Leviathan

Subnautic Light Dispersion, Sonar Visualization, and Visual Wavering in Unity

Victoria Lima
Computational Media
UCSC
Santa Cruz California Santa Cruz
vmlima@ucsc.edu

Buzz Tilford
Computational Media
UCSC
Santa Cruz California Santa Cruz
btilford@ucsc.edu

Gabriel McNeill
Computational Media
UCSC
Santa Cruz California Santa Cruz
gmcneill@ucsc.edu

ABSTRACT

We crafted an underwater scene that features a variety of screen and object space shaders. We use raymarching technique to simulate light diffusion in a uniform water environment and raycasting techniques to match locations on a mesh for an expanding sonar effect. We also apply a 'Wobble' effect in the screenspace of the scene to provide the feeling of constantly shifting water.

CCS CONCEPTS

• Applied computing~Media arts

KEYWORDS

raymarching, screenspace shader, Unity

1 Raymarching to Generate Light Rays

The light rays used in this project are created using the ray marching technique for volumetric rendering. The concept of ray marching was completely new to all the members on the team which led to some roadblocks. There are many different sources for learning about ray marching, however many dealt with distance aided raymarching, which did not apply for the style we wanted. We were able to discuss how raymarching works in an underwater scene with Dr. Oskar Elek. Then we were able to take the base program outlined in the Shadertoy shader Beneath The Sea by zel[4], which does not use ray marching for the light rays and apply a raymarch function to it.

The light rays in this project use a fragment shader and consist of three main parts. The fragment shader itself is straight-forward, it calls a light ray calculation function and adds the result of that function to the main texture being output to the screen. The light ray function takes the UV coordinates of the screen as an argument. Within it, we initialize a float value to 0 for our light, the beginning for all the rays, in this case, it is the camera's world space coordinates, and the direction the directions the rays will march. Then we add the results from our marching function, explained next, to the light value and clamp it between 0 and 1 before returning it. Finally, the raymarch function.

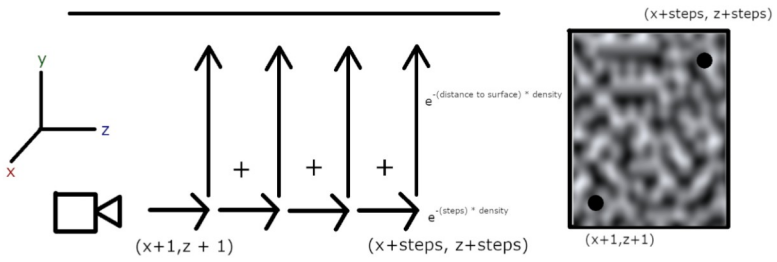

Figure 2: **Ray marches from the camera forward a certain amount of steps at each point checking the noise texture. If the light color value of the noise is above a threshold, multiply the distance diffusion from the water surface by the distance diffusion from the camera and add to the total light for the raymarching of that pixel.**

The base function used in the shader is the raymarch function. Raymarching is a way to render volumetric lighting. For each pixel on the screen, it will extend a ray from the origin position in a certain direction for a set amount of steps. For this example, at each step, we want it to sample a noise texture at the ray's x and z coordinates to determine if it would have light at that particular step in the pixel. If it does have light, it takes the distance to the surface of the water and multiplies it by the distance to the camera. We then update the ray's position and continue stepping through the loop. After finishing all the steps, it returns the color value. The function takes two float3s as arguments, the initial position of the ray and the direction the ray will march, and it returns a float value.

2 Sonar Ping Effect

To generate the 'sonar ping' (Figure 2) effect, we use four components: a C# script to take in the player input and to store the

parameters required for the possible impact points of the ‘ping’; a particle system which is place at each of the ‘ping’ impact points ejection particles off the normal and in a circle; a shader for the terrain that contains a sand texture, an animated texture for caustics, and parameters to take in three impact points and individual distance values; and a second similar shader for treasure chest objects which have a base texture but take ‘ping’ values to accomplish a variant of the sonar effect.

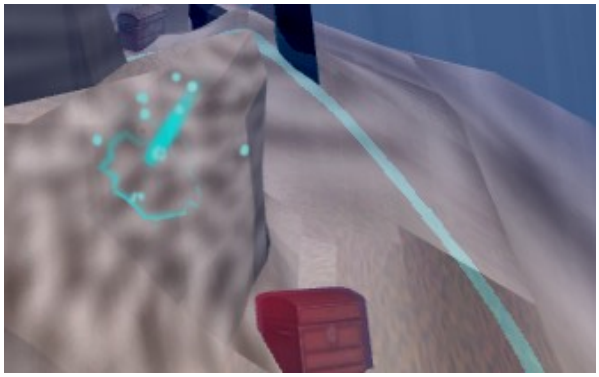

Figure 2: The final ‘sonar ping’ effect.

2.1 Particle System Components

The flat spiral which implies the expansion of the sonar’s hit is created using a rate over time emitter with a circular shape. Its mode is set to loop to make sure it outputs in steps around the circle. The particles in this component are note colored. Instead the shape seen is only visible with the teal-to-white gradient used to colors the single ribbon rendered over its lifetime. Thanks to the loop set in the emitter shape module, the ribbon neatly spirals as its guiding particle move away from its center.

The second component of the particle effect is an upwards burst, like a small vertical fire. Like the previous component, the particles are not themselves colored, but their trails are. Here trails are set per particle and fade out at the end of their lives. A vertical emission cone with no outward angle, and short particle lifespan, results in the short lived flaming cylindrical tower.

The last component is a singular burst of 30 particles emitted out of a hemisphere. These have variable speeds and lifespans and are effected by the noise module. Using a noise frequency of less them one, keeps these particles from veering too far from their expected paths, while still giving them some ‘juice’[1]. When each of these particles dies they release a simple spherical burst of 10 particles.

2.2 C# Script

The responsibilities of the C# script are as follows: perform raycasts from selected points on the camera; calculate their impact location and reflection across the normal of the mesh; cast another ray from this location; and a third if there is a second hit; pass the locations of these three possible collisions to the terrain and chest shader materials; continuously pass incrementing distance values to these shaders, slightly offset for each of the three impacts based on order; instantiate particle effects in line with the normals and locations of the impacts., scaling them down slightly for the later impacts. The most important component is the impact location and the distance value which controls the growth of the ping effect and the life of the chest glow.

2.3 Terrain and Chest Shaders

The terrain shader uses one small noise texture for sand, tinted with a color value, and a larger scaled noise texture for caustics. These caustics are transformed by moving offsetting the uv coordinates by the current time. The shader takes in three vectors for the three possible impact positions of sonar rays and three distance values.

In the terrain vertex shader we capture the world position and world normal. The normal is dotted with the y-unit vector and multiplied with the color in the fragment shader. This creates the appearance of shadowed areas with the assumption of vertical downward light.

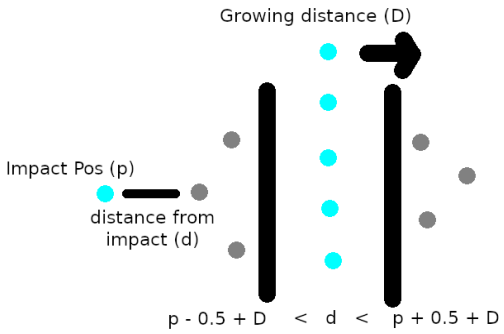

Figure 3: **Glow pixels if they are a specific distance from the impact that is covered by the range of the growing ‘distance’ value of the impact.**

In the fragment shader, after applying the shadows with the normal, we check the distance of every pixel’s world space from the locations of impacts. If this distance is within 0.5 units of distance value for that impact (a value being constantly incremented by the C# scripts), add teal to that pixel, scaling the amount added by its distance from the impact position (this cause the effect to fade with distance). It is possible for a pixel to gain teal from all three impacts via this method.

The chest shader lacks any texture, instead it was applied to a scaled up duplicate of each chest and is transparent by default. It takes in the same locations and distance values from the C#

scripts, but tests them slightly differently before adding opaque redness to the objects. It retains testing to make sure the chest pixel position is within the first line of the wave, so that it doesn't glow until the wave hits it. However it ignores the second line, losing its glow when the distance reaches an arbitrary point. The front end of the sonar wave hits it and then it glows until it suddenly stops. In theory, this should give the player time to detect its location.

3 Render Texture Screen Wobble

Creating the Screen "Wobble" effect was difficult due to the scattered information regarding it. The broad outline to achieve the effect was to create a shader that has the property, apply it to a material, create a script to make a render texture, apply the script to the camera, and then apply the material to the script.

Initially, we considered using a distortion texture map based on a unity forum post[2], but this caused some issues. When being linked to an OnRenderImage function the camera could not locate a target unless the texture was present in the scene on a 3D object, however, due to the way that unity handles external textures on it primitives the camera rendered everything upside down[3]. In addition to this, the distortion texture itself scrolled offscreen and did not wrap around the camera view.

Scrapping that we found that the next best method to create the effect would be to use a sin wave. The vertex positions and the UV for the screen will be stored in both appdata and v2f. This is done so that the vertex positions of objects on the screen and their UVs can be transformed relative to the position of the camera in the vertex shader. In the fragment shader, we took the x UV of the screen and added a sine function to it. In the sine function, we divided the provided unity time by 2 and subtracted the UV y value. We then multiplied it by our intensity slider and divided all of that by 20 to receive a smooth sine wave on the screen. When we returned this the first time we saw that the screen texture was stretching. To solve that we used a mirror method found in the first script called "Ping Pong" and "Repeat." These take the UV of the screen and set them so that should the UV be moved further inward the texture will repeat where it would have ended and will mirror itself.

ACKNOWLEDGMENTS

Thanks to Oscar Elek for sitting down with us and helping us develop a better sense of raymarching techniques.

Chest model available from the Unity Asset Store here: https://assetstore.unity.com/packages/3d/props/furniture/reinforced-wooden-chest-68139

REFERENCES

[1] https://www.youtube.com/watch?v=Fy0aCDmgnxg

[2] https://forum.unity.com/threads/horizontal-wave-distortion.295769/

[3] http://www.shaderslab.com/demo-20---distort-with-grab-pass.html

[4] https://www.shadertoy.com/view/4ljXWh