

CS460 Fall 2022

Your Name: Gabriel Vivas

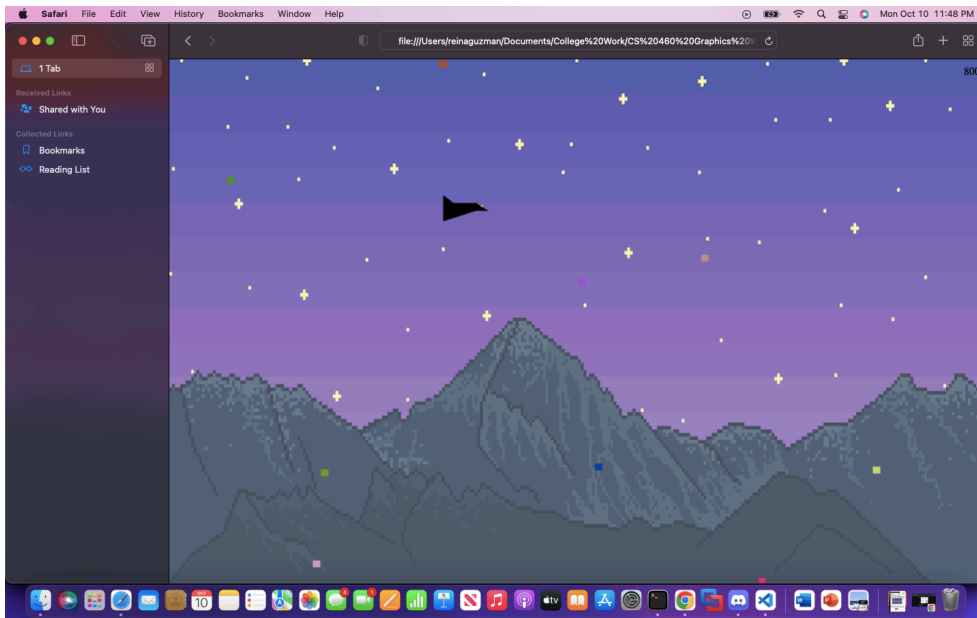
Github Username: gab-viva02

Due Date: 10/10/2022

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.



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:

The function `calculateBoundingBox()` takes in the airplane's default vertices, and the airplane's offset at the current frame. Then, for each of the vertices in the airplane, it calculates the given vertex's position by applying the current airplane offset to the vertex. Using the current vertex's coordinates, the function checks to see if the current vertex has either the largest or smallest `x`, `y`, and/or `z` coordinate out of all the vertices examined so far, before updating the current max/min `x/y/z` coordinate if needed. Once the function has finished going through all the vertices in the airplane, it has an array containing the minimum `x`-coordinate the airplane occupies, the maximum `x`-coordinate the airplane occupies, the minimum `y`-coordinate the airplane occupies, the maximum `y`-coordinate the airplane occupies, the minimum `z`-coordinate the airplane occupies, and the maximum `z`-coordinate the airplane occupies. With this information, we can imagine the dimensions of the smallest possible rectangular box that could contain the entire airplane. This rectangular box represents the "hitbox" of the airplane as a whole.

The function `detectCollision()` takes in airplane's current "hitbox" as calculated by `calculateBoundingBox()`, and it takes in the current coordinates of a given obstacle, which is equivalent to that obstacle's current offset value. Initially, `detectCollision()` assumes the current obstacle and the airplane aren't colliding with one another. Then, the function uses a nested conditional to see if the current obstacle is within the bounds of the airplane's "hitbox." First, the function checks if the `x`-coordinate position of the obstacle is between the minimum and maximum `x`-coordinates of the airplane's "hitbox." If so, the function moves on to the next conditional. Otherwise, the function knows that the two objects are not colliding with one another. The second conditional checks to see if the obstacle's `y`-coordinate position is between the minimum and maximum `y`-coordinates of the airplane's "hitbox." If the second conditional proves to be true, then the function moves on to the final conditional. Otherwise, the function knows that the two objects are not colliding with one another. The final conditional checks to see if the obstacle's `z`-coordinate position is between the minimum and maximum `z`-coordinates of the airplane's "hitbox." If all three conditionals prove to be true, then the current obstacle is somewhere within the bounds of the airplane's "hitbox," and thus the airplane and the current obstacle have collided with one another. In this case, the current round of the game ends, and a new round can start again.

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://gab-viva02.github.io/cs460student/04/>

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!

Attempted, see code

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!

Not Attempted

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 :).

Not Attempted