COVID-19 Visualizer Final Report

Team Tomoki Fukuzawa Fanclub
Tomoki Fukuzawa, 604920860, @tfukaza
Christian Rodriguez, 804789345, @christiaanrr
Kevin Tan, 704826225, @ktan17

1 Introduction

The COVID-19 pandemic has rocked the foundations of societies across the globe. It has redefined "normal" for billions of people, and in the US, it has exposed fundamental issues with the health care infrastructure at both state and federal levels. In order to help those in the present and future that may want to understand the spread of the coronavirus in the US, we decided to build a visualizer that illustrates the virus' spread over time.

2 Project Overview

The visualizer mainly consists of a planar 2D US map. As time progresses, bars (each of whose height represents the number of cases in a given county) begin to appear on the map and grow over time. The color of the bars also changes from blue to red as the height grows. The animation "begins" at January 21st, 2020, and continues until May 18th before it loops. The current date can be seen in the bottom-left corner of the website. We sourced our data from the New York Times' official coronavirus database, located at https://github.com/nytimes/covid-19-data.

A bar only appears for a county if their total number of cases exceeds 50; otherwise, the animation would be extremely data-intensive and laggy. Since availability of data varies by county, in the event that a county does not have data for an arbitrary date we treat the number of cases as 0.

Our visualizer has many elements of interactivity. At any time, you can hover over a bar and you will see the name of the county it corresponds to, the associated number of cases, and the associated number of deaths. You can also click and drag to maneuver the camera position. Finally, there is a light that is centered at your mouse's location; while the bars themselves have ambient lighting the planar map does not. You can illuminate or darken the map as you like for an additional immersive experience.

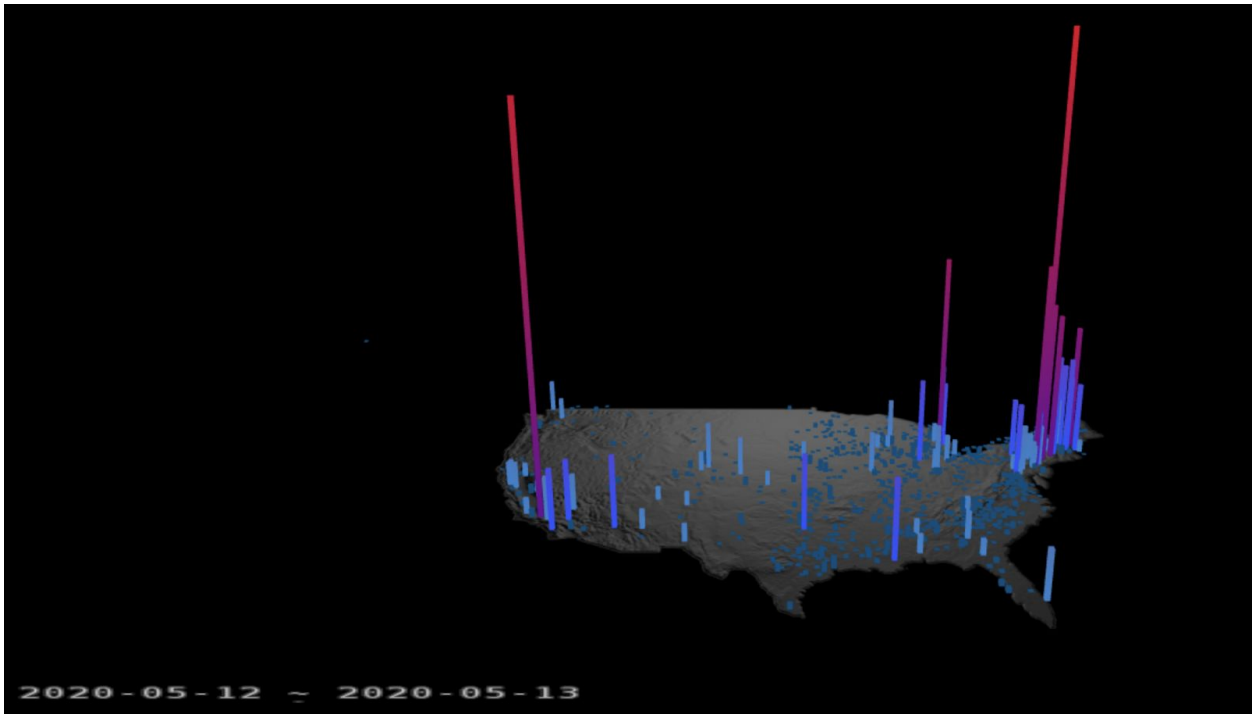

Figure 1: A snapshot of our visualizer showcasing the number of US coronavirus cases on May 12th, 2020.

3 Implementation

3.1 Core Functionality

Our project makes heavy use of the architecture used across our class assignments. That is, the website consists of an embedded WebGL canvas panel (`Canvas_Widget` class from `tinygraphics`) and the majority of our code is defined in a custom `Scene`. Denoting `/` as the root of our directory, the source code can be found at `/examples/covid_map.js`.

Nearly all of the elements in our visualizer rely on components in the `tinygraphics` library. For example, the bars are `Cube` objects whose y-scale is transformed over time. They are shaded using the `Gradient_Shader` class, a subclass of the generic `Shader`. We use the elapsed animation time to calculate the corresponding dates and current bar heights, as can be seen in the `lerp_date` function:

```
function lerp_date(a) {
 var start = new Date('2020-1-21');
 var end = start.setDate(start.getDate() + Math.floor(a * 117));
 var current = new Date(end);
 var current_2 = new Date(end);
 current_2.setDate(current_2.getDate() + 1);
 return [current, current_2];
}
```

We also use each county's latitude and longitude to determine the proper translation transform for each bar:

```
let bar_transform = Mat4
         .translation((lng+95)/5 + 23, 0, -1 * (lat-37)/3)
```

To implement the display of text, we used the `text-demo` located within this repo: https://github.com/encyclopedia-of-code/tiny-graphics-js. We imported `Text_Line`, a `Shape` subclass which uses an image of different text characters located at `/assets/text.png` to shade `Square` objects, for both the dates at the bottom of the webpage and the data displayed on bar hover.

To that end, we implemented hit detection as follows: for every iteration of the graphics loop, we check if the mouse's current position is contained within a bar given the camera's current orientation. If so, we instruct the `Text_Lines` to render themselves accordingly. Otherwise, they are not drawn during that tick. This logic can be primarily found between lines 342 and 385 in `covid_map.js`.

Our implementation of camera controls involved ray tracing.

3.2 Advanced Features

3.2.1 Normal Map

The main advanced feature that we implemented was a normal map. We decided that using a normal map to artificially simulate bumpiness on the underlying US map would add an extra level of creativity, interactivity, and beauty to our visualizer. To implement this, we adapted another class named `Fake_Bump_Map` from the previously

linked repo. `Fake_Bump_Map` is a subclass of the Textured_Phong shader that can be found in the same library.

We found a height map of the US, and converted it to a normal map using an online tool. We use the red and blue values of each pixel to modify the calculated normal vector at every point on the image. Then, when a light source "shines" on the texture, the resulting texture appears bumped. Unfortunately, `Fake_Bump_Map` did not work out of the box for our project, so we added our own modifications to display the normal map correctly. Specifically, we wrote our own GLSL code:

```
vec3 bumped_N  =    normalize(
             vec3(
               N[0] + (tex_color.x - 0.5) * 2.0,
               N[1],
               N[2] + (tex_color.y - 0.5) * 2.0
             ));
```

The map itself is merely a Square instance using this custom shader. Its ambient lighting is set to 0.5, whereas the bars all have ambient lighting of 1.0.

3.2.2 Mouse Picking

The mouse pick feature was added to allow users to hover over a bar, whereupon a text will show up to give precise numbers of Corona cases and deaths.

This works by first assigning a collision box to each bar on the screen, which will be used to check if the mouse is "colliding" with the bar. For efficiency, these "boxes" are not really boxes, but two planes crossed over each other, as shown below. This roughly cuts the number of collision checks we have to do by ⅔, while retaining sufficient accuracy.

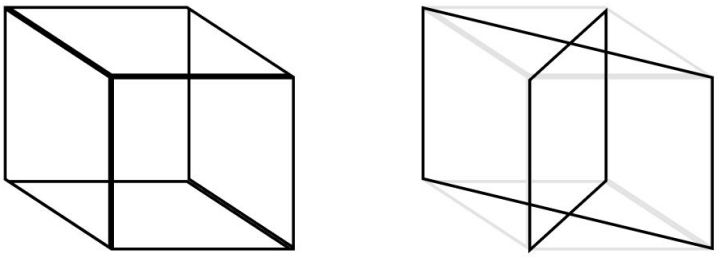

Figure 2: A conventional, box collision (left) and a approximate, 2-plane collision (right)

For every frame, the code checks to see if any of the boxes collide with the mouse. This is first done by projecting the box to screen space, and then doing a point-in-polygon test to see if the mouse is within the box. If the mouse is inside the box(projected onto screen space), it means the mouse is currently hovering over the bar.

In addition,at any given time it is possible that the mouse is hovering over multiple bars, especially when many bars are located close to each other. In such cases, the intuitive choice is to select the bar that is closest to the viewer. To perform this, the code has a 1-slot buffer to store the location of an arbitrary bar.

For each frame, the code parses over every bar it has to display. If a bar is being hovered over, it is put in the buffer. If there is already a bar in the buffer, its distance from the viewer is compared - if the current bar is closer, the bar in the buffer is rendered, and the current bar is put in the buffer. If the current bar is farther away, the buffer remains unchanged and the current bar is rendered. After all bars have been parsed, any remaining bar in the buffer is rendered along with its cases/death information, as it is that bar that is closest to the viewer.

3.2.3 Misc. Features

Though not advanced, there are several features that are noteworthy in this project.

Adaptive Scrolling

When the user drags the mouse to scroll (move) the camera, the speed depends on where the mouse drag is happening. For example, when one drags the mouse near the top of the screen, the camera scroll will be much faster compared to fragging at the bottom of the screen.

This is done by shooting a ray from the camera to where the mouse is. Then, the collision point of the ray and the "ground" (plane where the US map exists) is calculated. Camera movement is thus calculated based on where the mouse is dragging in world space, not screen space. This creates a scroll effect similar to that of Google Earth, and feels more intuitive compared to simply dragging based on screen space, as it gives the user the sensation as if he/she is pinching on the ground and moving it with his/her hand.

Movable Light

To make the normal map effect on the US map more noticeable, the light source in the scene moves along with the mouse. This is done by using the ray collision detection used for adaptive scrolling - except this time, the location where the ray and plane collides is set at the location of the light source.

Gradient Shader

The shader used on the bars is a custom shader built from scratch. Instead of one, it takes two colors, base_color and top_color. When the bar is short, it will only show the base_color, but as it grows vertically, the color slowly transitions to top_color, as shown in Figure 1. This is done by modifying the vertex color based on its world space coordinate position.

Billboards

The text in this project are billboards - that is, they always face the user and have a constant scale, no matter where they are positioned in the world space. This is relatively easy if the text is in a specific location on the screen - all that is necessary is to plot the text in screen space, and convert it to world space.

For texts that are displayed next to the bar, it was a little more difficult, as the world space coordinates first had to be converted to screen space, applied some matrix operations, and then converted back to world space. The following code is responsible for this task, where tt is the matrix that indicates the location of the text in world space, and b.t is the location of the bar:

```
let tt =
        world_to_perspective
        .times(
            new tiny.Matrix( [1, 0, 0, b.t[0][3]],
                    [0, 1, 0, 0],
```

```
                    [0, 0, 1, b.t[2][3]],
                    [0, 0, 0, 1]
                )
            )

            tt = new tiny.Matrix( [0.02 * (h/w), 0, 0, (tt[0][3] +
1)/tt[3][3]],
                                  [0, 0.02, 0, (tt[1][3] -1 + 2 * b.cases
/ cases_scale - i * 0.8 )/tt[3][3]],
                                  [0, 0, 0.02, -0.1],
                                  [0, 0, 0, 1]
            );

            tt = Mat4.inverse(world_to_perspective).times(tt);
```

4 Contributions

In all honesty, Tomoki led this group and his deep knowledge of graphics and strong coding skills are the reasons for our success. With this in mind, the other members (Kevin and Christian) think that Tomoki should receive most of the credit for this project.

One of the first challenges our group faced was adapting code from our assignments to use the slightly different `tinygraphics` library found at https://github.com/encyclopedia-of-code/tiny-graphics-js. We needed to display text, and the text demo within the aforementioned repo was perfect for this. All three of us tried to integrate the updated library into our code; Christian tried on the `usa-christian` branch, and Kevin tried on the `text-demo` branch. However, Tomoki once again delivered and managed to get it working.

Tomoki wrote the script to parse the raw COVID data, the plotting of bars, the animation, the camera controls, and mouse picking. Christian and Kevin worked on adding text and interactivity. Altogether over a call, we pair programmed to implement the normal map and finalize aesthetics.