MiniBART Project Milestone

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

MiniBART is a real-time web application inspired by Mini Tokyo 3D that aims to display a 3D map of the Bay Area, combining real-time data with interactive graphics to simulate the movement of BART trains. Our main goal is to create an immersive and informative experience by integrating dynamic environmental effects like fog within our 3D visualization. Currently, we use a tech stack that includes Threejs for 3D rendering, React for UI management, and MapBox for geospatial mapping. We've been making steady progress on the overall components of the project and are aiming to have a working prototype soon, which will include fog effects rendered over specific areas of the map. Here you can view our slides presentation and video.

1 Adding 3D Object to Mapbox World

Our final goal is to apply fog shading to the entire scene, interacting with several objects. However, for our initial milestone, we decided to start small by applying shaders to a 3D object: a simple cube model. This was implemented using a custom layer built with Three.js. First, we converted the geographic coordinates into Mercator coordinates, using a process similar to the coordinate conversions in our homework. We successfully tested our cube on transformations for translation, rotation, and scaling, which may be applicable in the future as we continue working with 3D objects. In the onAdd() method, we set up a basic Three.js scene with a camera, ambient lighting, and directional lighting. Then, we load the cube 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 cube is properly aligned. Finally, we call addLayer() to add this layer to the map. This lets us see the cube in the Mapbox scene, tying it to geographic coordinates.

2 Applying Custom Shaders

To apply custom shaders onto our cube, we 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.

3 Next Steps

Now that we have our MapBox environment set up with functional 3D models and shaders, our next primary goal is to write and implement our fog shader. We'll render the fog via .frag and .vert shaders, using ray-marching as a cost-effective alternative to real-time path-tracing. We'll test its success by applying it to a scene containing the 3D object we've defined. Once our fog has been implemented, we'll re-explore our transformation of 3D objects to simulate vehicles passing through the fog (representing the movement of Bay Area Transit vehicles).