MiniBay Team 59

SAGE AGUINA-KANG, EDWARD GILMORE, SEBASTIAN MEJIA, and JOVIN VALDEZ

MiniBay is a real-time web application inspired by Mini Tokyo 3D that simulates real world effects, projected onto a 3D map of the Bay Area. Our main goal was to create an immersive and informative experience by integrating dynamic environmental effects such as clouds within our 3D visualization. We use a tech stack that includes Three.js for 3D rendering, React for UI management, and MapBox for geospatial mapping. Here you can view our live webpage TODO!!, final video TODO!!, and project proposal.

1 Setting Up the Mapbox Scene

Before we designed clouds that can interact with several objects, we needed to figure out how to import a single object. It would have to fit in with the scene's coordinates, and it should be able to receive shaders from .frag and .vert files. We set up the scene using a custom layer built with Three.js. First, we convert the incoming object's geographic coordinates into Mercator coordinates, using a process similar to the coordinate conversions in our homework. This was done using MercatorCoordinate.fromLngLat(). In the onAdd() method, we set up a basic Three.js scene with a camera, ambient lighting, and directional lighting. Then, we load the object from a URL by constructing a GTFLoader(). Finally, we create WebGLRenderer() that is linked to the Mapbox scene. We update the projectionMatrix every frame, which ensures that the object is properly aligned. Finally, we call addLayer() to add this layer to the map. This lets us see the object in the Mapbox scene, tying it to geographic coordinates. Additionally, all imported objects are compatible with transformation matrices for translation, rotation, and scaling. These matrices invert the y-axis, because Three.js and Mapbox have different coordinate conventions for the y-axis.

2 Applying Custom Shaders

To apply custom shaders onto our object, we first implemented Phong lighting using WebGL. This involved writing a vertex shader file to compute the screen positions of vertices and a fragment shader file that computes the final color for each pixel based on the Phong model. We made a custom material with these shaders, passing in the appropriate uniforms like the view projection matrix and lighting properties. For the vertex shader, we take the vertex's original position in the model's local space and transform it using matrix uniforms to calculate the position and surface normal vector in world coordinates (Mapbox's Mercator coordinates), passing them to the fragment shader as varyings. The fragment shader then takes the varying variables and computes the lighting components needed for Phong shading: ambient, diffuse, and specular lighting, combining to produce a final pixel color. We referenced our Blinn-Phong implementation in HW4 to help implement this. With this custom material we can replace the default material with our own custom one, making sure to update the uniforms needed by the shaders in the render function. Afterwards, we updated the GUI to include

2.1 Cel Shading

To mimic the toony effect of cel shading seen in games such as The Legend of Zelda: The Wind Waker, we started with Blinn-Phong as our underlying shading. We normalized the world-space normal, and computed light direction with the light position and camera position. Afterwards, we calculate the diffuse lighting intensity as the dot product between the normal and the light direction. Finally, we create four bands of lighting: brightest, medium, dark, and darkest. We modulate the base color to one of these bands, based on the intensity computed.

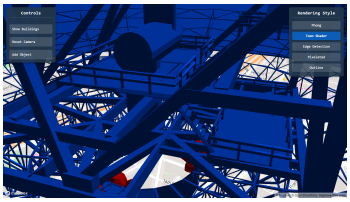

Fig. 1. Cel shading.

2.2 Edge Detection Shading

We implemented edge detection shading to clearly highlight . First, we compute standard diffuse and ambient lighting with the surface normal, light direction, and view direction from the camera. Then, we sample the screen texture at four neighboring pixels around the current fragment: top-left, top-right, bottom-left, and bottom-right. We use the Roberts Cross operator to obtain image-space gradients between the diagonally adjacent top-right and bottom-left pixels, as well as the top-left and bottom-right pixels. This lets us determine the abruptness of color change, which correlates to visible edges. We then feed the result into smoothstep() to create a smooth threshold near the edge strength. This final value is used to linearly interpolate between the base color, and the edge color (black).

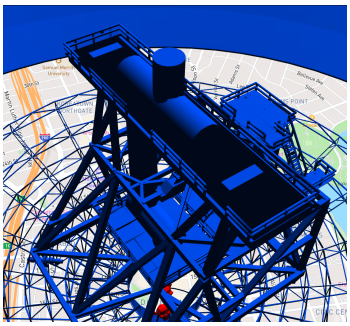

Fig. 2. Edge detection shading.

2.3 Outline Shading

Another way to emphasize silhouettes and curvature is through outline shading. We take the angle between the surface normal and the view vector, since their dot product provides insights for view-dependent edges. If a surface is mostly perpendicular to the view prediction, it will produce an outline. We then feed this dot product into smoothstep() to provide the range of fragments where the surface is mostly perpendicular to the camera. This range corresponds to contours, which we interpret as outliers. We proceed to linearly interpolate the outline color when the dot product is small, mixing it with the outline color (black).

2.4 Pixel Shading

After calculating realistic Blinn-Phong lighting, this shader converts the fragment's position into screen coordinate space of [0, 1]. Pixelation is applied by assigning screen coordinates to a grid of fixed pixel size (set to 10.0). After dividing by the pixel size, we apply the floor function to get a discrete integer. Afterwards, we multiply it by the same

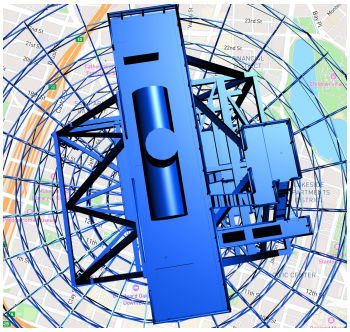

Fig. 3. Outline shading.

pixel size, restoring the original scale. The texture loses detail as many fragments are mapped to the same grid cell, creating a retro pixelated effect.

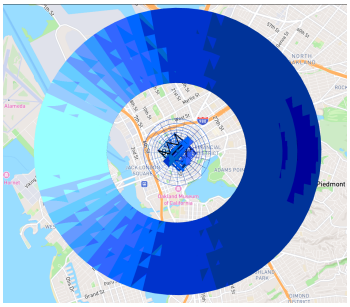

Fig. 4. Pixelated shading.

3 Backing Data Management

To properly load and manage multiple 3D objects in our scene, we used a dictionary to track objects. Since light simulation is computationally heavy, we wanted to reduce the workload by using a dictionary which has constant-time lookup and insert of objects. Each model is initialized in the dictionary with a unique identifier, named "BART-i" as the ith item of the list. Each entry contains the 3D model instance, material applied to the model, position/altitude/rotation/scale, and its ID. Additionally, we have a 2D dictionary of shaders that can obtain a shader by its name, then look up its vertex and fragment shaders.

4 Cloud

Rather than implementing clouds as a post-processing shader, we render it as a transparent three-dimensional object. This approach allows other objects to pass through a cloud, overlaying its effects on top of the objects' post-processing shaders.

4.1 Initial Approach: Bounding Box

Originally, we implemented a bounding box as the volume containing the cloud. We checked for box intersection by computing the ray's entry and exit points across all axes, using the inverse direction. We then determine tNear and tFar as the last plane that the ray enters, and the first plane that the ray exits. This implementation is theoretically more efficient, as it does not involve solving a quadratic equation.

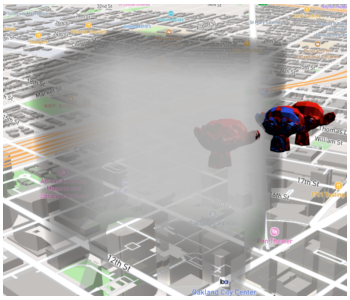

Fig. 5. Our bounding box with an early, rudimentary implementation of a cloud.

4.2 Bounding Sphere

We ultimately scrapped the idea of a bounding box in favor of a sphere, which produced better results visually. Although its Big O time complexity is theoretically slower, we found that it had a similar runtime on our devices in practice. This approach calculates the discriminant as b^2 - c, then uses its roots to find tNear and tFar. In the case of both shapes, we define the bounding volume's center, altitude, and dimensions by meters in geographic coordinates. We convert these to Mercator coordinates for rendering. The cloud model is assigned a model matrix, as well as an inverse model matrix which allows us to easily check if a point p (in world space during ray marching) is inside our cloud box, by transforming p into the box's world coordinate system. As a result, we made function scene(p) that returns the density sample at point p.

Fig. 6. Bounding spheres with volumetric clouds added in it.

4.3 Volumetric Ray Marching

In class we were introduced to ray tracing as a rendering technique to simulate light rays' behavior, which generates realistic lighting in images. This was feasible for Homework 3 because it only involved rendering small scenes. However, traditional ray tracing is too computationally expensive for the scope of our project, since its performance is not well-suited for real-time rendering on a web browser. To visualize the volumetric behavior of clouds over the Bay Area, we make use of ray marching. Ray marching discretizes the ray traversal which reduces physical accuracy, but provides

realistic results with reasonable performance requirements. We compute the entry and exit points of each ray within the bounding sphere, determining the ray's path through the sphere. Rather than perfectly solving the rendering question, we march along the ray with uniformly sized steps (0.02). At each step, we sample the cloud density with scene(p). If the density is greater than zero, we are inside of the sphere and proceed with lighting calculations. We compute color by multiplying the base cloud color (white) with the sampled density's opacity. Front-to-back alpha blending is used to accumulate the contribution of each step:

$$C_{result} = C_{result} + C_{current} \cdot (1 - \alpha_{result})$$
$$\alpha_{result} = \alpha_{result} + \alpha_{current} \cdot (1 - \alpha_{result})$$

In which C_{result} is the accumulated color, α_{result} is the accumulated opacity, $C_{current}$ is the color contribution in the current step, and $\alpha_{current}$ is the opacity at the current step.

4.4 Density Function

The scene function calculates density of a given point in the volume with scene(p) for signed distance and Fractal Brownian Motion (FBM) noise. FBM creates natural patterns in our cloud by layering noise across multiple scales, mimicking the fluffy appearance of clouds. We implemented FBM by summing multiple (set to 4) noise functions, known as octaves, where each octave has a higher frequency and smaller amplitude than the previous octave. We use the formula:

$$f(p) = \sum_{i=0}^{n-1} a^i \cdot \text{noise}(b^i \cdot p)$$

Where a is the persistence (how fast the amplitude decreases), b is the lacunarity (how quickly the frequency increases), and n is the number of octaves. We defined the spatial frequency, noise strength, and cloud opacity to values that produced clouds we though looked best. In addition to FBM, we combined our density function with signed distance functions (SDFs). The scene() function establishes a base density using the point's distance from the center, effectively creating a density field that's positive inside the sphere. This base shape is then distorted by adding multi-octave FBM noise scaled by the noise strength parameter, which introduces turbulent details and makes the boundary irregular and fluffy. To avoid a hard spherical edge, an additional falloff is applied using smoothstep based on the distance from the center. This smoothly reduces the density to zero between defined start and end falloff radii near the edge. The final density is the result of the noise-modulated shape multiplied by this smooth falloff factor, creating a soft, cloud-like appearance instead of a sharply defined sphere.

4.5 Temporal Animation

Rather than rendering a static image of a cloud, we manage elapsed time (in seconds) to re-render the cloud each frame. This is an integral part of our Fractal Brownian Motion implementation, which offsets the initial position by moving along the direction vector vector (1.0, -0.2, -1.0). We adjusted the speed by multiplying the time value by 0.1, to accommodate for a cloud's naturally slow movement in real life. Then, blue noise dithering is incorporated to reduce the artifacts of 3D shapes/planes.

```
float blueNoiseVal = texture2D(uBlueNoise, blueNoiseCoord).r;
float temporalOffset = fract(frameMod32 / sqrt(0.5));
```

```
float finalOffset = fract(blueNoiseVal + temporalOffset);
depth += step_size * finalOffset;
```

4.6 Lighting

A lighting model is implemented during the ray marching loop. At each point p accessed in the loop, we obtain p's direction to the light source. To estimate how much light has reached p, we sample the density of another point slightly offset from p in the light source's direction. If the offset point has a higher density than p, then the cloud is blocking light which means p should have a weaker light intensity. This calculation combines ambient light and directional sunlight, giving a self-shadowing technique based on cloud thickness that adds to the cloud's 3D appearance.

```
vec3 lightDir = normalize(u_lightWorldPos - p);
float density_along_light = scene(p + lightDir * light_offset); // uses modified scene
float diffuse = clamp((density - density_along_light) / light_offset, 0.0, 1.0);
diffuse = pow(diffuse, 1.5); // optional contrast
vec3 lin = AMBIENT_LIGHT_COLOR + DIR_LIGHT_COLOR * diffuse * DIR_LIGHT_INTENSITY;
```

5 Conclusion

Ultimately, our combined techniques of ray marching through procedually generated clouds and post-processing shaders have produced an atmospheric rendering that allows real-world scenes to interact with clouds in real time. We can now model clouds passing through the Bay Area with fluffy textures and realistic lightings, in addition to stylized renderings of the background scene.