Writeup: Homework #1

Malcolm Riley

April 28, 2019

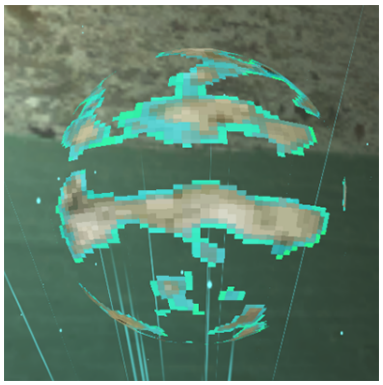
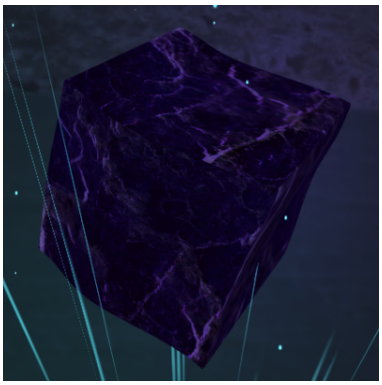
Introduction

All subtasks of this homework assignment have been integrated into a single Unity scene. No third-party Unity packages were used to create this scene. All code and scene assets including models and sounds were created by myself for the purpose of this assignment.

The viewer can use the mouse to interact with certain objects in the scene. If an object is interactable, a stylized mouse icon will appear in the bottom center of the screen. This is typically used to "peer" at a particular object in order to get a closer look (with two exceptions). In order to "lean back" from the "peering" view and return to the main scene view, right click with the mouse.

The terminal object can be interacted with using the keyboard, navigation controls are displayed on each screen of the terminal. The subtask demonstrations of this homework assignment are accessed through the terminal.

A. Three Objects and Three Lights


The scene contains three lights: One emitted by the hologram projector object, one that revolves around the displayed model, and the lightbulb which may be toggled on or off by left-clicking on the lightswitch object. To access the individual shader demonstrations, activate the terminal and select them from the main menu. The relevant object will be added to the scene as a hovering "hologram" above the hologram projector object. To inspect the individual objects more closely, left-click on the hologram projector. Click the links in the table below to view the code.

Name	**Link to Shader Code**	**Description**
Error Sphere	`Dissolve.shader`	This is an unlit shader that applies a pixelated texture to a sphere, but also a pixelated dissolve effect that changes over time.
Cube	`VertexDisplacement.shader`	This is an unlit shader that applies a texture to an object, and deforms the vertices using a sine function.
Crystal	`PhongLighting.shader`	This shader applies a texture and the Phong lighting process to an object. It supports up to four point lights.

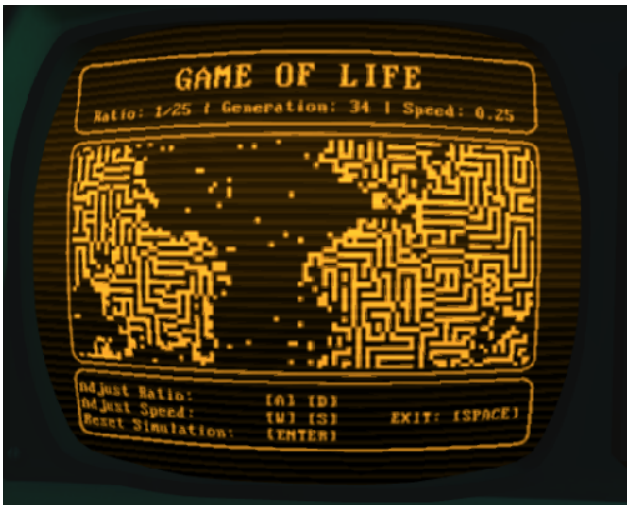
B. Image Processing Shader

The kernel-based image processing effect is implemented on the terminal screen object itself. Effectively, it is a standard box blur using a simple 3×3 convolution matrix. The effect increases over time and may be reset by "hitting" the terminal by left-clicking on the upper left corner of the terminal object. The intention is to simulate a CRT monitor with malfunctioning deflection coils. The convolution matrix for this effect is defined as:

$$K = \begin{bmatrix} 1-S & 1-S & 1-S \\ 1-S & S & 1-S \\ 1-S & 1-S & 1-S \end{bmatrix}$$

where S is an arbitrary "sharpness" value. This value is increased over time, thereby biasing the blur effect towards the neighboring pixels. A secondary `ScanIntensity` value is dispatched to the shader to control the intensity of the scan lines (themselves being provided by the `ScanLines` texture). The shader code can be viewed here: `ScreenEffect.shader`

C. Game of Life “Ping Pong” Shader



This shader is implemented as a “program” within the terminal object and may be accessed by selecting the “Game of Life” option in the terminal’s main menu. The rules of the simulation were discovered as a happy accident during implementation and produce a pleasing, almost mazelike end result. The simulation may be affected within the terminal “program” by using the controls described on that screen. There are several interoperating parts that are used to accomplish this effect:

Name	Link to Code	Description
Game of Life Shader	`GameOfLife.shader`	This is the shader that contains the actual Game of Life Logic.
Mask Shader	`SimpleUnlitMasked.shader`	This is the shader that is used to draw the Game of Life result texture onto a field `Image` on the terminal screen `Canvas`. It simply draws the texture, using the alpha value from a second provided texture. This allows the Game Of Life result texture to be masked.
Terminal Screen Code	`GameOfLifeScreen.cs`	This is the actual “ping-pong” code that generates and maintains the references to the `Texture2D` and `RenderTexture` objects and swaps-between them; see the methods `GenerateTexture()` `OnScreenUpdate()`, respectively.

D. Visual Effect Discussion

Depicted above is the "Celestial Character Effect" from the game *Path of Exile*. A video of the effect may be viewed by clicking the following `[YouTube Link]`. I find this effect to be visually interesting as it contains some pleasing soft particles and strongly invokes the "celestial" aesthetic. There are several components to the effect:

- **Character Model Highlighting:** This effect appears to be accomplished by the additive application of a bluish color to fragments whose normals are facing away from the camera.
- **Small Soft Particles:** These are simply, as stated, small soft particles of various colors. They appear to be additively blended to the scene during render.
- **Large Soft Particles:** This component is the most sophisticated, and also the most interesting. It appears to have been accomplished by a particle system with a special shader attached. The particle system emits "smokelike" particles that are then shaded with a static texture of a nebula using the screen-space coordinates of the fragment; this texture is also tiled and offset over time. The alpha of the original particle smoke texture is used.

If I were to try and implement this effect using a Unity shader, I would first try to implement the model highlighting as a second pass on top of the standard Unity shader (so that lighting is automatically consistent with the rest of the scene). The highlight color would be exposed as a property of the shader so that it could be altered if later desired. Then, I would implement the large smoky particles as stated above: with a custom shader that uses the alpha of the original particle, and the color value from a tiling, shifting-offset texture whose fragment sample UV parameter is taken from the screen-space of the render target rather than world-space. Finally, I would add the small, soft particles – they appear to be quite visually similar to Unity's default particles and no custom rendering would be required.