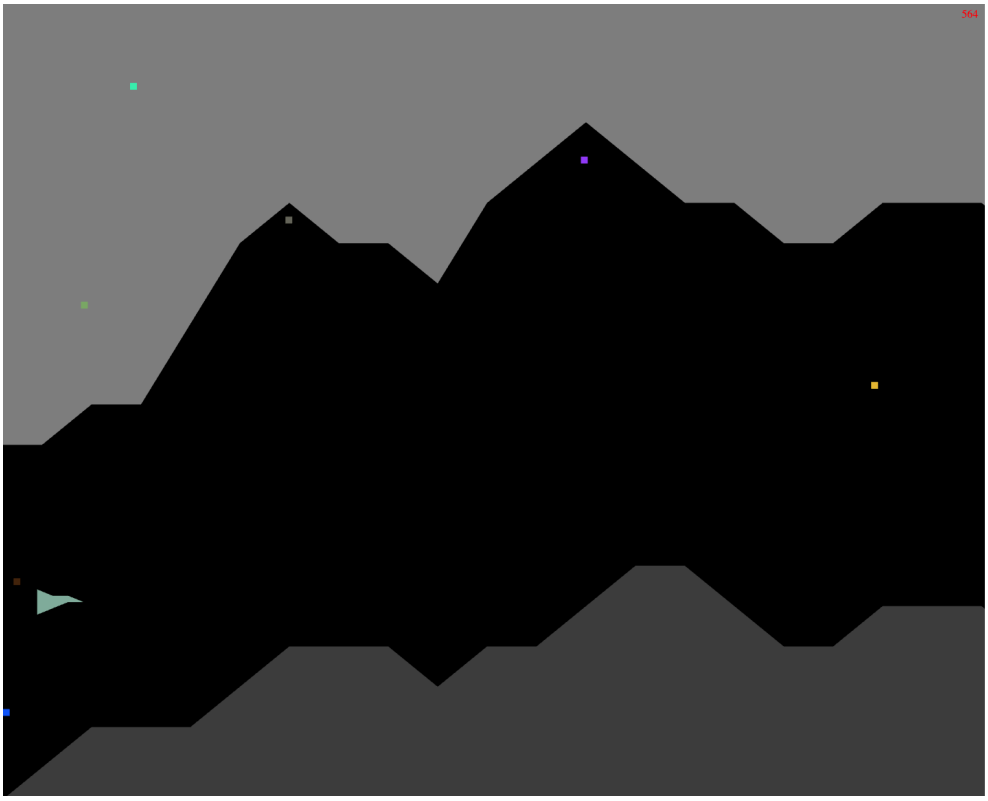
CS460 Fall 2020
Github Username: JamesEdMichaud
Due Date: 10/09/2020
Attention Loraine, I apologize that this commit/push was slightly after the deadline. I was working on parameterizing the wall segments in my game. If this version is not acceptable, please see the commit made prior to the deadline – it still has everything required for the assignment.

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).



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. Line 140.

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).

Line 167 (obstacle), line 178 (wall), line 234 (wall update).

I decided to add obstacles in the form of ”walls” above and below the airplane. This proved to be a much greater challenge than I anticipated. Each wall segment is 2/wallResolution x-values in width. When a wall runs off screen, it’s offset is set to the rightmost wall segment’s offset, plus the width.

I struggled for several hours with getting this to occur correctly. In the array of wall segments, walls[0] would always be placed with a space between it and walls[walls.length-1]. If I iterated backwards there would be a space between every segment **except** those two. This led me to realize that the wall being moved across the screen was not being moved in the regular way (left slowly), and thus a gap formed.

I then struggled for a while trying to implement a triangle collision algorithm to use in wall collision detection. That was a bit too much for this assignment, so I instead implemented a point-line intersection formula to handle approximate collision detection. If a corner of the bbox passes over a wall, game over.

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:

Bounding box is only calculated for airplane. The calculateBoundingBox method finds the minimum and maximum x, y, z coordinate values over all vertices that make up the airplane. This results in a cuboid shaped bounding box.

Obstacles don’t require a bounding box, as they are simply points – a single vertex. During each animation frame, each obstacle calculates whether it will collide with the airplane by checking whether it is within the bounding box.

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.

Lines 365 and 373.

I decided to use onkeydown and onkeyup so that the user may hold keys down to move the airplane. I also include WSAD controls, in case one prefers that instead.

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://jamesedmichaud.github.io`

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!

Line 493 (obstacle), line 431 (wall).

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!

Line 418.

I applied the same speed-up to the walls, and make the score increase faster as time goes on. Further, the airplane speed increases a bit to offset when everything begins to move so fast it's impossible to dodge.

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 :).

Line 151 & 159 (airplane), line 101 (wall). Line 132 has a method for creating the iBuffers.