

This means that when you play your game and you reach the end of the game, you're going to basically get a game over.

Our game will remember your top score.

Even if you turn off your phone when you come back, you will be able to see what your top score was.

This means that we'll have to be able to write to the phone's storage.

We will also have different game levels with different difficulty levels.

We will start with just three different levels and you will be able to add more levels if you wish.

You will be able to actually design them.

These are going to be defined in adjacent file.

Each level will have its own set of asteroids and bonuses.

Which level is being played will be shown on the scoreboard as well.

So you will be able to see that you're right now in level one or level two or level three or however many levels you have each level.

And this is very important.

Each level is repeatable.

This means that when you play the game again, the behavior will be exactly the same for any given level, including bonus events.

What that means is that the first asteroid that shows up, it will always show up in the same place with the same speed and velocity and with the same rotation.

This way you will be able to go back and try to beat the level.

In other words, we're not going to be randomly generating things.

We will also add sound effects in music, but music can be toggled out.

In other words, the player will be able to go into a menu and, for example, decide that they don't want the music to be playing.

We will also provide parallax that the user can also toggle out from the settings menu.

Of course, as usual, we also have a document for you going to show it to you right here.

This is the document that comes with study number eight.

With the game.

I would suggest that you actually go through the document first or at least go through the document before you look at the code for the game.

This is just a very quick asteroid game project.

And we have some assumptions.

We have, for example, the gameplay information, we have game elements, We then have high level game specification, which tells you something about the spaceship, the asteroid, the UFO, the bullet, joystick, explosions, parallax dashboard, etc. This is kind of like an iterative process.

This is high level.

Then you go further, we have more specification, more for development where for example, we have the details about the spaceship, which is what we had in this section.

And then here we have more information such as implementation notes.

And these implementation notes might be very specific to the flame engine and we talk about like a head box here.

We talk about an abstract factory, we're talking about using a timer.

So this gives you more kind of development information.

And then if we go even further, you can see that we're actually going to split this up.

This is like the high level game architecture draft where we talk about what the spaceship it what kind of data or methods we need.

And as we go through this even further, we talk about writing JSON file data, we talk about reading JSON file data, and then we have our controller, the controller with the handlers.

There's an asteroid example here.

If you go further, it gives you a little bit more information.

And if you continue going through this, you will see that we have a lot of different information.

For example, here we have objects and functions.

So we have, for example, the asteroid description, what the actual class is going to do, what are the data elements like position velocity rotation and the type of asteroid which could be large, medium or small.

We're going to have a constructor.

So if you go through this document, you will see that there is a lot of hopefully useful information to you that I suggest you go through before you look into the code.

All right.

This was quite a bit it's an introduction to what we're going to do in this final case study.

Please go through the document and I'm going to see you in the next video.

And as always, keep on coding and even more so, keep on architecting and designing.

All right.

We'll see you soon.

2. Top Level Architecture for Our Game

All right.

So now let's talk about the architecture.

What is the purpose of architecture?

Well, the purpose is to organize the main game elements into a co-operative and cohesive whole.

In other words, we want not only the game to correctly work and behave, but we also want the code to be nice and maintainable and basically look good.

We are looking for beautiful code.

So the aim is as follows We want to provide a good source of documentation about the workings of the game as code.

We would like to make it easier to maintain and enhance that code.

And also we want to reduce code redundancy.

We want to organize the structure of all our objects and their interactions as per object oriented design paradigms.

This means we're going to use encapsulation, abstraction, inheritance and composition.

The encapsulation means that each major element of the game will have its own class.

Abstraction means that we can abstract the elements into reusable components.
Inheritance allows us to even push the reusability further by creating, for example, a base class from which other classes inherit.
And of course, composition means that you can put together smaller objects into a much larger object to create a more complex object.
And you will see all of this in the actual architecture.
One other very important thing that we're going to do for architecture is we're going to identify useful design patterns.
All right.
So let's talk about design.
Our current code, the one from Exercise for Case Study seven has the following general design.
We basically have a spaceship object, which in our case is tied to our joystick.
We have an asteroid.
In our case, it was just a ball that was colliding.
It was my collided ball class.
We will have we don't at this point, but we'll also have a UFO bonus object.
We have a bullet, we have user gestures.
For example, when the user is moving a joystick, that's a user gesture.
When the user is tapping on the screen, that's a user gesture.
We're going to have serialization in our game.
And this is important because serialization means that we have the ability to read data in object form from a storage, like, for example, a file on the phone.
And also we have the ability to write data back into storage, also using the same paradigm.
So we have serialization and dx serialization.
Serialization is when you're writing something to a particular medium and the serialization is when you're reading it back.
We also obviously have an explosion and we have sound notice that all of these are just elements of the game.
These will be, to some extent our objects, or at least our functionality as broken up into these kind of standalone elements.
Let's look at it from a different perspective.
The main question, of course, when we have different elements in a game is how do they interact with

each other?

And so far, if you recall the interaction, the interaction is relatively complicated and not very easy to maintain.

Let's have a look.

We have a spaceship and we have a bullet.

We have an asteroid.

We have, let's say a UFO bonus.

We have civilization.

Now, what you see here is that we're going to use civilization to create the levels, and therefore we're going to create the asteroids and the UFO bonus by serializing levels from a file on the phone.

So that's one interaction.

Then we have user gesture, for example, whether you use a joystick or you tap on the screen, the user gesture works with spaceship and the bullet.

Now a spaceship fires a bullet.

This is what this arrow shows.

And then a bullet basically will create a sound.

And also, if a bullet happens to hit an asteroid, the asteroid will also make a sound.

For example, it will explode.

Not only that, an asteroid will also have an explosion.

Notice that the bullet can also hit our UFO bonus, and that will also create a sound and it will also cause an explosion.

We're not done yet.

Then an asteroid could actually hit a spaceship or vice versa.

A spaceship could hit an asteroid.

It's basically the same thing.

And then what do you think happens?

The spaceship will make a sound and the spaceship will explode.

Not only that, the spaceship can hit the UFO bonus.

And as you can see again, the UFO bonus then will explode and the spaceship will explode.

So what you see here is a set of interactions between the different objects.

And even though we don't have a lot of objects here, this is already pretty complicated.

And this points to the fact that our code will not be easy to maintain if this is how we have our objects

interacting, let's look even further.

So how can we improve the interaction of our objects?

What we can do is we can use something of a controller.

A controller is like a conductor in an orchestra.

It coordinates all the other elements.

So, for example, we have our spaceship, we have the asteroid, we have bullets, we have UFO bonus,

we have explosions, we have sound and we have civilization.

And we also have a user gesture.

So in other words, our objects haven't changed.

We just have added a controller.

So what is special about the controller?

The controller is a middleman or a broker between all the different elements.

Here we have a little legend.

When you see broken up arrows, that means that those are inter object communication.

So, for example, the spaceship can interact with an asteroid through the controller.

The spaceship can interact with the bullet through the controller.

On the other hand, spaceship cannot really interact with the explosion and it cannot interact with

the sound controller does that on their own.

So what I would like you to kind of understand from this is that we have reduced the number of interactions

by introducing a controller in the middle controller will basically organize all of the different interactions.

So using a controller, we're going to simplify things.

All right, let's go even further.

We could add handlers to the controller to nicely abstract, different functionalities.

So for example, if a spaceship is going to fire a bullet, we could have a handler in the controller

that will take care of this.

All that the spaceship has to do is basically make the controller aware that it would like to fire a

bullet so the controller has handlers.

All right.

Now let's look at how do we, first of all, create our objects in object oriented design?

Well, at a high level, it's pretty straightforward.

Here we have an asteroid, and let's assume that we've just created four methods that you can operate

on for the asteroid class.

We have an on head method, which basically tells the asteroid it has been hit.

We have a split method which allows the asteroid to split itself into two smaller asteroids.

We have a destroy method, which basically tells the asteroid that it has been destroyed.

And we could also have something like a get health, which in our case we're not going to use, but we could have a good health method, the idea being that we're going to look at a class like the asteroid and ask ourselves, what is it that we can do with that class?

So now the question is how do we make the controller work?

Think of it this way One simple way to work with the asteroid is to get an instance of an asteroid and then call the corresponding method.

So, for example, in this diagram, what we have is that the bullet could basically tell the asteroid, Hey, I am heading you so the bullet could call the asteroid's own hit method.

The main problem with this approach from an architectural perspective is that this means that the bullet would have to be aware of the asteroid class.

And not only that, it would have to have access to the specific asteroid instance.

And this usually means a lot of object passing which can get messy, especially as your game, for example, gets more complex.

Let's look at something a little different.

So we have this.

The typical execution of when a bullet hits an asteroid would be as follows Bullet calls the on head method of the asteroid.

Asteroid then updates the scoreboard and place the explosion sound.

Asteroids then destroys itself and removes itself from the game.

There's a problem with this, and the main problem is that the asteroid knows too much about things that do not concern it at all.

The asteroid should not have to know how to remove itself from the game.

It shouldn't have to know how to update the scoreboard and it shouldn't have to play the explosion sound or for that matter, show the explosion rendering that should be handled by something else.

So let's look at an alternative.

So one way to mitigate this issue is to create a notification mechanism.

Which would allow objects to send messages to other objects without having to directly interact with them.

That's the point of the controller.

So we have our bullet and let's say the bullet knows that it just hit an asteroid.

So what the bullet does is it sends a message to the controller saying, Hey, I just hit an asteroid.

What the controller does is the controller then knows how to talk to the asteroid and it calls the on head method.

And also the controller knows that it should update the score.

See, we're basically separating the asteroid and the scoreboard completely.

Next, let's say that upon hitting the asteroid, the asteroid is destroyed.

Like the asteroid knows that it cannot split itself anymore.

So what the asteroid does is it tells the controller I have been destroyed.

It's basically asking the controller to remove me from the game.

Again, in this situation, the asteroid doesn't have to know how to remove itself from the game.

And that is the point.

Let somebody else have that knowledge.

Next controller.

Knowing that the asteroid has been destroyed, it will actually generate an explosion the same way that it knows how to update the score.

It knows how to generate an explosion, and it will also know how to go to the sound module and play the specific explosion effects.

The nice thing about all of this is that we can develop the asteroid class tested separately and we don't have to worry about all of the other things about updating the scoreboard and everything.

So this is a much better approach.

Great.

So we have these messages, but how do we encode them?

How do we actually create messages?

In this context, messages are basically like events and they encode the event itself as well as the intended recipient.

So here are some examples, some high level examples of messages.

For example, we could have the following bullet messages.

Those are messages that are generated by the bullet to the controller, and that is a bullet create message.

Well, in this case it would not be generated by the bullet.

It would be generated by something else, like, for example, the spaceship, for example.

When the user taps on the screen, we know that tapping is supposed to generate a bullet.

So at that point, it will be the user gesture, the on tap up that would send a bullet, create message to the controller.

Another message could be bullet hit message.

This is where the bullet hits something.

We also have a bullet destroy message because of course, we will need to once the bullet is either out of bounds or the bullet hits something, it will need to be destroyed.

So those are the different notifications that pertain to the bullet.

Here are some messages that pertain to asteroids.

We could have a create message for an asteroid where we are requesting that an asteroid should be created.

We also have an asteroid hit message, which basically tells the asteroid that it has been hit.

We also have an asteroid destroy message, which basically removes that asteroid from the game.

And we also have an asteroid split message, which when the asteroid is hit but not destroyed, the asteroid will be split into two.

So these are just simple examples.

And I would like you to go through the slides maybe one more time and just kind of internalize what it is that we're doing here.

The most important thing is just the realization that something like a controller and the messaging design pattern are going to help us with simplifying the approach to our game.

I'm going to see you in the next video.

And as always, of course, keep on coding.

We'll see you soon.

3. Design Patterns We Use - Factory Method Pattern

All right.

Welcome back.

We're going to be looking at the object oriented design and specifically we'll be looking at the design patterns that we could use to help us with our game.

What we're going to look at right now is the factory method design pattern.

So let's see what we can do to improve our approach in architectural terms.

Let's start with object creation.

The following objects need to be created throughout the lifecycle of our game.

Bullet Asteroid Spaceship Explosion, UFO Bonus Ship.

We also have a couple other more general objects joystick background parallax, score data sound.

Those objects are usually created as part of the game environment and do not really get generated multiple

times based on gameplay.

For example, bullets or asteroids are generated constantly in our game, so we need to really take

good care of how we create those objects.

So let's look at the main ones.

So we have the bullet, the asteroid spaceship explosion and UFO bonus ship.

When we create an instance of a bullet or an asteroid, what we normally do is we're calling a constructor

of that specific class.

But what if we want it to be able to create different types of bullets or different types of asteroids?

How can we abstract that?

What would be the best way to do it?

To make the creation process as abstracted as possible?

We could use the factory method design pattern.

What is it?

This is a creation pattern and it provides one of the best ways of creating objects in the factory method

design pattern.

We create objects without exposing the creation logic to the color, and the color refers to the newly

created object through a common interface.

So as far as the color is concerned, they're always just working with a bullet.

This makes the code reusable and is perfect in situations where, for example, we want to upgrade the

game with new types of bullets or spaceships with minimum coding.

So let's look as an example at our bullet class.

Currently it is a concrete class.

This is just the code that we have from our previous examples.

And as you can basically see, we have a bullet.

It extends the position component.

We have a constructor and then we overwrite the update, the on load and the render.

That will work pretty well.

But what if we wanted to create different types of bullets?

We could use an abstract class.

Abstract classes are designed to gather all the common behavior for a specific class of objects under one recipe abstract classes.

They define the contract for all those objects to follow.

Consider this simple encapsulation of a bullet.

We can have a method like on head on, create, on destroy.

We can also get the health of the bullet and we can get the damage that the bullet renders.

All bullets could have this kind of behavior.

So if we abstract a class, what is the benefit?

If we make the bullet an abstract class, we can make it to be the parent class of all bullets.

So here's our bullet abstract class, and then we could derive specific concrete classes like, for example, maybe a fast bullet, a slow bullet, or maybe some sort of a splash bullet.

All of them would be children of the bullet contract.

So how does this abstraction help us?

We can now create bullet instances only based on the bullet interface, or we also call it a contract instead of a specific bullet definition.

We then use the factory method design pattern as follows Instead of going through a constructor, we use a bullet factory.

A bullet factory returns an instance of a bullet.

It could be a fast bullet, it could be a slow bullet, or it could be a splash bullet to the color.

It doesn't really matter.

They simply want to create some sort of a specific type of a bullet.

But what they get is the contract for a bullet.

So basically the bullet factory just creates bullets.

And the idea is pretty straightforward.

In the code, the color will ask for a bullet instance based on some sort of a abstracted name.

For example, we could say create a bullet asking the bullet factory, and we could basically ask for maybe a fast bullet or we could ask for a splash bullet.

Notice that in each case, my bullet 001 and my bullet 002 are both instances of a bullet.

The main advantage of that is that the color does not need to know how the factory creates this bullet.

The color only needs to know how to call the factory and with what initialization data, we could change

the details of the creation implementation if we wanted to.

But the.

Solar would be unaffected.

We could even make this more generic by using some type of a context data to initialize the instance.

For example, we could have some sort of a bullet built context.

We put some initialization data into it, and then we simply call the bullet Factory.

With that context.

One example advantage of using a factory method pattern is that, for example, we could cache objects in the factory for added efficiency.

Here is the factory used without a cache and here it is with a cache that is hidden behind the factory call.

So the caller does not know about this implementation.

So just like before, we have the bullet factory which creates a bullet, but the bullet factory could be caching the bullets, meaning that we are much more efficient with memory.

There's lots of things like that that you can do when you separate the concern of creation versus usage.

Now let's have a look at a UML diagram.

So here we have a UML diagram of our whole bullet abstraction.

First of all, we have an abstract bullet class and the bullet class does two things.

First of all, it creates a number of abstract methods, like on create, on Destroy and on Hit.

For example, we also define to constructors and we also have a getter for health and a getter for damage.

Now the bullet itself extends position component, so we also have the angle and the position.

But even more importantly, as you can see in the note, because we extend position component, we need to also remember that because this is a component in the flame game engine, if we want our bullets to actually do anything, all the child classes of the abstract bullet class will also have to override the render update and unload methods, which is why we show them in red in both implementation of fast bullet and slow bullet.

So here what we have is an example of actually two implementations.

We have a fast bullet which will be of a certain shape and we'll move faster and then we'll have a slow bullet, which will be maybe a different shape and will move a little slower.

We could also define, for example, a slow bullet to have more damage because obviously it's easier to avoid a slow bullet.

So if it really hits you, let's assume that it has more damage.

Now, once we have defined our bullet abstraction and we have the implementation of fast bullet in a slow bullet, for example, the next thing that we have is our bullet factory.

Notice that the bullet factory uses a bullet build context, and the bullet built context is basically just the information with which we're going to initialize our bullets.

The Bullet Factory has a single create method, and in that method we basically specify what kind of bullet we would like to get, and then we provide the initialization data.

Notice also that the create method, it's a static method, so we don't have to instantiate the bullet factory.

And in fact, if you look at the constructor for Bullet Factory, we made the constructor private, which means that you cannot actually instantiate this class.

The last thing in this diagram that I would like to put your attention to is the Bullet enumeration.

The Bullet enum basically enumerates all the bullets that we have right now, so we have a slow bullet and a fast bullet.

And inside our implementation of the Create method, we would have a switch statement which would use all of the different enemies as a way of figuring out which bullet class we actually want.

So now let's have a look at the actual code for this.

All right.

So let's have a look at our bullet glass.

As you've seen in the diagram, we have an abstract bullet class which extends the position component.

We have the two constructors.

We also have two getters, which is the get damage and get health.

And then we have three abstract methods on create, on destroy and on head.

We also have two implementations.

We have the last bullet and as you can see, we simply render that as a rectangle with the given size.

And then we also have a slow bullet which defines a slightly different bullet.

And you can see that that bullet is, for example, a circle.

And the paint that we use is red.

So these are just simple examples.

We also have the enumeration right here, and at the bottom we have our Bullet factory, and the Bullet factory is pretty straightforward.

We have a private constructor and then we have our switch, which based on the choice, which is our Bullet enum, it will create either a slow bullet or a fast bullet.

And finally, we have our bullet built context.

All right, So let's have a look now how this would be used to create an actual bullet.

And the way this is done is pretty straightforward.

We have our bullet built context.

We initialize it with particular values, and then we ask the bullet factory to create that particular bullet.

So let's have a look at what this would look like.

Let's run this on our emulator.

All right, so our code is running.

And if I'm going to just move it around, I'm going to shoot right now.

As you can see, I have little green bullets, fast bullets shooting right now.

What I'm going to do right now is I'm going to change this.

I'm going to make the bullets a little bigger.

And instead of fast bullet, I'm going to ask for a slow bullet to our effect.

And I'm going to change the speed here to, let's say 50.

Now, let's rerun this.

And now if I showed, you can see that it's a much slower.

All right.

So hopefully this gives you a pretty good idea.

What we're going to do in our game code is we're going to write such factories for all of the main objects such as asteroids and even the spaceship and the UFO, just as well as explosions.

But the main idea that I would like you to take from this is just that this makes it much easier in terms of breaking the bullet creation away from the bullet usage.

One other thing that I would like you to see, if we go back into the bullet glass inside the factory right here, notice this result on create result happens to be the bullet that is being returned.

So notice how inside the factory we're actually triggering an extra call that would allow us to maybe do some extra initialization or something like that on the bullet.

And that's really pretty powerful way of being able to add more functionality into a bullet and the color will simply not know anything about it.

So the main code sample from the HTML that we have seen and the code that we looked is basically this

We have a bullet built context.

We then initialize it with some specific speed, some specific position velocity, and a size.

Once we have created this context, we simply then ask the bullet factory to create a specific bullet for us, which is a fast bullet, and initialize that bullet with the context.

And what we get is a bullet back.

So as you can see in the above example, the bullet creation is abstracted out away from specific classes and we only rely on the bullet class as a contract.

All right, in the next lecture, what are we going to do is we're going to talk about the command pattern, which is a behavioral pattern which will allow us to now exchange messages, for example, between bullets and spaceships.

So we'll see you in the next lecture.

4. Design Patterns We Use - Command Pattern

Welcome back.

We're going to continue talking about the object oriented design.

And specifically, we will be talking about the second design pattern that we're going to use in our architecture, and that is going to be the command design pattern.

So what is it?

The command pattern is a behavioral pattern in which an object is used to encapsulate all the necessary information needed to perform an action or trigger an event at a later time.

The aim of this pattern is to decouple the request from the actual execution of the request.

The important thing here is that we're decoupling.

We achieve the following two objectives with the command design pattern.

One, we define a separate command object to encapsulate a request to then a client class delegates, a request to a command object instead of implementing a particular request directly.

This enables one to configure a class with a command object that is used to perform a request.

The class is no longer coupled to a particular request and has no knowledge is independent of how the request is carried out.

So what is it exactly that we're trying to achieve with this pattern?

The next step in improving our architecture has to do with how the different objects communicate with each other.

As stated previously.

Currently, most objects communicate directly with other objects.

For example, a bullet in our code makes a sound by directly invoking the sound API.

Here is the code from our bullet class.

As you can see right inside the on load we directly invoke the flame audio and this is not optimal.

We should separate the responsibility of playing the sound away from the bullet class.

So how do we do this?

Let's look at an example.

Consider just firing a bullet.

It starts with the user tapping on the screen and the ship, then shooting a bullet.

How can we abstract such an action in terms of abstracted commands?

Well, let's have a look at it this way.

We have our controller.

The user taps on the screen.

So there is a user tap notification that goes to the controller.

Then what happens is the controller sends a message to the spaceship, and the spaceship then sends a fire bullet message telling the controller that it wants to fire a bullet.

The controller then creates the bullet.

The controller also will play the sound and the controller will also update the scoreboard.

So as you can see, the spaceship doesn't have to worry really about things like how to create a bullet or how to make the sound.

And neither does the bullet have to worry about how to create the sound or how to update the scoreboard.

All of that is being done in an abstracted, decoupled manner.

Let's look at the sequence of the instructions to fire a bullet.

So as we've seen, user taps on the screen, controller gets the tap gesture, a spaceship fires a bullet, the bullet makes a firing sound.

How to encode this in terms of our commands or our messages, we can look at it as follows A user taps on the screen.

This creates a user own tap command, which is sent to the spaceship.

Spaceship then sends two commands to the controller.

A bullet fired command which creates an adds a bullet to the game and a bullet fired sound command, which generates the firing sound of the bullet.

The controller then creates and sends the increased bullet count command to the scoreboard.

We can think of this as a relationship between an Invoker, the command creator, and a receiver, the

actual command executor.

Let's put this in a slightly different format.

Let's put this in a table format for a little bit more clarity.

So you can see that we have an invoke column, we have the command, we have the receiver, we have the data that the receiver will need, and we have defined the behavior.

So, for example, the controller sends a user type command to the spaceship.

The location of the tap is the data.

In our case, we ignore it.

The behavior is that this lets the spaceship know that a user gesture was received.

This, in turn will generate a command from spaceship to the controller to create a specific bullet.

Then, for example, we have the invoker being the spaceship and the command is bullet fired command and the receiver is the controller.

The data is going to be the spaceship velocity and the bullet type.

Well, in our case it will simply be the spaceship instance.

And what the behavior of this command is going to be that creates a new bullet with the spaceship velocity and adds it to the game.

And as you can see in this table, we have a number of other commands.

The important thing here, though, is to see that there is a relationship between the invoker, the receiver, the data and the command itself.

But the idea is that the controller doesn't have to know anything about how the command is implemented.

Only the receiver has to know that the invoker is completely unaware.

So as we have seen in the previous slide, we have an invoker and a receiver.

The invoker is the agent that creates the command.

For example, when a controller wants the spaceship to know that the user has tapped on the screen, it will create and send the message to the spaceship.

Receiver is the intended recipient of the message and will execute the specific code for the command.

One last piece of the puzzle that we need here is some sort of a coordinator that will store and actually execute all those commands.

We will call that the broker.

And the purpose of the broker is to get all the commands in some sort of a list and then execute all those commands one by one at every game update cycle.

So here is how the broker will fit into our architecture next to the controller.

So as you remember before we have the controller, we have the other objects, the spaceship, we have the ONTAP, we have the scoreboard, we have the sound, we have the bullet.

This is the relationship between the different objects and these are the actual messages.

Nothing really has changed from previous diagram, but what we're adding here is we're adding the broker.

And what happens is that when anybody sends a message to the controller, controller uses the broker to schedule all those commands to be processed, the broker is going to store all those commands in a list and it will simply then process those commands.

So this is just even a slightly more abstracted solution where the controller just coordinates things and it's the broker that actually executes all the commands.

All right.

So let's have a look right now at.

A UML diagram of the solution for this.

So here's our HTML diagram.

And as you can see, it's relatively straight forward.

First of all, I would like you to look at the abstract command object.

And as you can see, the command object is the abstract class from which all the actual specific commands are going to be implemented.

So what does the command class actually have?

Well, first of all, it has two abstract methods execute and get title.

The executing get title are going to be implemented by the child classes.

The most important one is to execute.

The execute method is the actual encapsulation of what the command is going to do.

So for example, the user tap up command is going to execute some sort of code that should be executed

When the user taps on the screen, a bullet fired command is going to execute the actual firing of the bullet and then the bullet fired sound command is going to actually play the sound of the bullet being fired.

And of course, we would have many, many more commands that encapsulate all of the different interactions.

The idea, though, here is that each one of them is like a separate little piece of processing power, and this nicely abstracts it from whoever calls this command.

Now let's look a little bit more into the command abstraction.

Notice that the command actually has a controller inside it, and notice that we also have a method called add to controller.

So basically when you create a command object just by creating the command object, nothing happens.

You actually have to give this command to the controller.

This is why we have a controller inside the command.

Now let's have a look quickly at the broker.

The broker is pretty straightforward.

It has two lists.

It has a command list, which is just a list of all the commands, and it has a pending command list.

The main reason for the two lists is that when we're actually executing all of the commands on the command list, it's very possible that other commands are being scheduled since we don't want to add them to the currently being processed command list.

We have a pending command list, so we add all of the different commands that arrive at the time when we're processing.

We added into the pending command list and the broker has two methods.

One is process, which is basically when you call the process command, the broker is going to execute every single command inside the command list.

The add command method simply adds a new command to the list of commands that the broker holds.

So this is a pretty straightforward class.

Notice also that we have the controller.

Now the controller contains the broker.

That makes sense because the controller actually tells the broker to execute things.

Notice also that the controller has a method called get spaceship.

In general, the controller is going to have more methods that give you access to the state of your game, so it would also have a scoreboard, etc..

The most important two methods in the controller are the ad command, which is how you take a command and schedule it for execution.

What you do is you add the command to the controller.

We also have the update method.

Notice that the controller actually extends component, so the controller is going to be added as a game object into the tree of objects that the flame engine controls.

So the important thing here is that the controllers update method will be called at every game cycle.

Update every time there is a tick, the update method will be called and the controller at that point

is going to call the broker to process all of the commands.

So basically every time the update method is called, all of the broker's commands are going to get

executed.

So this pretty much makes sense.

All right.

So let's actually have a look at the code right now.

Here is our code.

First, let's quickly have a look at the controller.

As you can see, the controller extends component.

It has a broker.

And in the update method, all that we do is we simply call the brokers process method.

We also have an ad command method which simply delegates to the broker.

So the controller is pretty straight forward.

Next, let's have a look at our command.

So here is our broker class.

And as you can see, we have two lists, the command list and the pending command.

Notice that in the ad command, all that we do is we add a command to the pending command list.

Remember, we're not adding the command to the command list because the command list could be executing

at this point.

So we always add to the pending command list.

And then look what.

We do in the process method.

When the process method is called, we go and walk through all of the commands in our command list and

we simply execute each command.

Then what we do is we clear that list.

Then we transfer all of the pending commands into our command list and we clear the pending command

list.

And this basically gets repeated every time the update cycle happens.

So the broker's pretty straightforward.

Next, let's look at our command.

So our command is simply an abstract class.

We have access to the controller because the command needs to know about the controller.

We also have a method add to controller.

This is how we're going to add this command.

Once we create this command, it will not be automatically added to the controller.

We then have to add it to the controller and give an instance of the controller to the command so that it knows where it is actually being added.

And notice that in the app controller we simply add this command to the controller.

Then we have an abstract execute method and we have an abstract title method.

So now let's look at a very simple three implementations.

So we have a user tap command which extends command.

Notice that the receiver here is the player, so we initialize this command with the player and then inside the execute method, we actually delegate to other commands so we can actually aggregate commands.

So all that the user type command does is it basically tells the player the spaceship that oh, the user has tapped on the screen and what the spaceship is doing is basically says I would like to fire a bullet and I would also like to make a sound, but notice that it doesn't know how to do it.

It delegates that to additional commands.

This part creates the command.

We then add this command to the controller and that's basically it.

We also give name to this particular command.

Then we have another command which is bullet fired command.

Notice that this does not actually have a receiver because in this case it's the controller and the execute method simply just does what our previous on tap method did.

Basically, we get the velocity of the spaceship, we rotate it by a specific angle so that the bullet is going to travel at a certain angle.

We then create the bullet, build context by specifying the position of the spaceship and the velocity.

We give it a size and then we simply use the bullet factory to create the bullet.

Notice that we don't even know what bullet we're creating.

We're actually asking the spaceship what bullets are you shooting?

So we get from the spaceship the bullet type, and we then create the bullet and we add the bullet to

the game through the controller.

And that's basically it for the bullet fired command.

And then we also have a bullet fired sound command, which does very much the same when the execute method is called.

It simply plays the sound great.

So you can see that we have all the three different if you go back here, these are the three different commands that we have.

And the question is how well does this work?

Well, notice that we did just for debugging.

We've done a very simple thing here inside the broker.

Whenever we are executing a command, we're just printing and whenever we adding a command, we're also

printing.

All right.

So what I have right now is I have the code running.

Notice that inside the broker we're actually outputting something about the command.

So what I'm going to do is I'm going to filter this.

We're going to be looking for this command.

So notice what happens right now when I shoot the bullet.

So what you can see first, our code has added a command, which is the user tab command, and then it executed this command, which is what happens first, the user taps and that goes to the spaceship.

Notice then that the spaceship added to commands.

One is bullet fired and bullet fired sound command.

And notice that right after that, these two commands have been executed.

So the idea basically is that we have been able to very nicely clean our code, clear it out so that everything is compartmentalized right now into commands.

And I'm just going to show you one other thing.

For example, if I go into the space ship, I'm going to change right now the bullet type.

So as you can see here, there's a getter, which is the bullet type.

And what I'm going to do is I'm going to change it into a test bullet.

And if I rerun this right now, just by doing this change, notice that the command is going to be smart enough.

I have to change nothing.

And notice now that we're going to have a fast bullet and you can see that everything is exactly the same as before.

So this is pretty powerful and relatively easy to modify because all we have to do right now is basically just write more commands.

For example, right here we have the update score.

It's relatively easy to add that all we would have to do is just create a new command that would then update the score and that would be sent by the controller.

As you can see here, when the bullet is fired and the bullet fired sound is created.

Notice that the invoker is spaceship, but it's the controller that receives it.

So we could also, at that point tie a command.

As you can see here, it would be the controller that will be the invoker to call the scoreboard.

So basically all we would have to do is increase bullet count command, which could very well be done the moment the controller invokes the bullet fired command.

All right.

So this was a general introduction to the design patterns that we're using.

And the next thing that we're going to do is we're going to look at the final code for the actual game, which is going to use a combination of the factory method design pattern and the command design pattern to just make our code a little bit cleaner and a little bit more abstracted.

So it's a little easier for us to, for example, add new bullets or we could quickly write different commands and replace the commands because at this point what is happening is we could easily go in and just modify one command.

We don't have to go and look where all those different pieces are because we don't really worry about who invokes it.

All we really care about is that what does this command do?

If the command, for example, creates a sound for firing a bullet, then it's pretty simple.

If we want to change anything about that, we simply modify that command.

All right.

So I'm going to see you in the next lecture when we're going to go over the code of the actual game.

See you later.

And as always, keep on coding.

5. Putting It All Together - Architectually Speaking

Welcome back.

All right.

Let's continue with the game architecture.

What we're going to do here is we're going to look at how do you put everything together, architecturally speaking.

So let's take a bird's eye view of our whole game in terms of modules of responsibility.

We have looked at design patterns such as factory method, design pattern and command design pattern, which are very useful.

Using design patterns helps us with abstracting our architecture better.

But what exactly are all the pieces in our architecture?

We can break down our architecture into the following main modules.

We have our object factories that create all of the different objects that we need in the game.

We have our action commands which through the controller are able to manage all of the different object interactions.

We have state management, which is also going to be inside the controller where you're going to control the state of the application.

Of course there's the game logic.

In other words, we're creating asteroids, not, for example, Super Mario Brothers.

And one other important thing is the level generation.

We have to be able to generate different game levels for the players to play with.

So let's look at a slide from the beginning of this case study.

As you remember, we have initially identified most of the elements or objects that are going to be part of our game.

We have rearranged these objects to be a little easier to discern in terms of what their responsibilities are.

So here is a very high level view of our architecture organized in a architecturally better manner.

Obviously, we have our Flame Game engine, which is how we're writing this game.

We then have all of our object factories, we have a level generator, we have serialization which will be used by the level generator and the factories.

We have a controller and we have our commands.

In a nutshell, this is our game architecture and here we have a slightly more object specific view.

We have our UFO bonus and our asteroid, which are part of the factories, as you can see, of the blue coloring.

And we also have serialization.

Serialization will be used to read information about the UFO bonus and the asteroids.

When we're generating the new levels.

We also are going to use the serialization to write the high score into a file on the mobile device so that when you start your game again, you will see what your high score was.

Apart from that, the other factories that we're going to be using are going to be ones that create the spaceship, the bullets, the explosions and the sounds.

We have already seen the bullet factory.

The next step will be to create all of the game objects.

So we'll start with the factories first.

So as you see, this is what we're concentrating on and this is the list of the objects that we will need to create factories for.

Now, what I'm going to show you right now is just a quick UML diagram of the asteroid factory.

But what you will notice and this is where the design pattern methodology is very powerful, is that when you compare the bullet factory to the asteroid factory, you will notice that it's very similar in terms of the coding.

It uses very much the same structure.

Let's quickly have a look at the HTML for the asteroid factory.

So here it is.

And as you can see, it's basically the same as the bullet.

We have an abstract asteroid class, we have the asteroid factory, we have an asteroid built context.

We have an enum for the different types of asteroids.

And then we have three concrete classes for a small asteroid, medium asteroid and a large asteroid.

All of these asteroids are also positioned components.

They are collectable and they have hit boxes because we want them to be part of collision detection.

Notice that we have added two methods into our asteroid class, the abstract class that we didn't see in the bullet class, and that is can be split and get split asteroids.

The idea behind it is pretty straightforward.

When an asteroid is hit, for example, by a bullet, our controller will check whether this asteroid can be split.

So we'll simply call the can be split method.

And if it's true, then it can call the get split asteroids method, which is going to give the controller a list of all the asteroids that this asteroid should be split into upon the explosion.

The large asteroid is going to split into two medium asteroids and one medium asteroid is going to split into two small asteroids and a small asteroid will return, can be split as false.

All right.

So please go over this and as an exercise for yourself, try to create the factories for all the other objects that we have seen in that list.

And the next topic in our architecture is going to be level generation and serialization.

So we're going to see you then.

And as always, keep on coding.

6. Using JSON and Serialization in Our Architecture

All right.

Welcome back to case study number eight.

Asteroids, game architecture.

So in this segment, we will tackle serialization and the JSON file format.

As you remember from this architecture diagram, we are going to be creating levels based on a level generator because we want these levels to be repeatable.

We're going to permanently store them in a file, which means that at each time that the user starts the game, we will need to read those levels from that file.

This is where the level generation and the serialization comes in.

Of course, these two work together with the factories to create the objects that we're interested in specifically.

We're looking at being able to read UFO bonus and asteroid data for each level.

Additionally, we would like to also be able to read and write scoreboard information, specifically the high score.

Of course there are other factories, but they're not involved in the sterilization process.

The other elements of the architecture, like the user gesture, joystick and parallax, have nothing to do with either factories or sterilization.

So we're going to concentrate right now on the serialization aspect.

As you've seen in the diagram, what we have is we have our level generator and we have the serialization

part.

We will both serialize and serialize our data.

What that means is that we're going to be able to read the UFO and the asteroid data, which is the DX serialization, but we will also have the ability to read the high score of the scoreboard information as well as write it so that in between games, when the user turns off the device, for example, and comes back later, we can show them what their high score was.

So the serialization that we are interested in will have to be able to do the following.

We'll have to be able to read the asteroid data from adjacent file.

We'll use this to create a map of asteroid data so that we can generate game levels.

We also need to be able to read the UFO bonus data for the exact same reason.

And we would also like to be able to write and read high score data for the user.

How do we serialize the asteroid data?

We want to be able to read asteroid data from our JSON file and then using the asteroid factory, we want to create the asteroid objects from that JSON definition.

The serialization of an asteroid object needs the following data.

We need the starting position of each asteroid.

We need the velocity and we need the speed as well as the asteroid type.

Which is which asteroid are we actually creating for the level?

Note that the starting position of all asteroids will always have to be either off the screen or on the edge of the screen.

The velocity should always point into the screen, but even if any asteroid is pointing the wrong direction or the wrong way, we will do an automatic screen wraparound so asteroids cannot fly off the screen.

When the asteroids are generated, they will just fly towards the screen.

And then if they happen to be at the edge of the screen, they'll just wrap around and come out from the other edge of the screen.

So you, as the little spaceship, will always have to be watching out for them.

So let's look at the JSON data that we will need.

In order for us to create an asteroid object, we will use the context, which in our case is the asteroid built context as follows.

We have the JSON file.

We're going to ask the sterilizer to read the JSON data.

We are then going to create the context out of that data.

We're going to feed that context to the asteroid factory, which is going to give us the specific asteroid.

So it's a pretty straightforward workflow.

The JSON for asteroid will be quite simple.

Here is a single asteroid entity.

We are going to give it the name asteroid.

We're going to have position X, position Y so that we can create the vector.

We're going to have velocity X and velocity of Y so we can create the velocity vector.

We'll have the speed and we're going to have the name which is the enumerated name of the asteroid we would like to create.

And this is it.

This is a JSON entry for a single asteroid.

The above entry should be quite easy to parse and then build the asteroid, build context so that we can use that as an input to our factory.

Next, we'll look at how to define an array because we'll need more than one asteroid per level.

So here is how you would define an array of asteroids.

So we specify that we have an element called asteroids.

We open our array bracket and then within the bracket we will have the individual asteroid elements.

And that's basically it.

And finally, we can define an array of game levels where each array entry is level.

And so you can see that basically a level is an array of asteroids, which in turn is an array of asteroid objects.

So at this point, let's have a look at an actual code.

So let me show you the project here.

You have the running code.

It's a very simple serialization example, which is included with all the other code for this use case.

This particular piece of code has nothing to do with our game per say.

It's just a simple example that loads the level data for you to experiment with.

If you want to look a little bit more into serialization, notice that our sample, the JSON, sits inside the assets directory and you can also see in the pub Spectre YAML that we have specified the JSON in there.

Going back into our sample JSON.

As you can see, we have two levels here.

We have one set of asteroids and another set of asteroids.

So if I run this code right now, this simply loads level one with this data and then level two with that data.

So for example, if I add one more, I'm just going to grab the very first one.

So now we have three levels and let's see, I change the position here to some random stuff for now, 80, 80, velocity, 80, 80.

And if I run this right now, you can see that we have three levels.

And of course here is our 80, 80 and 80, 80.

So this is pretty straightforward.

And if you go a little bit further into the code, you will see that we have defined an asteroid class just like you've seen in the usmle.

But what is really important here is our context.

The asteroid build context has two extra methods.

One is a method that converts a string into an enumeration, which is pretty straightforward.

It's just a helper method because as you see here, the JSON data just puts a string.

We then have to convert this into an actual enum.

So as you can see, this asteroid from string returns an asteroid enum.

So it parses the data from the JSON and then creates a specific object.

In the main, what you really need to know is this code right here, which is the JSON read.

JSON is pretty straightforward.

As you can see, all we're doing is we're just loading a string from sample JSON.

So we take the whole sample, the JSON file, we read it in as a string, and then we use the JSON decode method that specifically comes from this import dart convert.

And once we have read that in, we set the state to go into items.

As you can see, an item is just a list.

And then inside here we're basically creating an item count for our builder, which is the length of the list.

In our case, there are three elements and then for each card, we're taking the array entry and we're passing it to our build asteroid data method, which then builds the context out of the JSON data.

And if you look at the definition for this, it basically gets a map and in there it goes into the asteroids.

As you can see in the sample JSON, that's what it is.

It's the asteroids.

And then all that we do is we build a new context and then we simply take the elements from the map and assign them.

Our object in this manual JSON serialization mode.

Dart treats JSON entries as just one large map.

For example, the key at the top level is levels.

Within that we have three elements.

The key for that is asteroids.

And then within each one of those we have additional elements.

And then the key for those is, for example, the name like position X or position Y or for example, speed.

Each one of these elements is a key.

So all that we're doing here is we're treating each element through its key and we map that.

So we have given you this code so that you can play around with it a little bit.

But the idea is pretty straightforward.

Let's go back now to our slide.

The next thing that we're going to talk about is what about writing data?

Specifically, how do we write high score?

We could use straight D serialization of the scoreboard class, but there is a better way for this specific circumstance.

It's shared preferences.

And of course, in this case we would be serializing meaning reading and we would be serializing meaning writing.

But there is a better way, since all we want is just high score, which is just one element.

There is no need to go through JSON.

We can use shared preferences.

This is a way for reading and writing user data to permanent storage on your device that is easy and reasonably reliable.

Here is a code sample.

So to obtain the shared preferences, you simply do a call like that.

Shared preferences dot instance.

Once you have that to save for example, an integer for the high score, all you do is you say press set int high score as the key, for example with ten.

And then if you want to read that you simply do this piece of code, that's all you would have to do.
We don't have to really go through JSON here.
We don't have to do any of that complicated mapping.
We can just go with that and that is what we're going to use in our code.
So this covers everything that we needed to know in terms of serialization and reading and writing high score data.
There is one more thing that remains.
One last additional element that we need to talk about is the UFO bonus.
The actual object data for UFO bonus is simple, and it's very similar to the asteroid data.
The main question though, is that of the timing UFO bonus should periodically show up in a given level.
But to make the game beatable, we would like to make the timing of the appearance of the UFO bonus to be exact, and define in the levels of our JSON data.
One simple way to achieve this, which is the timing aspect, is to encode the exact number of seconds or even milliseconds for better accuracy.
So the data that we'll use will encode the position, the speed and velocity, just like for the asteroid as well as the trigger.
Time for the appearance of the UFO bonus object.
This is what an entry for a UFO bonus would be in our level JSON data.
We simply specify the key as being a bonus, and then the position X and Y is the same as for asteroids.
The velocity x and Y is the same as for the asteroids.
The speed is the same.
But then we would have an additional element which would be triggered time, seconds, and we could set that to how many seconds from the start of the level we would like the UFO bonus to appear, and that is pretty much it for this segment.
What we're going to do next is we're going to actually see how levels are processed within our game so that when you start the game, the first level starts showing you the different asteroids from the JSON data that was read from our JSON file.
And you will see that each time you start the game because it reads the same file, the game will play out the same.
You know, that's the kind of expectation that a lot of players would have in terms of being able to beat something by learning how it behaves.
And in our case, we're making the behavior repeatable because the actual behavior of the game, which

is the levels are encoded permanently in adjacent file.

All right.

So I'll see you in the next lecture.

And as always, keep on coding.

Take care.

7. Generating Levels for Our Game Part #1

Welcome back.

Let's look quickly before we tackle.

Game level generation at an interesting issue with the game rendering.

When we're displaying the game on different platforms, when you look back at the video of a sample of the original Asteroids game, you will know two important things.

The original game has pretty much a standard resolution.

Basically, the original was an arcade game and thus the game terminals always had the same resolution.

The Atari console version also had a set resolution which would normally be played on a television.

Also notice that all the asteroid generation is done at the edges of the screen, which makes sense that when the game level starts, you want the asteroids to be coming at you from the sides rather than from the middle.

So what is the issue with resolution?

Let's have a look at an example of a 640 by 480 screen.

If we place, for example, a spaceship on it, it's going to have a certain width and a certain height.

Let's assume that it's 45 by 45 pixels within the context of the 640 by 480 resolution.

Let's also assume that we have an asteroid and let's say that the asteroid has a position given us a vector.

So we're showing it here as position X and position Y.

Let's say the position is exactly 580 100.

So this looks reasonable, but now let's change the resolution.

So when we change the resolution, this is what would happen.

Let's say the resolution is going to be 1920 by 1080.

We still have exactly the same little spaceship, 45 by 45, which now would look much smaller because there are more pixels to go around.

But what would happen with our asteroid?

Well, our asteroid wouldn't be at the edge anymore because the position X and the position Y that we had before, which was 580 by 100, now seems to be somewhere in the middle and at the top rather than on the edge.

So how do we solve the resolution issue?

The resolution issue comes from when you're switching between, for example, in a phone and a tablet.

If you go from a phone which has a certain resolution to a tablet, which might have a much larger resolution,

then your levels are not going to be starting in the same position, or at least visually, they will not be starting in the same way.

So we want to make sure that the experience is as much as possible, the same if you go from one platform to another.

So how do we solve the resolution issue?

We can use a resolution multiplier which would allow the size and the position to be properly mapped based on some sort of original resolution.

Let's assume that the original resolution, which is how we created all of our levels, let's say the original resolution is 640 by 480.

So let's say you have a position of 580 by 100.

Let's say now the new resolution is 1920 by 1080.

So what we do is we create a multiplier by simply taking the new resolution and dividing it by the old resolution.

That multiplier is going to be a vector.

It's going to give us, in this case, a three and 2.25.

So what do we do now?

We take the multiplier and we multiply the original position using it and we're going to get a 1740 by 225, which would actually be on the edge of the screen, just like in the original.

What about size?

Let's say you're given a 5050 as a size, which is mapped in 640 by 480 resolution.

If the new resolution is 1920 by 1080 and the multiplier is three and 2.25, then the new size could be one of the following.

We could simply multiply the x portion and we would get 150, 150.

And that is if we want to preserve the original symmetry.

The original was 50 by 50, or if we want exact mapping, we could basically just multiply it by the

corresponding x and y of the multiplier, which would give us 150 by 112.5.

This choice is really a matter of how you feel about it.

This is like your kind of aesthetic or even maybe artistic choice.

For myself, I would usually go with the 150 by 150.

In this case, unless we're talking about something that is scaled exactly, for example, like an asteroid, which is built carefully by using certain points, we would then map all these points using the multiplier.

Exactly.

So why is this important?

Since we would like to design this game for multiple platforms, it would make sense to be able to control the playability of the game on those platforms.

If we play the game on a tablet, the experience should still be comparable to playing on the phone.

When we're generating the levels and specifically the asteroids, we want to make sure that the game plays as much as possible.

The same on both devices.

So we will make the following assumptions.

We will map positions of asteroids and UFO bonus objects based on the base resolution.

On the original resolution, we will map the sizes of the objects to be based on the x dimension of the object unless we're dealing with asteroids which are rendered point by point.

So let's look at the changes that we would have to do to the code and the JSON for the game levels.

So this is.

The code.

Let's quickly have a look at the JSON.

So notice, first of all, that in the JSON, we have a resolution.

This would be our base resolution.

We're giving the X and the Y, and then the controller, for example, could calculate the multiplier based on the base resolution, which is coming from this JSON file.

And for example, the specific resolution of the device on which we're running the game.

Notice also that we have added a gain bonus element and the gain bonus element is exactly the same as asteroids.

We could have multiple game bonus elements and because we would like to have the ability to maybe create

different types of bonus elements in the game, we have done it exactly the same way as the asteroids.

We actually have a name which will be given to the factory to build a specific game bonus.
In our case, we're looking for the UFO bonus.
Notice also that we have the trigger time in seconds, so this is pretty much everything that we would need for our game data to create the levels we have, the resolution to create the multiplier.
We have the asteroids with the position and the names, and we have the game bonus, which tells us also the position and the velocity and the speed as well as when we should trigger or show the bonus on the screen.
Based on how many seconds have passed since the beginning of the level.
What we have done also is we have created simple JSON utilities.
Let's have a look here.
The JSON utilities is where we have abstracted all of the JSON reading.
And as you can see, we have read JSON any data which basically loads the whole file into memory.
And then we have extracting the gain levels we have extracting the game resolution and then we also have to helper methods which is build the asteroid data and build the game bonus data.
The nice thing about using a utility like this is that to use this, if you look for example at the controller, we have added to the controller a special method which is an init method, as you can see right here.
And in the image method, all that we're doing is we're using the JSON utilities.
We first read the JSON in a data, we then extract the game resolution, we then create the resolution multiplier.
This is just a little test that we did here because this piece of code is not currently connected to claim engine, but this gives you the idea.
And then we extract the game levels.
And if you look at the game level class itself, the game level is basically a list of asteroids and gain bonus objects exactly like what we have here.
One other thing to note is when you look at the controller is that the init is asynchronous because this is going to be done as a future.
So this is pretty straightforward as far as the code is concerned.
One last thing that I would like you to notice is this line right here, which is shared preferences.
In order for us to be able to read the data from shared preferences in the expected YAML, we have to include this dependency which is the shared preferences dependency, and that is basically everything.
So you can have a look at this code.

This is serialization example 002 But in general we are going to have all of this code ported to our game class.

This was more of just playing around with serialization so that you can just play around with flutter and serialization.

There is another design caveat that we need to be aware of the display orientation.

This could be either portrait or landscape.

Let's look at both.

This is simply portrait.

The height is greater than the width and this is landscape.

The width is greater than the height.

In portrait, we have more real estate in terms of height and in landscape there would be more in the width.

We could design our levels in such a way that the orientation would not matter.

But for this course we will stick with portrait and it might be a bit of a good exercise for you to think how to remap, for example, the generation of the levels so that it works similar in landscape like it does in Portrait.

But in general it's a good idea to lock a game like this into one specific orientation and in this case we're going to lock it into portrait.

All right.

So this is basically it for this short little lecture.

The next lecture is going to deal with level generation, with showing the asteroids as being generated for each level.

See you then.

And as always, keep on coding.

8. Generating Levels for Our Game Part #2

Welcome back.

Let's have a look right now at how it is that we're generating the levels.

Reading the data is relatively simple, as you've seen, but the main question is about the workflow of the data read.

So we want to read the JSON level data and store it in the controller state data.

As you remember, the controller is kind of like our nexus, like the center piece that coordinates

everything.

So once we have done that, once we have stored that in the controller state data, the game logic at high level will be as follows We generate the level for the player.

If the player beats the level, then we load the next level from the controller.

If it is available when the player is done, we store the high score in JSON to be read in.

Later, we will use a timer to notify the controller about time passage in one second intervals so that it can load gain bonus data when the time comes.

As you remember, game bonus data like the UDF bonus that shows up every now and again is triggered based on a timestamp.

So we need something to be aware of the passage of time.

So to generate the levels from our JSON file, we would need to do the following.

We need to read the JSON file into the controller state data, We read the asteroid and gain bonus data and this is done as part of the controllers init method.

We have made the init method a future so that way we will have the code.

Wait until that is loaded before anything else is done.

Because don't forget there's a game loop going on.

So we want to make sure that this is kind of in a serial sequential fashion.

So when we call in it, that's the only thing that will execute.

And once the entity is done, then it will continue.

So then what do we do?

We set the current level to be one.

We start the timer to monitor time passage to trigger any bonus data.

And then what we do is we generate the asteroid data for the current level.

So for each asteroid context, because if you remember in JSON file, we store information that will be used in the context we use the asteroid factory to create an asteroid object based on the asteroid context.

We then add the asteroid instance to the game and if the time triggers bonus, then we add that bonus to the game that was also stored in the controller state data.

If the user has beat the level, we then set the level to be the next one and we continue doing that until the player is done.

We simply then repeat step four.

All right, so let's have a look at the actual code.

So here is the game.

We have at this point simulated the asteroids to be just balls of different sizes.

So, for example, we have large ones, we have medium ones and we have small ones.

If you look at the JSON file, you can see what we have for level one.

We have four asteroids and we have two gain bonus pieces of information.

There are four asteroids here.

You can see that there's a two small ones, one large and one medium.

So if I run this right now, look exactly where the asteroids are going to come from.

So if you had seen that, what I'm going to do right now is I'm going to rerun this.

And you can see that the asteroids are coming from exactly the same spot.

I have simply paused the game right now so that you can see it.

So let's look again at the game configuration.

As you can see, we specify what the position is.

So for example, here, the position for the first asteroid is 50 and 300, so the X is 50.

So somewhere around here and then 300 at the bottom.

But notice this is the important thing.

We have velocity, X and velocity.

Why?

Why do we have like a one and a one?

Well, if you remember this little diagram, you can see that if we want to create the directionality, for example, the vector minus one, one points to the lower left corner and the vector one one points to the lower right corner.

And if you look at the code here, the one one corresponds specifically to the lower right corner.

You can see that we have another one.

The second one is going to be minus one one.

And if you look at minus one one, you can see that it's going to be pointing in the direction of the lower left corner.

If we want it to point, let's say straight down, then you would go zero one.

So, for example, if I change this into zero one and if I rerun the code right now with the zero one.

You can see right here that we have the actual asteroid pointing just all the way down.

So let's quickly have a look at the actual code.

So as you see, we have the resolution here is 720 by 1280.

And the reason why we did that is because we are in portrait mode.

So we have actually flipped it.

720 is the shorter edge and 1280 is the longer.

So basically we flipped to width and height.

Also notice one interesting thing we have actually commented this inside JSON.

Now JSON format does not allow you to put in comments like you do with code.

So the trick that we use is we simply put a comment element inside that can then be used to document what a particular element is.

So in this case we say we're using the 1280 by 720 as our base resolution, but we flipped it for portrait representation.

So let's quickly have a look at what our controller does.

So first of all, as we have said, if you look at the init method, you can see that the init method now is not only asynchronous, but it's also a future.

And the reason for this is that if you go into main dart, you can see that what we do is we create the controller, we have the controller to our game, but we then call initialization of the controller before anything else is created.

So this line is basically going to wait for the initialization to happen.

And then once we have all this done, we simply allow all the other components to be loaded.

Notice also that we have this controller timer.

Now, the controller timer is pretty straightforward.

It's at the top here and as you can see that the period is one ticks every one second.

And we also have a method here so that on tick it will call the controllers timer notification.

If we go back to the controller and if we look at the timer notification, you can see that there is a very simple timer hook.

And all that we do here is we basically check the following.

If the current level is finished, then we load the next game level.

Now this will need to be expanded a little bit because obviously once we run out of all the game levels, it would be nice to tell the user, Hey, you have reached the end of the game, but that's something that can be added a bit later.

And then what happens here is that if we go into the current level finished at this point, what we're doing is we're looking at an object stack, which is a list of all the objects that are on the screen.

And when the user shoots all those objects and this is empty, that means the level is done.

So this is a pretty straightforward check.
Every time we add, for example, an asteroid or a bonus onto the game, we put this into the stack and then every time an asteroid is, for example, destroyed, we remove it from the stack.
Once the stack is empty, we know that the level has finished.
So if we go back one more time here, you can see if the current level is finished, then we load the next game level and if we go into the definition here, you can see that all we're doing here is we're basically creating a list of asteroids and a list of game bonuses.
And then what we do is we go into the game levels, which is where we store the controller's data about what we read from JSON and we go into the game level for the current level index.
So we start, for example, with level one, well, level one would be zero.
So the first index and what we do is we basically say, okay, inside the game level, give me all the asteroid configuration data.
We put that as an asteroid here.
All we do is we also calculate the multiplier.
As you remember, the multiplier is used to make sure that the size of the asteroids and all the other objects is properly adjusted.
We then create a new asteroid, we create a new asteroid using the factory, and then we add this asteroid to the list of asteroids, and then we add those asteroids to our stack list.
And then all that we do is once we have all the asteroids created, we add them all to the game.
This needs to be, of course, improved because if the user doesn't beat the level, then obviously we would need to go back to the same level, but we'll cover that in the next iteration of this code.
So one last thing that is quite important for you to see is look at this asteroid multiplier.
As you remember, the resolution multiplier is simply created.
If we go back into the controller's init method, you can see that the resolution multiplier is simply the size of the screen divided by the base resolution where the base resolution was coming from the JSON file.
So once the resolution multiplier has been created in the asteroid, all that we do inside the on create method, you can see that we have created a vector multiplication utility and all that we do here is we simply multiply the expected size of the asteroid.
The resolution multiplier.
What that is going to give us is the size relative to the screen size as we have discussed in the previous lecture.

If I keep on running this, you can see what the size of the asteroids is.

If I stop this right now and if I change, for example, to Chrome.

What I'm going to do right now is I'm going to stop this and I'm going to run this right now using Chrome.

So when it runs in Chrome, notice that the relative sizes of both the asteroids and the ship have changed.

You will also see that there's an outline around the asteroids as well as around the bullets, because

at this point I'm showing the hit boxes.

So as you can see, everything is larger.

And if I shoot, for example.

You can see that we just destroyed an asteroid.

So as you can see, this works.

The relative sizes have been adjusted.

But what is important, if I run this one more time, look exactly where the asteroids are starting

from.

They're starting exactly from the same relative position as they did in the much smaller screen.

So we have achieved our result of making sure that at least from a perspective of gameplay, the gameplay

is very much the same on any any device.

All right.

So what we're going to cover next is the final piece of code, which is going to show you information

on the screen, such as your score.

It will also reload the high score from your previous session.

And we'll also be able to jump to a new level and that will pretty much finish the game.

And we will also be able to change if we want to.

The visualization of the asteroids, for example, make them look like actual asteroids.

It makes sense to run this just with like balls because it's so much easier.

But in the end, because you're using a factor, you can very easily change how the actual elements

are rendered.

All right.

So I'm going to see you in the next lecture.

And as always, keep on coding.

9. Putting It all Together: Architecture to Code Part #1

Welcome back.

Let's continue with finalizing our game before we deploy it.

So there's a few outstanding things that we have.

In the previous code base, we were able to generate a level of asteroids.

We were also able to shoot those asteroids and they would disappear.

But we did not have explosions, we did not have splitting of the asteroids, etc..

So let's talk about that.

Let's talk about splitting asteroids.

Splitting asteroids includes two actions determine if the asteroid has been hit with a bullet, determine whether it can be split.

If the asteroid cannot be split, then just explode it.

We then remove it from the game and we remove it from the stack and we're done.

If it can be split, then you will need to do the following.

Create two new asteroids that are in the next tier of size.

So we go from large to medium to small.

So for example, large breaks into two medium, one medium breaks into two, small and small doesn't break into anything.

Calculate the new velocity and speed for each of the new asteroids.

Remove the original asteroid from the stack and from the game and add the two new asteroids to the stack

and the game and also generate a very minor explosion to kind of show the impact that the bullet has made on the larger asteroid.

Note that we are removing the asteroids from the stack as this is how we keep track if a level has been finished.

So how do we calculate the velocity and speed of the split asteroids, which is the children?

The simplest way would be to do the following.

We start with a large asteroid, for example, or a medium one, and it has a certain trajectory.

It has a velocity vector.

At the point of impact, we're going to split this velocity vector into two other vectors, one for each of the new asteroids.

Notice, though, that what we're doing actually is we're using the 45 degree angle as a way of rotating the original velocity.

So what we do is we use the position of the collision as the position for the start of both asteroids.

So it's the position, the initial position of both asteroids.

We make the velocity of the two asteroids.

We make it based on the angle of the original asteroid as follows.

So you can see the dashed line is an extension of the original asteroids velocity.

Just for reference, we create a 45 degree clockwise rotation for the bottom asteroid and we create

a 45 degrees anticlockwise rotation for the top asteroid.

Take the dashed vector.

Imagine that we're simply rotating it 45 degrees up.

That will be one asteroid.

And then we rotate the same vector 45 degrees down.

That would be the second asteroid.

That's really all we're doing.

A rotation, which is something that we already know how to do because that's how we rotate our spaceship.

Let's have a look at the code for when an asteroid collides with a bullet.

And basically, how is it that we split the asteroids?

So inside our asteroid class, which is the base class for all asteroids, you will see down here that

we have a method which is on collision.

That's because asteroids collide.

What we do here is we create two commands.

If what we have collided with is a bullet, first, we send a message to the bullet telling it, Hey,

you have collided with something.

But the bulk of our processing happens inside the asteroid collision command.

So let's have a look at that.

The first thing that we do is we send another command, which is we are telling the scoreboard, please

update the score because a bullet has just hit an asteroid.

The reason why we have it here is because at this point, we're not really calculating a very complex

scoring system.

But it would be here that you could, for example, interrogate an asteroid.

How many points is it worth as an example?

So what do we do here is we send a message to update the scoreboard.

The next thing that we do is we remove the bullet from the game and then we check if the asteroid still

is on the stack.

And the reason why we do that is the bullet could possibly, as it's traveling, create multiple collision events, but we really care about only one of them.

So what we're doing here is we're basically saying, do we still have this asteroid in our stack if we do remove it so that any other collision with that asteroid is going to be dismissed because the asteroid can only be destroyed once.

Once we do that, what do we do then?

And only then is we check whether this asteroid can be split.

If the asteroid can be split and the split method is pretty straightforward, it checks whether there are any enumerated classes that tell us what kind of asteroids this asteroid splits into.

If that list is empty, this will return false.

So we check if the asteroid can be split.

If it can be split, what we do is we first grab the position of the current asteroid so we clone it because we're going to be working on it.

We want to have a copy of that position.

We know that we're going to be splitting it into two asteroids.

So what we're doing is we're giving the asteroid a one of them, asteroid B, we're giving them a clone of the velocity of the target asteroid.

And the reason for that is because we're going to be changing the trajectory of each one of them based on the parent's velocity.

Once we have the velocity copy, we simply rotate one of them by pi divided by four, which is 45 degrees up, and then the other one is negative pi divided by four, which is 45 degrees down.

Then once we have that, what we do is we simply create the two asteroids based on that.

So we give it the collision position, we give it the new velocity, and we also provide the resolution multiplier from the controller so that we can scale the size properly given whatever resolution we're running on.

And once this is done, then we just invoke the factory to create each one of those asteroids.

And then all that we do is we ask the controller to add both of them into our stack, and then we ask the controller to add both of them into the game.

And then we remove the original asteroid, the one that we have collided with.

Of course, we could have probably done this ahead of time and that's basically it for how the collision works.

So let's have a look at it right now.

I'm going to run it.

And don't forget that right now we're still working with just balls representing the asteroids.

All right.

So you will see that our first level is going to start.

We just have slowly moving asteroids.

So what's going to happen is I'm going to move back a little bit and I'm going to shoot this one.

It's just difficult to do it with a mouse.

We're going to hit this one.

And as you can see, it has split into a number of asteroids.

Let's let's hit.

This one that is just coming up on this side.

You can see it got split into two.

Then we can hit this one.

I want to hit one of the blue ones.

I'm just going to wait for it.

And so you can see that it seems to be working reasonably well.

Sometimes the asteroids get stuck a little bit together because it all depends on.

And now our ship got destroyed and a new ship right now should appear just going to pass this.

So basically what you've seen is that we have a pretty simple way of splitting the asteroids.

You can work a little bit more on making that a little more exciting in a sense that you can increase

the speed at the moment when there is an explosion, you can increase that speed to make these two asteroids

move away from each other much quicker.

Okay, So the next thing that we will look into is the game bonus generation.

Game Bonus is usually an extra event or an element for which the player gets extra points.

In asteroids there is usually a UFO bonus that can be shot down.

In the original game, the UFO can shoot at the player, but we will not implement that in our version.

You're welcome to implement that yourself.

Another behavior of the UFO is that it does not travel in a straight line, but in our implementation

we will currently have it travel in a straight line.

If you look at the original game, the UFO would maybe move a little bit to the left and then move a

little bit down so sometimes it could actually hit you.

We're simplifying the game mechanics a little bit by simply having the UFO travel in a straight line.

If you wanted the UFO to travel in a more complex path, you would then have to maybe describe the points

between which it travels in the JSON file.

The implementation for us is pretty simple.

The controller keeps track of the number of seconds that passed for the current level.

At each second we check if there is a bonus that should be spawned.

If yes, we create the bonus with the context data from the JSON file and then we add it to the stack

and the game just like every other element.

If the player hits the bonus, we remove it from the stack and the game.

Let's look at the scoreboard.

The scoreboard tracks the following in our game.

The high score, the number of shots fired, the current score, number of lives left, the current

level that the user is on, the number of max levels, which is how many levels does this game actually

have?

And we also track the following, but we do not display it.

The number of total seconds that has passed since start of the game and a number of seconds since the start of the current level.

We need that to figure out the bonus.

The implementation is pretty simple.

The controller will fully update all the scoreboard data.

We will achieve this through a number of simple commands.

We will have an update shots fired command.

We'll have an update current score command and we'll also have an update Current level command.

The scoreboard will be responsible for the data display, so we will make it a position component so

that in the render method it can decide how it wants to display this data.

What we have done with the scoreboard.

As you can see here, here is our class.

We have defined the scoreboard to contain all of the necessary information, such as the high score

number of shots fired, The score lives left current level, the max number of levels, as you can see,

for example, here, it says that this is right now level one out of three.

So max level tells you how many levels we have.

Notice that scoreboard is a position component.

That way we can use the update and the render to continuously show the statistics.

And what we have done is we have simply created a number of text banks.

Each one of these text paints is basically the text that you see here.

And we can control things like the font size, the font type and the color, all that we do here we have a couple of setters, we have a couple of getters.

All that we do here is basically have some business methods such as add a bullet when it was fired.

You can also add how many bullets were fired because you could, for example, have like a rapid number of bullets instead of going one by one.

If you're shooting a burst of three bullets, you might just as well add three automatically.

So this is pretty straightforward.

If you will look at the render, All that the render does is it basically takes each one of those paint elements and simply paints them.

In some cases we have like a little formatting helper function.

You can see right here, for example, the format number of lives is that if the number of lives left is greater than zero, then we output lives left, for example.

Three.

But if the game happens to come out of all the lives, then we're going to put game over.

If I continue playing this, I'm just going to get it destroyed.

I'm going to run myself so you can see the lives left too.

It's going to respond right now.

I'm going to run it again.

Now that we have.

We have only one life left.

And I'm going to run in one more time.

And as you can see now, it says game over.

Okay, so how do we achieve this?

If you look at the commands that we have here, you can see that we have a couple of scoreboard commands.

We have an update, scoreboard shots fired, update scoreboard score, command for updating the score.

And we also have updates, scoreboard level info commands so that you can, for example, show which

level it is.

And each one of those is pretty straightforward.

For example, updates, scoreboard shots fired command it accepts as its receiver the scoreboard and then all it does is it just adds a bullet and that's all.

And if you want to see where we actually do it, I'm just going to search for where this is used.

And it shows you that in our command we obviously have one.

And if I go here.

You can see that this command is used inside a bullet fired command.

So whenever a bullet is fired, the bullet command itself creates an update, scoreboard shots fired command and sends it to the controller.

So basically a message creates another message.

All right.

Please have a look at this code and see exactly how the commands are working.

One of the things that you see here is that when the game is done, everything just keeps on running.

We don't really have at this point, a way of getting rid of the game.

The most important thing here that we wanted to show you was the game mechanics rather than working with menus, etc., which is more of a flutter domain rather than a flame engine issue.

10. Putting It all Together: Architecture to Code Part #2

What about explosions?

How do we incorporate explosions within our current architecture?

We will follow the same design pattern as we do for asteroids, bullets and spaceships.

We will use here the factory method design pattern.

For each explosion type, we will have a factory and a context.

The context will capture the following data, the position in which the explosion happens and the explosion type.

This will work the same way as the other factories.

It will simply create a component which will then be added to the game.

We have the following explosion types the asteroid destroyed explosion.

The asteroid split explosion, which is a slightly smaller explosion and the ship destroyed Explosion.

Let's have a look right now at the asteroid explosions.

As you see here, what we have is a particle underscore utils class which contains all of our explosions

as well as any kind of utilities that we might need for some particle simulations.

Notice that the actual architectural paradigm that we're following is exactly the same as we have seen with other factories.

So we have an explosion in them.

If I go in here, you can see that we have an explosion in M, which has a large particle explosion, a medium particle explosion, a fiery explosion for the spaceship.

And we have a bonus explosion.

Then we have an explosion.

Abstract class, which defines the elements that are going to be common to all explosions.

Notice one very important method here, which is the particle simulation.

That's going to be the heart of the actual explosion class.

If we look at the concrete implementations, if we look at the explosion 360, which is just the generic name for an explosion that goes in all directions.

Notice that we extend the explosion and then what we do here is we have our particle simulation right here, which is exactly the same code as we've seen in case Study seven.

This is going to be just a normal explosion.

And we have also a particle bonus explosion, which is basically the same as this one here, except that it's just white color versus the original color here is red.

And then we also have a fiery explosion.

It's just an explosion that happens to be, in this case, a Sprite based explosion.

And as you can see here, we have the particle simulation that we get.

We basically have a Sprite animation particle and we get the boom explosion that we show when the spaceship blows up.

Then we have, as usual, a factory.

So in this case, the factory basically just looks for the different enemies and it creates the necessary elements.

And right here, as you can see, it actually creates the particle simulation.

So what the factory actually returns is a particle component.

So this is the factory.

And of course, just like in all the other examples, we have a built context.

So how do we use this?

Well, just like with all the other elements or objects, if you will.

In our application, we use mostly commands to tell the different elements what to do.

So let's have a look at some of the commands that we have.

And if you look here, this is all the commands that we have.

And if you look here, we have three commands.

We have an explosion of spaceship, which basically all that this command does.

As you can see, it gets the spaceship position, it gets an image information for the sprite.

And then what happens is it basically just asks the factory to create the explosion.

We also have an explosion of destroyed asteroid render command, which if we go here, basically the same idea, we're using the explosion factory to create an explosion.

And the same thing goes for explosion of split asteroid render command.

So if I run this right now, you will see that we'll have different types of explosions.

I am really bad at aiming with the mouse.

We don't have any sound at this point.

You can see there's another explosion.

This explosion happens to be the one that destroys the actual spaceship.

Another spaceship shows up.

So I'm going to.

Had that one.

As you've seen, it was split.

So you can see, like if you're looking at this, you can see that there is different types of explosions that we have.

Just going to show it quite a bit.

So this is working relatively well.

And then there's the spaceship explosion.

Now, let's say we wanted to add one more explosion here.

So what I'm going to do is I'm going to add an explosion for our bonus.

As you can see here inside the actual factory that we have an enum, which is bonus explosion.

But if you look at the commands, we don't have an explosion for that.

So what I'm going to do is I'm going to create one.

It's going to be basically very similar to this.

I'm just going to copy this whole command.

I'm just going to call it explosion of game bonus.

And all that we're going to do here is instead of the explosion in being medium particle explosion, we're going to use the one for the bonus.

And that's basically it.

The next thing that we would need to do is we need to wire this explosion so that whenever the bonus shows up and it's blown up, we can actually show that explosion.

Because if I run it right now without anything.

There it is.

That's our bonus.

And as you see, when I destroy it, nothing happens.

So what I'm going to do right now is I'm going to wire it so that we can actually get an explosion when we hit the bonus.

I just paused the game right now.

So let's go into where the actual bonus is exploded.

Let's go into our game bonus.

And if we look into the game bonus, you'll see that it's exactly the same as all the other ones.

We have an enum, we have a factory, but in our abstract class we actually have collision detection.

As you can see right here, we have the collision detection.

So in here we have a game bonus collision command.

But we also need to add the explosion.

So what I'm going to do is I'm going to go into our asteroid collision.

We'll see, for example, that in the collision we're using right here, you can see that we have explosion of split asteroid render command.

And for example, here we have an explosion destroyed asteroid.

So I'm just going to grab this and we're going to go back into our game bonus.

And we could either go inside the collision, So let's go in here.

We could actually call it from right within here.

So, for example, if the collision happens, what we're going to do is we're going to, for example, do it right here.

And of course, our target is not an asteroid.

So we're going to make it.

It's going to be gain bonus.

That's going to be our target.

And we just have to make sure that the explosion is added.

We get the position this is all good.

So now that we've changed the target, let's go back into the collision.

And instead of the target here, which is going to grab this target, we're going to put it here and that should be good enough.

And instead of, of course, this being explosion destroyed asteroid, we're going to have the explosion of game bonus.

So.

Okay, So let's run this right now.

And let's see what happens.

The game bonus is going to show about 15 seconds after, so we just have to survive.

So I'm going to shoot a few things.

Without trying.

There you go.

Without getting destroyed.

I missed it.

So unfortunately, now I have to wait 30 more seconds.

So let's try to get rid of some of these.

So.

We should have another bonus coming pretty soon.

There it is.

And as you can see, we have a wide explosion right now.

So this is how we have wired it.

Let me pass this.

And as you can see that all of this follows a certain pattern.

All the factories, all the objects, all the commands, they follow a very simple pattern.

Let's talk right now about commands within our architecture and our game.

How do we know what additional commands we need?

The command design pattern is not always useful in every circumstance.

So in your other projects, you need to really weigh the pros and cons of using a command design pattern.

In the case of our architecture, we have made a decision to use it to simplify the interaction.

Since there is a lot of messaging between different components, oftentimes a component will remove another component or a component will update a score, etc. This is handled pretty well by commands, but how do you identify the commands that you need when you're thinking about it?

Think of the following elements.

You want to let some object.

Know that you want it to do something, but you do not care about how the object does it.

So basically you have an action in mind like destroy this ship.

Commands are usually actions.

In other words, they follow the do this action for me paradigm.

Please look through the final code and study each command as follows.

Look at what it does and look at how it is invoked.

The idea behind the command pattern is that it's an encapsulation of some sort of an object.

If you interacted with that object directly, you would then have to know how to call the API for that object to do something.

That means you have to know exactly what the API is.

But your object, the one invoking the other one, does not need to know that.

So we encapsulate it inside a command where basically object A tells object V.

I would like you to do something for me, like let's say destroy the ship.

We don't really have to know how the ship gets destroyed.

We just want it done.

So this is the idea behind command design power.

One other thing that would be interesting to notice is when you look at all the different commands that we have, the way that these commands work is usually the commands themselves are pretty much on their

own, but sometimes a command can invoke another command.

That complicates things a little bit.

I mean, we do have some, for example, this user type command.

What it does, it fires a bullet and then it also creates the sound so that, for example, when I'm

when I'm shooting, not only is the bullet being fired, but we're also summoning the sound for the bullets.

So this is kind of nice because it's pretty straightforward.

If I wanted to do something else in here, I could just add another command.

And as you can see, we have a number of different commands.

If you look, for example, that came over command or that the game over command does is it checks whether

the score is greater and if it is, what it does is it saves the high score in persistence so that when

the game is done and the score that the user has now is bigger than the high score, then this will

write the actual high score into persistence.

So all of these are almost like messages.

We're just sending messages to all the different elements inside here and this is all done via the controller.

So please have a look at all these elements and just have a look at how they're working, especially

the broker, which is pretty interesting because all that the broker does is it facilitates for the

controller the execution of all the commands.

All right, let's go on to the next topic.

So let's talk about the final architecture.

We have provided you with the architecture as a full HTML image because it's a lot of elements.

It would be difficult to show it on a screen and go through it.

Given that we have already done quite a bit of HTML, you should have no problem following the final

architecture.

HTML image.

The image is high definition, so you should be able to zoom in and out of it.

The main thing to notice about the architecture is the role of the controller broker and the whole command

structure.

Most objects are created through a factory method and most interaction is through controller and the

broker via commands.

So basically what you see is a pattern that we have all sorts of classes like bullets, asteroids,

and then we have factories that create them.

And we also have a way of interacting between the different classes by using commands that are sent

to and from each other via the controller in the broker.

You can see that the organization of the code when looking at the HTML makes it pretty clear what is

going on when you look at the HTML.

You can see that there is a pattern and a structure.

If you want more actions, you can simply add more commands.

If you want different objects, like, for example, a new spaceship, you can create more entries in

the specific factory.

In general, the point of a good architecture is that you should be able to figure out where things fit in in the scope of the project just based on that kind of bird's eye view.

It is ultimately, in the end, about good organization.

Let's talk about issues with collision detection.

Inflame 1.0.0.

There are some issues with collision detection.

It's not really the greatest.

It does not really use something called CCD, which is the continuous collision detection algorithm.

This means that you might get some missed hits and missed collisions, especially if you have a lot of objects on the screen.

There are improvements in the later versions, such as, for example, the 1.1.0 release candidate five and the ones after.

At some point we're going to update this course, but that improvement does not have continuous collision detection.

If you want continuous collision detection, you can also explore Forge 2D, which can be relatively easily incorporated into flying projects.

If you want to know more about Forge 2D.

Here is a resource to check out.

The beauty of our architecture is that because of the fact that we're using command design pattern, the collision detection itself is relatively easily refactored.

You could easily bring in Forge 2D, and instead of doing the flying collision detection, you do the Forge 2D collision detection.

The question, of course, is when you detect a collision, what do you do?

Well, because we have everything encapsulated in commands, you simply call the right commands.

And that's exactly the same way as in the previous collision detection.

So refactoring our architecture to use a different collision detection mechanism will be relatively straightforward.

I would say give it a try.

Let's also talk about aspect ratio.

In the previous case study, we have seen that there is an issue with the resolution multiplier.

So here's a quick note about aspect ratios and issues.

With our resolution multiply resolution aspect ratio is the relative ratio of the resolution width and height.

For example, 1920 by 1080 has an aspect ratio of 1.7, seven, seven.

We're simply dividing the larger number by the smaller number, 1280 by 720 has exactly the same aspect ratio of 1.7, but on the other hand, 640 by 480 the aspect ratio is 1.3, three, three and so on.

When the ratios are different, our multiplier will skew the results.

So what is the solution?

When aspect ratios are drastically different, then more careful consideration is necessary as it could produce weird looking results.

In our case, we'll assume that the aspect ratios of the different platforms will be the same as our base resolution, which is 1280 by 720, which has the aspect ratio of 1.777.

It is an interesting problem though, to think about, and I implore you to do a little bit of thinking about it.

All right.

So now that you have seen the code working, I suggest that you go through the architecture and the code and have a look at how everything is working together and just play around with the code.

All right?

Remember, always keep on coding.

See you later.

11. Deploying Your Game to a Mobile Device

Welcome back.

Let's talk about deploying our application, as you remember, from previous sections.

We start with a code base.

We then have a flutter app.

Flame is built on top of flutter, so it actually becomes a flutter application.

We use the flutter UI rendering to either deploy and render our application on Android or we can deploy it and render it on iOS.

So how do we do that exactly?

Let's start with Android deployment.

What I'm going to show you here is how to deploy our game to an Android device.

The simplest way in which you can deploy your flutter application onto an Android based phone is to generate an APK file, which you can then site install.

So the way you generate an APK file in Visual Studio is very, very simple.

You go into terminal and then you have a run billed task.

Are you going to have a number of different options?

What we're going to do here is we're just going to say flutter build APK.

So I'm going to choose that.

And as you can see, this opens a terminal and what will happen right now it's going to run the gradle element of your application and it will generate an APK.

Now you might get sometimes hear a bunch of warnings or even what looks like an error, but in the end that's just probably Android studio complaining.

But in the end what you will see is the location of where your APK has been created.

You will then take that APK and you can copy it to your physical phone, which is what I'm going to show you next.

Once this task is done, this task takes a little bit of time.

So we're just going to wait for it to finish.

Now, one thing that we haven't done here, we haven't really set up an icon for the application and we haven't really set up a specific name.

That is something that you can do if you want to publish the game.

At this point, it's just going to show us a generic flutter icon application with the name of component underscore 008 underscore game.

Don't worry about the red stuff that you see here.

We're just waiting for the task to finish.

And there it is.

So you can see that it has put the application inside.

App Outputs Flutter APK.

App Release APK.

So what we're going to do is we're going to go in there and windows just hold the control key.

It shows you that's where it is.

You can right click on it and say review and file Explorer and it shows up right here.

We have it right here.

This is the one app release dot APK.

You can rename this if you want.

I'm just going to name this into, let's say Asteroids dot APK.

And what we're going to do right now is we're going to take this and drop it into our phone.

And the way you do that, I'm going to open one more file Explorer.

The way you do that is once your phone has been connected via USB, you can go to my PC and you're going

to see, for example, this right here.

This is my phone.

This is a Galaxy S10 Plus, if you go in here, you can go into internal storage and you can go, for

example, into just to give it a second, you go into download.

So what I'm going to do is inside the download directory for my.

Phone.

What I'm going to do is I'm going to create a new folder and I'm just going to call it asteroids.

So it's just easier.

And I'm going to go inside here and I'm just going to take this APK and I'm going to drop it in there

that has been copied.

What we're going to do now is we're going to go into my phone and actually just see how you would install

it.

So on my phone, all I basically do is I go into my file system and I'm going to go into internal storage,

then download and then I have right here asteroids.

So when I go in here, all I have to do is click on this and it will allow me to install it and then

I'm going to say install anyway.

And once it's installed, I'm not going to send anything.

We're done.

And you can now see that the actual application has been installed right here.

When you can see, for example, that here we have very simple right now.

This is like the older version of the code.

The code you're going to be running is going to be fuller.

But basically you can see that the application is working,

working pretty well.

All right.

So much for that.

Let's talk about iPhone deployment.

So what I'm going to show you here is how you would deploy your code base to an iPhone.