HydroGL

Summary

Building a realistic 3D scene is a crucial data visualization method for the hydrology field. By using HydroGL, researchers can apply their data to the framework, and adjust the scene using the client-side web system while customizing different options. HydroGL is a powerful framework for presenting hydrological knowledge and potentially increasing public awareness of certain problems.

Design Goal

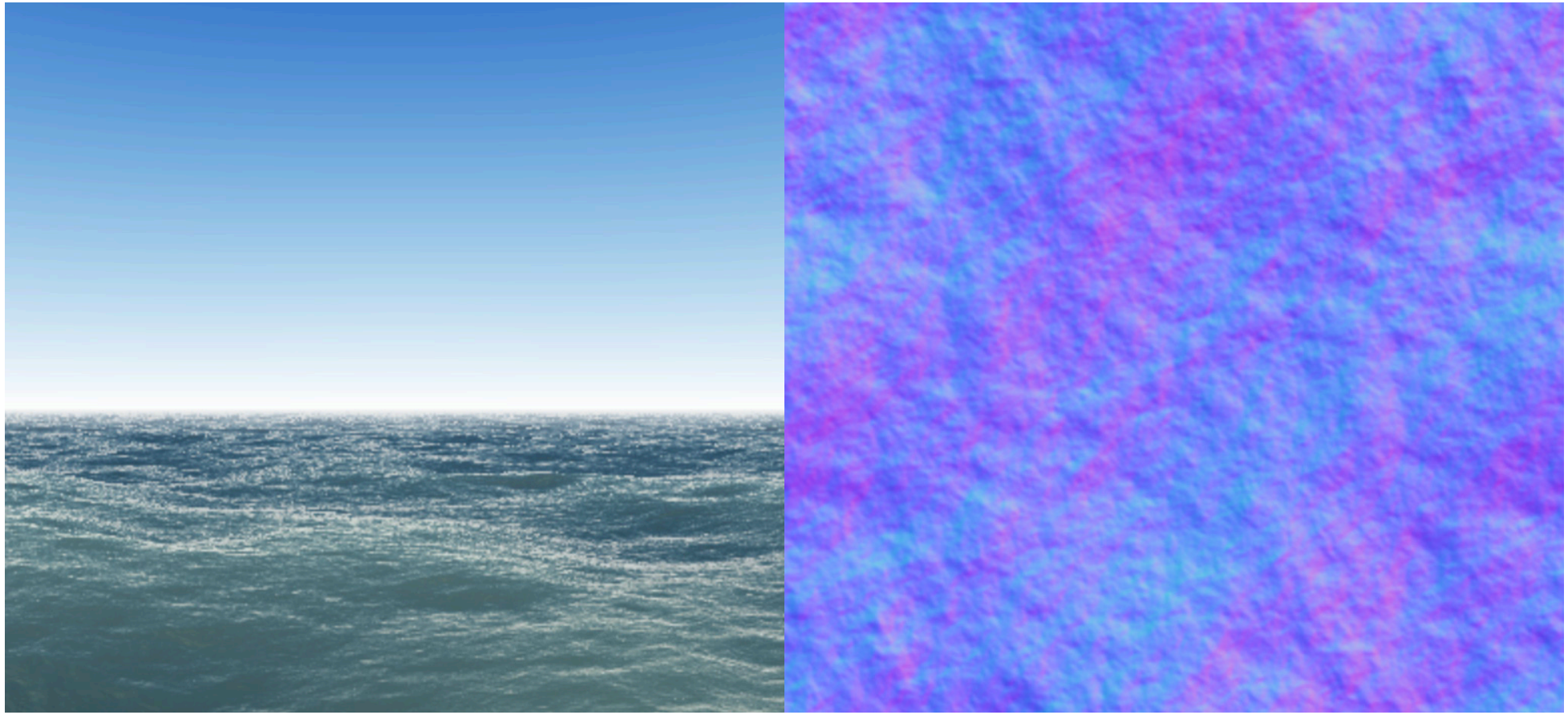
This project aims to incorporate Google Map API and WebGL to create a library to customize realistic water scenes based on hydrological data so that users can create a 3D scene efficiently and effectively. In addition, we want to introduce a form of hydrological data visualization that can be more interpretable to the public, which directly enhances public awareness.

Method

Javascript, WebGL, Google Map API

WATER SHADER

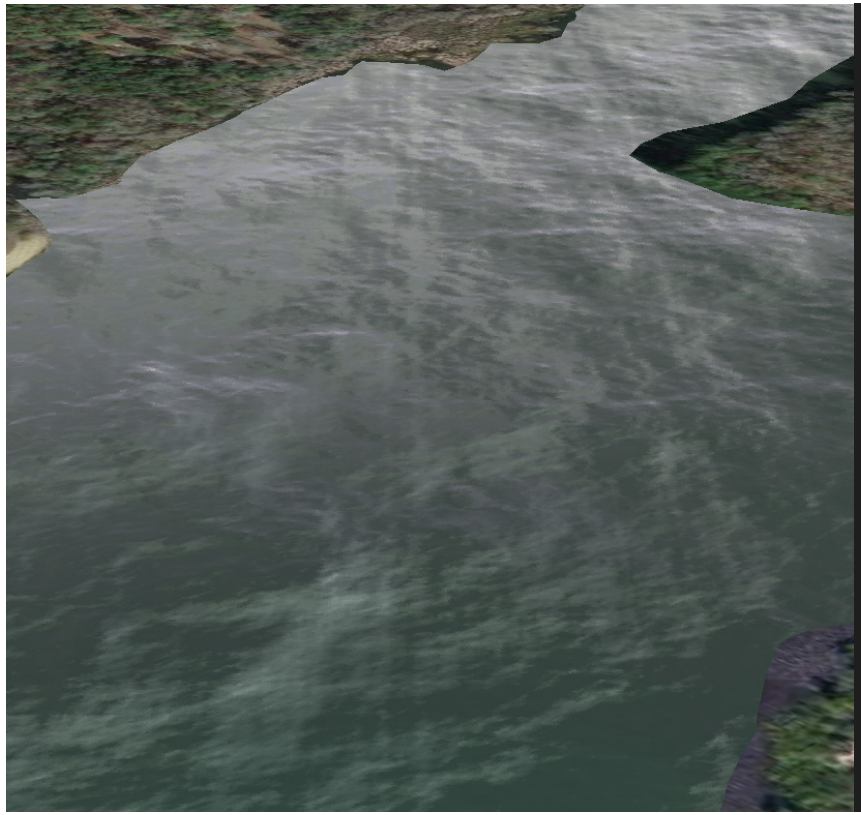
Realistic water is one of the core functions of HydroGL. It is achieved by using WebGL shader. We used a normal map to simulate the wave effects. A normal map is often an RGB-image in which each pixel represents the normal coordinate of that pixel. The normal of a vertice is the vector that is perpendicular to it. With given normal coordinate that simulates the movement and shape of wave, we can manipulate the water with simple lighting effect called Phong lighting. Phong lighting is an algorithm for simulating lighting effect on an object, where it consists of ambient light, diffuse light, and specular light. Diffuse light and specular light are determined by the normals.



WATER SHADER

The code structure of WebGL water shader consists of two parts: vertex shader and fragment shader. Vertex shader is the shader that controls every vertex of the geometry. Usually, to create a realistic water shader, a wave-like movement of vertices is required. However, because this project is web-based with limited computation resources and users barely see the geometric movement from a far point of view, we do not modify the vertices position.

Fragment shader is where we assign the normal coordinate of each pixel on the plane. Firstly, we map the normal texture onto the plane. Then, we use Phong lighting algorithm to get the realistic lighting effect. As a result, a realistic and animated water plane is created.



```
void main() {
    #include <logdepthbuf_fragment>
    vec4 noise = getNoise( worldPosition.xy * size );
    vec3 surfaceNormal = normalize( noise.xyz * vec3( 1.5, 1.0, 1.5 ) );
    vec3 diffuseLight = vec3(0.0);
    vec3 specularLight = vec3(0.0);
    vec3 worldToEye = eye - worldPosition.xyz;
    vec3 eyeDirection = normalize( worldToEye );
    sunLight( surfaceNormal, eyeDirection, diffuseLight, specularLight );
    float distance = length(worldToEye);
    vec2 distortion = surfaceNormal.xz * ( 0.001 + 1.0 / distance ) * distortionScale;
    vec3 reflectionSample = vec3(1.0,1.0,1.0);
    float theta = max( dot( eyeDirection, surfaceNormal ), 0.0 );
    float reflectance = rf0 + ( 1.0 - rf0 ) * pow( ( 1.0 - theta ), 5.0 );

    //vec3 worldNormal = inverseTransformDirection( vec3(0.0,0.0,1.0), viewMatrix );
    vec3 _eye = vec3(-10374890.885264954, 3843690.34055224, 1250.56563388561295);
    vec3 worldNormal = normalize(normalize(_eye-worldPosition.xyz)+surfaceNormal);
    vec4 envMapColor = textureCube( envMap, worldNormal.xzy);
    vec3 envColor = PI * envMapColor.rgb/ 3.;
    vec3 scatter = max( 0.0, dot( surfaceNormal, eyeDirection ) ) * waterColor;
    vec3 albedo = mix( ( sunColor * diffuseLight * 0.3 + scatter ),
                       ( vec3( 0.1 ) + reflectionSample * 0.9 + reflectionSample * spe
    vec3 outgoingLight = mix(albedo, envMapColor.xyz, reflectance);

    gl_FragColor = vec4( outgoingLight, alpha );

}
```

WATER GEOMETRY

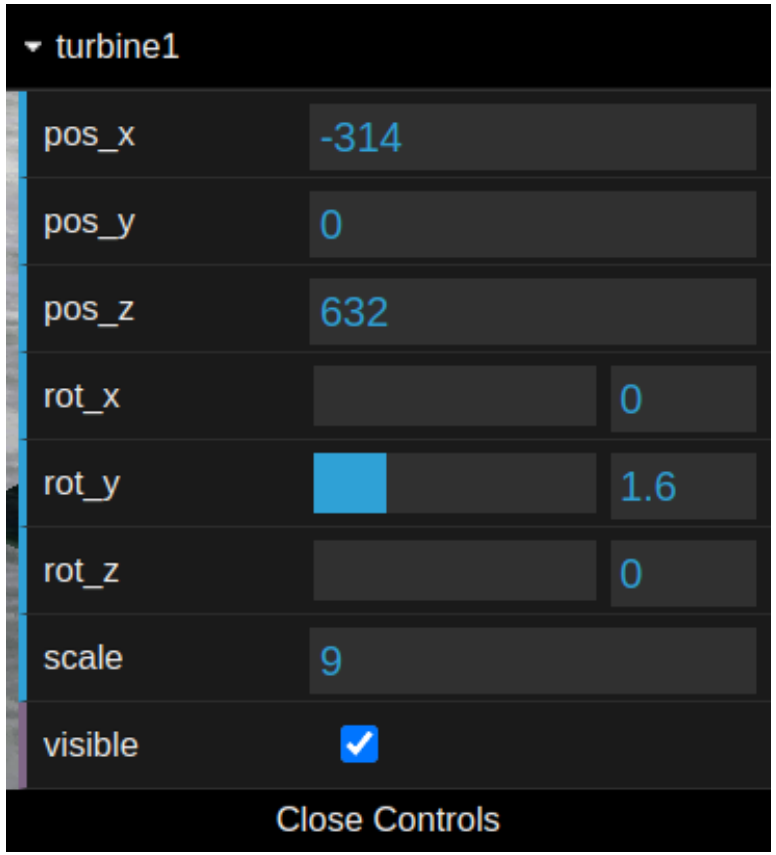
The outline of the lake is drawn by a map-to-polygon tool, where it can translate the river outline into coordinates of latitude and longitude. However, we need to further process the geometry since it only contains the outline, but not as a plane mesh, which means it does not contain vertices inside the plane. Thus, we applied Delaunay triangulation, which is to fill the inside part of the plane with triangles. The wireframe of the lake is shown as the figure below

REALISTIC SCENE

To create a realistic scene, we need to add more objects onto the lake such as houses, wind turbines, etc.. With ThreeJS, we can input these models with transformations. Combining realistic water effect and real-world model, the 3D visualization of hydrolic data becomes more attractive to the pulic.

USER INTERFACE

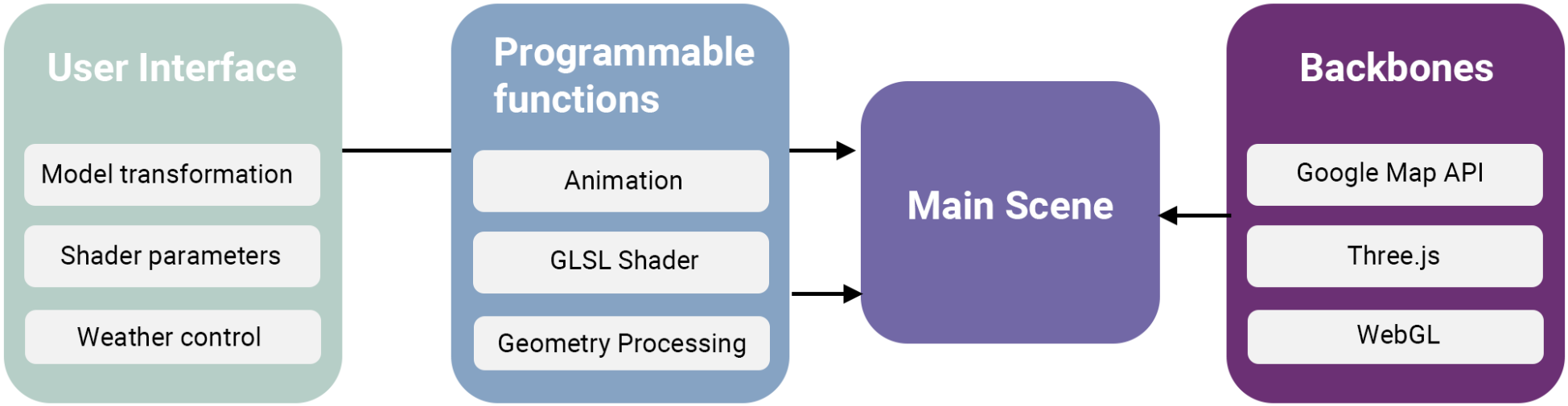
To lower the difficulties for users to manipulate the models, we created User Interface (UI) for each model to do transformations. The components of UI include position, rotation, scaling, visibility, animation, etc.. Users can create their own customized scene on client-side web, which significantly decreases the barriers to use.



GOOGLE MAP API

Google Map API allows satellite maps to be embedded on third-party websites. Google Map enables us to create realistic scene onto the terrain map.

WORKFLOW



The figure above includes several steps to create the main scene. The user interface enables users to have full control on **model transformation, water shader parameters, and weather control.** HydroGL also provides programmable functions for users to customize to their satisfaction. The functions include animations of different objects. In addition, users can create their **own shader program** in GLSL language. Plus, users are able to process the geometry of the water polygon.
The backbones of HydroGL's framework are **Google Map API, Three.js, and WebGL.** Google Map API is a set of application programming interfaces that can retrieve realistic google map data. Three.js is an open-source Javascript library that is used to display 2D and 3D scenes on a web browser. WebGL is a Javascript library for rendering 2D and 3D objects as well as a media that connects Google Map API and Three.js as an overlay.

CASE STUDY

We created three scenes on three different lakes in the US, which are Caney Lake in Louisiana, Pleasant Creek Lake in Iowa, and Cedar Lake in Iowa. After programming the essential parts of the library (loading models and adding animation), we built these scenes solely on UI parameters. Additionally, we can convert to different scenes by changing the directories of the main script. For each lake, we copied the script of the previous one and change the UI parameters directly on the web browser. As a result, CRATES is a flexible and customizable framework for users to build realistic scenes based on different hydrological data.