SHADERS IN MODERN VIDEO GAMES:
Rendering Dynamic 3D Objects in GLSL

Lauren Gardner

Advisor: Felix Heide

Submitted in partial fulfillment

of the requirements for the

degree of Bachelor of Arts

Department of Computer Science

Princeton University

May 2024

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

_____

Lauren Gardner

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

Lauren Gardner

Abstract

The recent growth of the computer graphics industry, specifically in the realm of gaming, has resulted in the desire for more realistic graphics. The initial assumption is that realistic graphics mean that object meshes should be more detailed—that they should have more vertices, edges, and faces, but this would lead to increased latency because of the file size and rendering demands on the GPU. The current solution to this problem is to use graphical shaders which implement graphics inside of the GPUs rendering pipeline, making them fast and efficient. This report details the implementation of the video game *Space Otterssey*, a desktop game with graphics that mostly rely on GLSL shaders. *Space Otterssey* proves that graphic shaders can be used to create a visually pleasing game that prioritizes the user's gameplay experience.

Acknowledgements

To begin with, I would like to thank the COS department as well as my advisor professor Felix Heide for support throughout this project. I would also like to thank professor Adam Finkelstein, whom I worked with over the summer of 2022, for introducing me to the field of computer graphics.

To my roommates Theresa, Zhaoran, and Sam. Theresa, thank you for becoming my first real friend at Princeton. I cherish the past three years, especially the late nights we spent together talking. Zhaoran, I didn't expect to become as close as we did, but I am so greatful to have had a dinning hall buddy all through junior year and basically another sibling through senior year. Sam, thank you for always being there and also putting up with my weirdness for the past two years.

To my closest friends Diego, Jeremy, Danica, and Warren thank you for the endless amount of smiles and laughs you guys have given me. The pressure of Princeton is astronomical but somehow you guys managed to keep me sane.

To Danica and Warren (specifically), without y'all I quite literally wouldn't be here now. My success as a COS major was entirly dependent on all those late nights we spent together working on all sorts of COS classes. I will never forget those painful nights... is it wrong of me to say that I will miss them?

To my family, Mom, Dad, Eiche, and Max who supported me all through my childhood and encouraged me to push myself so hard. Thank you for believing in me and helping me achieve my goal of attending Princeton. I couldn't have done it without y'all.

To my friends back home, Jade, Cass, and Andie. Thank you for continuing to be my life-long friends. I thank you for the countless late nights we have spent together over breaks and all the joy you guys have given me.

Lastly, I would like to thank AAC, Triple8, and HighSteppers. Thank you for being such supportive environments and giving me a home at Princeton.

Contents

List of Figures

Chapter 1

Introduction

1.0.1 Motivation

Over the past 25 years, the landscape of computer graphics has undergone significant evolution. Starting with the introduction of the graphics processing unit (GPU) in 1999 [8], which was a revolutionary innovation in computer graphics. Designed to execute the numerous computations required for rendering graphics with optimal efficiency, the GPU eventually led to the introduction of graphic shaders, allowing programmers to manipulate various stages of the, previously fixed, graphics rendering pipeline [12]. Resulting from the increased demand for hyper-realistic graphics across an array of applications, these advancements have fueled the desire for a high definition experience with minimal latency. The goal of this project is to prove that GLSL, a high-level shader language, can be used to render a majority of meshes in a video game. Rendering in GLSL has a few key benefits: the developer can represent dynamic objects using code, no outside software or knowledge in a 3D modeling software is needed, animating object meshes using GLSL shaders is easy, and GLSL shaders increase the control over the rendering process of meshes within a game.

Recently, there has been a shift towards visually pleasing games rather than just realistic games, as seen with *The Legend of Zelda: Tears of the Kingdom*-a 2023 Nin-

tendo Switch mobile game. This game uses high-end graphics to create an immersive environment without using hyperrealistic graphics. Based on Nintendo's report, *The Legend of Zelda: Tears of the Kingdom* was responsible for most of their profit in the first quarter of 2023 [3]. Additionally, many netizens regard *The Legend of Zelda: Tears of the Kingdom* as proof that a Switch game can still have pleasing graphics [13]. This game shows that the player's gameplay experience depends on the graphics performance and the atmosphere that the game creates.

1.0.2 Modern Shader Applications

The introduction of graphic shading APIs, such as OpenGL, allows developers to create efficient and visually pleasing graphics. The sources discussed in this report will detail the uses of shaders in a multitude of contexts such as in video games, mobile devices, and virtual reality.

1.0.3 Shader's in *Space Otterssey*

Space Otterssy is a game developed through the research of this report that is visually pleasing and also efficiently rendered. Using GLSL, almost every element of the game is dynamic. The landscape is randomly generated and slowly morphs throughout the game, giving an eerie quality to the landscape that is also enticing. The items are animated to look lively: waving, pulsating, and swaying. The main character also sways to mimic the physics of his jetpack. All of these graphics were created using Three.js libraries and GLSL which are both based in OpenGL, a shading API.

Using JavaScript, and implementing Three.js libraries, it was simple to import simple 3D shapes, objects, and use built-in functions. The goal was to use Three.js to access these basic attributes, then use shader functions to alter vertex positions to make dynamic meshes using GLSL. *Space Otterssy* builds off of the starter video game code from Princeton University's COS426: Computer Graphics course [15]. All scenes

and added items are unique to *Space Otterssy*, leaving only the base structure defined using the source code. Each item in *Space Otterssy*, with the exception of the otter, has unique fragment and vertex shader files. These files define the appearance of each object by altering a basic geometry imported from Three.js: usually an icosphere or a plane. The objects are animated using a time variable passed into each shader and a floating animation written in JavaScript.

Chapter 2

Background and Related Work

The desire to make video games more lifelike has resulted in the use of more detailed 3D meshes; these high-detail models are typically vertex-based and are used to represent things such as characters, items, weapons, and scenery elements. Animating a standard vertex-based mesh requires them to be rigged, a process where invisible 'bones' are added to a mesh that give it movement capabilities. Three.js offers limited functions to rig these meshes using code; therefore, rigging is usually outsourced to a 3D modeling software such as Blender or Maya which requires specialized knowledge of their functionality. Additionally, these vertex-based meshes require storage and rendering, which can put strain on the GPU causing lag. One common approach to these problems is by rendering, for example, scenery graphics using shaders rather than depending on just vertex-based models. Shaders are code that is able to alter certain steps in the GPU processing pipeline; this makes them efficient since they are in-place and dynamic since they are represented using a high-level programming language, GLSL.

GLSL is the most common shader language and can be used to build models and textures for objects internally and in real time [12]; it can also alter existing meshes, for example, a shader could be applied to a sphere geometry to create a spikey ball.

The following is a brief history of shaders in computer graphics, as well as their applications within the field.

2.1 A Brief History of Shaders

Shaders are a relatively new way of implementing computer graphics. Nvidia was the first company to introduce the graphics processing unit (GPU) to their devices in 1999 [8]. These GPUs operate by executing a fixed pipeline of commands, and until about 2002, this pipeline was strict and could not be altered or interrupted [12]. In the early 2000s, the first shader programs were introduced. Shader programs allow developers to interrupt the GPU at certain steps, for example, a fragment shader interrupts the step where the GPU renders pixel color. From here, programmers can use code to manipulate the graphics. Early shader programs were coded in machine dependent assembly-type languages, offering little versatility when it came to graphics manipulation and rendering. In 2004, a high-level shader language called OpenGL shading language (GLSL) was introduced, allowing the use of shaders across many platforms. At this time, the only portions of the graphics processing pipeline that could be replaced were the vertex and fragment portions. Thus, a vertex shader manipulates the vertex geometry and a fragment shader manipulates the color of individual pixels. In 2008 and 2012, two more replacements to the pipeline were made, adding geometry and compute shaders respectively.

2.2 Shaders in Video Games

The earliest examples of shaders being used in video games is around 2004 with games such as Half-Life 2 which were made using Direct3D, another API owned by Microsoft that is similar to OpenGL in functionality [10]. In this example, shaders were used to map textures onto 3D objects.

More recently, in the report "Evolving Pixel Shaders for the Prototype Video Game Subversion", Howlett et al. address the use of pixel shaders in their video game Subversion. Pixel shaders, also known as fragment shaders, control the color, texture mapping, and effects of individual pixels in an application. According to Howlett et al., pixel shaders are able to render effects more effectively than other methods of representing graphics [2]. Due to their effectiveness, pixel shaders are commonly used in modern video games to process graphics.

Subversion takes place in a procedurally generated city landscape with randomly formed landmasses, skyscrapers, and roads. Because the scene is defined using simple triangular meshes, shaders can be used to alter them on the fly; this allows the user to interact with these meshes in real time by selecting the desired color for the landscape. The use of shaders in Subversion allows the user to alter their environment while playing, allowing them to alter their environment based on their desired experience. [2].

The use of shaders in regard to Subversion is not only limited to the pixel shader, but includes the vertex shader to model the entire landscape as well; however, this use is static. In other words, the vertex shader is used to generate the initial landscape at runtime but otherwise does not change. The user input has no effect on the vertex shader or the geometries within Subversion, but they do affect the displayed pixel colors through the pixel shader. This means that the user is only able to alter the color of the landscape, but the rest remains static. One possible improvement would be to give the user more control over the landscape by allowing input for building height and shape. By giving the user more control over the scenery, they would feel a greater connection to the game [2].

2.3 Shaders and Mobile Devices

The second example of application of shaders is in the context of mobile devices. Alex S. C. Lima and Edson A. C. Junior, discuss this in their report "Experimental Approach of the Asymptotic Computational Complexity of Shaders for Mobile Devices with OpenGL ES" [1]. In this report, Lima et al. explore the rendering times of multiple shading methods on Android and iOS mobile devices. Their goal is to analyze these rendering times as the level of detail, number of polygons, increases. At the beginning of the report, Lima et al. acknowledge that these devices typically have weaker CPU and GPU capabilities than a computer or even laptop. This means that it is important for developers to properly know the rendering limitations of these devices in order to make an application that runs fast and efficiently [1].

Lima et al. cite that the first priority in creating a video game should be performance; graphics play a major role in the performance of a game based on their rendering speeds. Thus, they tested the rendering time of a variety of shaders on both Android and iOS mobile devices. These shaders were as follows: Gouraud, Phong, Red, Toon, Flat, Random Color, Simple Texture, Cube Map, and Reflection. Lima et al. created a mobile platform that allows the user to select the 3D object, shader type, and the amount of detail (polygons) to use when rendering; this allowed them to test all possibilities efficiently. Using OpenGL ES they loaded a variety of 3D objects to use for testing and applied each of these shaders to the objects. They tested these shaders for a range of polygon amounts and recorded their results [1].

Their findings showed that as the polygon count of an object increased, the rendering time increased regardless of the shader type; it is important to note that some shaders were inherently faster at rendering such as the Flat shader; regardless, their render time still increased with the number of polygons [1].

In general, this report acknowledged the essential role that shaders play in mobile game development. Lima et al. acknowledge that game performance and graphic

detail play major roles in the success of a game, and OpenGL allows developers to efficiently render detailed meshes [1].

2.4 Shaders in Augmented Reality

The final example of application of shaders is in augmented reality, discussed by Rikard Olajos and Michael Doggett in their technical report "Sparse Spatial Shading in Augmented Reality". In this report, Olajos et al. aim to create a seamless augmented reality experience that blends generated 3D graphics with reality. Olajos and Doggett achieve this seamless reality experience by color matching pixels from a live camera capture of the player's environment [11].

A major concern raised by Olajos and Doggett is one of program performance; they want to create an augmented reality program that's fast and efficient while still looking realistic. In the previous section, the use of shaders in the context of mobile devices was discussed. This is another example where the authors, Olajos and Dogatt, realize the limitations of a mobile device's GPU. Since they recognize this problem, they use pixel shaders to achieve their goal [11].

Using a quadtree data structure to hold the locality of the data, six directional vectors, and lighting data from the camera parameters, Olajos and Doggett were able to create a seamless transition from 3D graphics to reality. Since the lighting data was retrieved from the camera they were able to sample colors from the player's environment. They used these color values to shade the 3D objects in a manner that matches the surroundings. Put simply, they used the sampled colors to find the midpoint between the brightest pixels and the recorded camera data. The midpoint color, as well as some lighting functions, are applied to the 3D object. From there, the GPU is able to render the graphics based on these inputs, allowing for a dynamic use of graphics that changes depending on the input scenery [11].

2.5 Project Goals in Relation to Prior Research

Through the exploration of the past and current use of shaders in graphics applications, it can be concluded that shaders are still an essential part of graphics rendering. This is because there is a demand for more realistic graphics as well as a desire for a smooth running experience. Both of these factors play a major role in the use of graphics in a variety of contexts as shown previously.

One area of interest that has been overlooked by these sources is the use of dynamic graphics: using GLSL to dynamically create and render graphics that change over time. This concept was somewhat touched on in the Olajos et al. report, where they rendered an object based on the player's environment [11]. The purpose of this thesis project is to build upon prior implementations of shader graphics with the goal of making most of them dynamic; the resulting game should change and morph over time, giving the game a lively feeling no matter the setting or subject matter. This will be done by animating the meshes using the vertex shader.

Chapter 3

Approach

The main idea of this implementation is to create a video game using mostly shader-based geometries. This means that there will be very few imported mesh files, instead there will be quite a few shader files detailing the geometries of the landscape, objects, and effects.

As seen from the related work, shader-based geometries are beneficial to the creation and function of a video game. A video game typically has a multitude of required meshes, and if all of these meshes are represented by an object file, there may be a fair amount of processing required to both download and render these meshes on a client side. In terms of online gaming, an approach of using mostly shader-based geometries will avoid the need to download extensive object files in exchange for downloading one or multiple GLSL files which contain fast running C-based code. Furthermore, the use of shader-based geometries enables the meshes to be dynamically rendered rather than statically rendered. What is meant by this is that meshes can change based on user input or other factors active during game play. This is seen in the Howlett et al. article where user input could determine the color of the scenery and affect the user's gameplay experience.

In this implementation, the dynamic ability of shader-based meshes is used to

animate the scene and objects. This allows the landscape and objects, although their geometry is simplistic, to look and feel realistic. It allows this alien landscape to feel real and alive, something which is important to the user experience of a video game. This will be unpacked further in the following section.

Chapter 4

Implementation

One challenge facing video game developers is increased latency caused by rendering complex 3D meshes and objects. As they seek to improve the gaming experience, developers create more detailed meshes and objects, increasing latency. This is especially true for mobile devices, where the GPU processing power is far less capable than the GPU of a console or computer. One way that developers have accounted for this is to use shaders to render graphics. OpenGL Shading Language (GLSL) is a high-level shading language routinely used for rendering graphics in programming applications including simulations, visualizations, and video games. GLSL allows developers to alter 3D objects using shader files. These files, which are applied to the material of an object, can be altered to fit requirements of the programmer or application. This project will explore the ways in which vertex and fragment shader files can be used to create dynamic graphics while maintaining high-performance.

In the initial stages of this project, the goal was to understand the capabilities of GLSL. A plane geometry was generated by Three.js libraries, and a basic material containing the associated vertex and fragment shader files was also created. The goal was to turn this static plane geometry into an animated waving piece of fabric. The first wave-like fabric program was written in JavaScript and used Three.js libraries

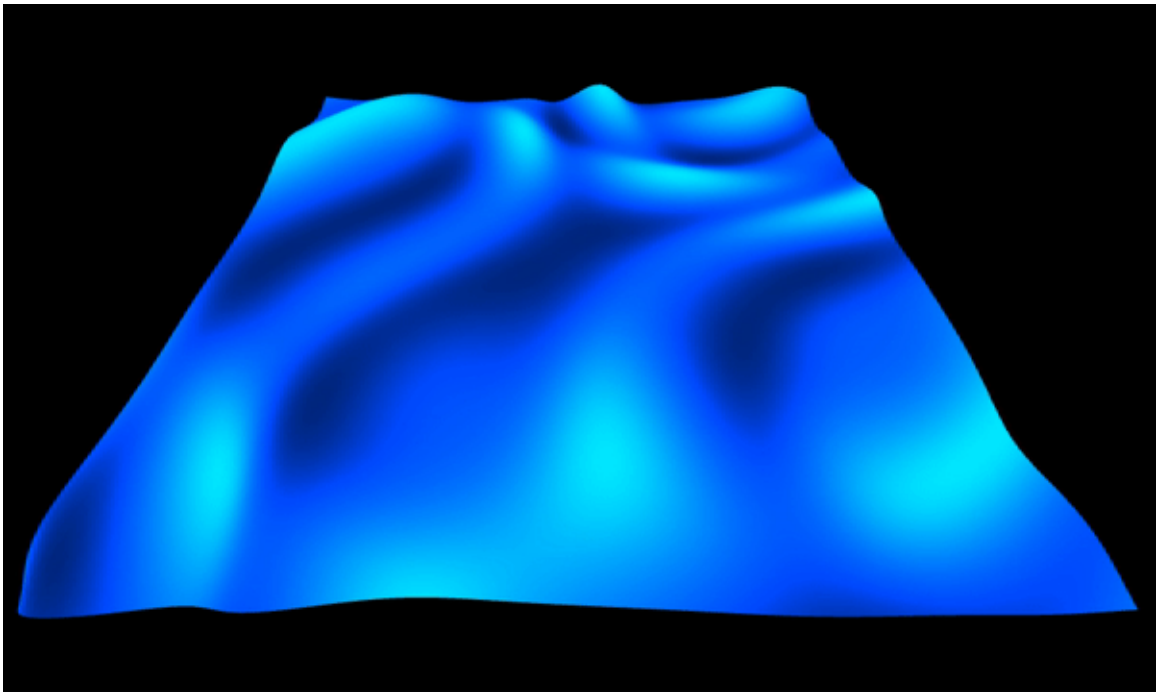


Figure 4.1: The final simple wave

to create a basic scene and environment for testing the mesh. Three.js, a popular JavaScript library, is built on top of WebGL, allowing it to work efficiently with GLSL, an OpenGL based language. On its own, the Three.js plane geometry is a stagnant rectangle of vertex points, a 1x1 plane with zero subdivisions along the geometry. These parameters were changed to result in a 10x10 plane with 50 subdivisions along the width and 50 subdivisions along the height. The basic material for the plane was defined using an imported vertex shader and fragment shader files.

In the vertex shader file, the individual plane vertices can be transformed and translated in 3D space. Since the end goal was to create a wave-like fabric, a smooth modulus function was implemented [14], rather than opting to use the standard GLSL mod() function. This is because the standard GLSL mod() does not provide a smooth transition between displacement values whereas the smoothMod() function does.

In the fragment shader file, the color and texture of each vertex is determined. In this initial implementation, a simple blue color was applied to the fabric. To provide

shading and dimensionality, the displacement factor for each vertex was also included in the fragment color calculation:

$$vec4(vec3(0.0, 0.2, 0.6) * vDisplacement * 5.0, 1.0);$$

Perlin noise was used to add some variance to the wave, making it a natural movement rather than a uniform movement [6]. Applying these two functions to the plane, and updating values using a time variable, the plane was successfully animated to look like wavy fabric. In order to see the displacement values over time, a time variable was added to the UV variable at each clock cycle before the noise and wave calculations. This resulted in a seamless animation which continues while the program runs; refer to figure 4.1 for the final result. This section of the GLSL programming experiments gave insight into the nuances of JavaScript, Three.js, and GLSL, leading to the next phase of development.

4.1 Initial Concept

The main goal of this project is to develop a video game using Three.js geometries and GLSL shaders. The first consideration was what genre of game would best suit this goal: action, artistic, adventure, puzzle, or fighting were just some of the genres considered. Since the challenge was to use basic meshes from Three.js with GLSL shaders, there is a limit to what can be represented efficiently. Because of this constraint, high-detailed meshes would be difficult to replicate. Instead, the game would rely on the dynamic capabilities of GLSL and animation techniques to create an immersive environment for the player. Therefore, it was decided that an appropriate genre would be an artistic-collection game. In this type of game, the design is just as important as functionality. The number of items needed to implement a collection game would allow ample opportunity to showcase the capabilities of Three.js and

GLSL, and challenge the goal of making an immersive gaming experience. The next challenge was to consider a theme based on this genre.

The scene of the game is on an alien planet in outer space. The player, an astronaut otter, has to explore the planet for food. The planet is covered with many varieties of delicious fruit, but is also teeming with poison. The user must navigate the space otter to all of the fruits while avoiding the poison to win the game. If the user successfully collects all of the planet's fruit without ingesting any poison, they win! With this theme and genre in mind, the first stages of game development began.

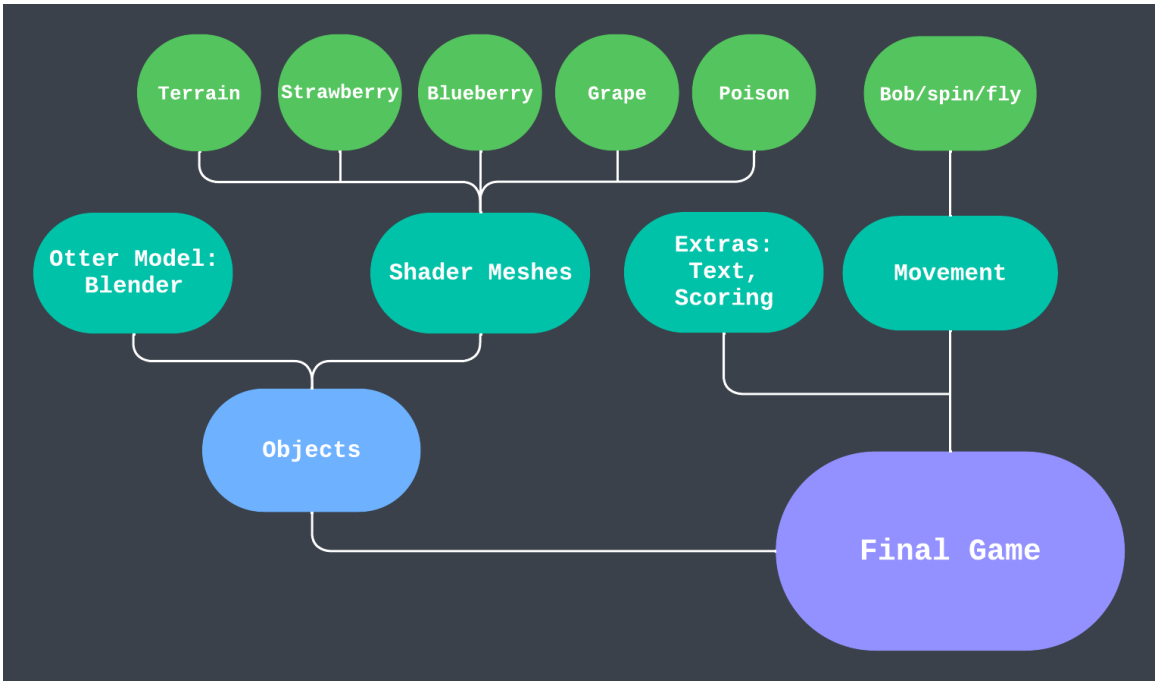


Figure 4.2: The *Space Otterssy* game structure

Built off of Three.js starter code from Princeton University's COS 426: Computer Graphics course, *Space Otterssey*, is a visually pleasing collection game. The structure of the game is detailed in figure 4.2. The game is composed of four total objects, one playable character, and an alien landscape, all of which are implemented using Three.js libraries and GLSL shaders. The following sections will detail the implementations of each of these additions.

4.2 Landscape

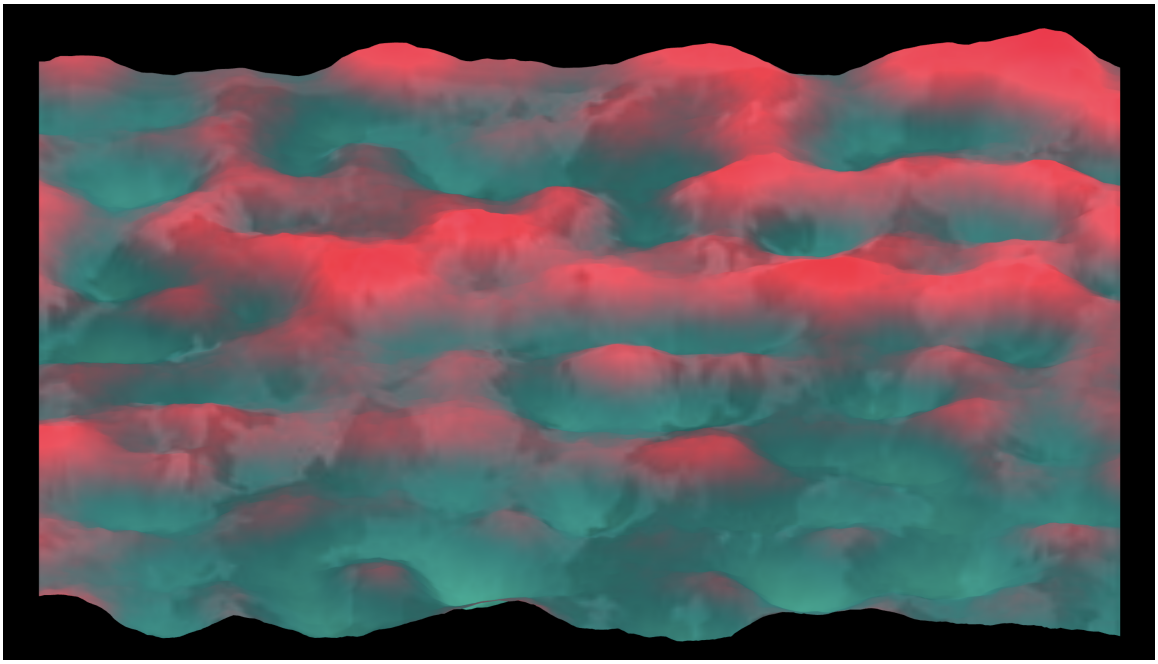


Figure 4.3: Zoomed-out view of the *Space Otterssey* landscape

Space Otterssey's alien scenery is a mountainous landscape with peaks and valleys that change and morph over time. The base plane for this model is a simple 100x150 unit Three.js plane geometry with 200 subdivisions along the width and height. The material applied to the plane consists of vertex shader and fragment shader files. Within these files, a displacement variable is calculated and applied to each vertex. This displacement variable is calculated by applying a noise function to the x and y position of the vertex. Three noise functions are used in total with frequencies equal to 1.5, 5.0, and 36.0, respectively. These noise factors are layered and averaged to give the illusion of a crumbly, rocky surface. At each clock cycle, the x and y position coordinates are updated using a time variable before calculating the respective noise factors, this results in a morphing surface texture as the device's clock cycle iterates.

The coloring of the landscape is handled in the fragment shader file. A simple rock-like gray texture is loaded as a png file, and a color with the following calculations

is applied:

$$vec4(0.9 * vDisplacement, 0.9 - 0.2 * vDisplacement, 0.9 - 0.1 * vDisplacement, 1.0) * 0.3$$

The *vDisplacement* comes from the previous displacement calculations in the vertex shader. Since the *vDisplacement* is applied to each color of the RGB scale independently, this results in different color changes. The result is a landscape that shifts from pink to green hues, which can be seen in figure 4.3.

4.3 Fruit

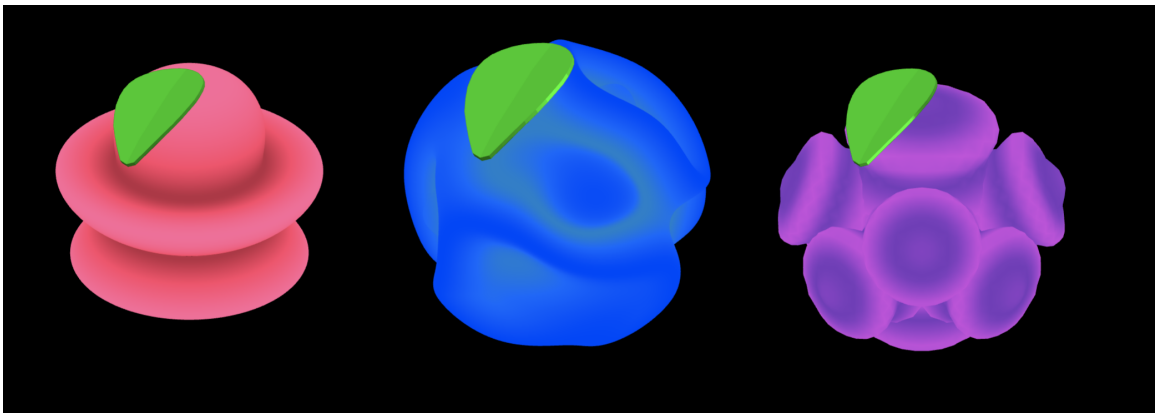
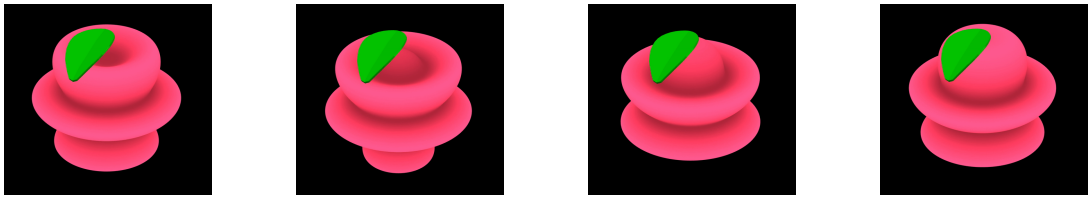


Figure 4.4: Three unique fruit objects from *Space Otterssey*. From left to right: Strawberry, Blueberry, and Grape.

Out of the four items added to the game, *Space Otterssey* has three possible fruit types; all of which are formed using simple Three.js icosahedron geometries and GLSL shader files. Each mesh also contains a simple leaf mesh constructed in Blender and imported to the object's main JavaScript file. For the purpose of describing the implementation of each fruit, let the fruit be denoted by *Strawberry*, *Blueberry*, and *Grape*: refer to figure 4.4 in order from left to right.

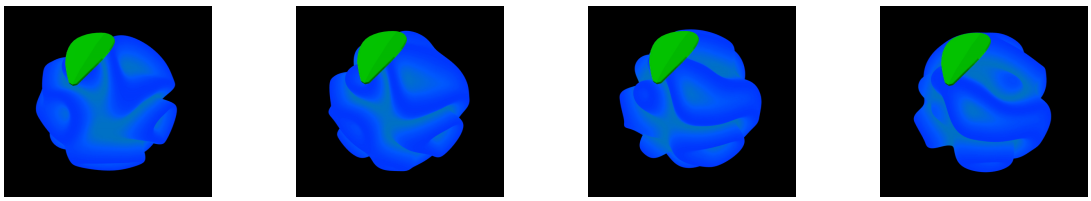
Starting with the *Strawberry* object, a base icosahedron mesh with 100 subdivisions is imported to that item's main JavaScript module. The basic material is formed



using Three.js libraries and unique fragment and shader files. In the vertex shader, the position.y factor is adjusted based on the current time factor and smoothMod() is applied in the y-direction. Since a noise function is not applied, the resulting wave is uniform. This creates a lively wave animation that moves down the fruit in a pulsatile manner. In the fragment shader, this displacement factor is used to create a shaded object:

$$vec4(vec3(1.0 - 0.2 * vDisplacement, 0.2, 0.3) * vDisplacement, 1)$$

The *vDisplacement* factor is applied to the RGB values as a whole and individually to the red factor. This results in a smooth shaded gradient that loses red saturation as the *vDisplacement* increases.

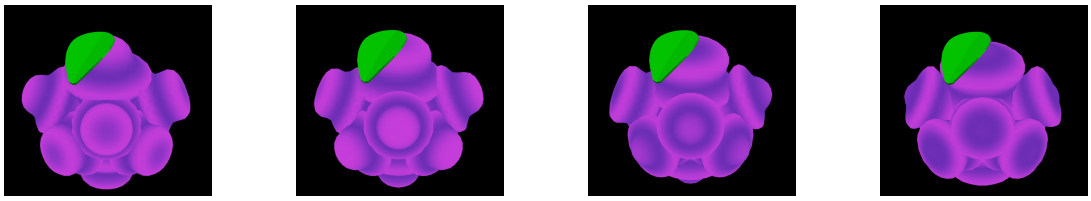


Next consider the *Blueberry* object. A base icosahedron mesh with 100 subdivisions is imported to that item's main JavaScript module. The basic material is formed using Three.js libraries and unique fragment and shader files. In the vertex shader, the position factor is adjusted in the x,y,z directions based on the current time factor. The noise() function is applied to these coordinates and then passed into the smoothMod() function for all directions x, y, and z and averaged. This creates a lively wave animation that moves across all axes of the object in a random pattern.

In the fragment shader, this displacement factor is used to create a shaded object:

$$vec4(vec3(0.0, 0.9 - 0.35 * vDisplacement, 0.7 * vDisplacement), 1)$$

The *vDisplacement* factor is applied to the green and blue RGB values. This results in a smooth shaded gradient that loses green saturation and increases blue saturation as the *vDisplacement* increases.



Finally, consider the *Grape* object. The base mesh is a bit more complicated than the previous objects in that it is represented using ten icosahedron meshes with five subdivisions each. The basic material is formed using Three.js libraries and unique fragment and shader files. In the vertex shader, the position.z factor is adjusted based on the current time factor.and smoothMod() is applied in the z-direction. Since a noise function is not applied, and the base icosahedrons are rotated before displacement, the resulting wave is uniform. This results in a wave that moves throughout the different sections of the fruit. In the fragment shader, this displacement factor is used to create a shaded object:

$$vec4(vec3(0.4 * vDisplacement, 0.2, 0.9 - 0.2 * vDisplacement) * vDisplacement, 1)$$

The *vDisplacement* factor is applied to the RGB values as a whole and individually to the red and blue factors. This results in a smooth shaded gradient that loses blue saturation and gains red saturation as the *vDisplacement* increases.

4.4 Poison

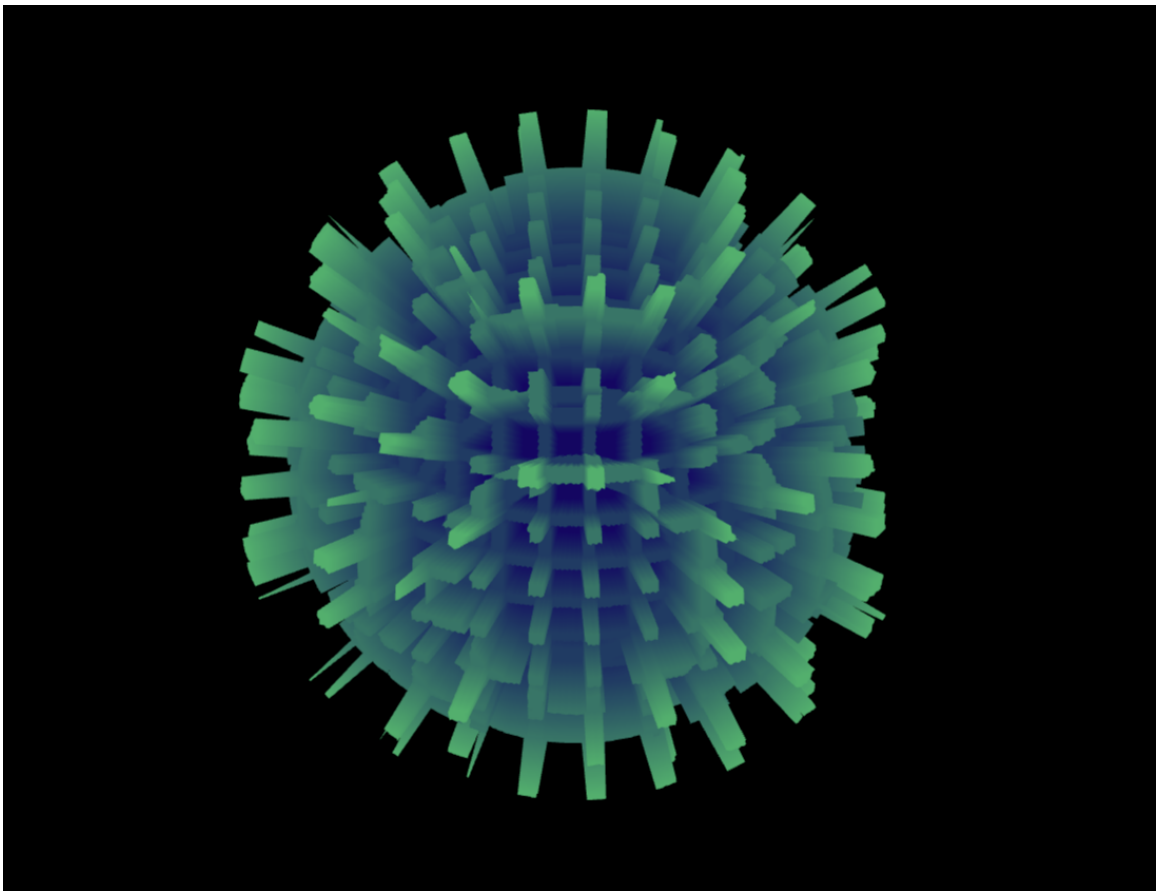


Figure 4.5: Poison object

Space Otterssey has one additional object meant to represent a poisonous fruit. This mesh is also formed using simple Three.js icosahedron geometries and GLSL shader files. The base icosahedron mesh has 100 subdivisions. The basic material for this icosahedron consists of a vertex shader and fragment shader. In the vertex shader, the position factor is adjusted in the x,y,z directions based on the current time factor. The noise() function is applied to these coordinates and then passed into a waveSpike() function. This function takes the vec3 position coordinates and applies the step() function to create a stark displacement value. This results in a spiky, pulsating object. In the fragment shader, this displacement factor is used to

create a shaded object:

$$vec4(vec3(0.1, 0.7 * vDisplacement, 0.4), 1)$$

The *vDisplacement* factor is applied to the green RGB value. This results in a smooth shaded gradient that gains green saturation as the *vDisplacement* increases.

4.5 Otter

Figure 4.6: Otter object

The otter mesh is the only mesh in *Space Otterssey*, besides the leaf mesh, that is rendered from an imported object file. The base otter mesh was downloaded from an open source 3D model website [9]. This base mesh was uploaded to Blender, a free 3D modeling software, where a jetpack, helmet, boots, and textures were added. Three.js and GLTF only supports a few select textures including principled BSDFs which were used in this case; although the materials do not look as realistic as desired, they fit within the restrictions of Three.js and GLTF. After being exported from Blender, the final mesh was imported to the game using the Three.js GLTFLoader() function.

The smoke from the Otter's jetpack is created using a modified version of a sample JavaScript Three.js low-poly smoke object [7].

4.6 Bobbing Animation

In order to make the game feel more realistic, a bobbing function was added to the Strawberry, Blueberry, Grape, Poison, and Otter objects. The original bobbing animation was altered from the COS426 starter code [15]. The bobbing function works by changing the object's position or rotation based on a trigonometric function. For instance, the bobbing function for the Strawberry, Blueberry, Grape, and Poison changes the y-position of the object using a cosine function applied to the timestamp value. This means that these items just bob up and down in a loop throughout the game. In the case of the Otter, the bob function is a bit more complicated. In order to differentiate it from the items, the y-position is changed using a sine function applied to the timestamp rather than a cosine function. Furthermore, the z-rotation and x-rotation are changed by a sine and cosine function respectively. This means that the Otter sways around at the same time as bobbing up and down, making his flight pattern look realistic.

4.7 Other Mechanics: Character Movement, Text, Scoring

Some additional features of the game include character movement, text, and a scoring system. The Otter character can be moved holding down either WASD or arrow keys. The movement speed can be increased by holding down the shift button in addition to a directional key. When the shift button is activated, the Otter enters sprint mode which doubles the movement speed and increases the amplitude of the bobbing

motion. This creates an effect where the Otter looks even more unstable the faster he goes.

Figure 4.7: Start screen

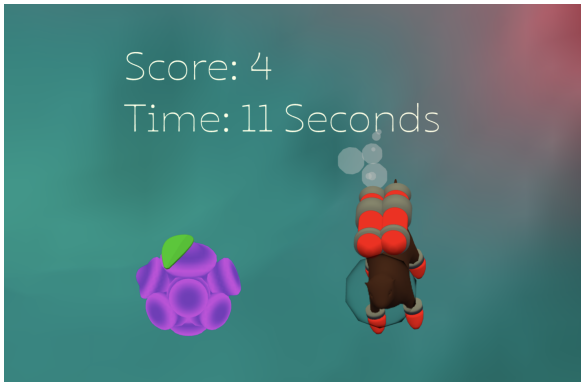


Figure 4.8: Score screen

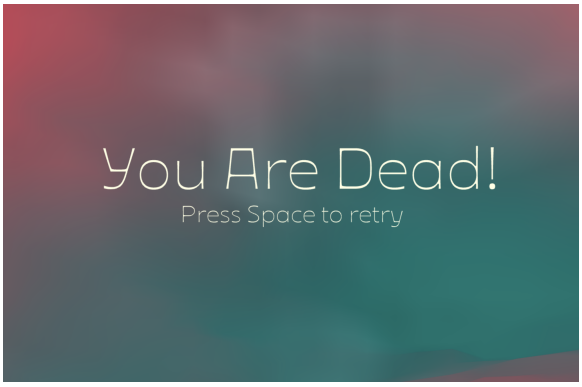


Figure 4.9: Dead screen

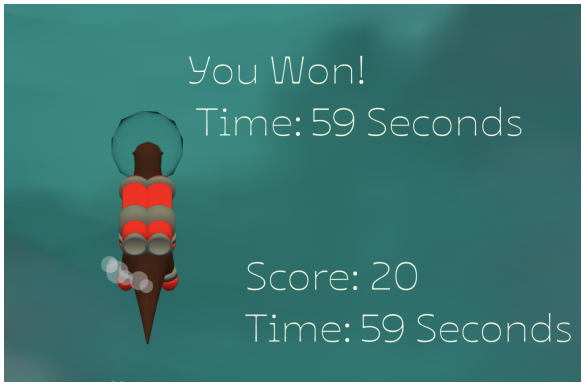


Figure 4.10: Win screen

All text in the game uses the font "Ojuju" imported online from Google Fonts [5]. At the beginning of the game, instructions are displayed for the player. After consuming a Fruit, a text prompt will appear telling the player their current score (how many Fruits have been collected) and the elapsed time in seconds. If the player manages to eat all 20 Fruits, the text will congratulate them and display their final time. If the player accidentally eats a Poison item, the text will prompt the user to start again by pressing the spacebar; this will reload the whole website. Refer to figures 4.7-4.10.

Chapter 5

Conclusions and Future Work

Figure 5.1: A screenshot of the final version of *Space Otterssey*

In conclusion, this project has shown that a modern video game can be implemented using mostly shader based rendering. The result is a fully functional and visually pleasing video game created using mostly GLSL shaders to render objects (refer to figure 5.1). The full code for *Space Otterssey* can be downloaded from Github

and run easily on a local computer [4].

Some limitations of this project are as follows. *Space Otterssey* was originally intended to be hosted on GitHub Pages, which would allow it to be played by anyone. Unfortunately, GitHub Pages does not support GLSL files which are essential for this game to properly run. Some future work may include researching ways to host the game through AWS services. Additionally, *Space Otterssey* was intended to be a multiplayer game, but since the website hosting didn't pan out, neither did the multiplayer aspect. In future, a multiplayer feature could be added. Some other future work would be to add more scenes to *Space Otterssey*, as well as a high score tracker.

Bibliography

[1] E. A. C. J. Alex S. C. Lima. Experimental approach of the asymptotic computational complexity of shadersfor mobile devices with opengl es, 2014. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7000046.

[2] C. B. Andrew Howlett, Simon Colton. Evolving pixel shaders for theprototype video game subversion, 2010. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=34aec673f874dda0922dd3ec0b6b26aaf80d84a4.

[3] N. Co. Consolidated financial results and outlook. https://www.nintendo.co.jp/ir/pdf/2023/230803_2e.pdf.

[4] L. Gardner. Space otterssey code. https://github.com/Lauren-Gardner/Space-Otterssey-Thesis.

[5] GoogleFonts. Ojuju font. https://fonts.google.com/specimen/Ojuju.

[6] S. Gustavson. Perlin noise. https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83.

[7] F. Hung. Low-poly smoke particles in three.js. https://medium.com/geekculture/low-poly-smoke-particles-in-three-js-acd3942fd250.

[8] Investopedia. What is a graphics processing unit (gpu)? https://www.investopedia.com/terms/g/graphics-processing-unit-gpu.asp#:

~:text=Nvidia%20was%20the%20very%20first,trading%20around%20$645%20per%20share.

[9] Nishio. cgtrader. https://www.cgtrader.com/free-3d-models/animals/mammal/lowpoly-otter.

[10] PCGamingWiki. Half-life 2 wiki. https://www.pcgamingwiki.com/wiki/Half-Life_2.

[11] M. D. Rikard Olajos. Sparse spatial shading in augmented reality. In *19th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2024. https://www.scitepress.org/Papers/2024/124293/124293.pdf.

[12] J. Rodriguez. *GLSL Essentials*. Packt Publisher, 2013. https://subscription.packtpub.com/book/programming/9781849698009/1/ch01lvl1sec08/a-brief-history-of-graphics-hardware.

[13] S. Stein. Zelda tears of the kingdom, reviewed: The switch's swan song. https://www.cnet.com/tech/gaming/tears-of-the-kingdom-reviewed-the-new-zelda-game-is-the-switchs-swan-song/.

[14] C. Stiles. Smooth mod, 2021. https://www.curiouslyminded.xyz/til/smooth-mod/.

[15] P. U. Cos426 final project: Starter code. https://www.cs.princeton.edu/courses/archive/spring22/cos426/assignments/Final-Project/.