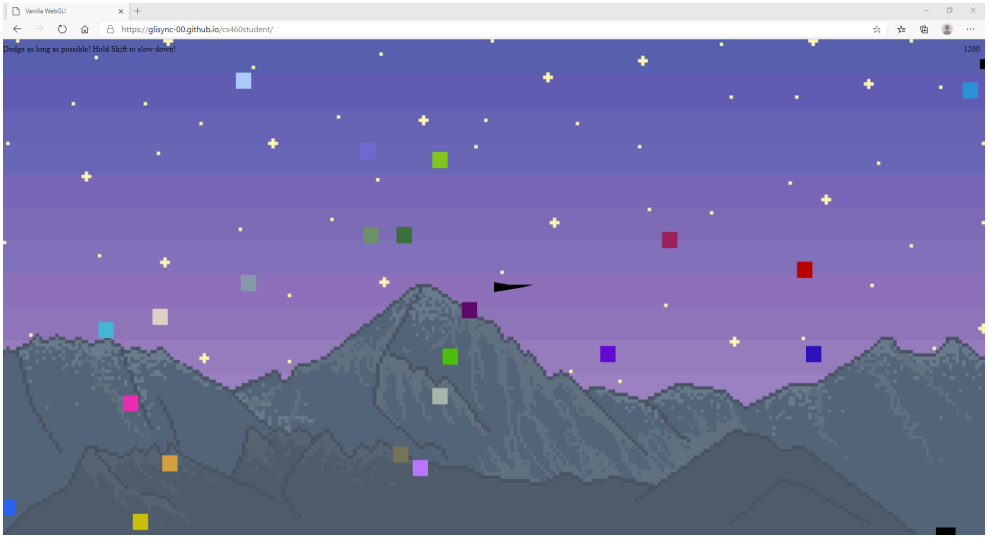
CS460 Fall 2020
Github Username: Glisync-00
Due Date: 10/09/2020

Assignment 4: A WebGL Game!

WebGL without a framework is hard. But this makes it even more rewarding when we create cool stuff!

In class, we learned how to draw multiple objects (rectangles) with different properties (colors and offset). We also made the rectangles move! In this assignment, we will create a simple but fun video game based on the things we learned. In the game, the player can control an airplane using the `UP,DOWN,LEFT,RIGHT` arrow keys in a scene to avoid obstacles. The longer the player can fly around without colliding with the obstacles, the more points are awarded. Once the player hits an obstacle, the game is over and the website can be reloaded to play again. The screenshot below shows the black airplane roughly in the center and multiple square obstacles in different colors. The mountains, sky, and stars are a background image that is added via CSS (as always, feel free to replace and change any design aspects).

The screenshot shows my game though you should play it to see how it works



Starter code: Please use the code from `https://cs460.org/shortcuts/16` and copy it to your fork as `04/index.html`.

Part 1 Coding: Extend the `createAirplane` method. (25 points)

First, we need to create the airplane. Take a look at the existing `createAirplane` method. **This method needs to be extended.** We will use triangles to create a shape similar to the one pictured below. Please figure out the triangles we need and set the `vertices` array. We can assume that the center of the airplane is `0,0,0` in viewport coordinates. Then, please setup the vertex buffer `v_buffer` (and remember create, bind, put data in, unbind). There is no need to change the `return` statement of the method. This returned array contains the name of the object, the vertex buffer, the vertices, an offset, a color, and the primitive type—the drawing code of the `animate` method needs this array in this exact order.

Part 2 Coding: Extend the `createObstacle` method. (25 points)

Now we will extend the `createObstacle` method. This method creates a single square obstacle. There are different ways of rendering a square but the simplest is to use a single vertex and the `gl.POINTS` primitive. Make sure that `gl_PointSize` is set appropriately in the vertex shader! We use `0,0,0` as our vertex and then control the position of the obstacle using the `offset` vector. Please modify the code to set the x and y offsets to random values between -1 and 1 (viewport coordinates). The color of an obstacle is already set to random and the `return` statement follows the same order as in Part 1. Once this method is complete, multiple obstacles should appear at random positions on the screen (9 in total as added to the `objects` array after linking the shaders).

Part 3 Explaining: Detect collisions using the `calculateBoundingBox` and `detectCollision` method. (20 points)

In class we learned about bounding boxes. The starter code includes collision detection using bounding box calculation of the airplane and the offset of an obstacle. Please study the existing `calculateBoundingBox` and `detectCollision` methods and describe how it works and when the collision detection is happening:

CalculateBoundingBox() tries to find the minimum x, minimum y, and minimum z coordinate values in all the vertices of an object. Its input is the object's initial vertices and its offset vector which helps calculate the current position of each vertex. The x, y, and z coordinate for each vertex is calculated by its initial position plus the offset vector. Initially we compare the min x/y/z to to -1000 and max x/y/z to 1000 before we search each vertex for the min and max x/y/z value respectively. The function returns these min xyz values as an array.

detectCollision() takes as input a boundingbox array (output for CalculateBoundingBox) and a xyz point. It then checks to see if the point is within the bounds of the minimum/maximum x, then the bounds of the minimum and maximum y, and finally within the bounds of the minimum and maximum z. If the point passes these 3 conditions, the function returns true. Otherwise it will return false.

Part 4 Coding: Extend the `window.onkeyup` callback. (20 point)

We want to allow the player to use the arrow keys to move the airplane. Please take a look at the existing `window.onkeyup` method. The `if` statement checks which arrow key was pressed. Please extend this method to move the airplane. Hint: Like in class, we just need to set the `step_x,step_y` values and the `direction_x,direction_y` based on which arrow key was pressed.

Part 5 Cleanup: Replace the screenshot above, activate Github pages, edit the URL below, and add this PDF to your repo. Then, send a pull request for assignment submission (or do the bonus first). (10 points)

Link to your assignment: `https://glisync-00.github.io/cs460student/`

Bonus (33 points):

Part 1 (11 points): Please add code to move the obstacles! Flying the airplane around static obstacles is half the fun. The obstacles should really move! Please write code to move the obstacles every frame. The obstacles should just move in `x` direction from right to left to create a flying illusion for the airplane. This can be done with little code by modifying the offsets accordingly at the right place!

see link. The obstacles are moving

Part 2 (11 points): Make the obstacles move faster the longer the game is played! Right now, the game is not very hard and a skilled pilot can play it for a very long time. Currently, the scoreboard updates roughly every 5 seconds. What if we also increase the speed of the obstacles every 5 seconds? Please write code to do so. This can be done in one line-of-code!

see link. The obstacles move faster every 300 score for balance

Part 3 (11 points): Save resources with an indexed geometry! As discussed in class, an indexed geometry saves redundancy and reduces memory consumption. Please write code to introduce a `gl.ELEMENT_ARRAY_BUFFER` for the airplane. Of course, we do not need to change anything for the obstacles since a single vertex does not need an index :).

see code snippet below.

```
function createAirplane() {

  //
  // Part 1 Starts (uses indexed geometry)
  //
  var vertices = new Float32Array([0.0, 1.0/50, 0.0, // V0
                                   0.0, -1.0/50, 0.0, // V1
                                   3.0/50, 0.0, 0.0, // V2
                                   1.5/50, 0.5/50, 0.0, // V3
                                   4.0/50, 0.5/50, 0.0, // V4
                                   6.0/50, 0.0, 0.0,  // V5
                                   ]);
  var indices = new Uint8Array([0, 1, 2,  // Triangle 1
                                2, 3, 4,  // Triangle 2
                                5, 2, 4]) // Triangle 3

// Create Geometry
var v_buffer = gl.createBuffer(); // create buffer
gl.bindBuffer(gl.ARRAY_BUFFER, v_buffer); // bind
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW); // put in data
gl.bindBuffer(gl.ARRAY_BUFFER, null); // unbind


var i_buffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, i_buffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```