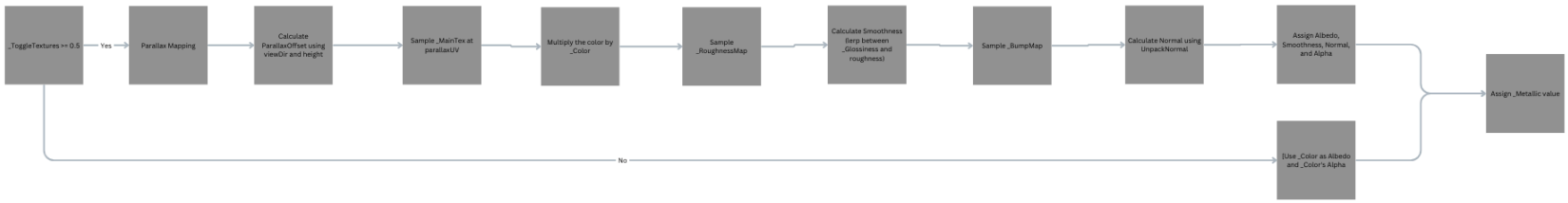
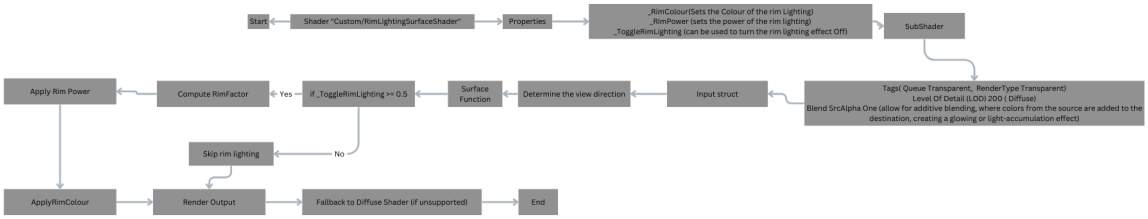
The Gold Cube in the scene is the object the player (basketball) needs to try to push off the concrete platform to win the game. The gold cube's first shader "GoldCube", uses a combination of parallax mapping, texture sampling, and material properties to create a dynamic, realistic material effect. The core logic begins with parallax mapping, which adjusts the UV coordinates of the texture based on the height map to create the illusion of depth and surface variation, especially when viewed at an angle. The ParallaxOffset function calculates a displacement for the texture coordinates based on the height map and the viewer's direction, which is then applied to the main texture (Albedo) to give a sense of 3D surface relief. The shader also uses a normal map to simulate surface details like bumps and wrinkles, and this is influenced by a BumpScale value. The roughness of the surface is controlled through a RoughnessMap and blends between a smoothness value (Glossiness) and the roughness map's value to adjust how reflective or matte the surface appears. Additionally, the shader includes an optional toggle for enabling or disabling these textures, and when the toggle is active, the HeightMap and associated maps modify the visual appearance. Finally, the shader incorporates a Metallic property to determine how metal-like the surface looks, with values ranging from non-metallic to fully metallic, affecting the material's interaction with light and reflections. This combination of algorithms results in a detailed material that can simulate complex surface properties like depth, roughness, and reflectivity, making the surface appear more lifelike. The significance of this it makes the gold

pirate skull texture stand out and look more visually appealing on the screen.

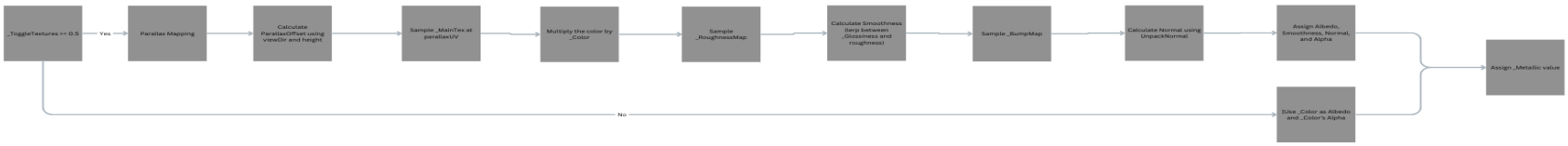


The Gold Cube also has an additive rim lighting shader called “RimLightingSurfaceShader”. This shader creates a rim lighting effect by calculating the angle between the camera's view direction and the surface's normal. The shader first computes the dot product of the normalized view direction and the normal to measure how aligned the two vectors are. This value is then used to

determine the "rimFactor," which is highest when looking at the edges of the surface (perpendicular to the view direction) and lowest when the surface faces the camera directly. The rimFactor is raised to the power of a user-defined "_RimPower" to control the intensity and falloff of the effect. If the toggle "_ToggleRimLighting" is enabled, this value is multiplied by a rim colour ("_RimColour") and applied as emission, producing a glowing edge effect. The result is a visually striking highlight around the edges of the surface, with the intensity and appearance adjustable based on the parameters. This allows for the gold cube edges to glow slightly to help give information indicating the importance of this object.

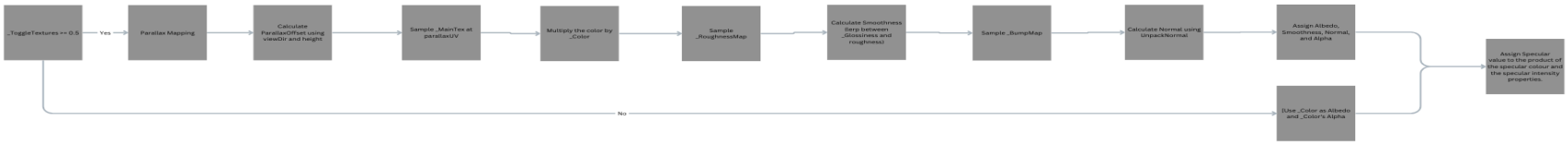


Like the “GoldCube” Shader the volcanic walls shader “VolcanicRock” uses the same logic and algorithms and logic to apply the volcanic walls look to the object at the end of the lava. This allows the scene to look like it is in some volcanic cavern.



The basketball in the scene aka the object the player controls tries to push the gold cube off of the platform into the lava without falling off itself. The basketball shader “Basketball”, is designed to simulate the appearance of a basketball surface, incorporating various material features such as normal mapping, parallax mapping, roughness, glossiness, and specular reflections. The shader includes parameters for customizing the base colour, textures, and material properties such as specularity and bump mapping. It utilizes parallax mapping to simulate depth by adjusting the texture coordinates based on the height map, providing a more realistic surface detail. The shader also adjusts the glossiness based on a roughness map, which controls the smoothness of the surface, and combines this with a specular reflection to simulate the shiny highlights that are typically seen on basketballs. The two-sided rendering toggle allows for more flexibility depending on the mesh's geometry. If textures are disabled (via the "_ToggleTextures" parameter), the shader defaults to using a simple colour with a glossiness effect. The combination of these effects creates a visually

detailed surface with realistic lighting interactions. This allows for the object to look like a basketball. The textures used by this shader were procedurally generated in Blender and baked into a PNG image so that way they would map properly over a sphere.



The ConcreteBricks shader uses two passes to create a progressive glow effect that interacts with the platform's environment, particularly when it collides with lava. The first pass of the shader handles the basic surface properties of the material, including texture mapping, normal mapping, roughness mapping, and height mapping. These elements are applied based on the shader's settings and are influenced by a toggle that can turn each of the texture maps on or off. The _ToggleTextures property controls whether the height, normal, roughness, and albedo maps are used to modify the appearance of the platform's surface. When enabled, the shader uses the parallax mapping technique, calculating a UV offset based on the height map to simulate surface depth and detail. This gives

the platform a more complex and realistic look, enhancing its visual appeal. The albedo colour, roughness, and smoothness are then calculated from these maps to finalize the platform's appearance. The toggle offers flexibility, allowing the shader to operate with or without these additional surface details.

The second pass of the shader is responsible for the progressive glow effect, which is determined by two primary parameters: _VectorBounds and _VectorRange. _VectorBounds specifies the center of the glow effect, which is dynamically updated by the C# script based on the platform's interactions, while _VectorRange defines the radius of the glow. The glow effect is calculated using the distance between each fragment (pixel) and the glow center. A smoothstep function is used to create a smooth falloff for the glow, so that it gradually fades out at the edge, forming a soft and natural circular glow. The glow's colour and intensity are determined by the _GlowColour and _GlowIntensity properties and the _GlowMask texture is used to add patterns or noise, enhancing the visual complexity of the effect. This second pass is responsible for rendering the glow based on the calculated parameters, allowing it to envelop the platform in a glowing aura that reacts to its interaction with the environment.

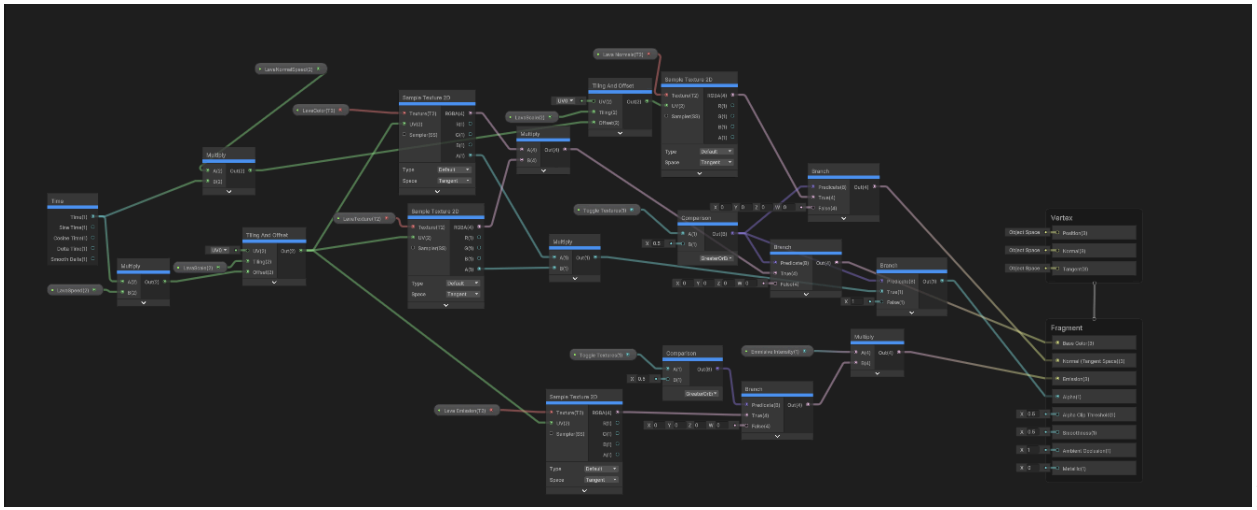
The GetCollisionContact C# script works in conjunction with the shader, enabling the dynamic nature of the glow effect. When the platform collides with the lava, the script detects the collision and updates the _VectorBounds property with the average contact point of the collision, which shifts the center of the glow effect to the point of contact. The script also manages the _VectorRange property, adjusting it over time based on the platform's proximity to the lava. When the platform is in contact with the lava, the glow range decreases smoothly, creating a contracting glow effect, while the range expands back to its maximum when the platform is no longer in contact with the lava. The script uses Mathf.SmoothDamp to ensure that the transition of the glow's range is smooth and not abrupt. This real-time update of the glow's center and range ensures that the glow effect reacts to the platform's position and interaction with the lava. Together, the shader's logic for rendering the glow and the script's collision detection creates a visually engaging effect that dynamically responds to the game environment.

The significance of the progressive glow effect is that it visually represents the heat transfer from the surrounding lava to the stone brick platform, with the glow expanding from the edges inward to the center as the platform heats up. This behaviour simulates how heat moves, with the edges heating first and the center gradually becoming hotter. The C# script dynamically controls this effect by adjusting the glow's range based on whether the platform is colliding with the lava. When the platform stops colliding with the lava, the glow reverses, progressively fading from the center outward, simulating the cooling process. This logic ensures that the center, which heats up first, is also the first to cool, providing a realistic and visually engaging way to show the platform's interaction with its environment.

The lava shader “Lava”, is the only shader in the entire scene created using Shader Graph, rather than a scripted shader. This shader generates a material that simulates a moving lava flow when applied to an object, specifically the object that triggers the collision with the ball and the push object beneath the platform. When the _ToggleTextures value is greater than or equal to 0.5, the base colour is determined by the product of the _LavaColour and _LavaTexture textures. The UVs for both textures are adjusted using a tiling offset calculated from the _LavaScale input and the product of _LavaSpeed and the Time node's time output. This offset is

then applied to the base object's UVs. The emission texture is sampled using the same tiling offset, and when _ToggleTextures is greater than or equal to 0.5, the _LavaEmission texture is used for the emission output.

The alpha output is calculated by multiplying the sampled alpha values from both the _LavaColour and _LavaTexture, with this product applied when _ToggleTextures is greater than or equal to 0.5. For the normal and tangent space outputs, the process is similar to the other textures, but the property affected by the Time node is the _LavaNormalSpeed. This property allows for separate control over the speed of the colour and normal maps, enabling different animation speeds for each if desired. When _ToggleTextures is set to less than 0.5, all emission, colour, and bump textures are outputted as black since no textures are active. In this case, the colour is handled by the texture, not through the standard RGBA properties, and the alpha output defau



lts to a general standard value.

The enhanced particle effect for the ember particles in the scene applies the EmberShader, an unlit shader designed to simulate glowing, flickering ember particles commonly used in fire or flame effects. The shader works by sampling a base texture (_MainTex)

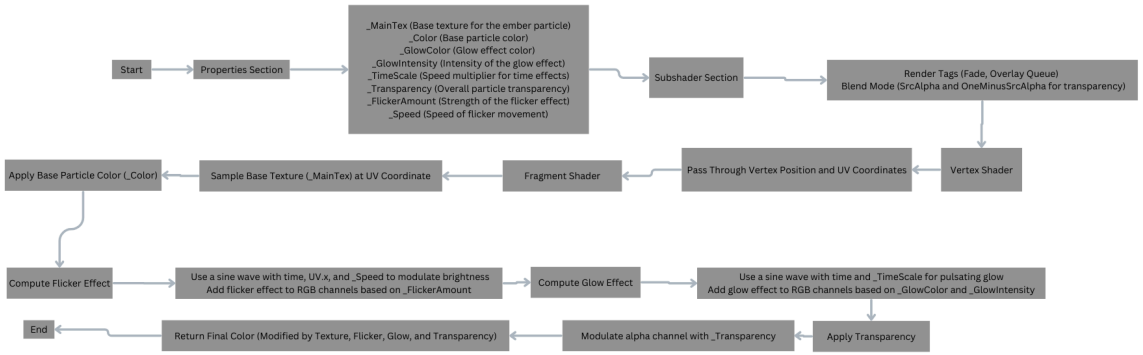
and modulating this texture with a colour tint from the _Color property, which is typically set to an orange hue to resemble the colour of embers. The shader then incorporates several effects, including glowing, flickering, and transparency, to create a dynamic and organic look that enhances the appearance of the ember particles.

A standout feature of the EmberShader is its glow effect, which is controlled by the _GlowColor and _GlowIntensity properties. The glow colour is generally yellow or white to represent heated particles, while the intensity controls the brightness of the glow. Using a sine wave function, the shader animates the glow over time, making it pulse in and out. This pulsing effect mimics the behaviour of glowing embers, providing a realistic flickering effect as the particles glow and fade. This dynamic feature adds to the authenticity of the ember simulation, contributing to the fiery, glowing look of the particles.

In addition to the glow, the shader also simulates flickering, a critical effect in fire and ember visualizations. The _FlickerAmount property determines how strong the flickering effect is, while the _Speed property controls the rate at which the flickering occurs. The flicker effect is generated using a sine wave based on time and the particle's UV coordinates, specifically affecting the X component. This results in a shifting brightness across the texture, which helps replicate the randomness and inconsistency found in natural ember behaviour. The _TimeScale property allows further control over the speed of flickering, offering flexibility in the shader's visual output.

Transparency is another important element in the shader. The _Transparency property controls how opaque or transparent the ember particles appear. Lower values make the particles more transparent, while higher values make them more opaque. This effect simulates the gradual fading of embers as they burn out. The shader ensures that the particles naturally fade to transparency as their glow diminishes, enhancing the realism of the effect and making the embers look more lifelike as they lose their intensity and slowly disappear.

Overall, the EmberShader provides a highly customizable tool for creating ember or fire-like particles in Unity. It offers flexibility in adjusting the colour, glow, flicker, and transparency of the particles, allowing for the creation of dynamic, organic effects. The use of sine wave functions for glow and flicker animation ensures that the ember particles behave in a natural and unpredictable way. This shader can be easily integrated into particle systems, making it an invaluable asset for developers seeking to create realistic and visually compelling ember effects for fire or flame simulations.

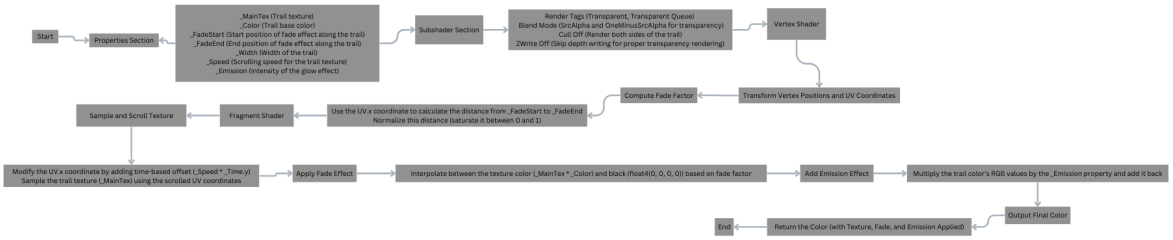


The EnhancedTrailEffect shader creates a highly customizable trail effect that is applied to the basketball as it moves through the scene. This shader gives the user control over various aspects of the trail's visual appearance, such as texture, colour, fading, width, speed, and emission intensity. The trail texture (_MainTex) is tinted by the _Color property, allowing for flexible colour adjustments.

A fading effect, regulated by the _FadeStart and _FadeEnd properties, ensures that the trail gradually becomes more transparent as it moves, simulating the dissipation of the trail over time.

A standout feature of the shader is the scrolling effect applied to the trail texture. The _Speed property controls the rate at which the texture scrolls, enhancing the sense of motion and providing a dynamic look to the trail. The transparency of the trail is modulated as it extends, gradually fading from fully visible to transparent along its path. The _Width property allows users to adjust the width of the trail, offering further customization to the visual design, whether it’s a thin, subtle trail or a broad, pronounced one.

In addition to the above features, the shader incorporates an emission effect, which creates a glowing trail. This effect is controlled by the _Emission property, where the trail's colour is amplified based on the emission intensity, giving it a luminous, glowing quality. This glow effect is particularly useful for scenarios where glowing or fiery trails are desired, such as in sci-fi or fantasy scenes. The shader uses standard transparent blending to seamlessly integrate the trail into the rest of the scene, ensuring it blends well with other visual elements. In summary, the EnhancedTrailEffect shader is a powerful tool in Unity for creating dynamic, visually captivating trail effects.



The GlobalIlluminationShader is a highly customizable shader designed to simulate global illumination effects by applying ambient, diffuse, and specular lighting to materials. It offers a range of adjustable properties, such as controlling the base colour, specular colour, shininess, texture, normal map, and ambient lighting, allowing developers to fine-tune the lighting and appearance of materials. The shader includes toggles for dynamically enabling or disabling the influence of ambient, diffuse, and specular lighting on the final output. This flexibility is further extended by properties like _BumpScale, which adjusts the intensity of the normal map, and _AmbientLightColor, which modifies the colour of the ambient light. By toggling the ambient, diffuse, and specular components, users can control how each lighting type interacts with the surface, providing a more customizable and responsive visual effect.

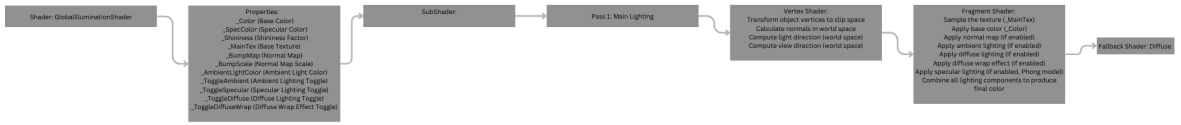
In the vertex shader (vert), the object's vertices are transformed into clip space, and the normals are transformed into world space, which is crucial for accurate lighting calculations. The light direction and view direction are also computed in world space to ensure

the lighting interactions are correct for each fragment. The fragment shader (frag) handles the core lighting calculations. It starts by sampling the base texture and applying the base colour. If a normal map is present, the shader modifies the surface normal using this map, enhancing the lighting effects to simulate more detailed surface details and depth. Depending on the toggle settings, the shader then calculates the ambient, diffuse, and specular lighting components.

For lighting calculations, the shader supports ambient, diffuse, and specular components. Ambient lighting is applied if the corresponding toggle is enabled, using the base texture colour modulated by the ambient light colour to simulate light that uniformly illuminates the entire surface. Diffuse lighting is computed using the Lambertian reflectance model, which calculates the lighting interaction based on the angle between the surface normal and the light direction (dot product). This model is then enhanced with a diffuse wrap effect, softening the lighting and providing a more natural, subtle shading effect across the surface. Specular highlights are calculated using the Phong reflection model, where the specular intensity is determined by how light reflects off the surface, taking into account both the light and view directions and modulated by the shininess factor to control the sharpness of the highlights. The shader combines these components—ambient, diffuse, and specular—into a final output colour that is rendered on the screen, providing a realistic and adjustable global illumination effect.

When used in a post-processing pipeline, this shader provides a dynamic way to manipulate the visual appearance of materials based on real-time inputs. By toggling different lighting components on or off, developers can test various lighting configurations, allowing for an interactive way to modify the scene's look. The flexibility of this shader makes it ideal for applications where lighting needs to be adjusted dynamically or customized per object, providing a powerful tool for achieving a wide range of visual styles and effects. This improves upon the previous assignment one illumination because the illumination from assignment one is now applied as a scene render rather than individual materials. Instead of affecting only the materials of individual objects, the illumination is applied globally to the entire scene, enhancing the overall visual consistency and interaction between lighting and objects in the environment. This

change makes the lighting more integrated and dynamic across different elements of the scene, further improving the realism and depth of the rendered image.



contributions

Xavier Bastalla

Made a smoke effect particle effect using a material (https://youtu.be/8LyZNNlfJzk)

Made the second LUT Look up colour table

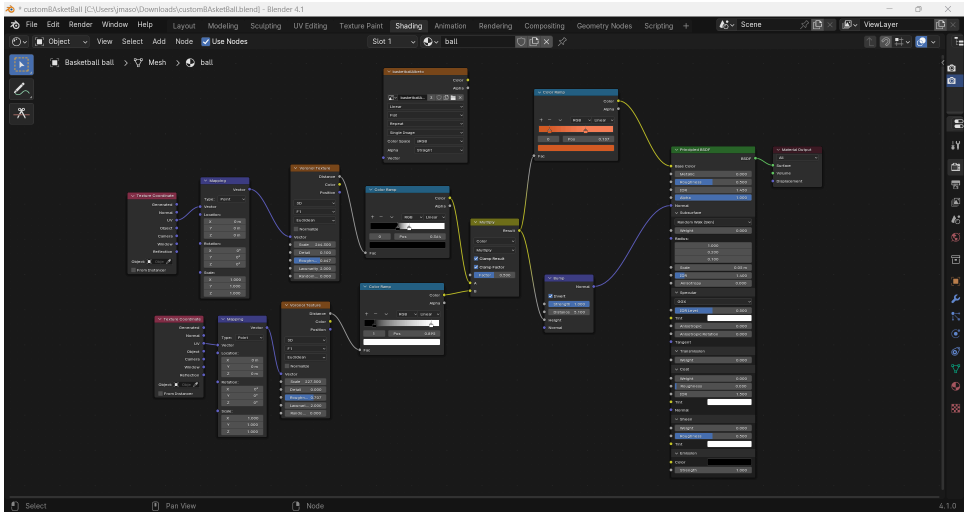
I coded the win and lose conditions for the game involving a ball and a box. If the player (ball) falls off, the game states that the player lost, but if the box falls off, the game states that the player has won. I also helped with the layout of the game world, making adjustments and changes as needed. Additionally, I assisted with some of the shaders for the project. I reviewed the code and understood how Jamie implemented it, then repeated it for my own project to see how it works. Jamie is fine with the work I've done.

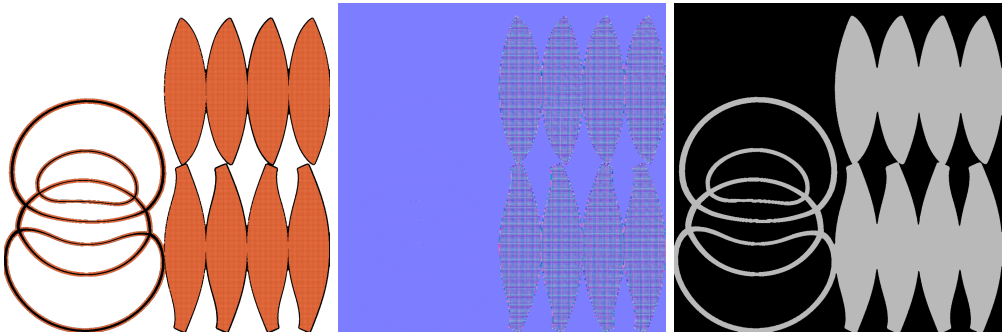
From this course, I learned how to create shaders. Initially, I didn't think I would enjoy making shaders, as it seemed like a lot to take on. However, as the course progressed, I began to enjoy it more and even applied it to my own side project. It felt great to know that I didn't have to rely on pre-made shaders but could create my own. I've learned so much about how shaders work and the many possibilities they offer. I enjoyed my time in the class and am considering taking the next class to continue learning.

Jamie

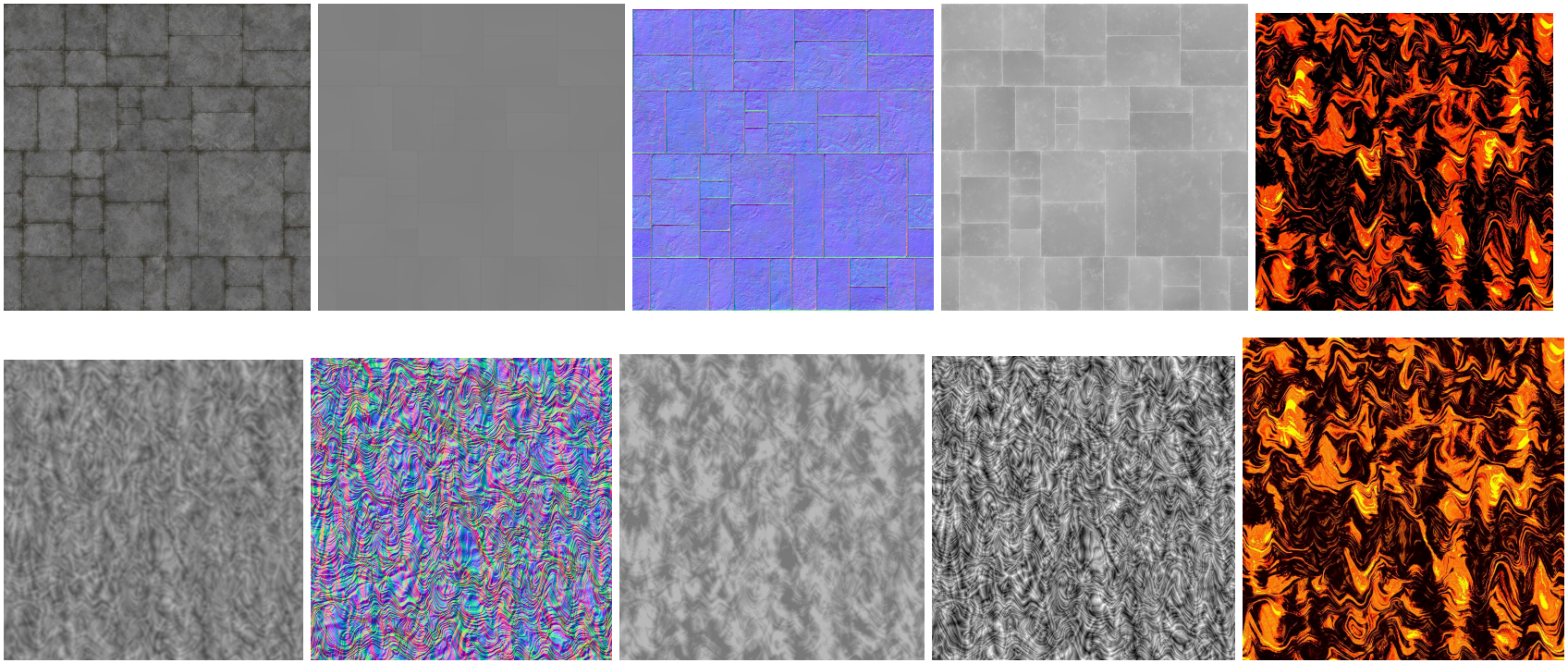
I created the player movement system, implementing smooth and responsive controls for the ball to navigate through the environment. I also developed methods for handling collisions with the cube, ensuring that the ball can knock the cube backward when enough force is applied upon impact.

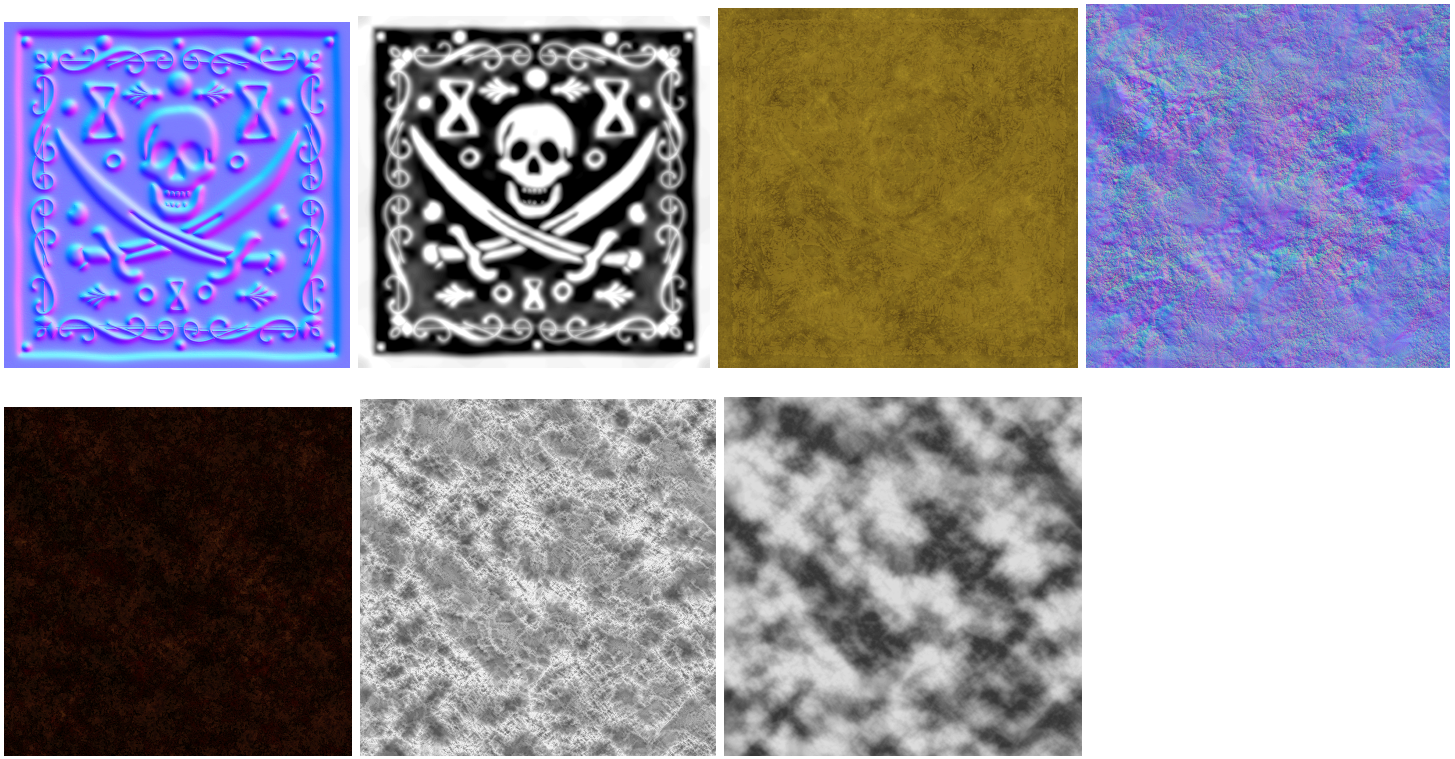
For the basketball texture, I procedurally generated and baked the texture in Blender, which allowed me to achieve a more seamless mapping of the texture onto a sphere. This method not only enhanced the visual quality of the basketball but also prevented texture stretching or distortion, which could occur with a traditional 2D PNG texture.





Additionally, I took the initiative to source PBR (Physically Based Rendering) texture images for critical assets in the scene, such as the gold cube, stone brick platform, and volcanic wall. This ensured that the materials reacted to light in a more realistic and immersive way, enhancing the overall visual fidelity of the environment.





As part of my contributions to the project, I created shaders for all elements in the scene, developing custom lighting models to ensure materials and objects interacted with light in an accurate and visually appealing manner. These shaders were crucial in achieving the desired visual style and maintaining performance optimization.

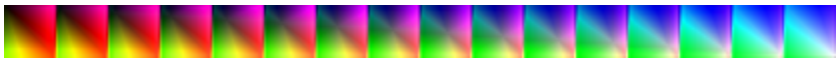
In addition to the shaders, I developed several visual effects to enhance gameplay and visual appeal, including a trail effect that tracks the movement of the ball, an explosion effect for when the ball collides with specific targets, and ember effects to simulate particles

and add a dynamic environmental atmosphere. I also created various material variants for these effects, which allowed for greater flexibility and variation in how they were rendered across the game world.

To elevate the game's visual presentation, I created the first colour look-up table (LUT) for colour correction. This adjustment helped fine-tune the game's overall colour scheme, enhancing its cinematic quality and ensuring that the colour grading aligned with the tone and atmosphere intended for each scene.

Also, I created the script that renders the scene with the applied colour correction and allows the user to modify its contribution to the UI of the game.

I also created the camera follow and orbit Cinemachine object to follow the ball as it moves and allow the player to rotate their camera around the ball with the mouse.



My takeaways from this course are invaluable, as I learned how to apply a variety of visual effects to objects in Unity that I didn’t initially think were possible. This has opened up an entirely new area of game development, one where I can now design more realistic and seamless environments that are visually appealing and offer a greater degree of flexibility in their design. The course also introduced me to new ways of applying visual effects, allowing me to create effects without needing to build entirely new meshes in Blender. This is especially useful for incorporating complex visual elements without adding unnecessary complexity to the development process unless the effects require specific interactable shapes.

Additionally, I gained insights into leveraging the GPU for creating visually immersive worlds, which was something I hadn't fully understood before. A great example of this is how I applied the rim lighting effect to create a ghost object in one of my personal games that I am currently developing. Overall, this course has expanded my skill set and opened up a whole new field of possibilities for producing better-quality games.