CMPM 163 Final Project

Eric Lanini
Computer Science
UCSC
elanini@ucsc.edu

Rachel Trail
Technology Information Management
UCSC
rtrail@ucsc.edu

Tayla Rund
Art & Design: Games & Playable Media
UCSC
trund@ucsc.edu

ABSTRACT

This project creates a daytime, outdoor scene with a terrain, sunlight, fire, and smoke. The terrain was created using a noise function, the sunlight using crepuscular rays, the fire using a particle system, and the smoke using fluid simulation.

CCS CONCEPTS

- Post-processing
- Particle System
- Perlin Noise
- Deferred Rendering

KEYWORDS

Post-processing, Particle system, Perlin Noise, Fluid Dynamics, Radial Blur

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 INTRODUCTION

The scene for this project was created using THREE.js and WebGL as its base. Each part of it was created separately and then combined into one scene that showcased each effect. The crepuscular rays were created by Tayla Rund. The smoke and fire simulation was created by Eric Lanini and Rachel Trail.

Originally, the scene was going to be set at night with moonlight instead of sunlight, but the blue sky looked better with the terrain. The smoke simulation was also more complicated than was expected, so two group members worked on that and the fire simulation together. The result can be seen in Figure 1.

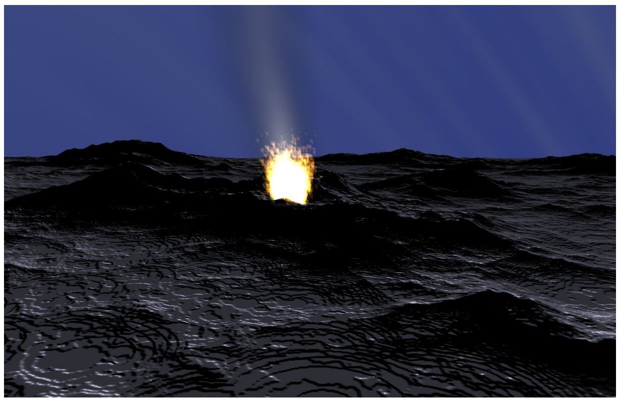
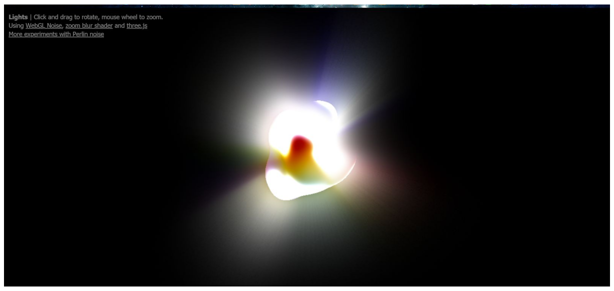


Figure 1

2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1 Crepuscular Rays

Crepuscular Rays, or God Rays, are mostly created using a post-processing blur effect, but this can be created in different ways. For some implementations, an object is the source of light, such as in Figure 2.



Lights | Click and drag to rotate, mouse wheel to zoom.
Using WebGL Noise, zoom blur shader and three.js
More experiments with Perlin noise

Figure 2

Here, the scene in rendered normally, and then a blur shader is applied and it is rendered again. This allows for the scene to be just the unblurred object, just the blur, or a combination of both. The shape of the blur is dependent on the variables applied in the shader. This isn't quite what the god rays for this scene were meant to look like, since the sun isn't actually visible, but the blur effect applied is useful for different types of god rays. The effect can be manipulated to show the original, the blurred version, or a combination of them.

When the light source that is creating the rays is visible as well as the occluding objects, god rays are created by first rendering the scene with the lightsource as white, or whatever color the color the rays are supposed to be, and everything else is colored black. This way, when a radial blur is applied to the scene, it creates rays emanating from the light source with the other objects blocking parts of the light. This scene with just the god rays is then combined with a rendering of the scene normally to create the effect of a light shining past the objects in front of it. This method is shown in Figure 3.

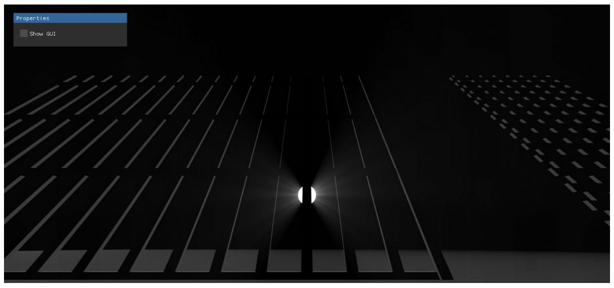


Figure 3

The god rays for this project were based off the first example, but edited to look like rays that would come from a light source that isn't visible in the scene. Like in real life instances of god rays created by the sun shining through clouds, the source isn't visible, but the rays it creates are. To get this result, a plane with a noise texture was added to the scene using the classic Perlin Noise that was introduced in class. A blur was applied to make it look like beams of light from an unseen source.

2.2 Smoke Simulation

Approximating smoke means using a fluid simulation. Stam outlines the algorithm in detail. The fluid, the smoke in this case, is separated into a grid of cells. The cells accept and give fluid to their 4 adjacent neighbors. The execution of this task is similar to that of Conway's Game of Life. The next state of a cell of smoke is determined by the current state and the states of the cells around it, using a diffusion equation.

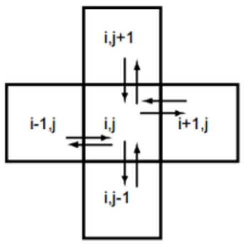


Figure 4: Through diffusion each cell exchanges density with its direct neighbors.

$$(L_{x,y},\ R_{x,y},\ D_{x,y} \text{ and } U_{x,y})$$

$$F_{x,y}^{in+out} = f \cdot \begin{bmatrix} F_{x+1,y}L_{x+1,y} + F_{x,y+1}D_{x,y+1} + F_{x-1,y}R_{x-1,y} + F_{x,y-1}U_{x,y-1} + \\ +F_{x,y}\left(L_{x,y} + R_{x,y} + D_{x,y} + U_{x,y}\right) \end{bmatrix}$$

Figure 4

The equation above outlines the diffusion equation. Making the smoke diffuse upward was implemented by adding more weight to the contribution by the cell below current. Some sway was added to the smoke by calculating diffusion both to the right and left using the above equation, and linearly interpolating that based on the current time. The coefficients for each of the above effects were determined through trial and error.

An issue that had to be dealt with was lack of precision. Some negative or positive number is added to the current cell value. Due to floating point number issues, the value being added can become small enough (but still nonzero) that the cell value is unaffected by the addition. Therefore, some cells cannot get down to a cell value of 0. A minimum value must be used which is large enough that it affects the cell value and small enough that it does not look like a large jump.

This is all done on a simple plane, using buffer swapping. In order to integrate this into the final assignment, the output buffer is taken and used as a texture on a plane and added to the final scene. Obviously it had to be translucent, but not fully transparent - this was done by setting the alpha to the smoke value. More smoke in a cell meant more opaque, less meant less opaque. Anything that isn't smoke has an alpha of 0. As well, a source was necessary for the smoke - if the initial texture was all black, there would never be any smoke. Since this is coming from a fire, the smoke should be continuous from the same point. Therefore, using gl_FragCoord, if the current fragment is at the center of the screen the color is set to 1.0. The smoke then flows out of this point.

2.3 Fire Simulation

A particle system was used to implement this simulation with multiple GUI options. A particle system begins with an initial emitter, which acts as the source of the particles. The emitter is essentially the source of creation for all of the point sprites and its location determines where are the points go in the 3D scene. The emitter can be manipulated by being connected to sets of parameters which determine how the particle system will look. This is where the GUI options are used to change the look of the fire. The GUI options allow the user to change the velocity randomness, position randomness, size, size randomness, color randomness, lifetime, turbulence, spawn rate, and time scale of each sprite in the fire scene. Once these options are changed, the particle system is updated in real time. This means that each sprite's state in the system will be saved to incrementally update for next iteration of rendering.

The fire simulation was stuck on a specific green/yellow color because there was a bug with the given GPUParticleSystem.js file from CMPM 163. A new GPUParticleSystem.js had to be downloaded online from Charlie Hoey for a successful output. This new js file allowed illumination to automatically be added to the center of the fire to look more realistic. To maximize efficiency, a smaller PNG file was added as the flames' texture. This simulation was easily added to the final scene as a vertex and fragment shader with deferred rendering in order to update the GUI inputs in real time.

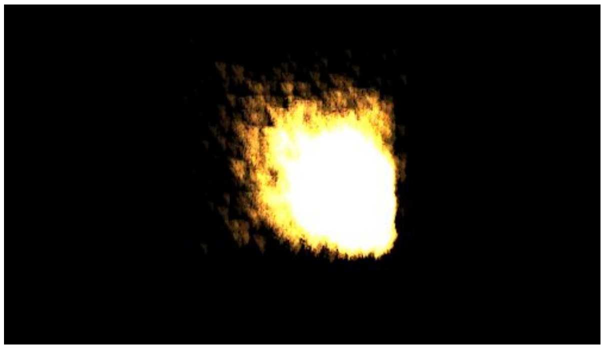


Figure 5

2.4 Terrain

The terrain was created using concepts similar to the 'elevated' demo by Iñigo Quilez. Fractal Brownian Motion (FBM) in combination with perlin noise is used as height of a plane. The height was written out to an image, then read in and the texels in each four directions are used to calculate the normal of the current texel. This data is then saved as a normal map. In the final terrain code, both heightmap and normalmap are read in, and the heightmap is used for vertex displacement in the vertex shader and the normal map is used for lighting in the fragment shader.

One thing that FBM includes as a parameter is octaves, or amount of iterations. The fewer the octaves, the smoother the output. The heightmap and normalmap were output again, but at fewer octaves. Iñigo Quilez realized that the normal from the smooth noise at some location, approximates the fact that that area is in shadow. If the regular normal and smooth normal are in the same direction, it is not in shadow. So that is used as well for lighting.

For integration into the final project, this is simply a plane mesh added to the scene.

2.5 Deferred rendering

The concept for deferred rendering is fairly straightforward. It requires two passes for a full render. The first pass renders all geometry, without lighting it, writing the position, normal, and color into separate buffers. The second pass uses these buffers as input, calculating lighting only for each pixel on the screen, rather than per fragment of geometry. With many lights and objects, performance increase is expected.

Executing deferred rendering directly in WebGL is fairly straightforward as one has access directly to all framebuffers, textures, drawing, etc. However, once research began, it was realized that using deferred rendering in the context of three.js is difficult if not impossible without modifying three.js internally. Because the effects developed in the project use three.js, using deferred rendering was out of the question and had to be dropped.

3 RESULTS AND DISCUSSION

3.1 Crepuscular Rays

The crepuscular rays created for this project simulate how rays would look shining through clouds. For the final scene, the colors of the texture used to create the rays was changes from black and white to blue and white so that they would look more like sunbeams.

In order to make the rays look like they were formed by shining through clouds, they needed to be in the background of the scene and not overlay the fire, smoke, and terrain. To do this, they were rendered to an off screen buffer and made into a texture that was applied to a sky box. This makes the effect more of a background element of the scene, which fits with what god rays are.

Figure 6

Although this does make the god rays less changeable in the scene, it allows them to create the sense of light and a light source in the scene without having fog or any actual light.

3.2 Smoke Simulation

The smoke simulation was relatively effective. The translucency technique used made the smoke more authentic, with the edge a dark gray and the inside a bright white. Randomness would have been a welcome addition, so that the smoke does not always have the same movement, but as it stands, it does look like good fake smoke. This effect could definitely be used with some modifications for a stream of water or other liquids, but it would be nowhere near accurate.

3.3 Fire Simulation

This effect looked quite realistic since illumination and some turbulence was added. Also, a blur effect between particles was implemented in the new GPU Particle System. As seen below, there is a large difference in how much more realistic the new fire simulation looks, as opposed to a simple, yellow particle system.

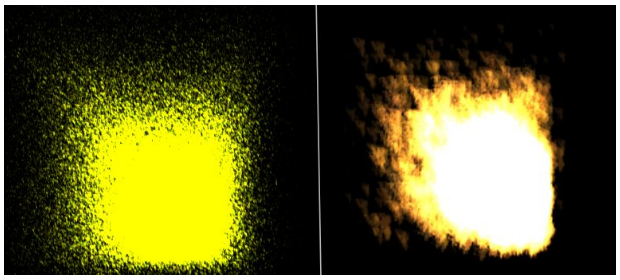


Figure 7

3.4 Terrain

The terrain and fake shadow generation produced fairly high detail results without needing many vertices. However, it was a harsh terrain in some areas with stiff edges. Taking samples of heights further away as well higher sample density might have fixed this.

4 CONCLUSIONS

In order to successfully combine all processes, the main problem faced was that the crepuscular rays utilized a post processing blur effect. When combining all elements, the effect from the God Rays blurred all other objects in the scene. To solve this, a Buffer to Texture had to be implemented in order to apply the post processing effect just to the God Ray Frame Buffer, and apply the texture as a plane.

Another time consuming problem faced was that the smoke created is 2D. In order to enable camera movement through the scene, the smoke had to rotate with the camera, without cutting the fire simulation off, and instead to slice through it. To solve the problem, getWorldPosition() had to be used in order to update the smoke's position based on the camera to ensure the correct side was always facing the camera.

5 REFERENCES

https://gamedevelopment.tutsplus.com/tutorials/how-to-write-a-smoke-shader--cms-25587
https://www.alanzucconi.com/2016/03/09/simulate-smoke-with-shaders/
https://creativecoding.soe.ucsc.edu/courses/cmpm163/
http://www.iquilezles.org/www/material/function2009/function2009.htm
http://bkcore.com/blog/3d/webgl-three-js-volumetric-light-godrays.html
https://www.clicktorelease.com/code/perlin/lights.html
https://github.com/Erkaman/glsl-godrays
http://charliehoey.com
http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf

---

*Produces the permission block, and copyright information
[†]The full version of the author's guide is available as acmart.pdf document
[1]It is a datatype.