Pixelation, Voronoi, and Crepuscular Rays

Andrew Gwinner
agwinner@ucsc.edu

Jacob Le
jamale@ucsc.edu

Nicholas Ho
nidho@ucsc.edu

ABSTRACT

A project that is required by Angus Forbes's course involving computer game graphics and real-time rendering that is part of the University of California's Computational Media department. The scene that was created for said project includes shaders that cause pixelation, creation of voronoi diagrams, and crepuscular rays written in HLSL with the support of the game engine, Unity.

1 INTRODUCTION

1.1 PIXELATION INTRODUCTION

Censoring portions of an image through pixelation is a common effect on TV. Its ability to be used for humorous effect made it an intriguing effect to pursue implementing. To generate pixelation, slices are made across the screen both vertically and horizontally. Initial attempts at pixelation implementations involved the use of arrays that stored the position for each slice. Quickly it became clear this was inefficient and the strategy of storing all slice positions was abandoned.

1.2 VORONOI INTRODUCTION

Emulating the distortion effect that water has on light is a GPU exhausting task. Doing so in real time is even more taxing. As such, this project seeks to cheat by using Voronoi Diagrams to mimic caustics. The organic and cellular nature of Voronoi Diagrams lends greatly to the caustic effect.

1.3 CREPUSCULAR RAYS INTRODUCTION

Crepuscular rays, also known as volumetric lighting, is a popular effect which involves a light source emitting light shafts that are popularly referred to as "God Rays" or "God's Light" in some cases. The basic algorithm to create this effect is with shader that takes the camera's view and finds the light source along with anything that obscured the light's emission. The data is stored in an occlusion map and them blurred outwards, away from the light source to create the signature light shafts.

2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1 PIXELATION

2.1.1 Determining Which Slices Surround A Pixel.

A desired number of vertical and horizontal slices across the screen is given. For each pixel our goal is to identify which two vertical slices and which two horizontal slices surround it. The formula in Figure 1 is used to locate the horizontal slice that will be below the current pixel. In Figure 1, i will increase from 0 to the number of vertical slices until currentHorizontalSlice is larger than the y value of the current pixel. When that occurs, the vertical slice below the current pixel has been found.

$$currentHorizontalSlice = \frac{(i+1) * screenHeight}{(numberOfHorizontalLines + 1)}$$

Figure 1. Formula used for determining the position of a horizontal slice.

The last result from the formula is stored as well. When currentHorizontalSlice is larger than the current pixel's y value the last result produced from the formula in Figure 1 is therefore the horizontal slice that is above the current pixel. Should the very first result from the formula in Figure 1 be larger than the current pixel's y value the top slice that surrounds the pixel would be at a y value of 0.

An identical process is used for determining the vertical slices that surround the current pixel.

2.1.2 Sampling Color.

In this implementation of the code the method for selecting which color within the four slices to sample is done by simply finding the center of the square by finding the midpoint between both sets of slices.

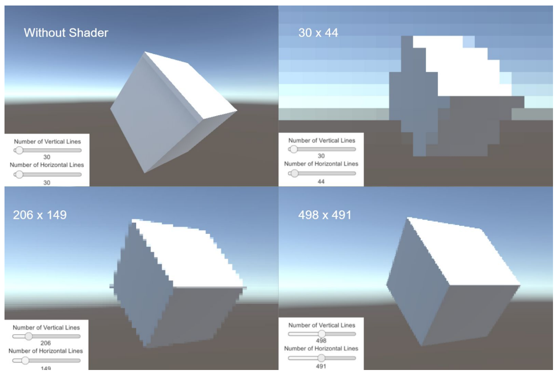

Figure 2. Example of results from shader with varying number of slices made across the screen horizontally and vertically.

2.1.3 Restricting The Pixelation to a Set Area.
To emulate a censoring effect that pixelates only a certain portion of the screen, a check can be made if the pixel is less than or more than values along the x and y axes and in those case return the exact color that pixel already has.

2.2 VORONOI

2.2.1 Voronoi Diagram Usage.
Voronoi diagrams create organic cellular shapes, similar to real life caustics. The use of Minkowski's distance function avoids the straight, inorganic lines that a traditional euclidean distance function would create. Let a and b be two dimensional vectors in the xy coordinate plane. Let d be an arbitrary value.

$$Distance = (|a_x - b_x|^d + |a_y - b_y|^d)^{\frac{1}{d}}$$

Figure 3. Formula used for Minkowski Distance between a and b.

In this case, d is a user defined value set to 1.5. With our distance formula, we can build the Voronoi diagram. Let E be a set of 200 points in the xy coordinate plane, and set B to one of the points in E. With A being the world coordinates of the vertice, we can find the Minkowski distance between A and B. We find the distance between each point B in set E, and choose the shortest one, dubbing it “minDist”.

2.2.2 Reflectivity
Water is reflective. To model this, we use a traditional cubemap reflection when determining the color of the material. Simply multiply the base color of the material (1, 1, 1, 1) by the color sampled from the cubemap. The result is used in Unity's surface shader Emission built-in value.

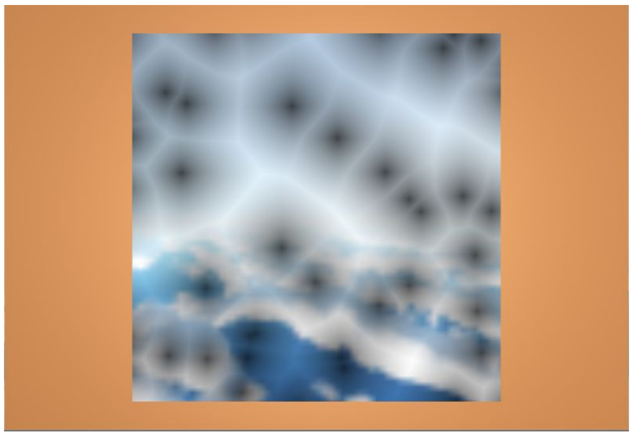
Figure 4. Minkowski Voronoi Diagram with Cubemap reflection.

2.2.3 Voronoi Diagram and Transparency.
Not all water is reflective. We use the Voronoi Diagram to set the transparency of the material. Using a log function on minDist, we can achieve a gradient of transparency with the most transparent part of the cell at the center, and the most opaque part of the cell at the edges.

$$Alpha = 0.5 * (\log(minDist + 1))$$

Figure 4. Log function for Alpha.

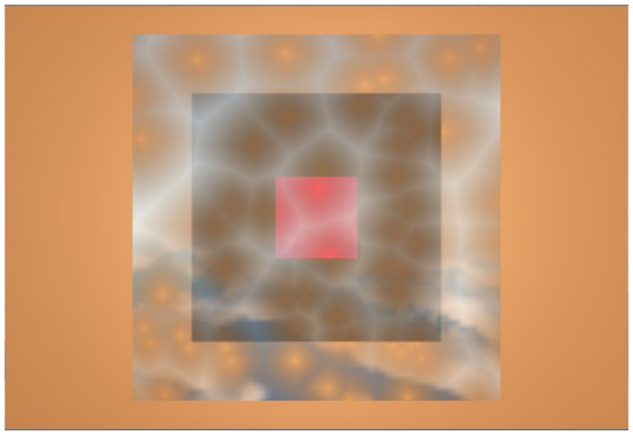
Figure 5. Cubemap reflection with Voronoi Diagram with log function influencing the alpha value.

2.2.4 Using A Noise Texture and Adding Color
This lacks the blue tint and any further distortion that moving water normally has. To solve this, we introduce an arbitrary shade of blue “blue” and a noise texture.

$$Color = blue * 0.6 * Noise * sin(2 * Time)$$

Figure 6. Formula introducing noise as a function of time.

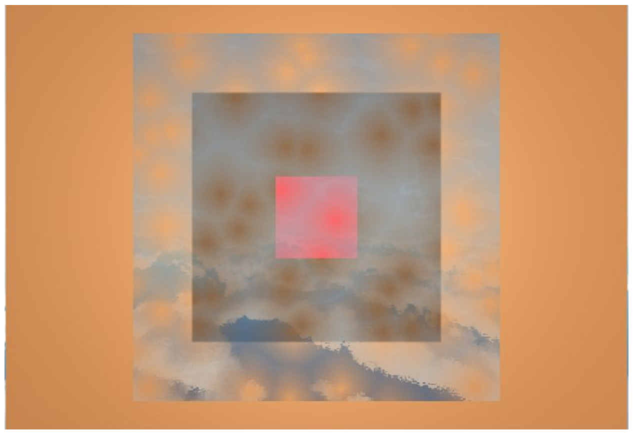
Figure 7. Using Arbitrary Blue Color Value and Noise

The result is washed out. Caustics are created by light distorted by water, and as such needs to be bright at the edges. Much like before, We can use a log function to increase the whiteness at the edges of the Voronoi cells.

$$Color = (blue + 0.75 * log(minDist + 1)) * 0.6 * Noise * sin(2 * Time)$$

Figure 8. Using Log Function to Increase Brightness

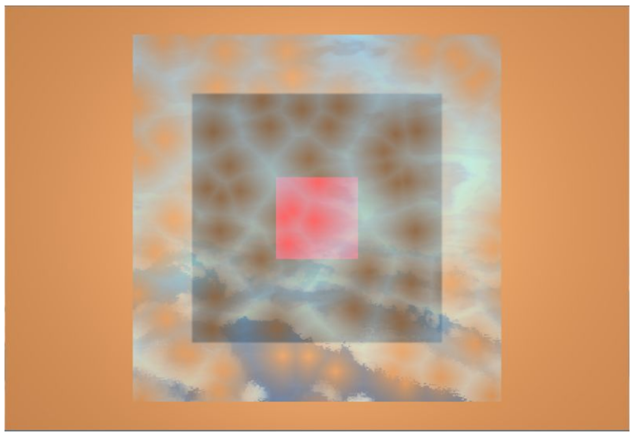

Figure 9. Final Caustics result.

2.3 CREPUSCULAR RAYS

2.3.1 Components of Crepuscular Rays.

The main components to integrate crepuscular rays into a scene is the occlusion map and a shader to blur the map outwards, creating the lights rays. For the occlusion map, all we need are the color values of each fragment to decide the if it is light or background. The blur shader requires the position of the light source within the camera's view to decide the direction of the light rays as well as samples of the neighboring pixels to create the blur effect.

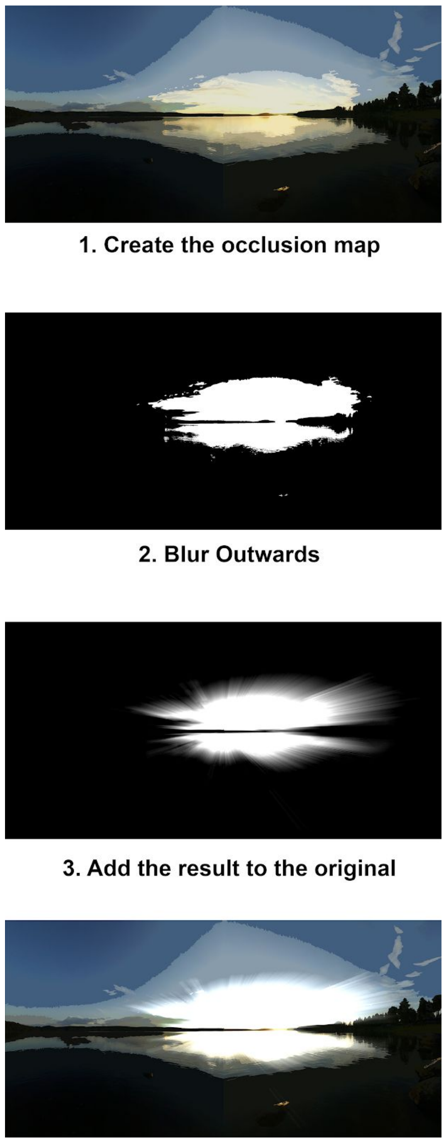
1. Create the occlusion map
2. Blur Outwards
3. Add the result to the original

Figure 10. The Process of creating God Rays

2.3.2 Creating the Occlusion Map.

Occlusion maps define the light source in white and anything that is not the light source in black. To create this map, each fragment is converted to greyscale to make the red, green, and blue values equal. If the the new color is greater than a certain threshold, the fragment is part of the light source and colored white, else the fragment is colored black. The resulting occlusion map is then passed into the shader that generates the crepuscular rays.

2.3.3 Direction Blur and God Rays.

The occlusion map is taken into the main shader and each fragment samples its surrounding pixels. The pixels being sampled are determined by the light's position, each sample coordinate moving in the direction of the light and away from its source. Each sample is them multiplied by an illumination value and a weight value. The weight value is a constant number that defines the light's strength and the illumination value is multiplied by a decay value after each sample to fade the light shafts being created. The sample is them added to the original fragment and then moves onto the next sample to be processed added onto the original fragment. This is repeat 100 times but can be repeated more to make smoother light shafts at the cost of GPU computation. At the end, the fragment is multiplied by an exposure value to increase or decrease the overall brightness of the scene.

2.3.4 Variables and DIfferences

The shader hold multiple variables that can be changed based on the context of the scene. The weight value of the shader define the strength of the light. Increasing this value will increase the intensity of the light overall. Illumination value defines how far the the light will spread. This work in conjunction with the decay value, which defines how fast the illumination value decreases as the light spreads. The sample size, as stated in 2.3.3, is how many samples are taken each fragment. It is not recommended to go any lower than 100 samples as it would not look like crepuscular rays below that value and increasing the value will decrease aliasing which can make the scene look cleaner and more natural. This cost of the GPU depends on hardware and it is safest to keep the sample size at 100. The exposure value defines the overall brightness of the entire scene. Lowering the value below 1 can create a better definition of the light shafts in the scene (i.e. using crepuscular rays in a dark scene to highlight the lighting source). Increasing the value past 1 will brighten the entire scene, obscuring the light shafts but puts an emphasis on the brightness of the context.

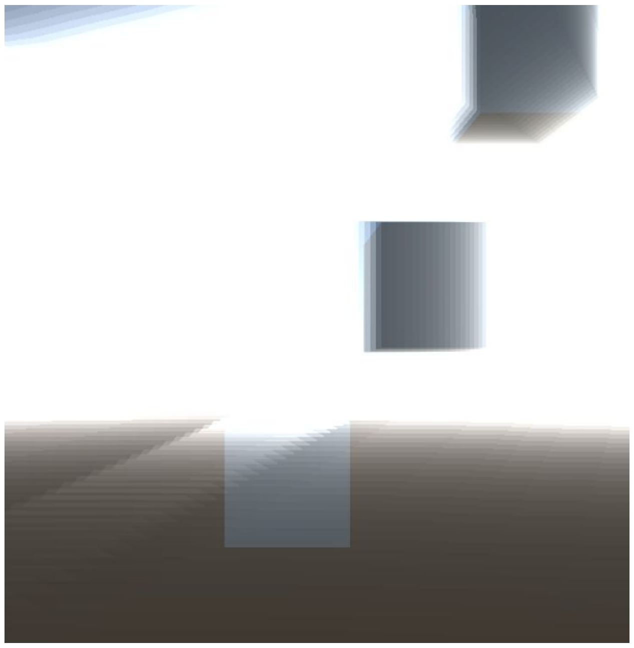

Figure 11. Crepuscular rays from light obscured by three cubes.

3 RESULTS AND DISCUSSION

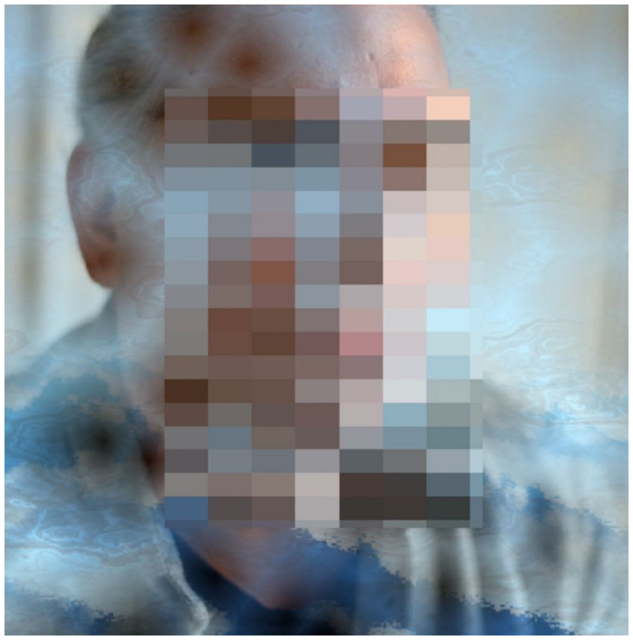

Figure 12. An early integration of the Voronoi caustics and the pixelizer.

3.1 PIXELATION RESULTS AND DISCUSSION

Due to the slicing process being a division of screen height and width the “pixels” it generates are often rectangular. Further exploration into different processes for deciding the color to sample within the 4 slices could produce results further in line with traditional pixel art if desired. Additionally an attempt to keep the “pixels” generated more square-like would help further to reach that aesthetic. For the goal of this project which was to emulate the censoring effect found in TV, restricting the output to be more in line with pixel art was not a priority.

3.2 VORONOI CAUSTICS RESULTS AND DISCUSSION

The end result resembles what one would see looking directly into water, 90 degrees from above. The effect isn’t truly scalable, and underwhelming when viewed from the sides. In the future, remedying these concerns as well as displacing vertices according to the same log function used to change the alpha value would allow for lighting models to create more accurate specularity.

3.3 CREPUSCULAR RAYS RESULTS AND DISCUSSION

The shader resulted in success, as it is both flexible and easy to implement onto any scene. There is, however, issue of it interacting with other shaders as it process the camera’s view, the render queue must be set up properly to achieve the desired effect. Due to Unity’s limitations on multiple passes in shaders, the occlusion map is being passed into the main shader through a render texture. The accuracy of the rays are severely limited as the render texture cannot update every frame, making certain rays not match up properly to the scene.

3.4 COMPILATION OF THE SCENE

We decided a Greek bath would be a nice setting to combine all of these effects together. It would provide a pool area for the caustics and pillars that would work nicely with the crepuscular rays.

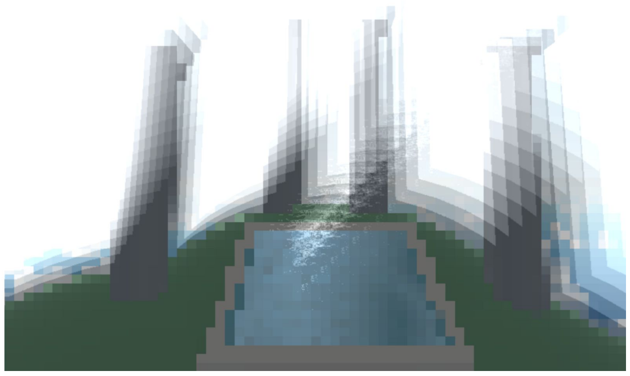

Figure 13. The final result for our compiled scenes with all three shader.

4 CONCLUSIONS

In summary, we finished what we set out to do and implemented shaders to perform pixelation, voronoi diagrams, and crepuscular rays into a scene in Unity. While Unity does fully support HLSL, many built in features that can be passed into shaders as uniforms is locked behind Unity Pro. Because of this, the project became a bit more involved, as we had to program many components from scratch rather than use the full efficiency of the engine. Additionally, the way Unity builds to WebGL flipped a few effects in strange ways that had to be accounted for.