Outdoor 3D Scene

Unity 3D Version 2018.3.11f

Alex Covert
Computer Science: Game Design
UC Santa Cruz
Santa Cruz, CA
acovert@ucsc.edu

Dongbo(Bob) Liu
Art & Design: Game and Playable Media
UC Santa Cruz
Santa Cruz, CA
dliu28@ucsc.edu

Jordan Lee
Computer Science: Game Design
UC Santa Cruz
Santa Cruz, CA
jlee385@ucsc.edu

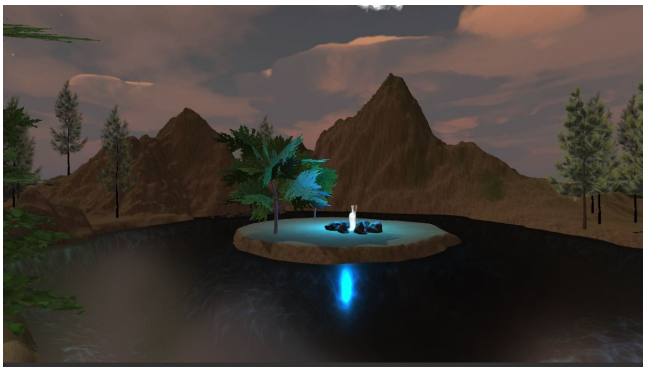
Bryan Arvizu
Computer Science: Game Design
UC Santa Cruz
Santa Cruz, CA
bdarvizu@ucsc.edu

Alexander Bradtke
Art & Design: GAmes and Playable Media
UC Santa Cruz
Santa Cruz, CA
abradtke@ucsc.edu

Samuel Wolfe
Computer Science: Game Design
UC Santa Cruz
Santa Cruz, CA
sajwolfe@ucsc.edu

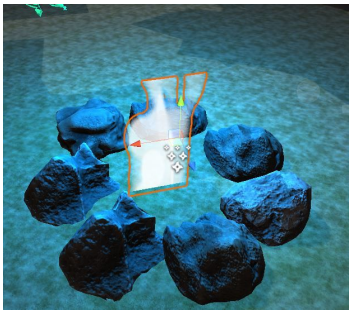
ABSTRACT

Our project aims to simulate an outdoor scene - a small island in a lake surrounded by mountains. To create this scene, we implement a variety of different visual effects that simulate outdoor experiences. The main effects we attempted to replicate were smoke, fog, water caustics, water reflection & specular highlighting, a skybox, subsurface scattering, and various particle systems.



In order to accomplish fog, smoke, caustics, and other water effects, we implemented a combination of C# scripts and HLSL shaders. Particle systems were created through the use of Unity's built-in particle systems and modified to fit our needs. In order to create a subsurface scattering effect on the tree leaves, a custom surface shader must be written that applies some lighting to the back of the object, to create the illusion that light is passing through the material.

1.1 Alex Covert - Smoke



The campfire smoke in this Unity project is created on a quad-shaped GameObject. This quad contains a few additional components: two C# scripts that generate new smoke and update the texture as well as a smoke shader that causes the smoke to slowly dissipate.

The shader itself causes the smoke to dissipate, rather than actually generating new smoke. The code works similar to how many Game of Life shaders work - it samples pixels from one overall texture on the quad. The vertex shader is relatively simple; the bulk of the work is done in the fragment shader. Nearby pixels are sampled on the main texture and, depending on the alpha value of each, updates a buffer texture with changed alpha values on each pixel. [1]

The shader is applied in a C# script - ApplySmokeShader.cs. This script does 2 things - firstly and unrelated to the shader or texture, it makes the fog quad constantly rotate to face the camera in Update. This ensures that the fog is always visible to the camera, as our camera can freely move around the scene. Secondly, the script changes the texture in Update, using Graphics.Blit() to "ping-pong" between the main texture

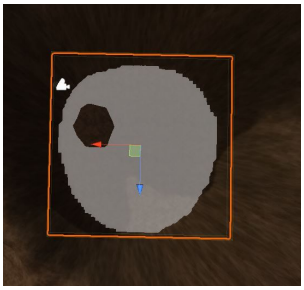
and buffer texture and actually cause the smoke to look animated.

The final component implemented is a C# script - GenerateSmoke.cs - which draws new smoke onto the texture at a given time interval. All the script actually does is handle the time interval, and calls an "AddSmoke()" function inside the ApplySmokeShader script. This function creates a random offset position for the new smoke then uses Graphics.SetRenderTarget() and Graphics.DrawTexture() to actually draw new puffs of smoke onto the quad. One difficulty that comes with the DrawTexture() function is that it only works in specific situations - at first, it seemed to be working fine when called from Update() in editor, but no textures would draw in the standalone build of the project. To fix this, you must call DrawTexture() from Unity's built-in OnGUI() function.

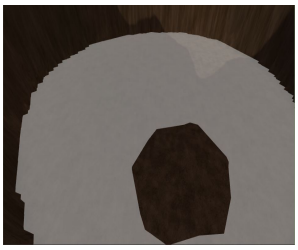
In the project, the smoke quad is located on the center island as part of the campfire.

1.2 Alex Covert - Fog

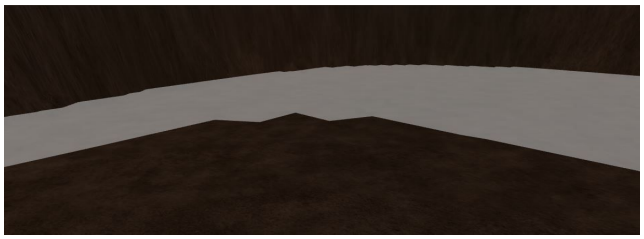
The fog in our Unity project is created using a plane object, a shader, and a few C# scripts. This was largely based off of code written by Roystan, a popular creator of Unity tutorials and resources [2]. Unlike the smoke shader, the fog shader has the bulk of its code in the vertex shader. All the fragment shader does is return a uniform "Color" variable, allowing the user to change the color and opacity of the fog to their liking. The vertex shader, on the other hand, samples a heightmap ("Displacement Texture") and updates the vertex positions based on it.



This heightmap is generated at runtime through a C# script, "Fog.cs." If nothing is interacting with the fog, all it does is slowly return to a flat plane using Mathf.Lerp(). If another object, which must have the "AffectFog.cs" script and trigger collider attached as components, collides with the fog, it calls the fog's Impact() function. Each object that affects the fog can have a given radius, which is taken in by the Impact() function to decide how much of the fog will be displaced. The "heightmap" is actually a Color array which gets applied to a texture; in order to change the heightmap, all the code does is set specific pixel colors and applies the array to a texture, which in turn updates the texture sampled by the shader. [2]

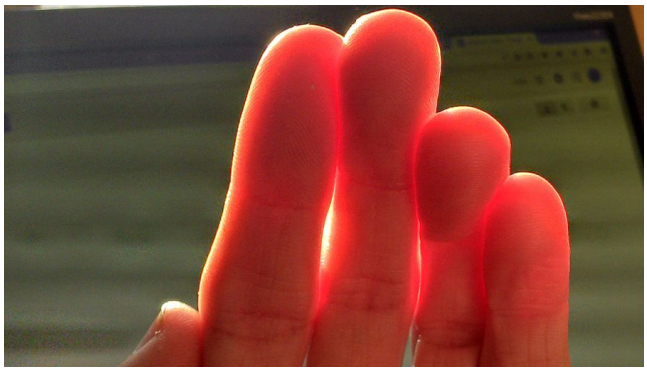


In the actual build of the project, the smoke can be found towards the corner of the terrain surrounded but mountains. It is easiest to visualize if you navigate the camera right around it - as you "walk through" the fog, you can see the fog immediately around you lower into the ground.



1.3 Bryan Arvizu - Subsurface Scattering

The subsurface scattering effect applied to the leaves on the tree within the scene was made using a custom surface shader. The shader is based off of a tutorial from Alan Zucconi, in which he dissects a talk from a 2011 GDC talk from DICE employees Colin Barré-Brisebois and Marc Bouchard, who developed a technique to emulate real-time subsurface scattering cheaply. [3]

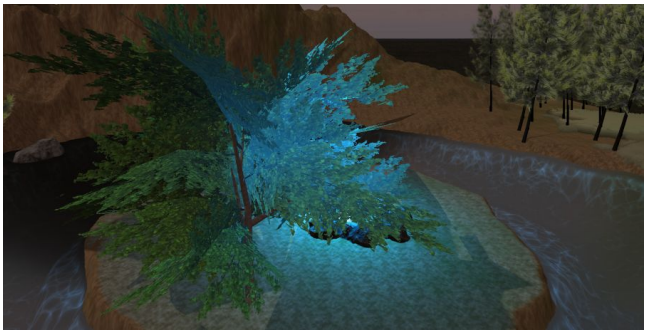


In reality, subsurface scattering is a real-world phenomenon that occurs when light particles pass through translucent objects, bounce around, then exit at another random location. To the human eye, this effect shows itself as a glow within the translucent subject. This is why covering a light source with your finger makes it appear to glow.

Simulating this effect in computer graphics, however, requires the emulation of the light particles bouncing around within an object. Additionally, since light can enter in one region and exit in another, this requires us to render all of the different faces of the

object and calculate where the light they recieve goes. This makes accurate recreation of subsurface scattering unfeasible in real-time. This is why we must use a cheap recreation in order to properly render on time; we do not have to accurately recreate the effect, we merely have to make it appear believable enough.

The real-time subsurface scattering effect that was implemented ultimately works like the standard lighting model, with the addition of having lighting affect the rear-facing triangles of the model. If you look back to phong shading, the dot product between the light vector and the surface normals is used to determine how far forward the surface is facing; a value of 1 means it is directly facing the light, and a value of -1 means it is directly facing away from the light. Using this method, we can directly identify when we should apply backlighting. Because backlighting depends on perspective, however, we must force calculate lighting using the dot product between the view vector and lighting instead when the surface face is facing away.

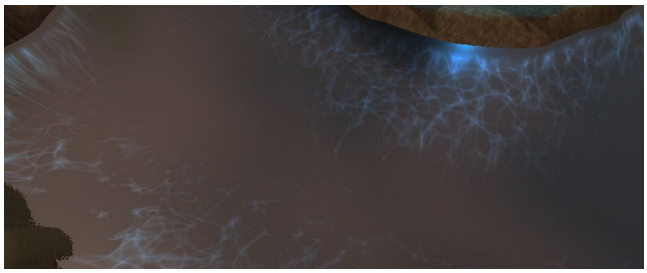


Even then, however, the surface normal must have some influence over the effect. This is where the DICE employees devised subsurface distortion, which forces the light vector closer towards the surface normal. This is done by multiplying the halfway vector between the light and the normal by a distortion value between 0 and 1, then taking the dot product of that vector with the view vector.

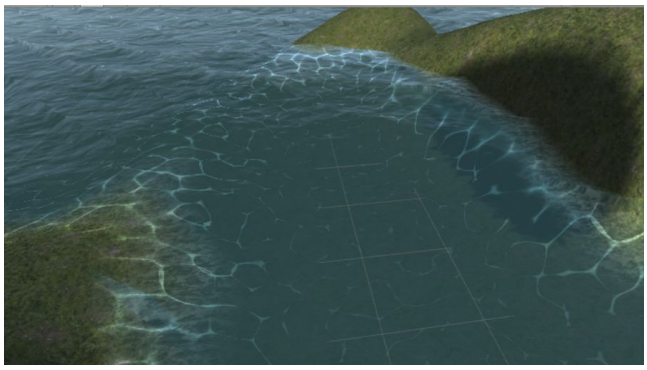
These steps should ultimately create the effect, but without taking depth into account. Accounting for depth is crucial, but can be very complex. So the simplest way to do so is to create a texture that holds the depth value in any particular area on the mesh. The sampled value from the mesh can then be multiplied with the effect to strengthen/weaken it in particular areas. While this texture does not hold accurate values for depth, as depth can vary depending on the viewing angle, it should be sufficient for fooling the viewer's eyes.

1.4 **Jordan Lee - Water Caustics**

The caustics effect for the project was created by Chris Cunningham using Unity's projector object and shader scripting. In the same fashion as a real-world projector, a material is "projected" onto the environment with adjustable settings in order to have it fit the scene. In this case, the material is a caustics texture that is animated over time using a noise texture to distort for a fairly realistic and inexpensive implementation.



Caustics refer to any light that is distorted by a curved surface in the real world, with glass objects and water being the most obvious examples. As Unity relies more on blob-like lighting, creating this effect in real-time is a challenging endeavor, hence why using the projector is a more practical solution for a similar payoff.



Such an implementation has some obvious drawbacks, such as appearing artificial or not aligning with a changing water level. One implementation that uses an array of textures simulates caustics relatively convincingly, but may come to feel unnatural regardless of the length of the animation; water and caustics are normally more erratic than the finite nature of an animation. Cunningham addresses this issue by relying on a noise texture, making the animation pseudorandom and more natural looking.

In order to fit the scene, the caustics had their color changed to match the surrounding lighting and the water level had to be set beneath the plane acting as the

reflective water, or else the caustics would be projected on top of the water as if it were a sheet of ice.

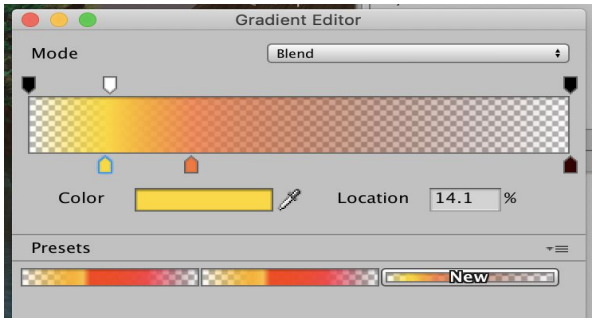
In terms of programming, the shader in question relies heavily on the noise texture. The noise texture is sampled and distorted by adding the in-built time variable and then uses lerping with the caustic texture. Finally, the quality of the render is determined by the LOD distance, using the difference between the set value and the actual view distance value to change this. Of particular note, the shader uses many variables that can be manipulated by the user to suit their needs; for example, Water Height changes the vertical position of the caustics base level or rather the “water level”, Caustic and Distortion Speeds determine how fast the caustics animate, and Height and Edge blends determine where the cutoffs for rendering the caustics to give the illusion of containment and depth.

1.5 **Dongbo(Bob) Liu - Fire Particle System & Unity scene built**

The fire flame/ember targa files are made in Photoshop by simply using the brush tool and the blur tool.

In Unity, in order to achieve a realistic fire effect, multiple particle systems are required. Besides the main parent one, there is a basic particle system for the main flame; an additional one at the second layer is for increasing the brightness for the central high light spot; the third layer is for glowing effect; and the last layer is for creating the embers (small white dots). This fire particle system is created based on Sirhaian'Arts’s Youtube tutorial.[5]

By adjusting the start lifetime, start speed, shape and noise in the particle system, we are able to change the shape, height and lifetime of the fire. By changing the “color overtime” in the particle system, we are able to change the gradient and the colors of different parts of the flame such as inner flame, outer flame, ember etc.

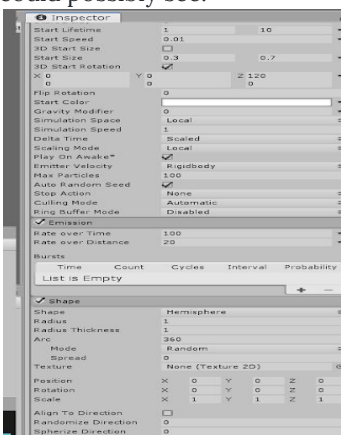


The Unity scene is created by me and Alex Bradtke together. We raised up the height of the basic terrain in order to create a pool for showing our water effects. We then created the trees, rocks and grass by using the assets we found in Unity store. There are two tree objects created on the central island particularly for showing the subsurface scattering effect. We have the cloud particle system and the skybox above the terrain. The fog is shown among some trees.

1.6 **Alexander Bradtke - Cloud ParticleSystem & Scene Design**

The smoke & fog image was used from one of the textures from a unity asset pack. This was based on the particle system tutorial for clouds and smoke by xOctoManx.[6]

In the tutorial the person made a mesh for the particle system to form onto but decided not as I was getting errors from it and used multiple particle systems to create clouds that looked believable as well as being able to adjust and change into any shape and size we want without much difficulty. I adjusted the particle system using random speed when it bursts out as well as randomizing the time it stays alive. I also had the image texture rotate at different speeds to create a more lifelike cloud. For the standard shape of the cloud i used a circle and hemisphere shape to create a cloud that you could possibly see.



The scene was created in Unity with Bob. We originally created a scene using borrowed rock assets and increasing the size. We used “se height” to increase the base, so we could dig out a place for the water shaders as the hole feature hasn't been implemented yet. We then added a grass texture across the map and then added grass models all over it and then finally trees. We also made mountains by raising the height and certain brushes to form unique forms. We made an island in the middle of the lake putting a tree and created a fire pit using rocks and a fire and smoke shader. We added my cloud particle system spread throughout the scene above the map as well as using a skybox. We finally added a fog shader placed between the trees.

REFERENCES

[1] https://www.alanzucconi.com/2016/03/09/simulate-smoke-with-shaders/

[2] https://twitter.com/roystanhonks/status/990755998280265728?lang=en

[3] https://www.alanzucconi.com/2017/08/30/fast-subsurface-scattering-1/

[4] https://www.patreon.com/posts/21987965

[5] https://www.youtube.com/watch?v=5Mw6NpSEb2o&t=319s

[6] https://www.youtube.com/watch?v=F8conwqx2kY