HARVARD UNIVERSITY

Bloxorz Unity Project

Author:
Adriana ROTARU
Sebastian REVEL

Guided by:
Professor Shlomo GORTLER

A final project based on Graphics built in Unity

CS175 - Computer Graphics

May 12, 2021

Contents

List of Figures

0.1 Project Description

For our final project, we worked on making a game akin to Bloxorz in the *Unity* game engine. We were playing Bloxorz when we were young and we thought it would be fun to recreate our own version of the game. The game was built in the 3D framework of the Unity Hub, version 2020.3.6f1. To implement our functioning prototype, we utilized various tools within Unity such as its shader tools, physics engine, scene hierarchy, particle system, lighting engine, and more. We will break down each component of our project and discuss our different design decisions as well as hurdles we ran into.

THE GAME CAN BE ACCESSED AT THIS LINK.

0.1.1 Scene Setup

Description: The game scene contains the important bodies (GameObject) with their assigned shaders, textures, materials. Each object in the scene has a position, rotation and scale properties, which can be controlled from the Unity's `Inspector` Window. Some objects can be parents of others, in which case, each child's position and rotation are relative to the parents' and a change in the parent's properties changes the properties of all children. The player is able to see the number of moves it makes, by looking at the **Moves** counter, which increments with every roll of the brick. The goal is to finish the mission (getting the brick on the lava puzzle) in as few moves as possible.

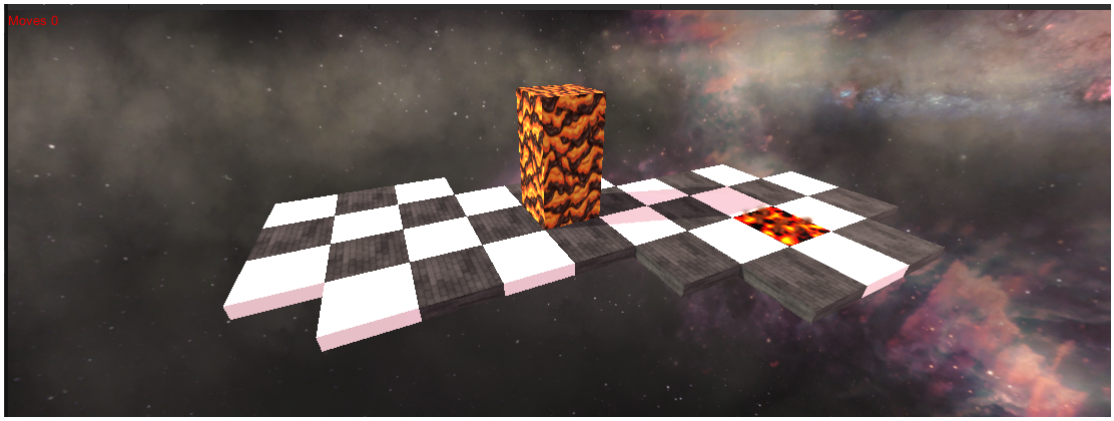


FIGURE 1: Game View

Implementation: The game has a single SampleScene which includes the main GameObjects. The GameObjects are :

1. The **Player** (shown above as the brick)
2. The **Directional Light** (acts as a lightbulb emitter of natural light)
3. The **Floor**
4. The **Focal Point** (discussed in 0.1.2)
5. The **Fog** (discussed in 0.1.6)
6. The **Smoke** (discussed in 0.1.7)

The **Floor** is in fact a rectangular ensemble of puzzles, of which some are not rendered, to create a non-uniform floor shape. The Floor Game Object is a single cube at the lower left corner of the platform with many children (the other puzzle pieces). The puzzles that are

not rendered are used to detect collision when the brick falls out of bounds, which is the condition for game over, described in 0.1.4 and in 0.1.9. The following figure shows a bone structure of the floor:

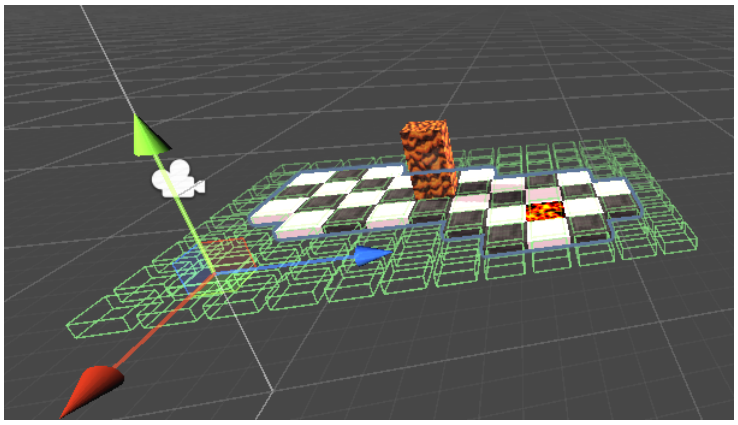


FIGURE 2: Floor Structure

The textures for the **Floor**, the **Player**, the spatial **Sky** were selected from the free sample textures on Unity Asset Store. The Environment that imitates a galaxy was applied 3D as a skybox, wrapped around the scene. The **Lava** shader applied on the special puzzle, is described in 0.1.7 and was created from scratch.

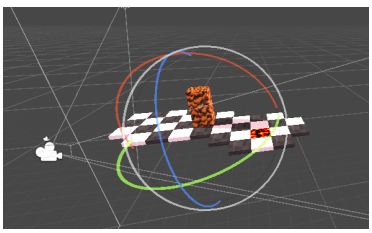
The **Moves** counter is a text display of a private variable counter that is incremented with each roll of the **Player** object on the **Floor**.

Difficulties and Lessons: The **Floor** object could not be turned into a RigidBody object, because it would fall under gravity. RigidBody properties are added to the floor when the effect of gravity is needed. The camera view was hard to manipulate so that it captures the entire game scene by simply using the rotation and positioning tools in Unity, so, first, the eye view was centered on the scene and then the camera view was aligned with the eye view, using `GameObject->Align with View`.

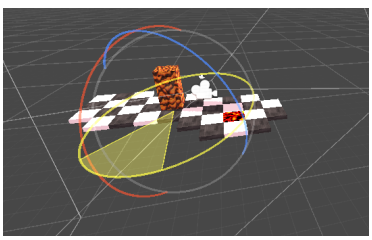
0.1.2 Player/Camera Movement

Description: Instead of having a static scene, the game player should be able to orbit around for a more enjoyable game experience. With orbiting around the directions of the brick rolling should align with the eye view and camera view, respectively. The **Player** rolling is disabled while the view is changing.

Implementation: To create the camera orbiting w.r.t. the center of the **Floor**, we deployed a trick, where we create an empty GameObject, the **Focal Point** having the camera object as its child. The **Focal Point** is located in the middle of the **Floor** GameObject. Since the camera object is a child of the **Focal Point** an arcball rotation of the **Focal Point** would cause the camera to follow it, as shown below:



(A) Initial Rotation Vector of Focal Point - Camera



(B) Rotation of the Focal Point - Camera pair at an angle

Applying `Ctrl + RightArrow` or `Ctrl + LeftArrow` as keyboard input causes the view to orbit around the scene, by changing the Quaternion of the **Focal Point** by an angle scaled

by the input amount and the `Time.deltaTime`. This is implemented in the `RotateCamera.cs` script which controls the **Focal Point** GameObject.

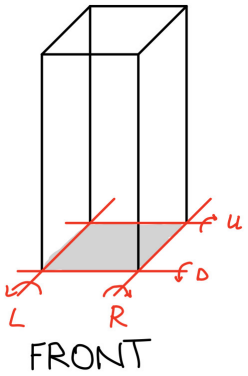
Testing: To test this functionality, simply orbit around the game scene using the `Ctrl + Left/RightArrow` in Windows.

0.1.3 Rolling

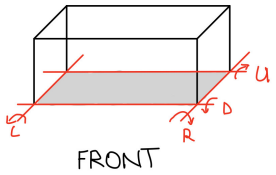
Description: The **Player** brick can be rolled on the floor such that it would eventually end up vertically on the **Lava**, which is the winning the condition. Rolling the **Player** out of the floor bounds, even if half on the edge, would cause the scene to collapse and the player to lose. The **Player** can roll in 4 possible directions (Left (1), Right (2), Up (3), Down (4)). Since only 4 directions of rotation are allowed, the brick has to exactly align with the floor puzzles in order to reach a winning condition.

Implementation: The entire rolling functionality is implemented in the `Rolling.cs` script, which is a component of the **Player** GameObject. The **Player** brick has 2 possible positions (vertical and horizontal), 4 directions of rotation (L, R, U, D) and the directions also depend on the camera angle view. The default camera angle is shown in Figure **??**.

First, let's discuss the rolling independent of the camera view, that is in default camera view position and rotation. Then we will get into how the directions of rotation are determined when the camera orbits around the scene.For the brick rolling, the brick will rotate around the axis of its base, and the choice of the axis of rotation is determined by the direction of the rotation. Therefore, starting from either a vertical or horizontal position, the brick can end up in the same or in the opposite position (orientation). The possible directions and states of the brick are shown in the figures below.



(A) Axes of rotation from the vertical position



(B) Axes of rotation from the horizontal position

Hence, there are 8 possible states of the brick (4 that start from the Horizontal and 4 from the Vertical). The rotation around an axes is implemented using the `Transform.RotateAround(Vector3 point, Vector3 axis, float angleSpeed)`. To make sure that the rotation is smooth, this function is called in a loop with linearly incremented angleSpeed, as a function of time, so that, the transition from one state of the brick to the other is animated across multiple frames.

In addition to computing the rolling in the default Camera/Eye view (from the front perspective of the floor), we had to figure out how the axis would change if the cameraView is rotated using the `Ctrl + Left/RightArrow`, as described in 0.1.2.

We define 4 possible quadrants from which the camera can capture the floor, and map the directions of rotations based on how the axes of rotations map to each quadrant, as shown in the diagram below:

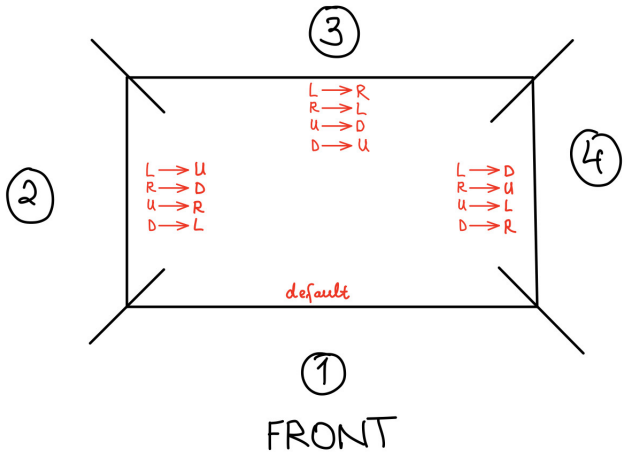


FIGURE 5: The Floor representation with the four quadrants of the camera view

There are 4 possible quadrants from which the camera can capture the game scene, and for which the directions (L, R, U, D) change accordingly. The function `computeQuadrant()` implemented in `Rolling.cs` computes the correct quadrant, by looking at the y component (the angle of rotation w.r.t around the Y axis) of the EulerAngles of the Focal Point quaternion. Certain ranges of these angles correspond to a specific quadrant. Then, the directions L, R, U, D are mapped based on the quandrant.

Testing: Simply manipulate the **Player** using the L, R, U, D arrows from the keyboard and use `Ctrl + LeftArrow/RightArrow` to control the camera view. The direction of roll should be intuitive. For ex, if your camera is in the 2nd Quadrant, a RightArrow should correspond to the motion of the brick out of the page in Quadrant 1.

Difficulties and Lessons: Instead of using rotation around an axis, we tried to implement linear interpolation between the 2 states of the brick, but the result was not physically accurate. Given that, the brick collides with the floor during the interpolation, especially near the rotation point, using SLERP and LERP led to inaccurate physical interaction between the floor and the brick. The intermediate frame "passes" through the floor, which would cause the base corners of the brick to pierce through the floor if the brick were not a rigid body, but since LERP and SLERP require RigidBody properties, when the brick becomes a rigid body, the interpolation causes the brick to dislocate and to fail to allign with the puzzles perfectly). We tried to disable/enable the RigidBody property of the brick, but there were other issues arising, so we went with completely disabling the RigidBody property of the **Player** brick and use rotations around an axes instead.

0.1.4 Platform Collapse

Description: In the event that the player goes out of bounds, we wanted to create an effect where the platform that the **Player** is on falls apart. Consider the following image:

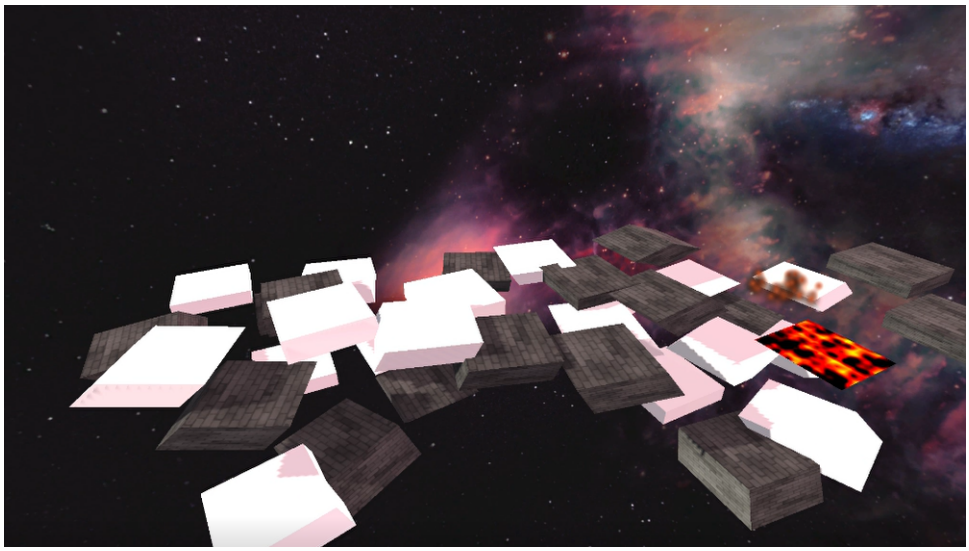


FIGURE 6: Floor Collapse

Implementation: In order to achieve this affect, we first needed to maintain access to all the tiles within the scene. From here, we need to modify each tile's Rigid Body component. Then, for each tile, we disable kinematics and apply a velocity to each object in a random direction. The direction does not matter since the force applied is not strong enough to send it too far. The random direction just ensures that the tiles do not fall identically. Disabling kinematics also allows gravity to affect the tiles which creates the falling.

Difficulties and Lessons: The main difficulty with this was interacting with the scene hierarchy and understanding how Unity handles rigid bodies. First, we needed to maintain a list of all the tiles in a scene and be able to reference it across scripts. After learning this, then it was understood that the rigid bodies **isKinematic** field allows for forces to be applied to the object based on whether or not it is true or false. Since we want the tiles to be static during game play, **isKinematic** allows us to maintain control over the objects physics within the rigid body component in Unity.

0.1.5 Player Fracturing

Description: Whenever the **Player** brick falls out of bounds, we add a special breaking effect to it, such that the brick shatters into tiny pieces. The **Player** object also gains RigidBody properties, so it falls along with the floor. The final result should look like this:

FIGURE 7: Breaking Effect

Implementation: There is no special feature in Unity that creates a breaking effect. Therefore, to implement the effect, we first used Blender, which is a free and open-source 3D computer graphics software toolset used for creating 3D printed models, motion graphics, interactive 3D applications, virtual reality, and computer games, etc. It is similar to Unity, but it allows for a more complex and detailed manipulation of textures and shapes. A similar **Player** brick was created in Blender, then it was fractured into many cells of random size and shape, using the `Cell Fracture` tool. A similar texture as the one for the brick was applied, and then the model was exported to be imported as a prefab in our Unity project. The fractured prism looks like this:

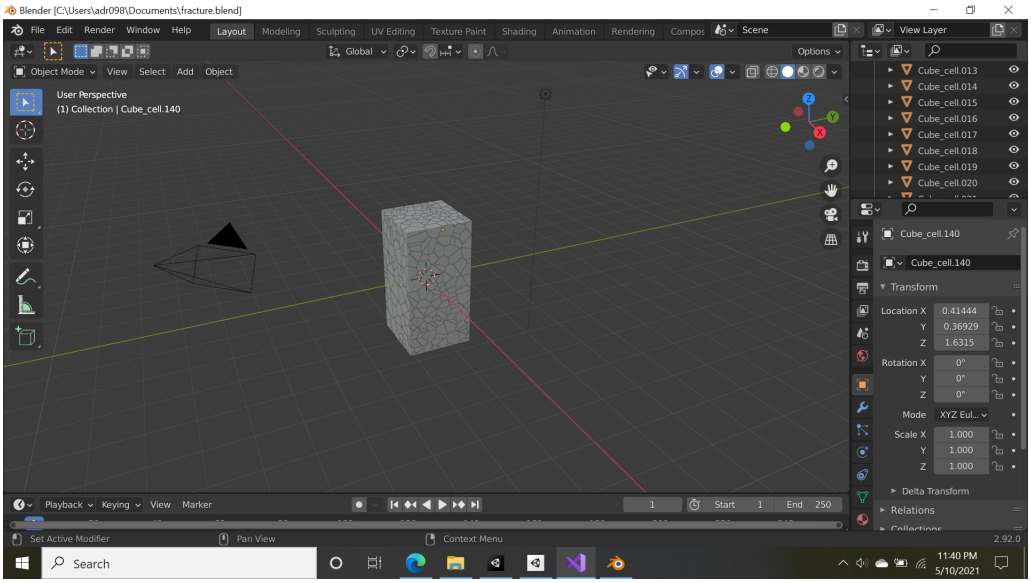


FIGURE 8: Fractured Object

The functionality of the breaking is implemented in `fracture.cs`, a component of the **Player**, which instantiates a a copy of the **Player**, but using the fractured material object created in Blender. It also calls `Destroy(Player)` to destroy the old **Player** object. It finally applies an explosion force to each of the cells in the fractured object. The power of this force can be controlled from the Unity `Inspector` panel.

Difficulties and Lessons: Fracturing the brick into smaller pieces actually did not result in the desired result of a more spectacular shattering effect. It would cause the brick to shatter into more pieces, but equally large. To fix this, we had to tweak the mass of the brick and the mass of the fractured cells that replace it, when the brick gets destroyed.

Testing: To test the breaking effect, simply roll the brick until it falls out of the floor bounds. It should cause the brick to shatter.

0.1.6 Fog and Particle Effects

Description: We wanted to add *atmosphere* to the game scene and decided to implement fog using particle effects. The final result looks like the following:

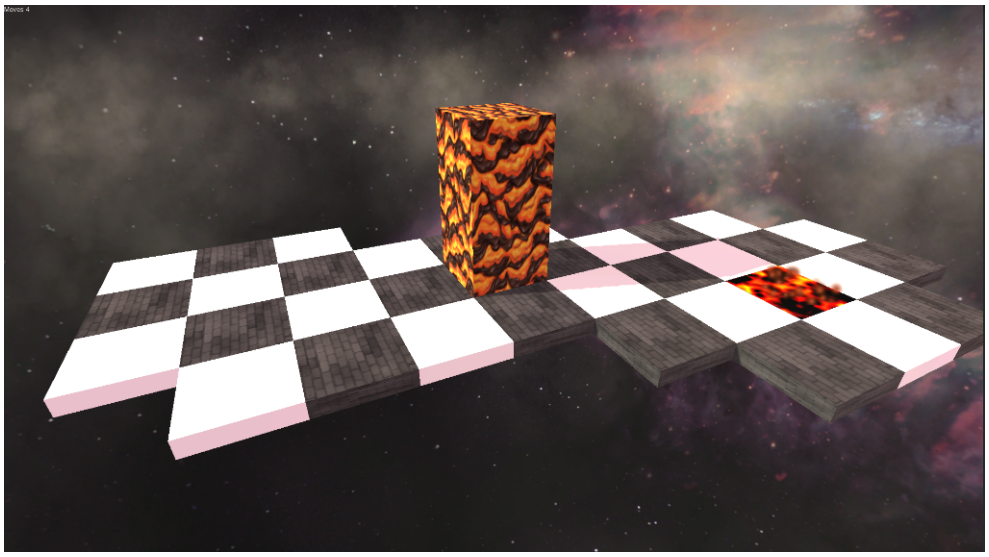


FIGURE 9: Fog

Implementation: We implemented volumetric fog using Unity's Particle System. Within the particle system, we used a stock material from within Unity for dust storms and tweaked its color and opacity to give a more misty look rather than dust. From here, we tweaked the system's duration, speed, looping, and emission. We also had the emission occur in a box shape so that the fog would cover a designated box area (i.e. the platform). We also changed the color's opacity over time in order to achieve the affect of fog fading in and out of the scene.

Difficulties and Lessons: The main, and only, difficulty of this task was understanding Unity's particle system. While Unity handles most of the work, the various parameters and effects baked into the system make the task daunting at first. After tweaking with examples and trial and error, the effect system becomes easy enough to understand. However, there are still aspects to this system that were left untouched or unobserved. There was also a difficulty in finding the right material to base the particles off of in the renderer, but this was also solved through trial and error within Unity.

0.1.7 Lava Shader

Description: The special puzzle, where if the brick lands, the player wins the game, has a lava-like looking shader. Since the brick looks like solidified lava, it makes sense to have a shader that simulates liquid lava for the winning puzzle. The shader also has bubbles of smoke that is emitted by the lava. A closeup look of the lava pit looks like this:

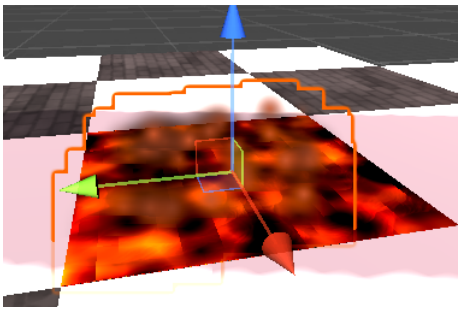


FIGURE 10: Lava

Implementation: In Unity, most textures are pre-made and can be downloaded from the Asset Store and easily applied as materials to objects. However, a customized dynamic texture, such as the lava texture requires a complex implementation. Our lava was created using the Universal RP package, which is a newer version of the PBR shader graph, and it allows for creating textures from scratch using various tools that form a tree graph, called the shader graph.

A shader graph is composed of various nodes, with input and output parameters, that control the `Albedo`/the `Base Color`, the `Smoothness` which define the actual texture; and the `Emission`, and `Normals` which define the dynamic properties of the shader. The output nodes are called the `Fragment` and the `Vertex` which are analogous to the fragment and vertex shaders in OpenGL.

Our shader graph looks like this:

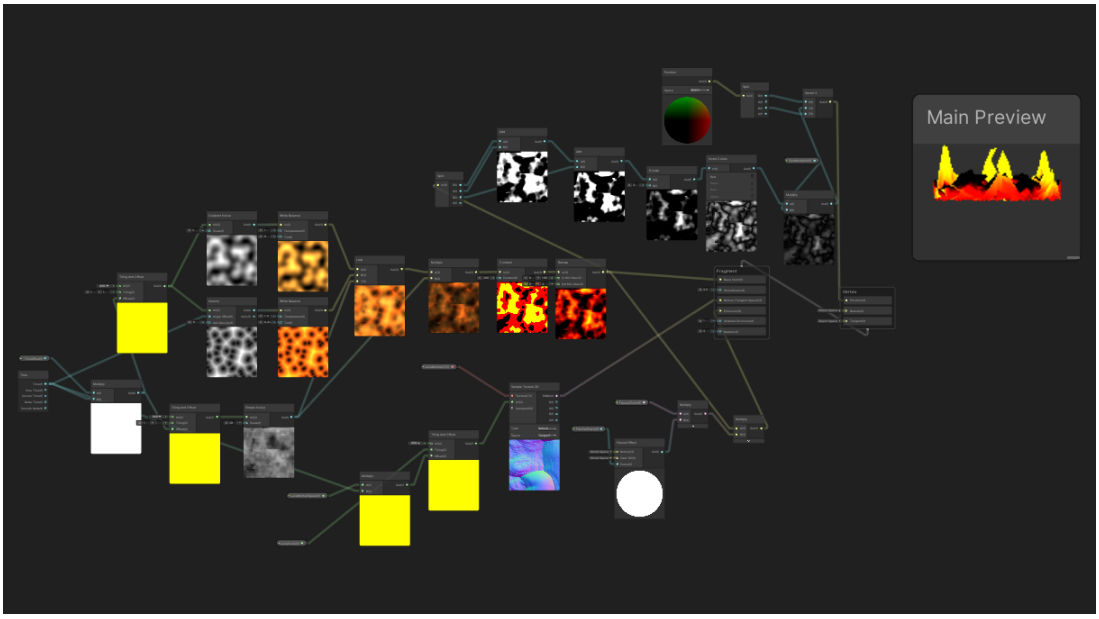
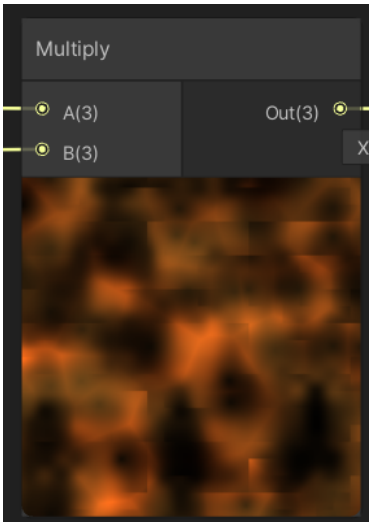


FIGURE 11: Lava

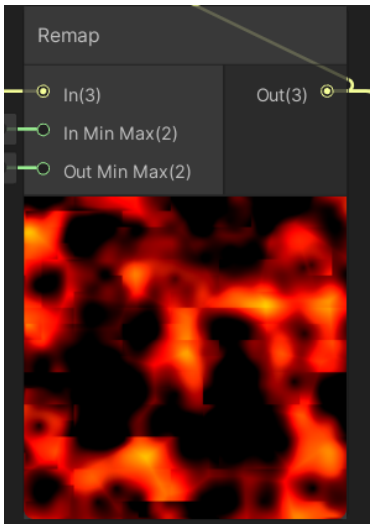
Where the middle branch handles the `Base Color` and the simple dynamics of the lava, the lower branch handles the horizontal flow and the `Emission` of the lava, and the upper branch the vertical motion (the ripples) of the lava. The Universal RP graph also allows for the final result (`Main Preview`) to be tracked with every node addition or parameter change, showing what the final result looks like.

The middle branch is the quintessential branch, so we will discuss some of its nodes in order from root to leaves and how they affect the shader.

1. **Gradient Noise** - determines the direction and the value of the vectors of motion
2. **Voronoi** - acts at the cell level (determines density, spacing)
3. **TimeOffset** - is connected to the AngleOffset in **Voronoi** to control the angle for the particle direction of movement as a function of time.
4. **Lerp** - applies lerp between the **Voronoi** and **Gradient Noise** outputs. The ratio of the 2 nodes (the alpha) in lerp will change dynamically at the same rate that the **TimeOffset** (the rate at which the angle of the cell movement changes)
5. **Contrast Node** : Blends/Enforces the cells boundaries (in the texture they appear as squares if you zoom in). Using contrast parameters gives contrast between the cells and allows control over the color contrast in the shader, as well. As a result, the bright orange cells are more easily differentiated from the dark ones.



(A) Before Applying the Contrast Node



(B) After Applying the Contrast Node

For the lower branch, which controls the horizontal motion, some of the most important nodes are:

1. **Simple Noise** - a node that takes in the **TimeOffset** and uses the timeoffset amount to create randomness in the motion of the shader.
2. **LavaNormals** - a Vector2 parameter (defaulted to the y = -x direction) that is controlled by the user and is multiplied by the Simple Noise using a `Multiply` node to determine the direction of horizontal motion (flow in the lava). The output of the `Multiply` node is connected to the `Normals` parameter of the FragmentShader. If you zoom in on the lava puzzle in the scene, one can see horizontal waves that simulate a flow in the y = -x direction.
3. **Fresnel Effect** - is connected to the `Emission` parameter of `Fragment` output node and adds emission glow to the lava.

And, finally, the upper branch that handles the vertical motion, aka the ripples of the lava, has the following functionality:

1. The branch takes in the output from the middle branch, that is the output of the **Remap** node. It uses a **Split** node to split up the color spectrum into R, G, B and then a series of `Add` nodes to obtain an identical texture, but in black and white.
2. The default behavior in Unity URP is to make darker colors appear deeper than brighter colors, so we have to invert that effect, because darker colors are solid lava and it floats above the lighter burning liquid. We use the **InvertColors** node.
3. A **Position** node that will create the height difference between colors.
4. Define **SizeMultiplier** float that controls the height of the lava. This parameter can be controlled by the user in the `Inspector` window.

To make the lava even more interesting, we added Smoke bubbles using the `Particle System` in Unity. The smoke particle are emitted through the surface area of an imaginary box positioned in the center of the lava puzzle, with bubbles emitted in the upward direction, with a fading property, and with their velocity decreasing as a function of height. The parameters of the particle system, such as velocity, density of the particles, direction, position, etc can be directly controlled by the user through the `Inspector` window of the **Smoke** GameObject located in the scene.

Difficulties and Lessons: Part of the lava shader implementation was inspired from this tutorial. However, the majority of the graph was modified to include better dynamics and make the lava look more real. In the tutorial example, the lava ripples would cause the

shader to look like a flying carpet above the floor puzzle, with its edges disconnected from the puzzle. This did not look realistic, so we had to modify the upper branch of the graph tree to remove that effect and to add horizontal motion in the lava using the lower branch to simulate horizontal waves. Adding the smoke bubbles actually made the lava look much better by simulating a boiling effect, as if the smoke bubbles came off from the simmering liquid, and thus creating a vertical effect that looked more realistic than the initial ripples.

This part of the project was significantly more challenging to implement than other parts, because we were new to how the Universal RP shader graph works, and we were not familiar with the tools (what each of those nodes controls) and how to use them to make the lava look as realistic as possible. It took some understanding of their effect through lots of trial and error.

0.1.8 Sound Effects

Description: To add some ooomf to the game, we added a click sound upon the collision of the brick with the floor puzzles, and play the sound at every roll.

Implementation: The sound was downloaded from FreeSounds and it is played in the `Rolling.cs` script that controls the **Player** motion. The sound is also added as an `AudioSource` component to the **Player**, so its volume, pitch, intensity can be tweaked in the `Inspector` window, as preferred.

0.1.9 Game Over and Boundary Detection

Description: Whenever a player fails the game by falling out of bounds, we fade in a "game over" screen and prompt the user to press 'R' to restart and try again. This then reloads the scene and grants access back to the user. After losing, the screen will look like this:

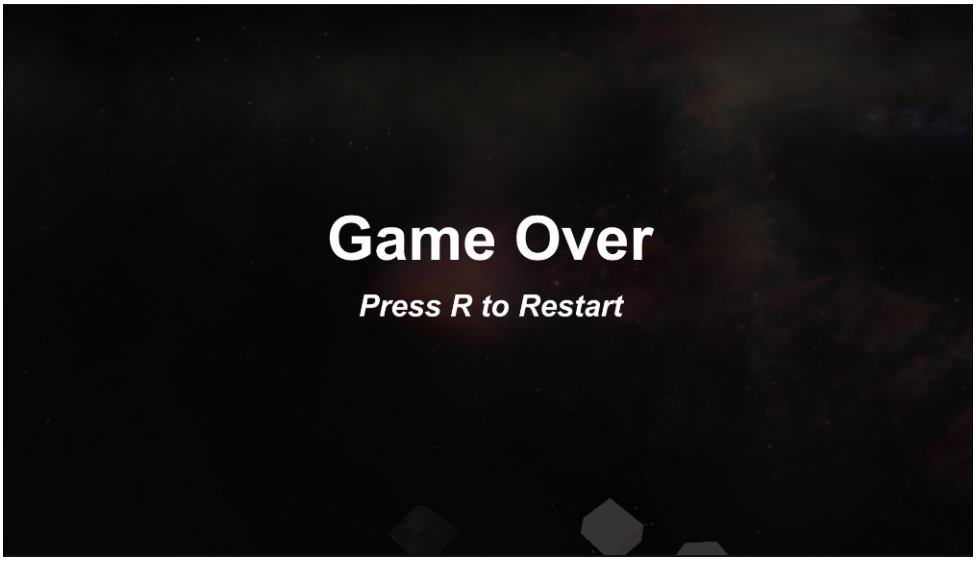


FIGURE 13: Game Over Frame

Implementation: Before prompting the game over screen, we needed to detect whether the player fell out of bounds. To do this, we had the platform include layers of unrendered tiles. These are not visible for the player, but are necessary when detecting collisions out of bounds. Therefore, every time a new collision occurs, we check to see if it is with an unrendered tile. If it is, we flip a game over flag which then prompts the floor and player to collapse and fracture respectively. In addition, we revoke controls away from the player. Then, we increase the opacity of a canvas which has the game over text objects as children. In game play, the canvas is entirely transparent, and when the player goes out of bounds,

the screen seen above fades in. As for restarting the game, we detect whether the player inputted R during this time, which then re-loads the original game scene.

Difficulties and Lessons: By this point in the project, Unity's higher level features and capabilities were better understood by us so this feature was relatively easy to implement. This utilized cross-object and cross-script referencing which we better understood during the floor collapse part of the project. One minor difficulty was understanding how to detect collisions within Unity, but this feature is well supported and easy to learn once adding a box collider to the player and all the tiles.

0.1.10 Winning Condition

Description: When the player successfully solves the puzzle by fitting the player object into the lava, we prompt a screen similar to the one seen during a game over just with different text (as explained in 0.1.9).

Implementation: To detect the winning condition, we needed to maintain a list of objects currently colliding with the player object. We know that if the only object in collision is the lava, then the player must have solved the puzzle. During each collision entry, we add the collider object to a list of colliders and we remove the collider during collision exit. This way, we can easily check to see if the current object in collision with the player is the lava. We had to cover a few edge cases, however, where the player rolls over the lava but does not fit in the square. In this case, we need to only check the collider list whenever the object is not in mid-roll. This is a flag already maintained during rolling so this was straightforward enough to check. When the player satisfies this winning condition, we disable the lava tile under them, and disable kinematics for the player's rigid body which allows the player to fall through the level and prompt the success screen.

Difficulties and Lessons: Since most of the canvas logic was implemented during game over, that implementation was straightforward. There was difficulty in covering the aforementioned edge case and making sure that the lava was not disabled too soon. Once checking if the player was mid-roll these issues were resolved. There was also an issue on maintaining a list of objects currently in collision with the player. Unity does not directly support this so the list we maintain is a work around for this problem.