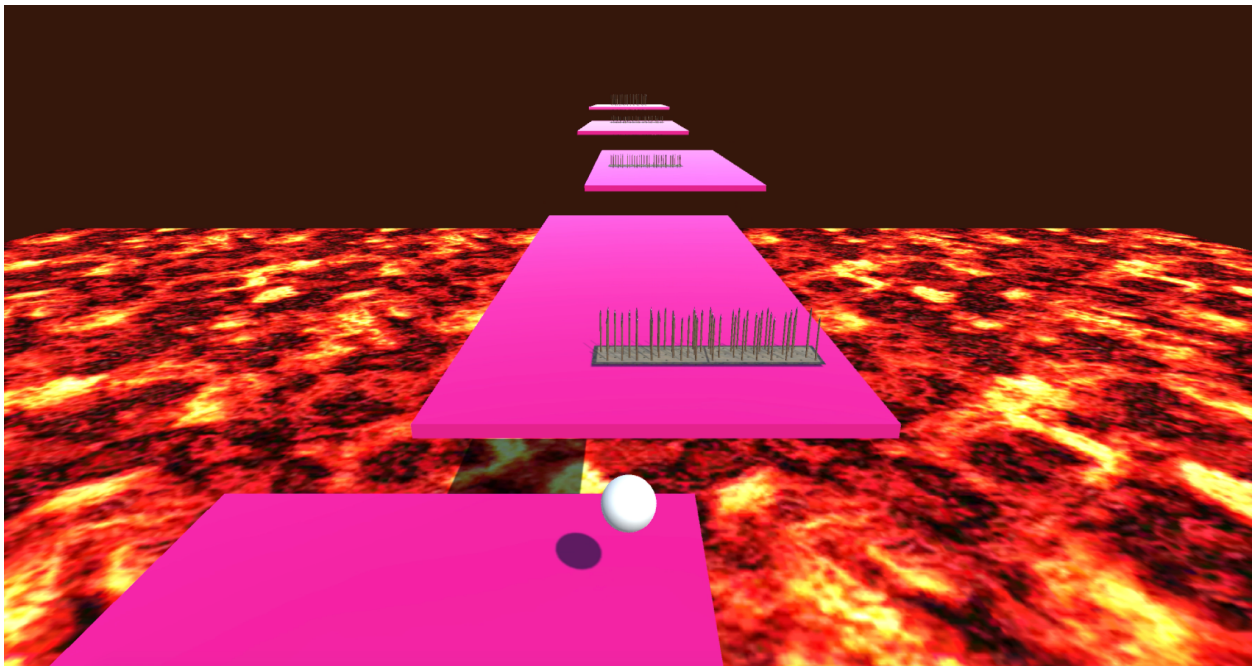
Victor Alvarez
Mia Johansson

CS175: Lavascape Unity Project

Github: https://github.com/miajohansson9/175finalproject/tree/master



For our final project, we implemented a game in Unity called Lavascape where a player jumps through staggered platforms and avoids moving spikes. The game starts off slow and speeds up, increasing the difficulty. Eventually, the player will fall into the lava or run into the spikes. When this happens, the player will die and a “Game Over” screen shows up. The player’s score is displayed and the player can restart to play again.

Section 1: Implementation

Initializing Objects

Before we could code any of the game mechanics we had to have objects that we could manipulate. We declare a sphere object for the player model and name it “Player” in the

hierarchy. We then create a cube object for the floor and name it “Ground” and another cube object for the lava and name it “Lava.” We created two materials:

- One material that uses the standard shader and we changed the albedo so that the material had a hot pink color. This material was assigned to the floor
- The other material uses a “diffuse” shader and we give it a picture of lava to use as a texture. We resized the “Lava” object to be very large, so we had to change the tiling of the texture so that it would not look too pixelated or distorted. We decided on a 5x5 tiling of the texture which resulted in a smooth enough image of lava when playing the game.

Player Movement

The player movement allows the player (the white sphere) to move left, right, and jump. We created a script for the player and called it `PlayerMovement`. Unity has a built-in “Input Manager” which has a built-in “Jump” input. Inside of the `Update` function in `PlayerMovement`, we check for the “Jump” input and then increase the y’s velocity by 7.0. We get the horizontal input with `Input.GetAxis("Horizontal")` and use that value, multiplied by a speed of 7.0, to update the x velocity. We intentionally do not allow the player to move in any other directions.

Collision Checking

Unity makes it very simple to make objects “solid” in that they cannot phase through each other. This allows the ball to not fall through the floor. It’s as easy as adding rigid body and box collider components to the objects. However, there are specific collisions that we want to check for so that the game plays as expected. For example, we want the game to end when the player touches the lava or spikes, and we only want the player to be able to jump if they are currently touching the ground.

Checking for these specific events required adding some code to the player movement script (PlayerMovement.cs). First, we add a method to the PlayerMovement class named `IsGrounded()` which, as the name suggests, checks whether the player object is touching the ground. However, we first had to define what the "ground" was as we don't want the player to be able to jump on the lava. To do this we define a new layer in Unity named `Ground` and set the "Floor" objects to have this as their active layer. We then add a layer mesh around the floor objects. This allows us to access the layer in the PlayerMovement.cs script. Then, we add an empty object to the bottom of the player model which contains only a transform component. We title this component `groundCheck` and give access to it in the PlayerMovement.cs script. Now that we have access to all of these components the `IsGrounded()` function simply calls the following:

```
Physics.CheckSphere(groundCheck.position, .1f, ground)
```

This function creates a sphere of radius 0.1 centered at groundCheck.position which will be the bottom of the player model. It then returns whether or not the sphere is colliding with the final argument which in this case is `ground`. We then call this function in the `Update()` function which is called every frame to determine whether or not the player is grounded when they attempt to jump. If the player is grounded, then pressing space will cause the player to jump. Otherwise it does nothing.

To check for collisions with the lava we take a very similar approach except that we now define a new layer named "lava" and set it as the active layer of the `lava` object. We then also create a layer mesh component for the `lava` object which allows us to access it in the script file. Then, using the same `groundCheck` object defined earlier we call

```
Physics.CheckSphere(groundCheck.position, .1f, lava)
```

which returns whether or not the bottom of the player model is colliding with the lava object by using the same method as earlier, creating a tiny sphere at the bottom of the player model and checking for collision.

Lastly, we check for whether or not the player has collided with the spikes. The spikes were a game object that we purchased on the Unity Assets store. These game objects come with a mesh so that they can be drawn, so we very similarly added a layer mesh component and created a new layer named “spikes” and set it as the active layer. For this collision we had to take a slightly different approach because we didn’t only want to check if the bottom of the player model was colliding with the spike, but whether any portion of the player model collided with the spike. For this we create a new empty object titled “frontCheck” that is a child of the player object. This object consists of only a transform which is centered at the center of the player model. Since our player model is a sphere it was very easy to then change the function as follows:

```
Physics.CheckSphere(frontCheck.position, .5f, spike)
```

This function creates a sphere of radius 0.5, which is the same radius as our character model, and checks whether it has collided with a spike. This has the effect of returning whether or not the player model has touched the spikes at all.

We call the `TouchedLava()` and `TouchedSpike()` functions in the `Update()` function to determine whether the game has ended or not. If they return true, we display the game over screen. Otherwise, the game continues.

Moving Platforms and Floor Generation

To create the illusion of the player moving, we move the platforms back with a speed of 7.0 in a script called `FloorMovement`. We only have one Floor defined in our Hierarchy which

is obviously problematic. The player will quickly reach the end of the platform and fall into the lava. To generate new platforms, we call `Invoke("GenerateFloors", 0f)` in `Start()` until we reach the floor limit of 5. We set this limit to not generate too many floors at once and to also re-use already generated floors. `GenerateFloors` uses `Instantiate` to create a new floor.

In `OnBecameInvisible()`, we update the transform's z position to be `transform.position.z + 14f * floorLimit`. This updates the z position to be right after the last platform. Because our platforms are staggered like a staircases, we also update the y position to be `transform.position.y + (1f * floorLimit)`. We chose to re-use platforms instead of just creating new ones for performance purposes. This also avoids having to generate and destroy platforms every couple of seconds.

Moving Spikes

We include moving spikes as a deadly obstacle for the player. These spikes move up and down and have random positions along the x axis and z axis. Their widths are scaled randomly so that some are longer than others. To do this, we first downloaded the dungeon trap asset found on Unity's store here. We imported it and added it to the first floor defined in our scene hierarchy. Because the trap was nested inside the floor, it was automatically generated each time a floor was generated. The position also follows the floor when it gets moved.

The moving spikes were more difficult to implement. The trap object has two children objects called the base and the spikes. The spikes are initially hidden and move up through the floor. We added a `Rigidbody` component to the spikes as well as a script called `TrapMovement`. Inside of `TrapMovement`, we generate a random float between 0.5 and 1 to determine the distance the spikes come up. Based on the spikes' current position and this value, we set

global min and max positions for the spikes. Their positions are changed in Update() by using the `Mathf.PingPong` to move between positions.

A problem occurred after the floor was re-used. Because the spikes are moving as this happens, their current position becomes their startPosition and then the min, max values don't reflect this. Furthermore, min, max are initialized in Start() which doesn't get called after the spikes are generated the first time. To fix this, we made min and max *public* variables and changed their values in the `OnBecameInvisible` function in `FloorMovement`. This was to ensure we used the correct start position. The `OnBecameInvisible` function was also not called in `TrapMovement` since the floor gets moved before this happens.

Finally, we randomly set the z and x positions of the traps to vary the traps along the platforms. We also changed the scale by a random value to make some traps smaller than others. To do this, we created a new script `Trap` and defined `transform.position` with the new values. We used `transform.parent.position` in generating the new random positions to ensure the trap stays on the platform.

Score System, Difficulty Increase and Game Over Screen

The score system for the game is very simple. We decided to use a continuously increasing counter as the score for the game, essentially serving as a timer that says how long the player survived. To do this we made use of the fact that the `Update()` function is called every frame. We created a `frames` variable and increment it in every call to `Update()`. We also created a `points` variable. Every time the `frames` variable reaches 60 we increment the `points` variable and set `frames` back to 0. This gives the effect of the score incrementing approximately every second as the default frame rate is 60fps.

To display the score on the screen we created a `Canvas` object which allows you to create UI elements that will then be overlaid on the screen. We created a simple text box element that we placed in the upper left corner of the screen. We then give the PlayerMovement.cs script access to this text box and pass in as text the current score with every call to `Update()`. Thus, every 60 frames you should see the score increment in the top left corner.

We used this same `Canvas` object to create the game over screen. To do this we created an image object as a child of the `Canvas` object and named it `Background`. We made the image a black background, and added as children of the `Background` element two text boxes. The first text box simply says “Game Over” and nothing else. The second text box is filled in with the score at the time of the game over event through the PlayerMovement.cs script. We also added a button as a child of `Background` and added the text “Restart.” Unity then allowed us to attach a function to be called when the button is clicked. The function we attached was the `Restart()` function which uses the Unity ScreenManager to set the “Game” scene which is the initial state of the game. To bring up the game over screen to begin with we have a `Setup()` function defined in the GameOver.cs script which is called whenever the player touches the lava or a spike.

Section 2: Issues and Bugs

One of the bugs that we were not able to resolve is the spikes occasionally going off the edge of the platform. This doesn't affect gameplay but we would ideally want the spikes to always be within the platform. In the Trap.cs file we attempt to define the x-position of the traps in such a way that they will always be contained within the platform, but they still appear off the edges of the platforms. We think the bug may have been caused by the scaling of the traps, since

we also randomly assign the x-dimension of the traps. Then, when we tried to define the interval of positions as follows:

(platform x-position) ∓ (platform x-dimension - trap x-dimension) / 2

the traps would occasionally be partly off the sides of the platform. To access the x-dimensions of the objects we used the lossyScale attributes of the transforms which returns their scale in world measurements. We thought this would work as Unity uses world coordinates when setting position vectors but the bug persisted. Ultimately we decided to define a strict interval of positions that the traps could be placed between instead of relying on the math to get the traps to appear anywhere on the platform.

A feature we did not end up adding was the platforms going down or up randomly. Instead, we just incrementally increase them like a staircase. This made some of the math either because when we change the y position in `OnBecameInvisible()`, we know how much higher the platform should be.

We had originally looked into generating random holes for our player to fall into but realized that this was a bit too advanced for us as it involved creating a custom mesh and altering it at run-time. It was also expensive to do this every couple of seconds, and we would ideally have holes in different locations for every floor. One option was to manually create a lot of platforms in the scene hierarchy and create holes beforehand, but this would prevent us from just being able to have one Floor that we re-generate and re-use its copies.

We also noticed after we created the builds for Mac and Windows that the font size on the Mac build is very small whereas the Windows build looks normal. We didn't notice this as we were coding since the simulator never showed any differences on the Mac vs Windows.

Section 3: What We Learned

Working on this project we got a better understanding of how collision detection with rigid bodies works, and how difficult it can be to correctly identify different collision events. We also got some experience with designing a UI for the game which was not something we covered in class, but was interesting to experiment with. Since Unity allowed us to abstract away a lot of the lower level functions that we would otherwise have to implement, we were able to focus on game design which was rewarding and eye opening. Even with this level of abstraction it is not completely trivial how to efficiently design code that handles different events for various objects simultaneously.

Before this project, neither of us had any experience with Unity and there was definitely a learning curve to working together and navigating the scene view. With OpenGL, everything was created with code and so creating objects in a scene hierarchy was new to us. Youtube videos helped a lot with learning in the beginning.

Section 4: Resources

We used the first few YouTube videos in this tutorial as baselines for some of our features such as ground checking. We used this video as a guide for the game over screen. We then expanded on it to create the score counter.

As mentioned in Implementation, we also used the dungeon trap asset found on Unity's store found here. When we ran into bugs, we searched through forums, StackOverflow, and blogs to find help.