Zefeir Shaders

Shayne Hayes
University of California, Santa Cruz
United States
shmihaye@ucsc.edu

Sebastian Shelley
University of California, Santa Cruz
United States
seshelle@ucsc.edu

Autumn Washington-English
University of California, Santa Cruz
United States
amwashin@ucsc.edu

ABSTRACT

We are developing a game called Zefeir, a 3D platformer where you run, glide, and dive through a large, open skyworld as Zefeir the Griffin. In this paper we break down three shaders central to our art style: a rock shader, a fluff shader, and a procedural skybox.

CCS CONCEPTS

• **Computing methodologies** → *Graphics systems and interfaces*

KEYWORDS

Real-time rendering

ACM Reference format:

G. Gubbiotti, P. Malagò, S. Fin, S. Tacchi, L. Giovannini, D. Bisero, M. Madami, and G. Carlotti. 1997. SIG Proceedings Paper in word Format. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97)*, 4 pages. https://doi.org/10.1145/123 4

1 ROCK SHADER

1.1 Motivation

The core aesthetic of our game is discovery, and so we have designed a large, open world of floating islands for players to explore. However, because we have such a limited timescale, UV unwrapping every piece of world geometry would not be practical. Our solution is a shader that textures geometry based on world coordinates instead of texture coordinates. This technique is referred to as World-Space UV Mapping [1] and allows us to texture objects without having to specify UV coordinates! Our goal for this shader was to render several different rock layers to achieve an effect similar to the rocky walls in Super Mario Odyssey (see Fig. 1).

1.2 Implementation

We implemented this effect as a simple surface shader in Unity so we could let the engine handle lighting and shadows for us. We can achieve simple rock layers very easily with 3D Perlin noise. In pseudocode:

```
int bandInt = bandColors * perlin(0,
        y * (1/bandHeight), 0)
float band = strength * bandInt * (1/bandColors)
o.Albedo = baseColor - band
```

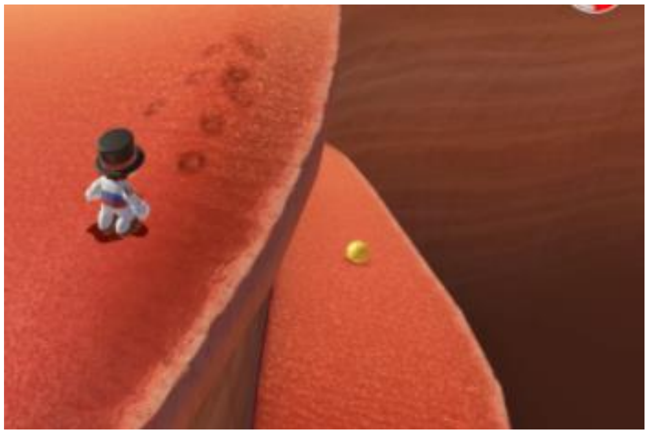

Figure 1: Rock layers in Super Mario Odyssey

For a given fragment, we start with some base color and darken it by some value *band* that is between 0 and some *strength*. The amount the fragment is darkened is tied to the fragment's y world coordinate, thus all fragments with the same y coordinate will be assigned the exact same color. We round to an integer and then back to a float to achieve distinct color bands instead of a gradient (which appears blurry). The variable *bandColors* controls the number of distinct band colors output by the shader and *bandHeight* scales the heights of the bands up and down. Finally, we use Perlin noise to add some irregularity to the band heights.

We now have a shader that colors in fragments based on their y coordinates. However, with just those three lines, our bands would be perfectly straight. To make our bands wavy, we can offset the y coordinate based on its position in the world. Again, in pseudocode:

```
y += amplitude * perlin(x * wavelength,
y * wavelength, z * wavelength)
```

We are essentially tricking the shader into thinking two y coordinates are slightly different, even when they are the same. Perlin noise makes the waviness a little more irregular and natural looking compared to sine or cosine. The variables *amplitude* and *wavelength* can be used to control the height and length of the waves, respectively.

In addition to the bands, we chose to add splotches to our walls to give them a little more texture. This can be achieved in a very similar manner to the bands, only the amount we subtract from the base color is based on the fragment's x, y, and z coordinates and not just its y coordinate. We decided to add three layers of this noise at different granularities -- some large splotches, some medium splotches, and some small splotches.

1.3 Results

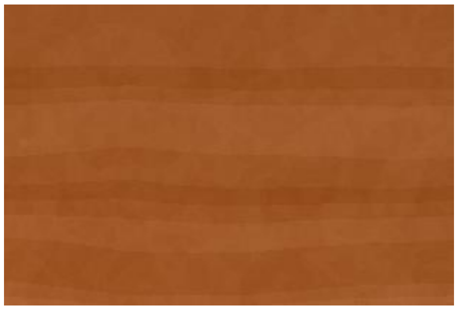

Figure 2: The rock shader in all its glory

The result, shown in Fig. 2, is exactly what we were hoping for -- wavy rock layers with a little added noise. And because it is all tied to world coordinates, we can mash together pieces of world geometry and our shader will texture them seamlessly, as shown in Fig. 3.

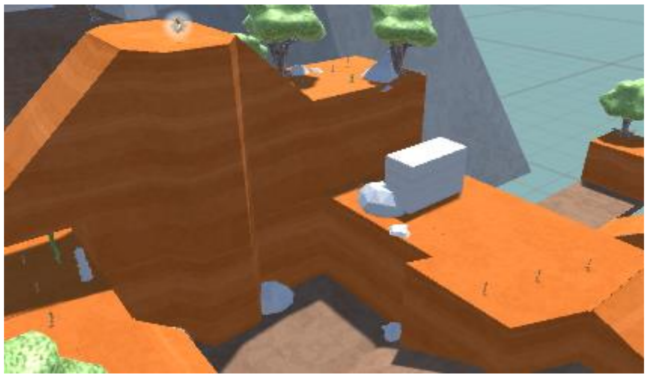

Figure 3: The rock shader on several objects put together

As an added bonus, this technique hides z fighting, or when two objects overlap and flicker between two textures. Because the overlapping pixels share the same world coordinates, they are given the exact same color, and thus z fighting is practically invisible.

However, this approach does come with one drawback: your source of Perlin noise must be chosen wisely. Our first implementation of this shader calculated 3D Perlin noise directly on the GPU as it was needed, which led to a noticeable drop in framerate since our shader calculates Perlin noise five times per fragment. For our final implementation we pass in Perlin noise as a Texture3D, or a 3D lookup table representing volumetric data, so we do not have to calculate it on the fly [2].

1.4 Evaluation and Future Work

To improve this shader we discussed using bump mapping or vertex displacement to add some more depth and texture to make it even more rocklike. However, our first attempt at adding bump mapping ended up looking ugly and introduced a wide range of lighting bugs, as shown in Fig. 4. We plan to revisit this topic if we have extra time for polish.

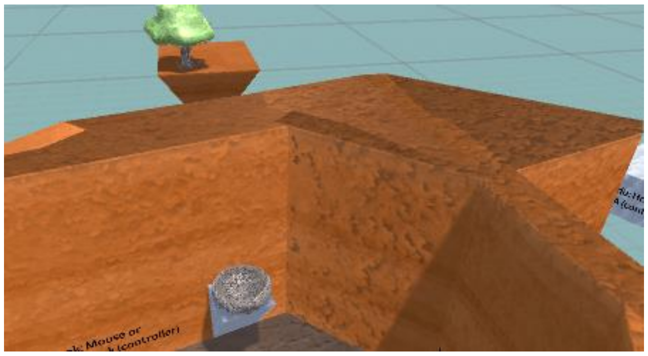

Figure 4: Some very ugly bump mapping

2 FUR SHADER

2.1 Motivation

Creating a mesh with hair takes a greater amount of time than just creating a simple mesh. Our character has gone through multiple design changes over the course of the game’s development. A shader that automatically adds fur to a mesh reduces the amount of time our artists need to spend on modeling. Procedural hair generation on the mesh was a possibility, but it increases the number of vertices that have to be animated.

Figure 5: Similar approaches appear to have been used in the original *Shadow of the Colossus*

2.2 Implementation

The mesh is a vertex and frag shader that uses built-in Unity lighting functions to react to light and shadow. Our fur shader is composed of multiple passes of a vert/frag shader. Each pass renders the mesh, scaled up larger than the previous in the vertex shader. This is done by taking each vertex and moving it along its normal. These repeated meshes form shells which make up the body of the fur. The first pass renders all the fragments of the mesh, and does not expand the vertices of the mesh. This forms the base layer of the hairy object. This prevents holes from appearing where you can see through the mesh.

The fragment shader is a basic fragment shader with Unity lighting functions so that it can react to light inside the game. The shells of the mesh look furry because more and more of the fragments of the shell are discarded as the shells become larger. The shader uses a 2D noise texture to accomplish this. Fragments are discarded when the alpha of the texture is below a transition value. As the shells become larger, the transition value also becomes larger. The fragments become sparser as the shells become larger, just like how fur works in real life. The noise texture has many strong points interspersed with larger low areas.

Fragments also write to the depth buffer. If a fragment is found to be behind another, it is also discarded. This is necessary to prevent overlapping fur from being overwritten by fragments behind it.

2.3 Results

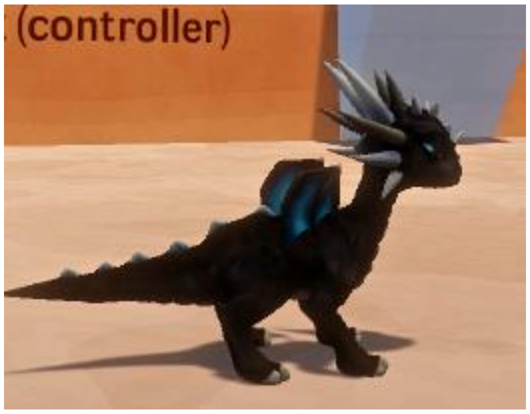

Figure 6: Basic mesh without fur

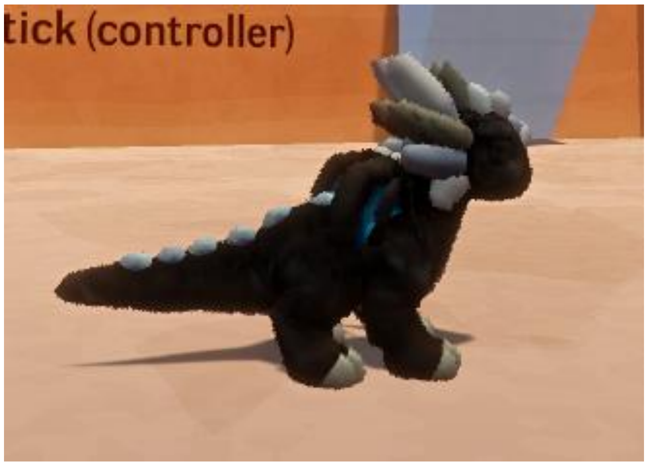

Figure 7: The fluff shader

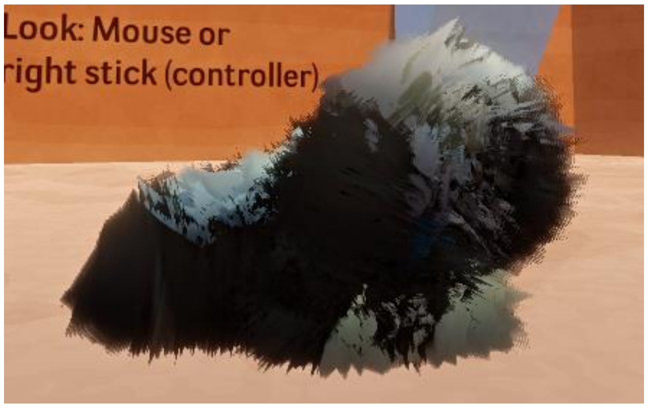

Figure 8: Larger shells to illustrate how it works. Notice the more obvious banding around the edges of the mesh where the shells are visible.

2.4 Evaluation and Future Work

A slight wind effect could be produced by moving the vertices gradually over time with a sine function. The amount is then multiplied by the size of the shell. This results in the hairs appearing to be bent by wind. The effect changes over time and could be modulated by the game, such as by the character's speed.

This same effect could also be used to represent gravity bending the fur. By supplying the shader with the world down direction, the hair would become more realistic.

The most important future feature is using UV coordinates in the vertex shader to determine how long the hair is in that position. The UV coordinate would look into a texture that would multiply the hair length by the red channel in that texture. This would allow us to remove hair in spots where it does not belong such as around eyes and on claws.

3 DYNAMIC SKYBOX SHADER

3.1 Motivation

The primary aspect of creating a dynamic procedural skybox is to add not only a sense of time but something to populate the sky with objects to add to the environment. This shader's purpose it to primarily reduce the amount of objects in our senior project, instead of creating individual clouds or volumetric clouds which kills frame performance. The goal is to keep maximum performance but with a beautiful environment to explore.

3.2 Implementation

Achieving the colors of various times like Dawn and Dusk is essentially Color math.

```
float3 v = normalize(i.texcoord);
float p = v.y;
float p1 = 1 - pow(min(1, 1 - p), _Exponent);
float p2 = 1 - pow(min(1, 1 + p), _Exponent2);
float p3 = 1 - p1 - p2;


half3 c_sky = _Color1 * p1 + _Color2 * p3 +
_Color3 * p2;
```

This takes the chosen colors in the unity interface and applies the results along the Y axis. P1/P2 determines the numerical application of the _Exponent/_Exponent2, which basically means how much of this color should fill the upper/bottom half of the sky. P3 is the necessary math to make _Color2 Mix with _Color1 and _Color 3 for a hard code transition, where c_sky comes in to complete the sky gradient.

3.3 Results

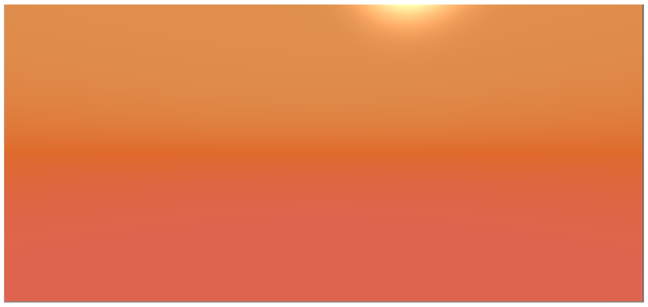

Figure 9: Sky Gradient with three colors and Sun.

3.4 Evaluation and Future Work

There is a section in the code where there will be horizon manipulation. The purpose is to take an _Exponent of the horizon to apply a curvature. This is to add an effect of the further up you go in the atmosphere the more you can actually see the curvature of the earth.

There is also the presence of ”float4 _Gradient_ST;” as of right now this does not do anything to the code yet. The plan is to figure out to add an additional gradient to tint the sky. This is to save the process of choosing 3 colors over and over again for the different color phases of Dawn, Dusk and Night. The tint will be a single color overlay using that gradient , so that it applies that color over the base skybox.

Sun/Moon and Sky time lapse. I need to implement a movement function where the Sun/Moon rises and sets over a certain time triversed in the game as well as change the colors of the sky to go through the phases of: Dawn, Day, Dusk, and Night..

Populating the sky with starfields and planier clouds. One of my fellow team members tested with volumetric clouds and it really killed the frame rate from what was reported. My intentions is to optimize frame rate but still have a beautiful backdrop to enjoy while exploring in our senior project.

ACKNOWLEDGMENTS

We would like to thank our professor Angus Forbes for introducing us to shader programming and our TA Lucas Ferreira for showing us how to port GLSL shaders to Unity.

REFERENCES

[1] "World Space UV Mapping." *Shader Forge Wiki*. N.p., 5 Oct. 2014. Web. 25 Mar. 2018. <http://acegikmo.com/shaderforge/wiki/index.php?title>.

[2] "Texture3D." *Unity Documentation*. Unity, 1 Mar. 2017. Web. <https://docs.unity3d.com/ScriptReference/Texture3D.html>.