**Intro to Computer Graphics**

**Dr. Mahmoud Eldefrawy**

**Assignment 2 Report by Jose Marquez**

* Part 1 - Unit Square Implementation

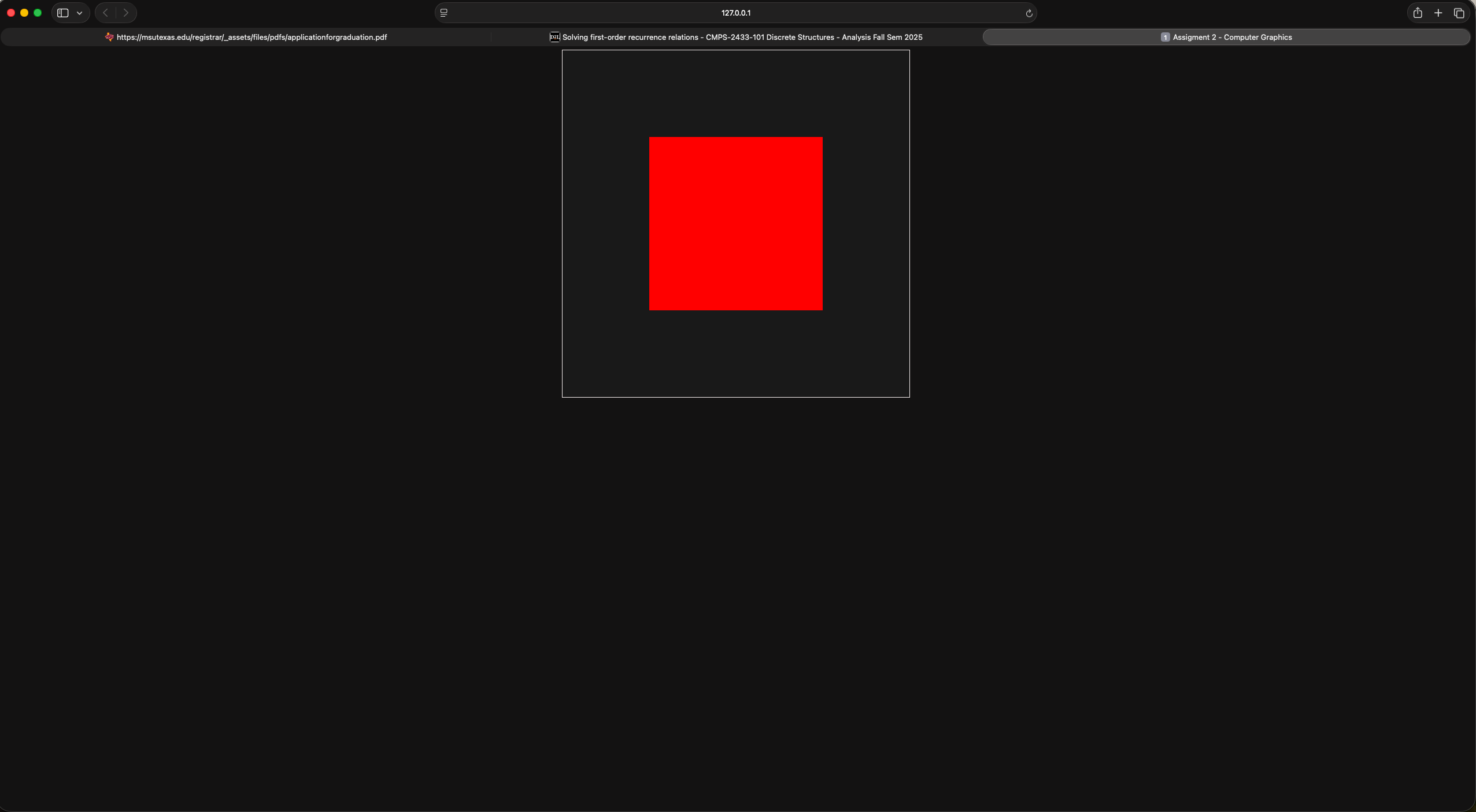
I have created a unit square centered at the origin in WebGL. As the assignment instructed, I was able to follow the implementation details given to me in this assignment. Defined the square vertex coordinates as **(-0.5, 0.5, 0),\ (0.5, 0.5, 0),\ (-0.5,\,-0.5, 0),\ (0.5, -0.5, 0),** giving a 1x1 square centered at (0,0,0).

Used a TRIANGLE\_STRIP primitive to draw the square efficiently with only four vertices, and I was able to set up the WebGL context with depth-testing and a dark grey clear color at first, then I was able to play with my code and turn it to pitch black.

Implemented minimal vertex and fragment shaders. The Vertext shader applies the model matrix (uModel) to each vertex, as well as the fragment shader uses a solid color red implemented at (uColor = 1,0,0,1). I have applied an identity transformation matrix for this part to keep the square at the origin wih no scaling or rotation

At last, verified the correct viewport configuration and pixel ratio handling to ensure the square appears properly centered on the canvas.

The result was a red unit square that appears centered in the WebGL canvas with correct proportions and orientation, as you will see in the picture below.



**Part 2 - Multiple Squares & Two Transform Paths**

I extended the program to render multiple squares and implemented the two required transformation approaches:

* **GPU transform path:**

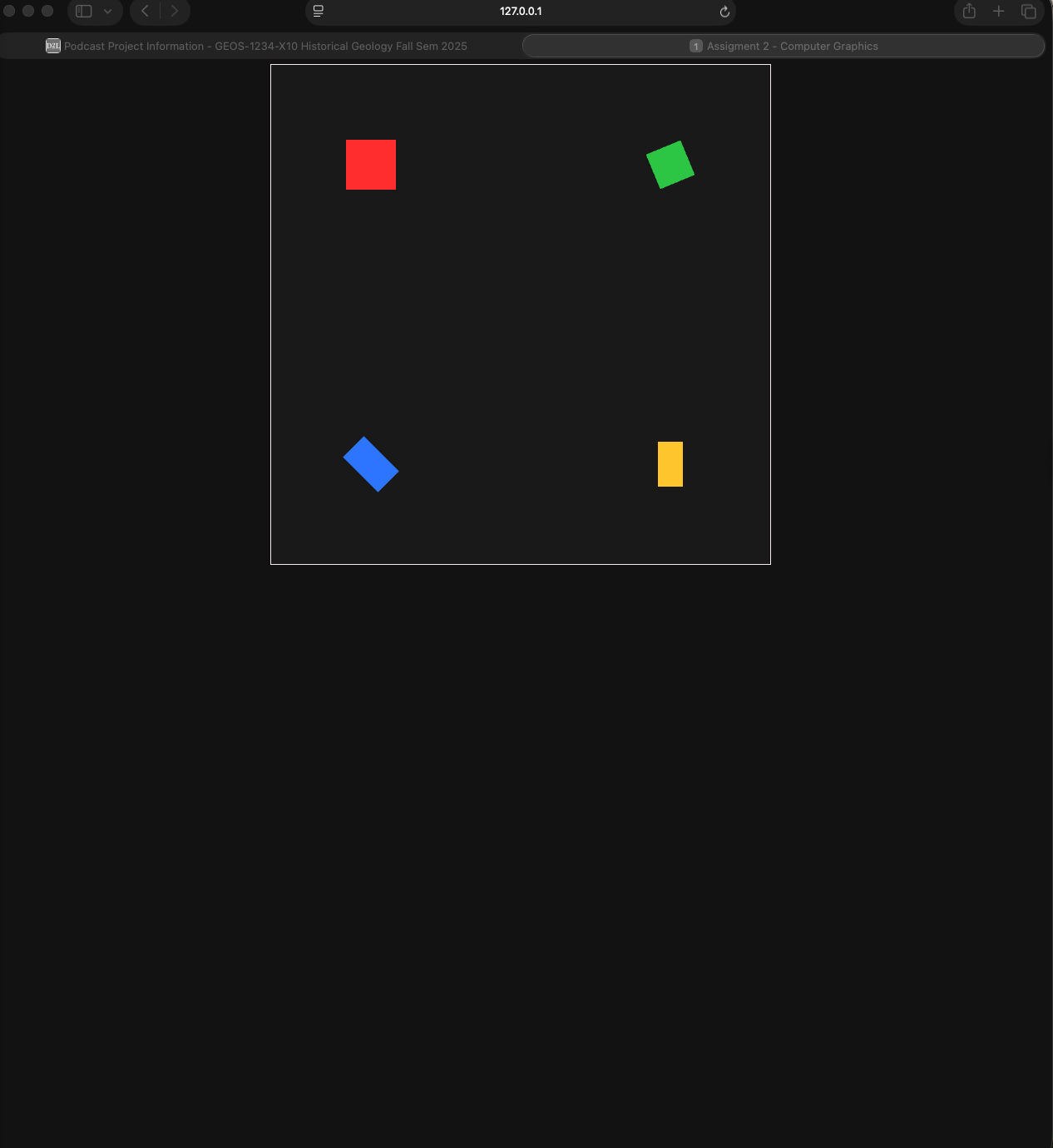
I have kept the original square on the GPU and, for each square, sent a model matrix and color as uniforms, then issued one call per instance.

* **CPU transform path:**

I added a method to transform the 4 vertices on the CPU by multiplying each point (x,y,z,1) with the model matrix, uploaded the transformed vertices to the buffer, and drew using an identity uModel in the shader.

For this part, I have created a small instances array with per-square position, scale, rotation, and color. The render loop iterates over the instances and draws each square using the selected path.

The GPU path results were multiple squares of different colors, and the CPU showed the same visual result. Below is the picture of the final result from this part.



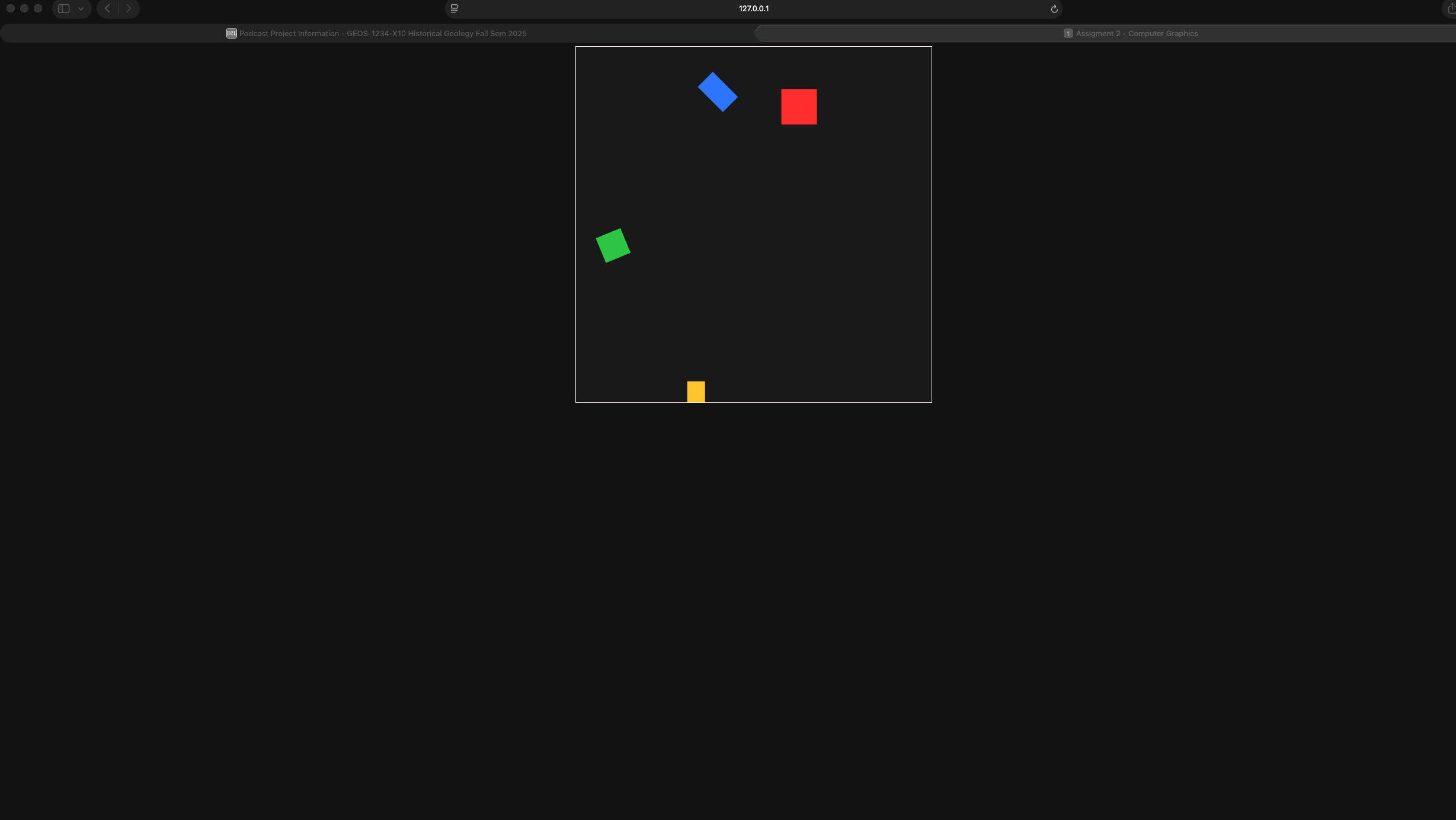
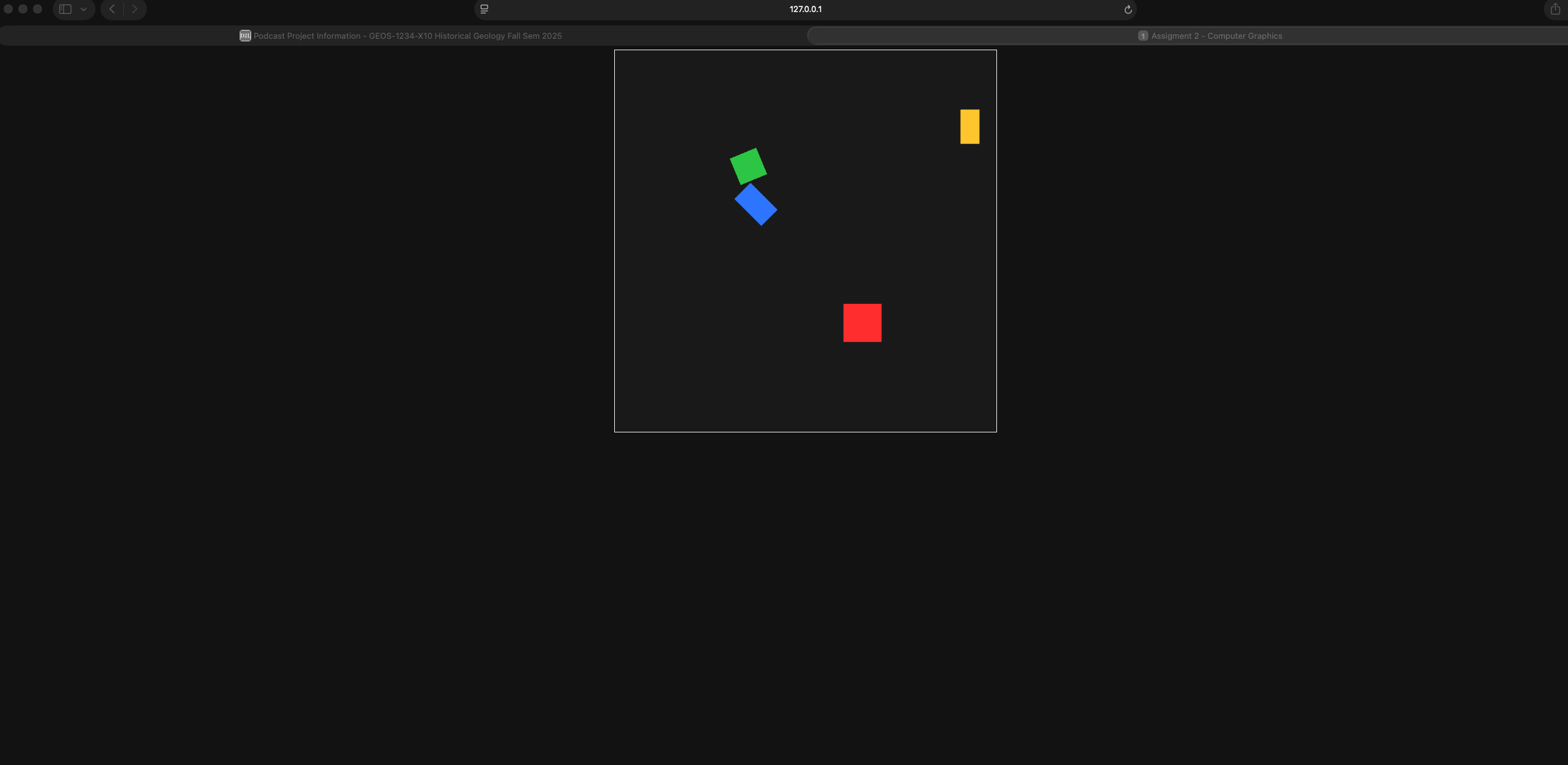
**Part 3 - Rain Implementation**

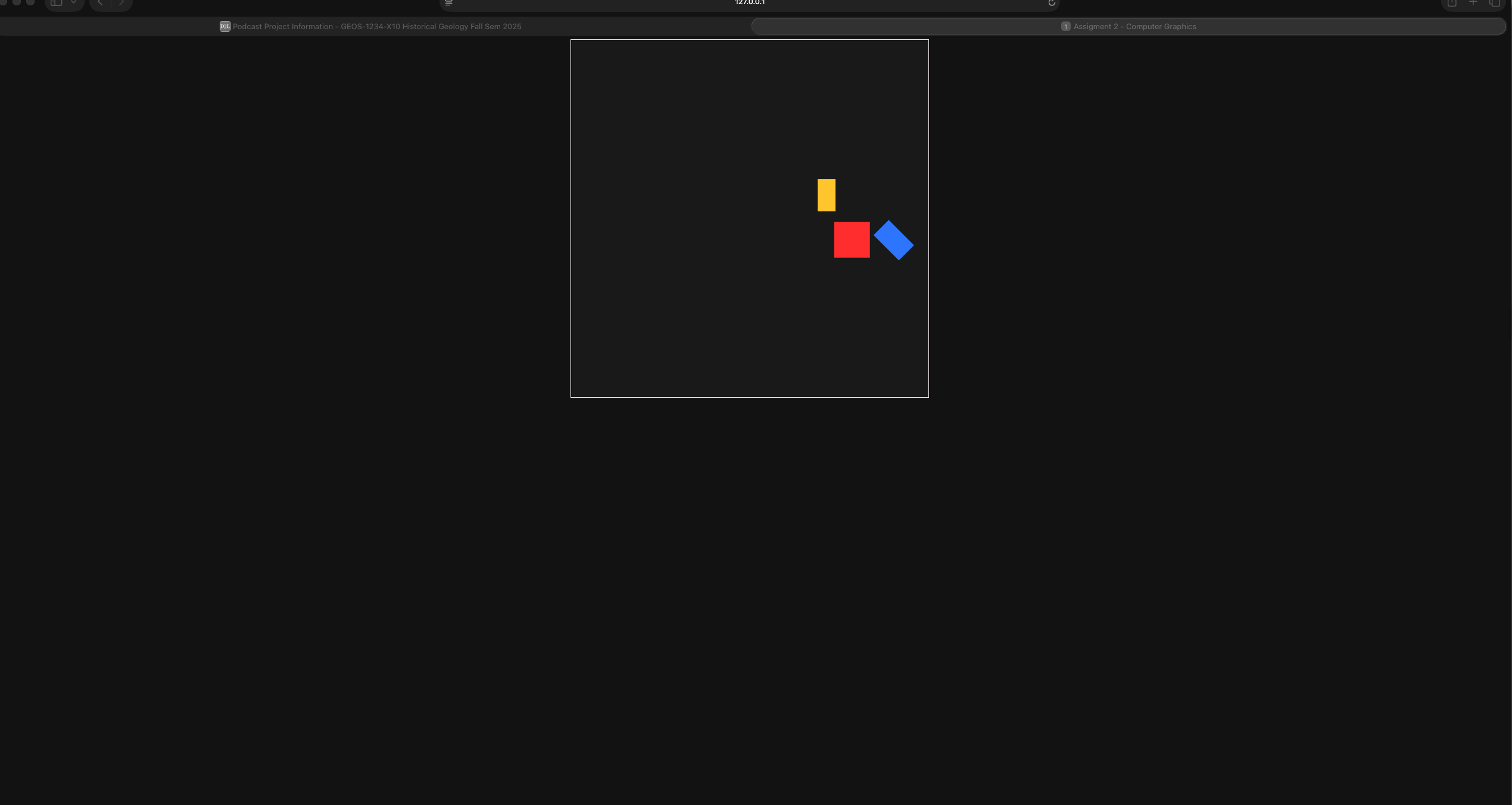
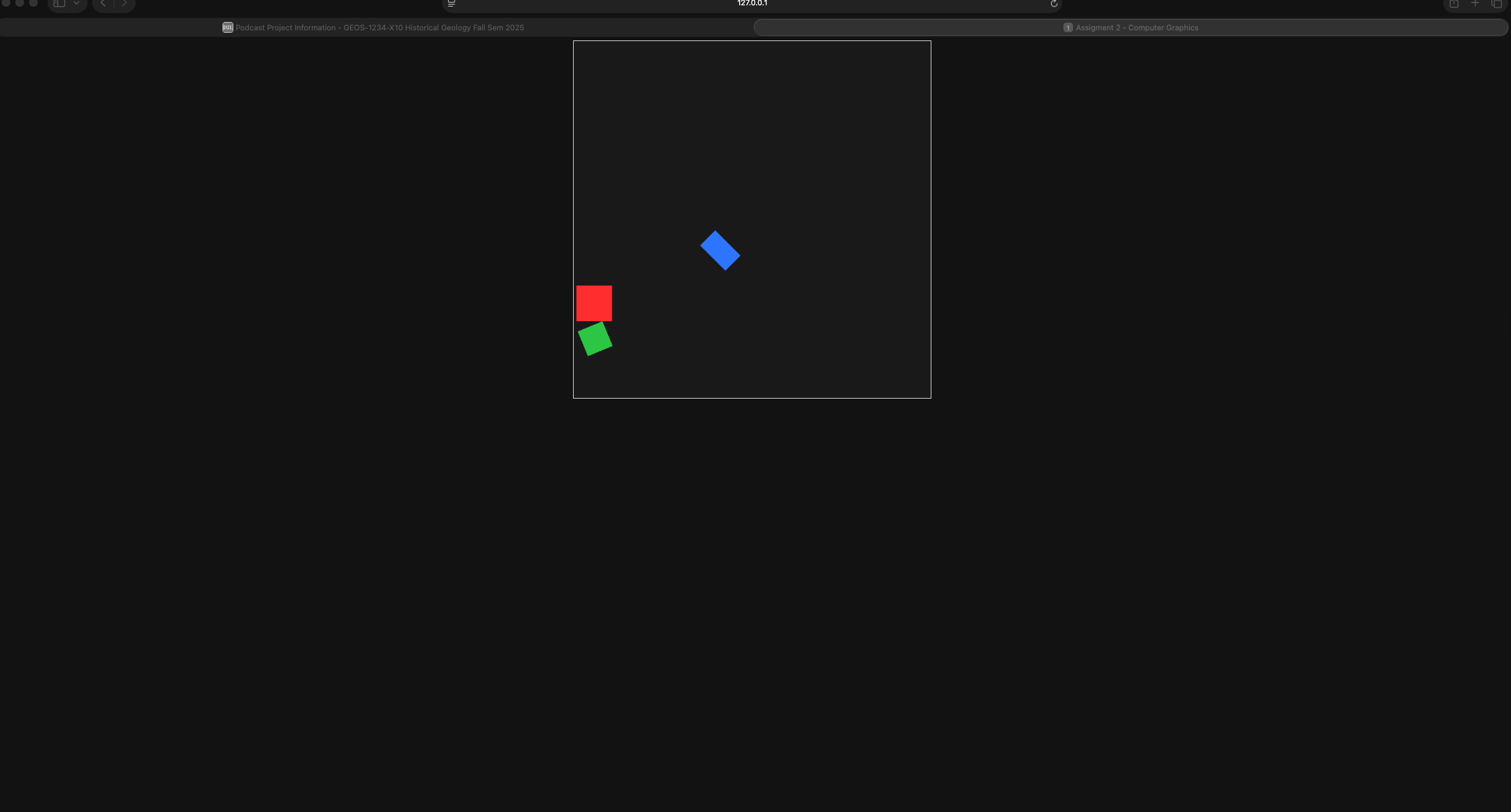
For this part, I implemented a continuous animation loop using requestAnimationFrame that updates each square and transforms every frame to create a “rain” effect.

**Implementation details:**

* Added per-instance velocity to each square in the instance array
* Created an update function that moves each square downward by vy (velocity) \* dt ( update) every frame.
* When a square moves below the bottom of the view, it is respawned at the top with a randomized x position within the visible range.
* Built an animation loop frame that computes a time step, calls update, renders (), and schedules the next frame with requestAnimationFrame(frame).
* Preserved both transformation paths so the animation runs correctly.

The result of this was a continuous rain of multiple coloured squares falling across the canvas, with proper redraws each frame and seamless respawn logic at the top boundary.





**Conclusion**

**Added extensions:**

•Implemented a real-time falling animation for each square.

•Added velocity and random respawn logic for natural motion.

•Ensured compatibility with both GPU and CPU transformation paths during animation.

**Sources used:**

•MDN Web Docs (WebGL API reference, animation timing using

requestAnimationFrame).

•OpenGL ES 2.0 Specification for understanding matrix multiplication behavior.

•Instructor-provided lecture slides and in-class shader examples.

**Experience and challenges:**

Throughout this assignment, I spent approximately three hours per day for the past week and a half refining the code, debugging matrix transformations, and ensuring identical visual output between the GPU and CPU paths. One recurring challenge was maintaining consistent behavior between both paths during animation, as even a small matrix-multiplication order error caused flipped or distorted motion. Debugging WebGL without high-level tools also required frequent console testing and visual validation. Completing this project deepened my understanding of transformation pipelines, buffer management, and frame-timing control in WebGL.