Hi. In this video, I’m going to take you through how I used C++ and Splashkit to create a short animation which recreates the famous gun barrel sequence from the James Bond movies.

I’m going to be taking you through the two main systems that I have written which work together to make this animation possible: The rendering system and sequencing system.

# The Sequencing System

The way I went about animating using code is through a system I call ‘The Sequencing System’. The idea behind the sequencing system is to split the animation into smaller sequences. I’m going to be referring to sequences a lot in the video so keep in mind that sequences refer to smaller parts of the complete animation. This will start to make more sense once I explain how the system works.

I’m going to start in the program.cpp file and up here you’ll see that I have defined a constant up here that tells us the number sequences the animation is made up of.

The programmer does not have to know the exact number of this straight away and can increase and decrease the number as they wish, should they want to add or remove sequences from the animation.

Down here in int main, I have defined a vector that stores sequence\_progress structures. Let me show you what a sequence\_progress structure consists of.

So here have two Boolean values. The first indicates whether a particular sequence of animation is complete and we are therefore ready to display the next sequence.

I will get into what the second Boolean value does later on when I explain the rendering system.

So back here in int main, we have a for loop here that populates the sequence\_tracker vector with the number of sequences in the animation.

How the program uses this array is that it identifies the index number of each sequence\_tracker element as a sequence in the order of sequences in the animation. So for example, the sequence\_tracker element at index 1 is going to tell us whether the second sequence of the animation is finished or not.

Similarly, the sequence\_tracker element at index 3 is going to tell us whether the 4th sequence of the animation has finished or not.

The way the gun barrel program knows that the full animation is complete is when all of the sequence\_tracker elements’ complete values are all set to true.

I’m going to move down here now to another for loop which determines what sequence the program has to actually play at any given moment throughout its runtime.

The idea is that this block of code iterates through the elements in the sequence\_tracker array and checks for the first element that contains a complete value set to false.

Once this happens, the program calls the sequence using the index location of that element and the corresponding sequence is then carried out on the screen.

I’m going to show you what the sequences procedure actually does now. So, I’m going head over into the rendering.cpp file where it lives.

So the sequences procedure consists of a switch statement whose case is chosen based on what index number we call the procedure with. You’ll also notice the populate procedure here which I will again, get into when I explain the rendering system.

Basically, what’s happening in this procedure is that it is choosing which sequence to display on screen.

So if I go up here to the sequence 1 procedure, you’ll see the code that I have defined that makes up all of the moving parts on the screen for the first sequence of the animation.

This sequence is for the two circles that move across the screen at the very beginning. You’ll see that it consists of a couple of procedures that tell the circles to move across the screen. These procedures are located in gfx\_animations.cpp, which I will quickly show you now.

So the gfx\_animations module is dedicated to housing the procedures that helped me with the manipulation of objects on the screen. So for instance, we have two procedures here that I used as a base: animate\_circle and animate\_bitmap.

Essentially all you had to do with these functions is specify which property of the object you wanted to manipulate and then set the value of that property.

The procedures that come after it down here take those base procedures and do something more specific with them. Like here with circle\_translate. It takes the circle we want to manipulate, along with the horizontal location as the property we want to change and then we also have the distance that we want to move the circle.

We also have the same down thing down here for moving bitmaps. This version of the procedure is a little bit more flexible as it allows for movements in both the vertical and horizontal direction.

I’m now going to head back to our rendering module and finish off explaining how these sequences operate on an individual level.

So how these sequences operate is that each of them stores code that describes what is to happen in the sequence one frame at a time. The reason for this is because the gun barrel program repeatedly executes these 60 times a second, giving us the illusion of movement through the code it runs from these sequences.

What I also needed to make sure of is when the program knows that a sequence is finished. So every sequence procedure has what I call a complete condition and in sequence 1, you will find that right here. This condition put plainly, says that if the circle has not completely moved off the screen, then execute this code right here.

Otherwise, if it has, then we’ll say that this sequence is now complete by changing the complete value of the sequence\_progress element whose index we ran the sequences procedure with. Now the program is ready to move onto the next sequence.

Further down here we have the sequence procedures for the other sequences that make up the gun barrel animation and these are all set out similarly to the first one.

They all contain the code that is executed every frame and they all have a complete condition that sets their respective sequence\_progress element’s complete value to true once the condition is met.

Once all of the sequences have been ran and all of their complete conditions have been met and all the complete values of the sequence\_tracker array are set to true, the full animation is complete.

So that is a break down of how the sequencing system works in the gun barrel animation. I will now explain the process of how the sequences are actually drawn onto the screen using the rendering system.

# The Rendering System

The idea behind the rendering system is pretty straight forward. You give it all the arrays that store all the things you want to draw onto the screen in a single frame and it calls the draw procedures provided by Splashkit for each thing you want to draw.

To accomplish all of this, I’m going to start from ground level here and show you the data types that I have defined that the system works with to render everything.

Down in the middle here, we have circle\_properties, rectangle\_properties and bitmap\_properties. Each of these structures contains the properties that the rendering system needs to work with when it comes down to drawing things onto the screen, such as where it needs to be drawn, how big it may be and in the case of bitmaps, the name of the image being drawn.

I’m now going to take you back to the program.cpp file just quickly so that I can show you what artefacts I used for this rendering system before I go in-depth with how it all works.

So in our int main function, I have a few vector arrays defined here, circles, rectangles and bitmaps. These store the shapes and images that are needed on screen at any given time.

In terms of the images the animation needs, it’s also important that we load them into memory using the load procedures provided by Splashkit up here before we can start putting the images into the bitmap arrays.

I’m now going to take you back into the rendering module to have another look at the sequences procedure.

So right here back in the sequences procedure, the procedure calls another procedure called populate.

The rendering system has two rolls: to ‘populate’ the arrays I mentioned in the program.cpp file with the required shapes and images needed to be drawn on the screen as well as actually drawing everything in the arrays onto the screen.

I’ll now take you up to the populate procedure to show you what it does. So the populate procedure works by determining what shapes and/or images it will need during a specific sequence.

It first determines which sequence it is executing, which it already knows from the sequence procedure being called. It takes the sequence number and it executes the code for loading in the shapes and images for that sequence.

Right before it does this however, it checks to see if these have already been loaded into the arrays. Let’s have another look at the sequence\_progress structure in the datatypes header.

So the sequence\_progress structure has a second Boolean value called graphics\_loaded. This keeps track of whether we have loaded the shapes and images into the arrays for this sequence or not.

Back here in the rendering module, each sequence in this procedure requires that the program check to see if the shapes and images for the sequence that it is working with has been loaded.

If it has not been loaded, it will load what the sequence needs and then says that the graphics for the sequence has been loaded.

The reason for this check is to make sure the program only loads the graphics once and not repeatedly which will make everything to double up on the screen and cause graphical glitches.

Once the rendering system has figured out which graphics to load into the arrays, it is now ready to draw everything to the screen and it is done through this procedure that is called here in program.cpp, the render procedure.

The render procedure calls three more procedures, all of which do the same thing, just for different kinds of graphics. You can see these procedures up here, they run some sort of draw procedure for each of the elements in the array they are given.

The render\_bitmaps procedure works a little bit differently, however. The rendering system draws images from the array it’s given by starting from the end of the end of the array and working its way to the start. The reason for this is so that images that are towards the end of the array are considered as background images and ones that are more towards the front are drawn over the top of those more towards the back and are hence, considered foreground images.

I chose to do things this way so that it makes more sense to the programmer as to what images should go on top of other images during an animation.

Ok so this brings me to the end of the video. This is how I managed to use C++ and Splashkit to make an animation inspired by the gun barrel sequence from the James Bond movies. I hope you found this really interesting to watch and thank you for watching.