Venkata Chakradhar Reddy Challa

Rasterization

Compiling Instructions:

In order to compile the downloaded source code on your machine, you need to do the following:

1. Install CMAKE
2. Download the submission.
3. Each task has a main file typically name as "main_task<task number>". Rename the main file (of whatever task you want to execute) to "main".

   Note: main_task5 and main pretty much carries all the functionalities developed on the preceding tasks. So, you can just use main/main_task5 to check all the tasks functionalities. If some functionality is not available or doesn't work as expected, please do check the respective task file.

4. Create a directory called build in the downloaded folder directory by typing in a terminal window: cd TOPDIR; mkdir build
5. Create the necessary makefiles for compilation and place them inside the build/ directory, using the CMAKE GUI (windows), or typing: cd build; cmake ../
6. Compile and run the compiled executable by typing: make; ./ Rasterization_bin

Note: If your machine runs on the latest macOS Mojave, you need to resize the window once to see OpenGL output.

More : https://developer.apple.com/macos/whats-new/, https://stackoverflow.com/questions/52509427/mac-mojave-opengl

Task1: Triangle Soup Editor

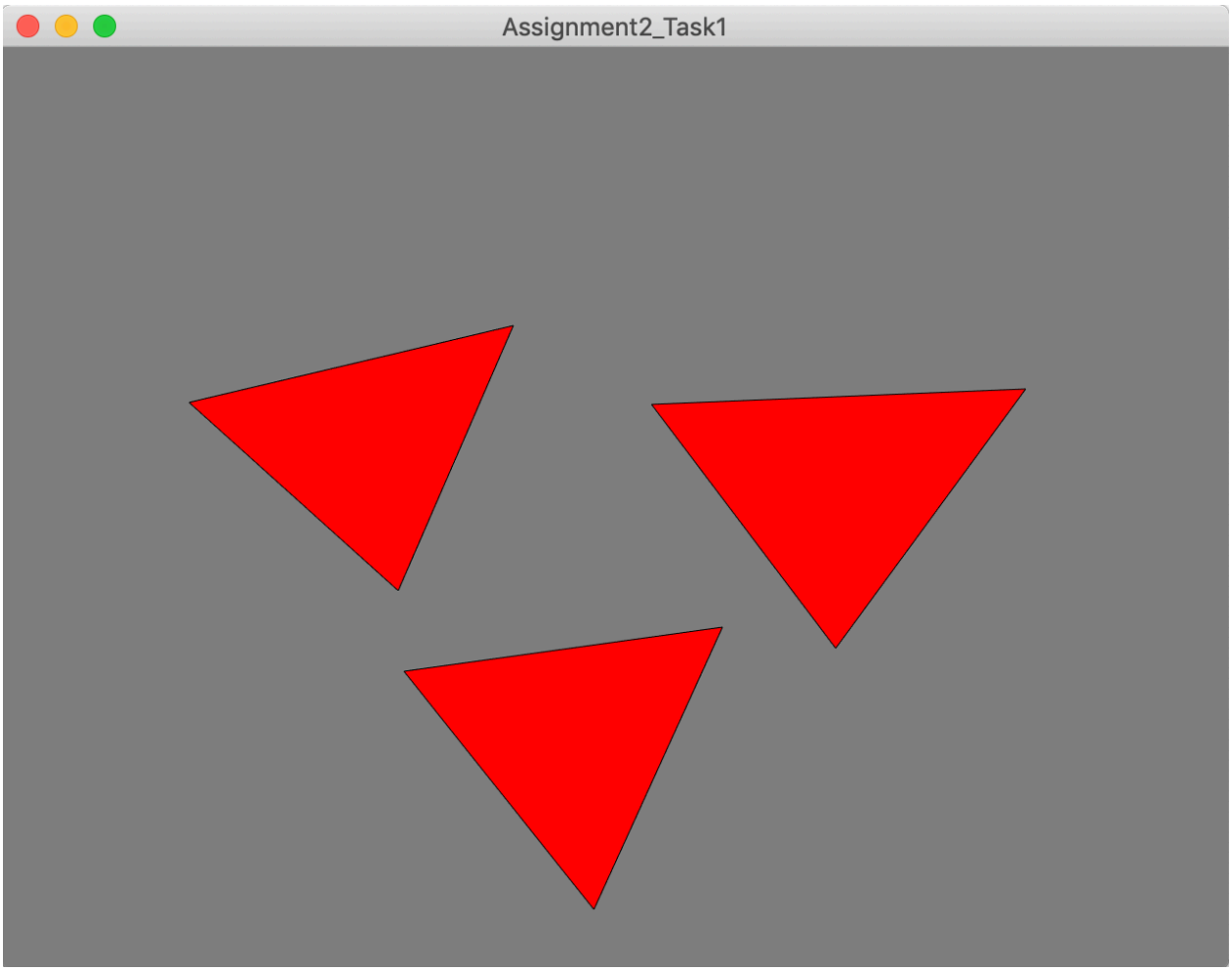
Implementation:

1. Created a matrix 'V' to hold vertex values generated dynamically on mouse clicks and movements as indicated in the requirements. V is of dynamic size and is resized without losing previous data whenever there is insertion or deletion.
2. Created a matrix 'view' to hold view changes to the entire screen/a particular object.
3. Created VAO and VBO to hold vertex value for the Vertex shaders. VAO and VBO are binded to the program and VBO is updated whenever there is a change in vertices matrix 'V'.
4. mouse_button_callback, cursor_position_callback, and key_callback are used to trigger respective events and detect mouse, cursor and key data.
5. Created "vertex shader" and "fragment shader" to process and draw respective objects in the GPU using the passed on information from CPU through VAO, VBO and uniform variables.
6. Program, VAO and VBO are freed at the exit.

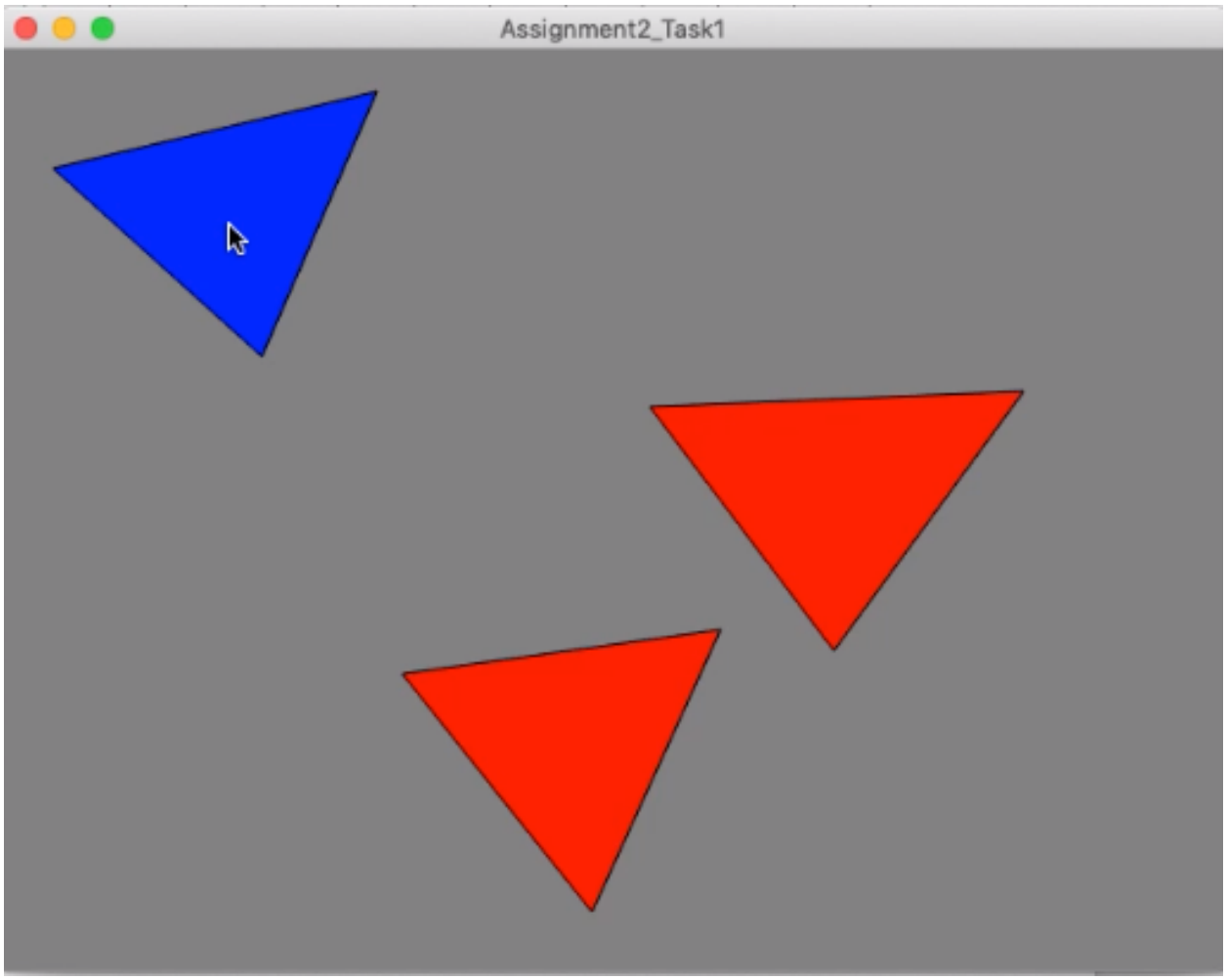
7. Insertion mode:
    a. This mode is detected using the key press of key ‘i’ and enables the user to create points, lines, line loop, and triangle using the respective mouse clicks and movements as per the instructions from requirements document.
    b. V is updated, Also VBO to let the shader know of the updates, when there is a new vertex generated with user actions.
8. Translation mode:
    a. This mode is detected using the key press of key ‘o’ and enables the user to select a triangle/object and move it around the screen with mouse drag.
    b. View is updated with the transformation done by the user following the instructions on requirements document.
    c. An algorithm(ptInTriangle) is used to detect object/triangle selection for translation.
    d. Once the transformations are finalized (when the user moves out of translation mode either by other key press or mouse click) the vertex matrix is updated followed by VBO update.
9. Deletion mode:
    a. This mode is detected using the key press of key ‘p’ and enables the user to delete a triangle/object by clicking on it.
    b. An algorithm(ptInTriangle) is used to detect object/triangle selection for deletion.
    c. The vertex matrix ‘V’ and ‘VBO’ are updated with changes.

Result:

1. Insertion

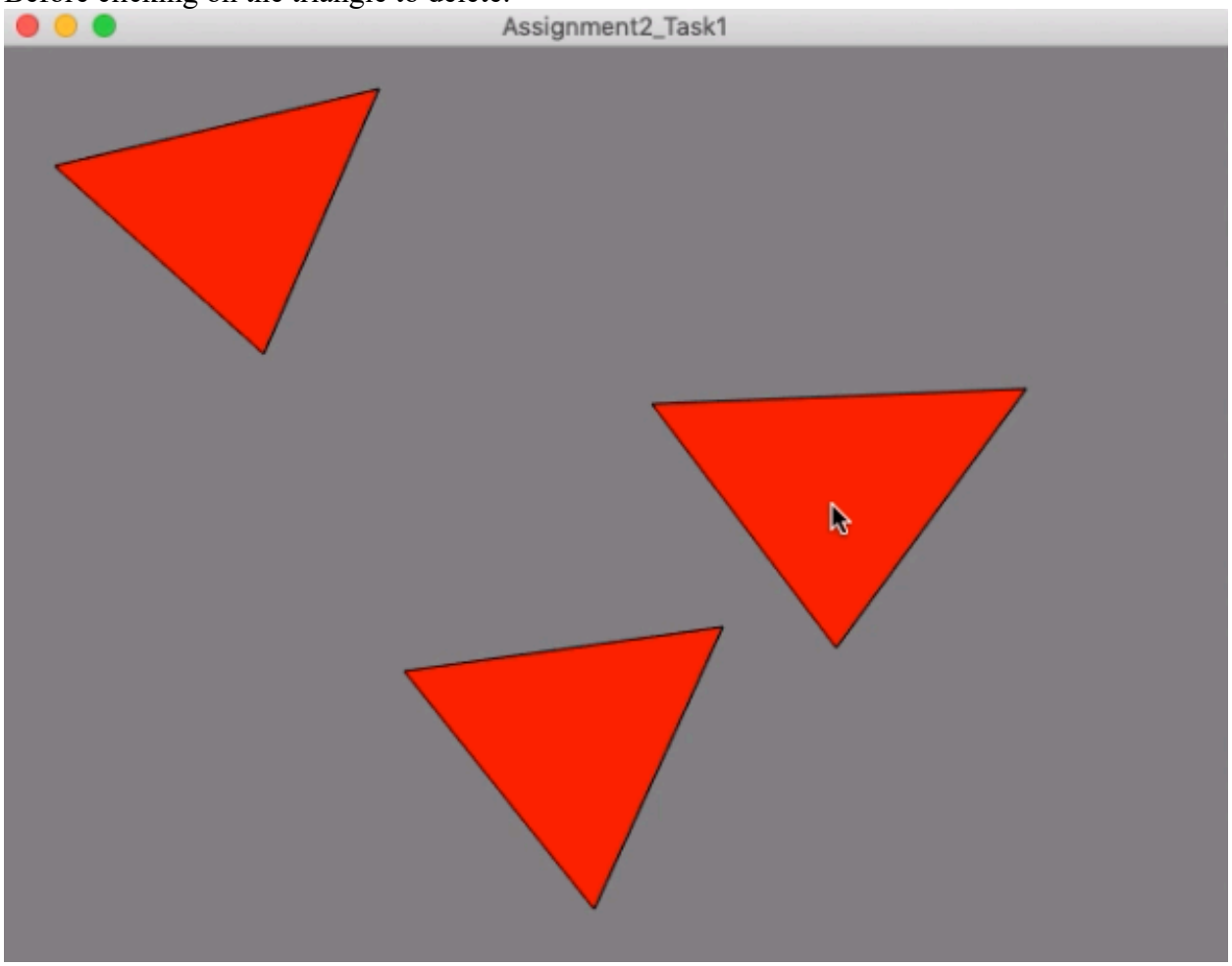


2. Translation:

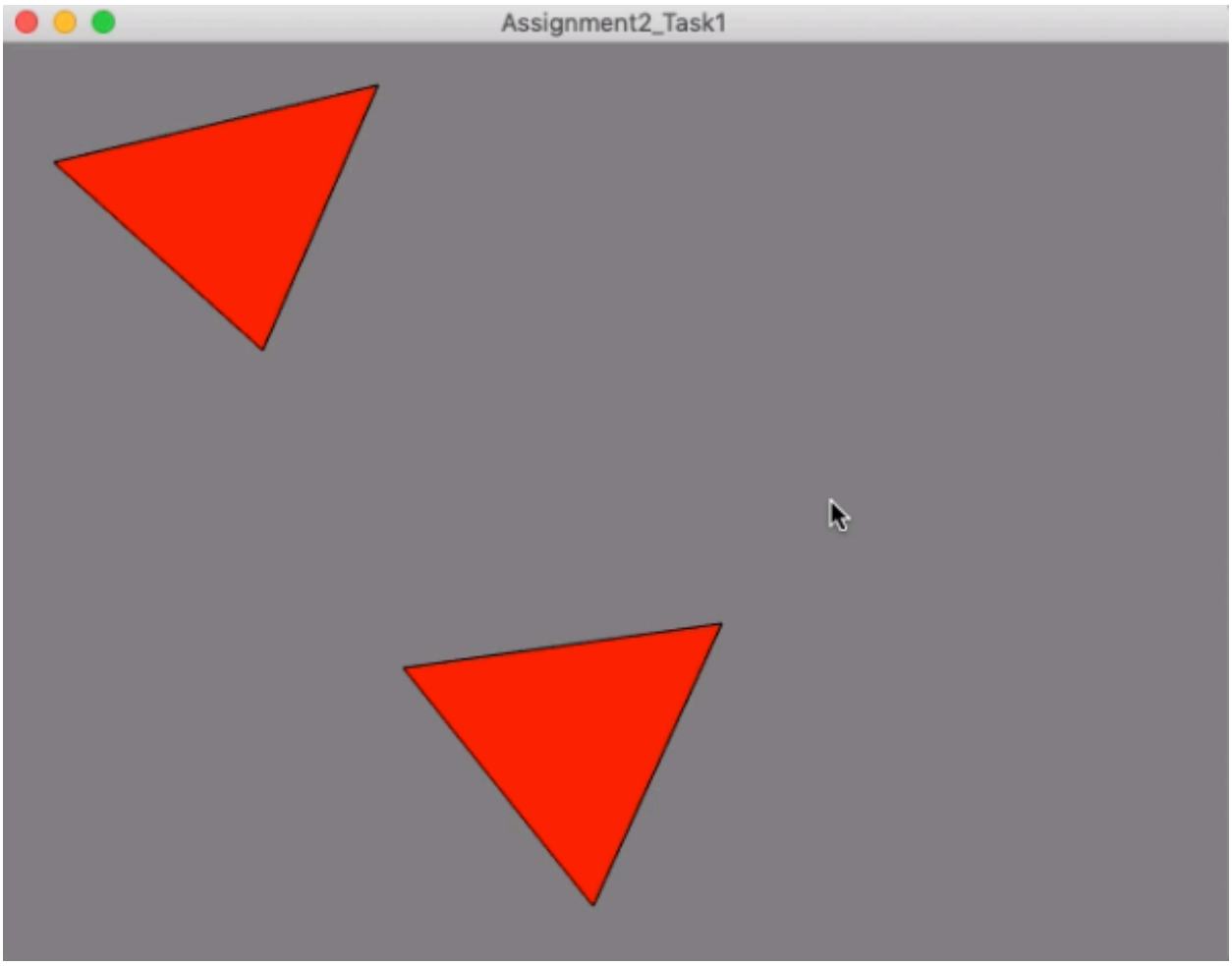


3. Deletion

Before clicking on the triangle to delete:



After clicking on the triangle to delete:



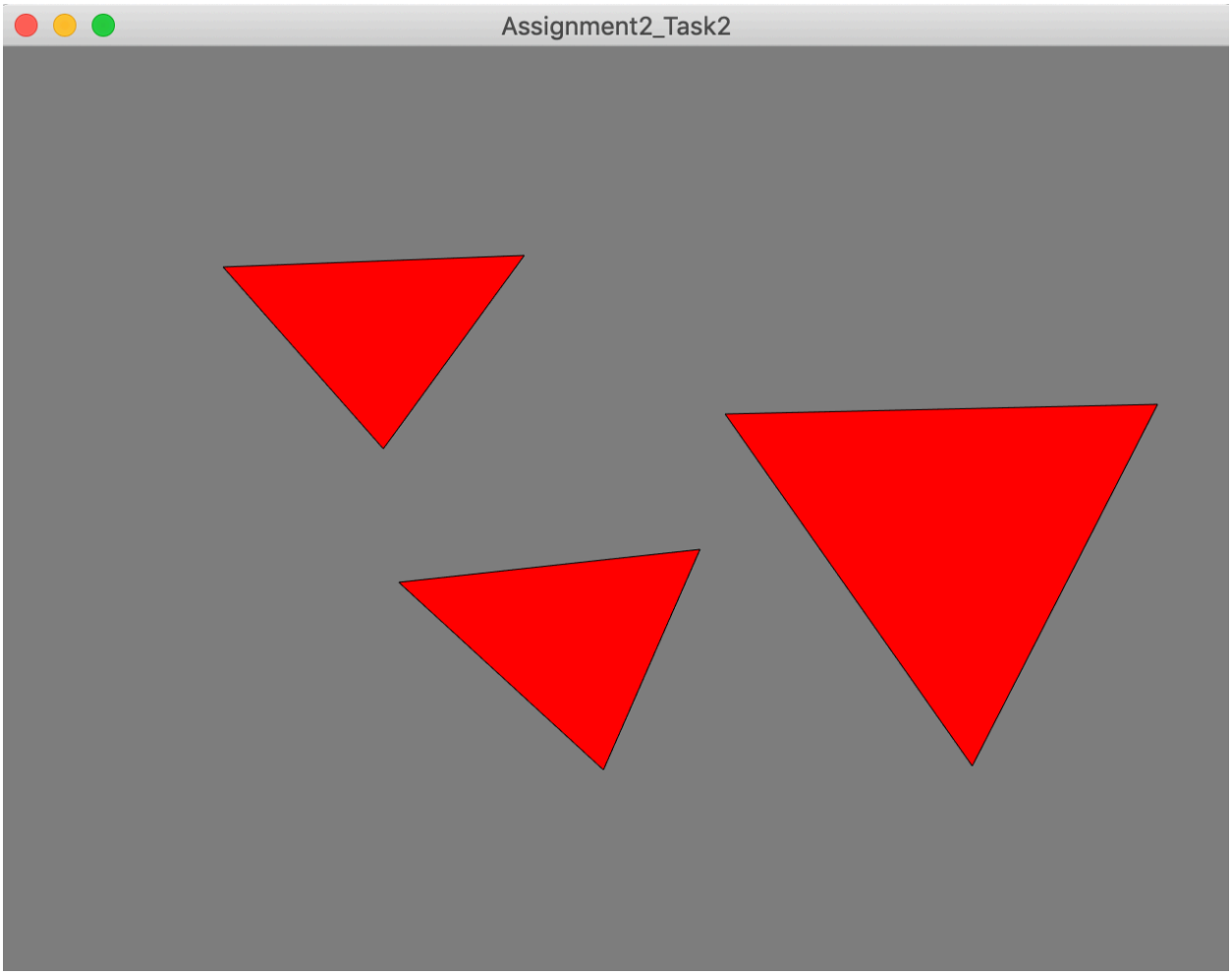
Task2: Rotation/Scale

Implementation:

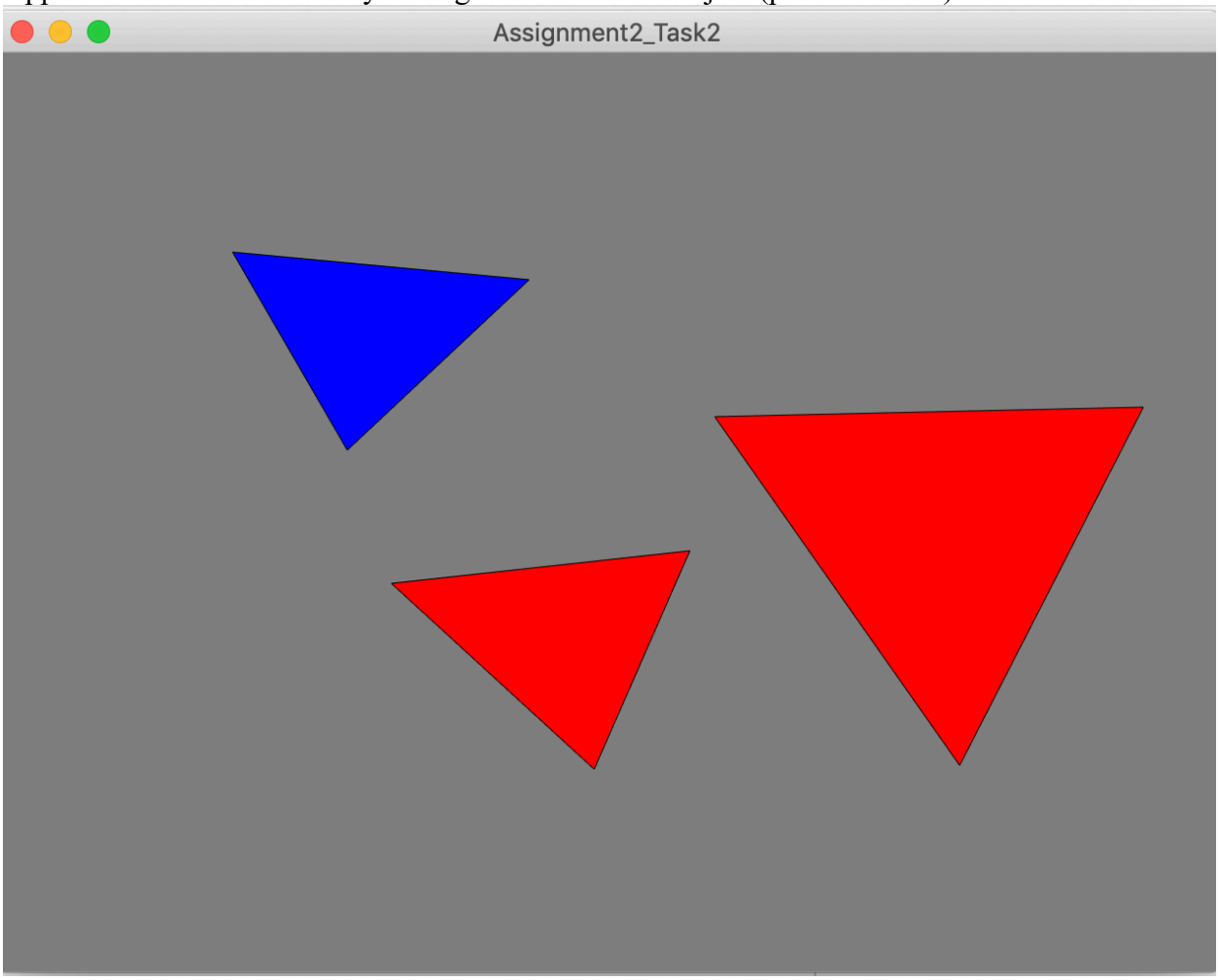
1. Includes most of the steps from task1and user must be in the translation mode for this task.
2. Selecting the triangle is more like task1 translation mode but the only difference is that in this task object stays selected even after the first mouse click release.
3. Respective key strokes ‘h’, ‘j’, ‘k’ and ‘l’ are detected using key_callback event
4. Rotation and scaling algorithms are used according to the requested functionality through key strokes.
5. To perform rotation and scaling around the barycenter, first the barycenter (px, py) of a triangle/object is found using the written algorithm (centroid_of_triangle(float v0x, float v0y, float v1x, float v1y, float v2x, float v2y, float &px, float &py)).
    a. Scaling: translate(px, py) * scale(1.25) * translate(-px, -py)
    b. Rotation: translate(px, py) * rotate(-10) * translate(-px, -py)
6. Only view matrix is updated but not the vertices values in V matrix based on applied transformation. So, vertex shader takes the responsibility of updating gl_Position with the requested/required transformation (as expected in task8).

Result:

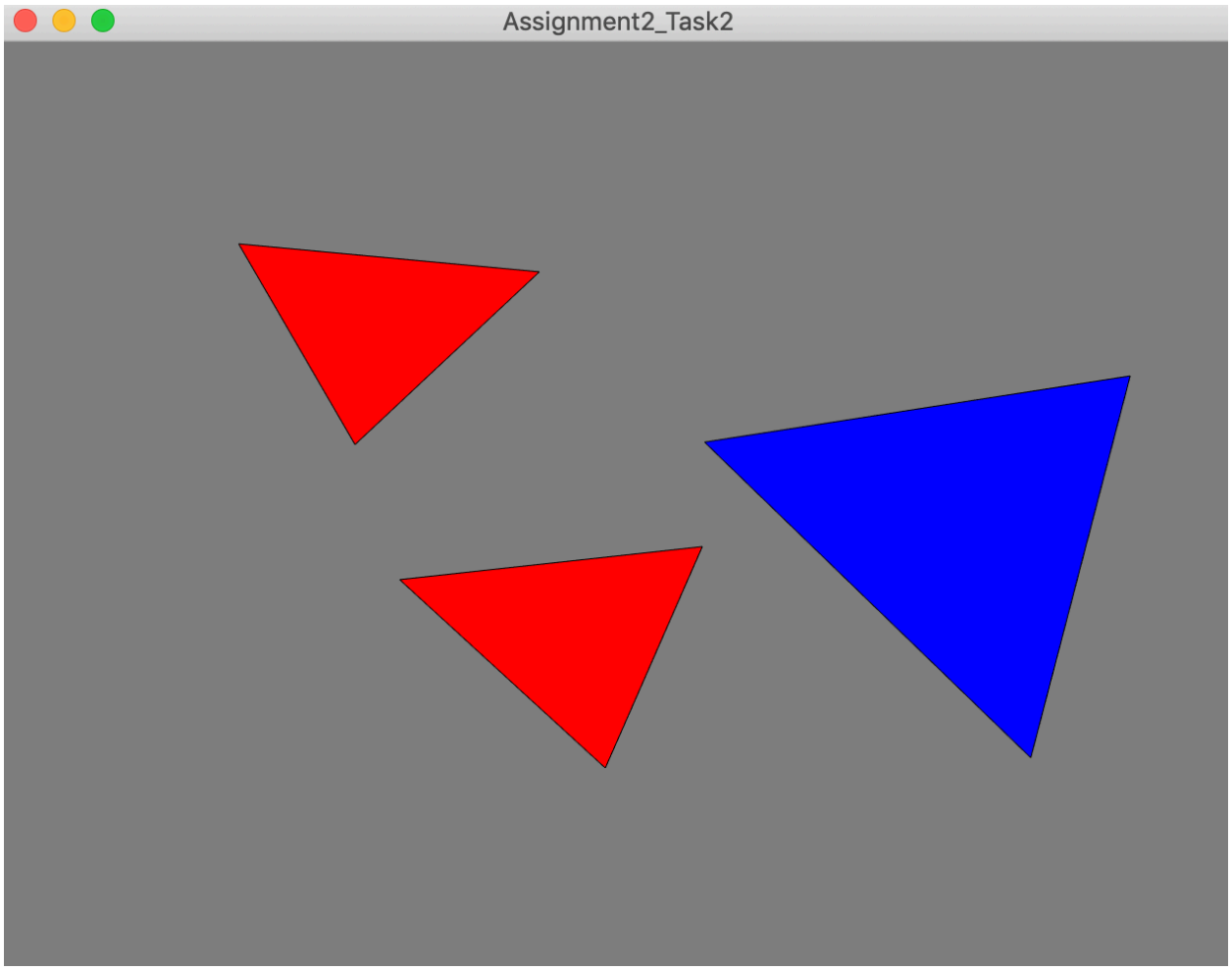
Original (before transformation):



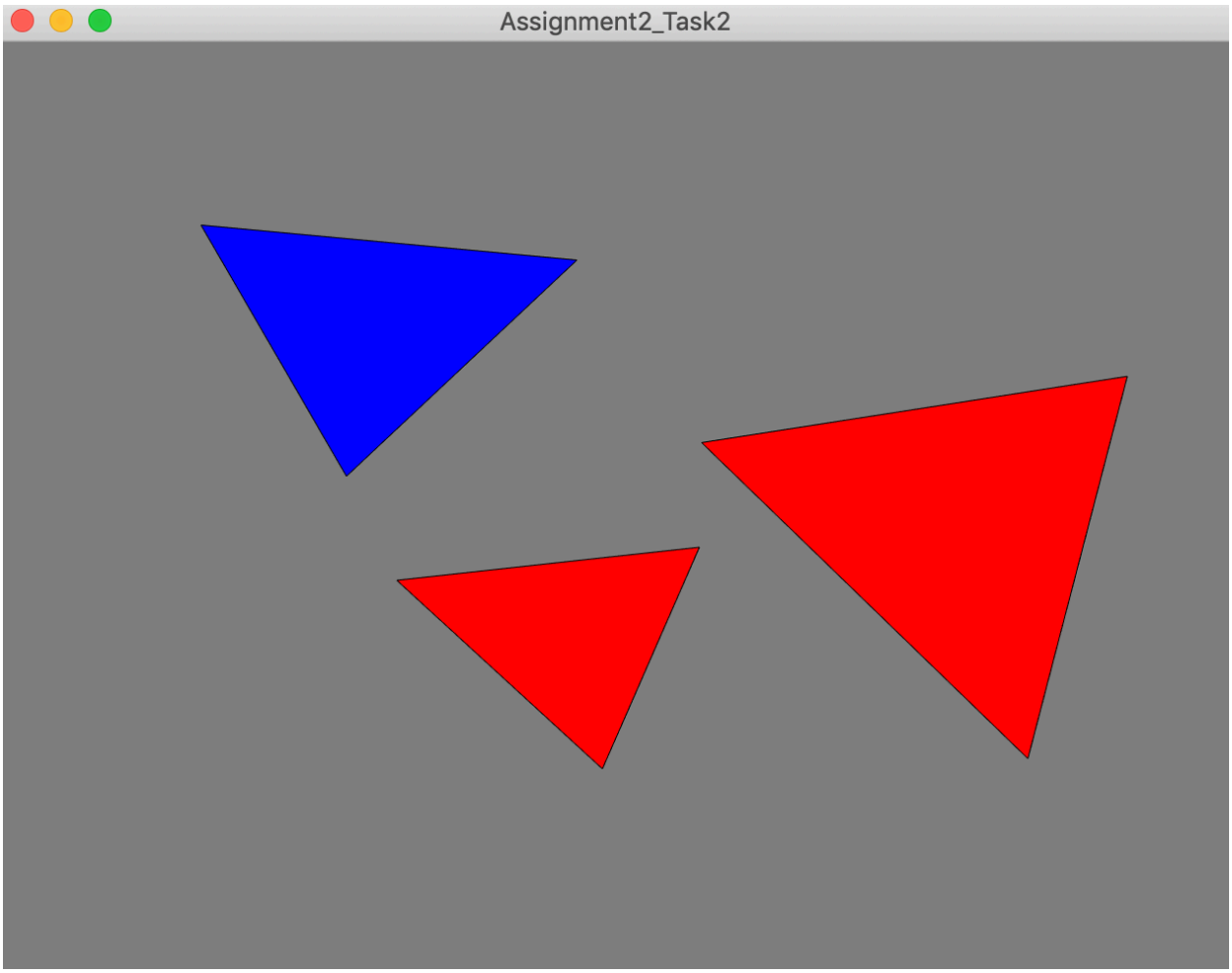
Applied clockwise rotation by 10 degrees on selected object (press ‘h’ once):



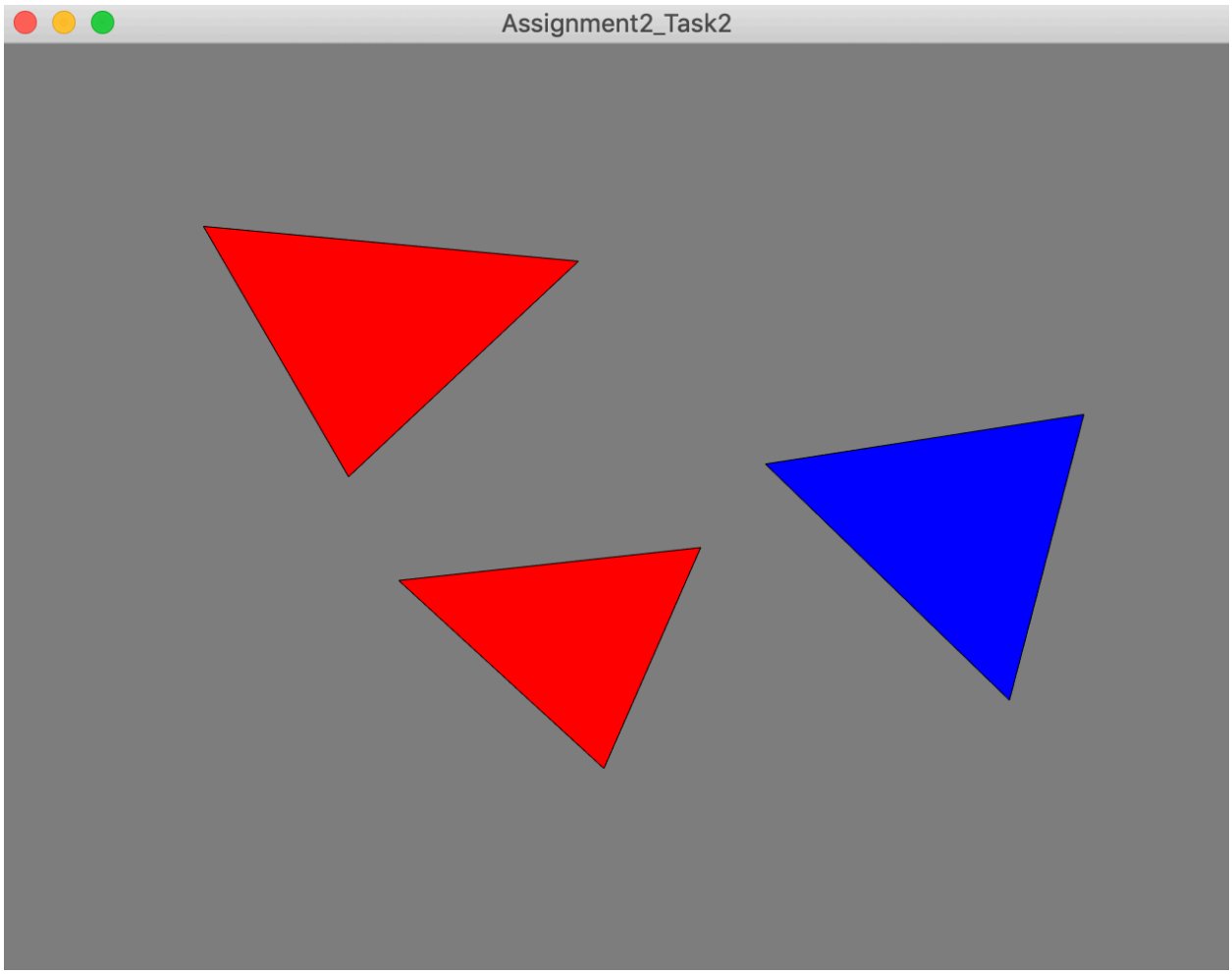
Applied anti-clockwise rotation by 10 degrees on the selected object (press ‘j’ once):



Applied scale up by 25% on the selected object (press ‘k’ once):



Applied scale down by 25% on the selected object (press ‘l’ once):



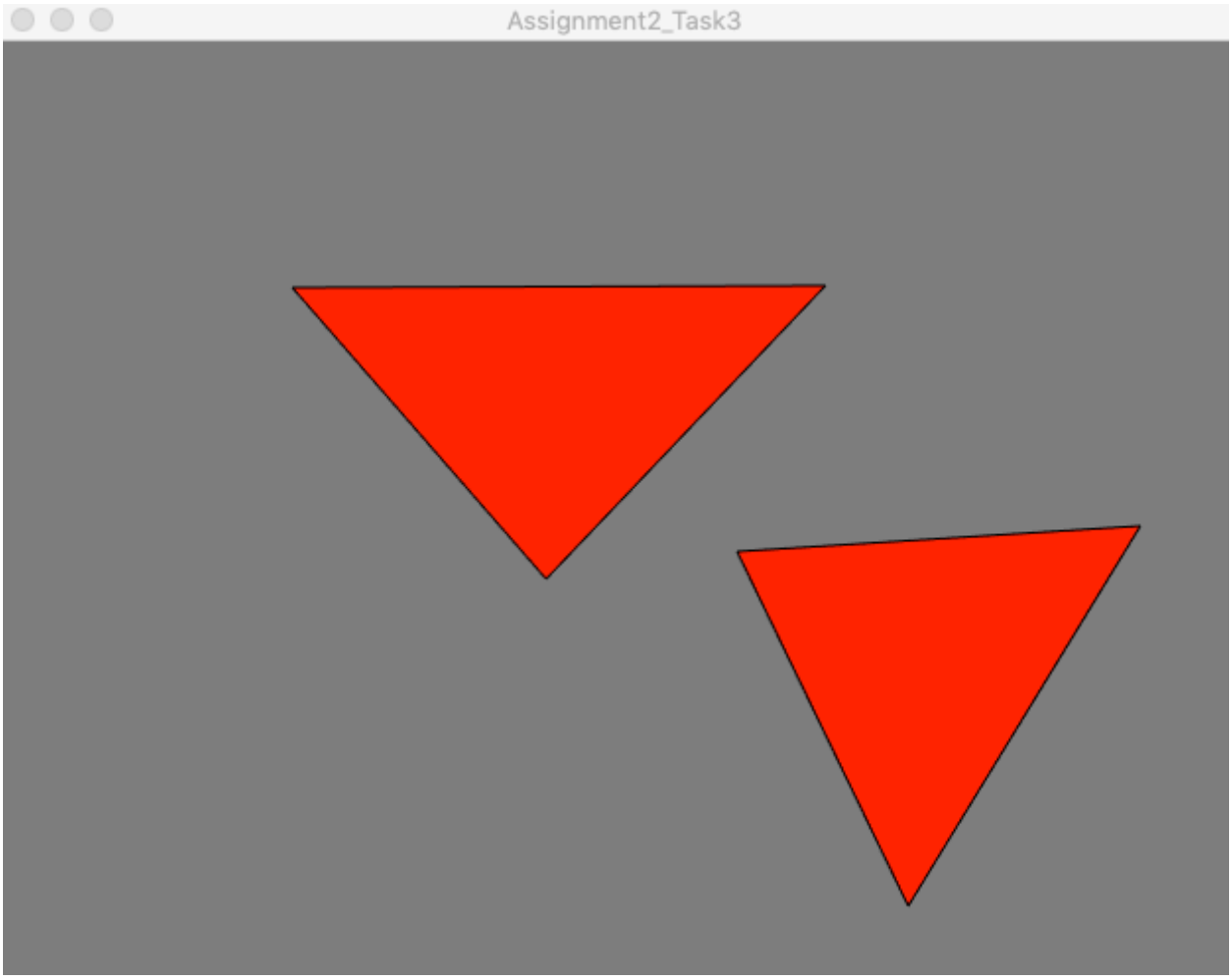
Task3: Rotation/Scale

Implementation:

1. Includes most of the steps from task2 and user must be in the color mode for this task. Color mode is enabled by pressing key ‘c’
2. Once the color mode is enabled, mouse click is tracked by mouse_button_callback and an algorithm (findNearestVertex(float click_x, float click_y)) is written and called to find out the nearest vertex to the mouse click.
3. 12 types of Color RGB values are saved in a matrix ‘colorCode’ and 9 of them are used with the user key strokes from 1-9 respectively. A matrix C is created like V to store color values for the respective vertices. C is updated as per the user actions.
4. A VBO is created i.e., VBO_C to push color values as per the selection by the user to the vertex and fragment shader. VBO_C is eventually updated with C matrix values.
5. Shader use VBO_C values to apply color to the fragments.

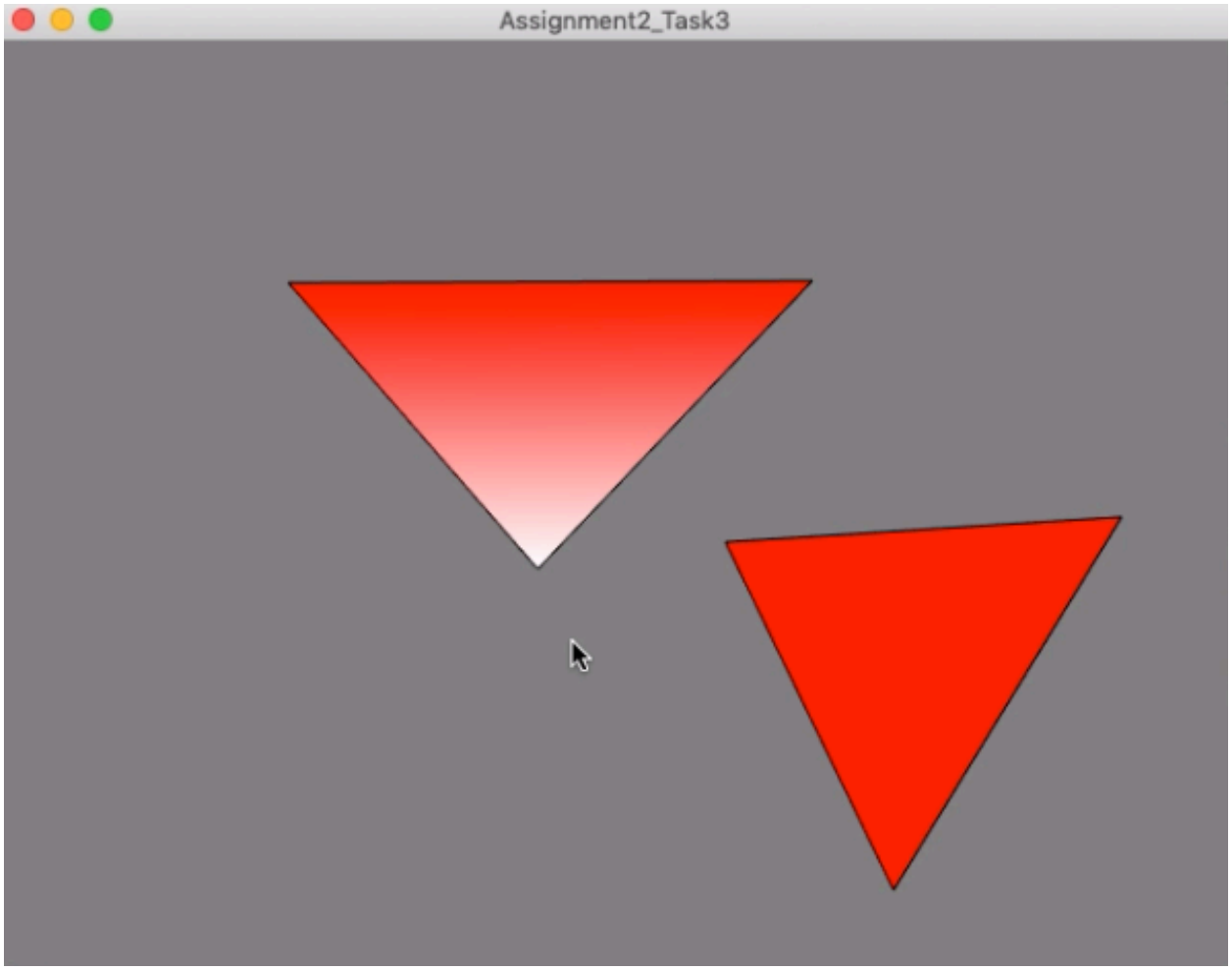
Result:

Before applying colors:

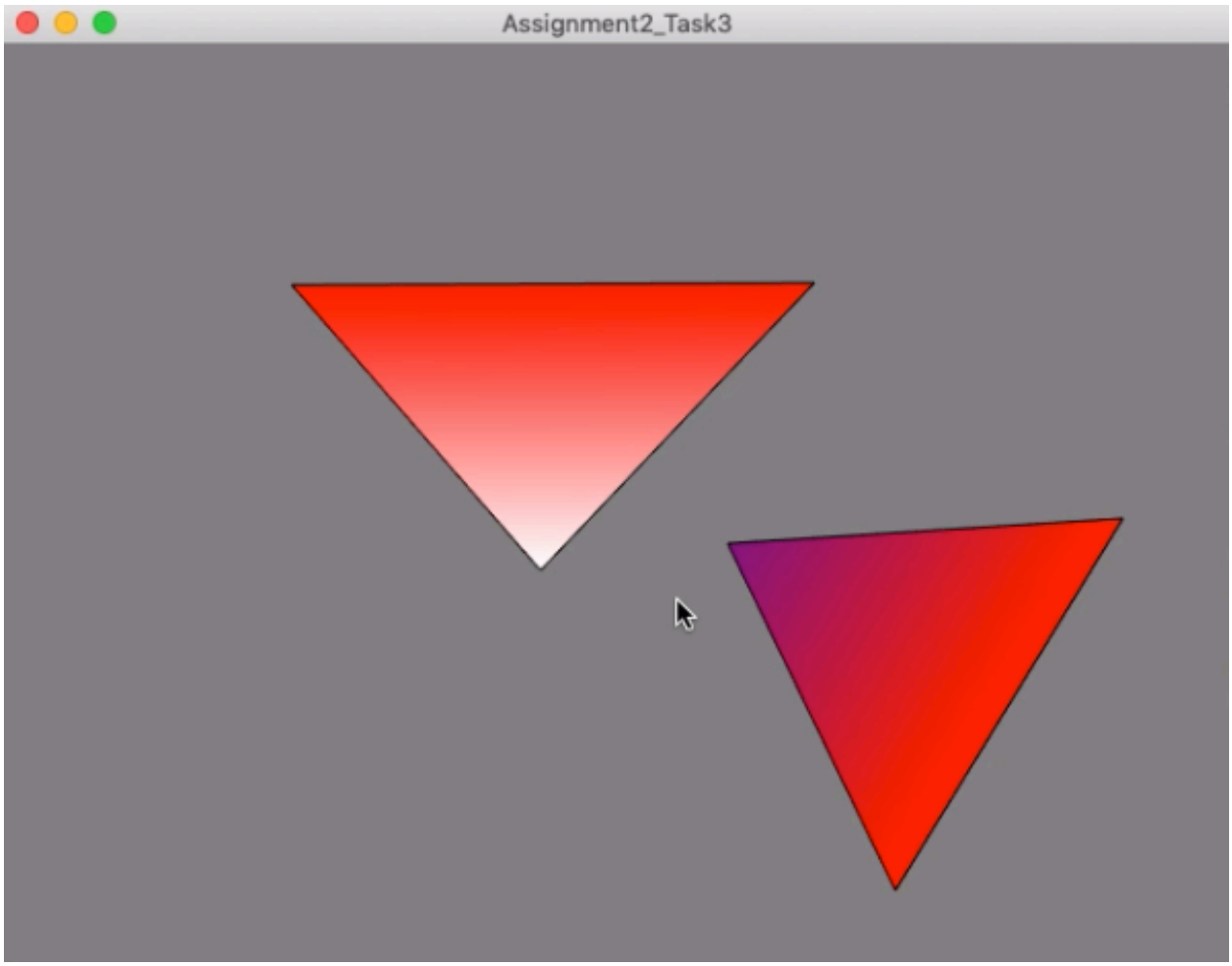


Applying colors:

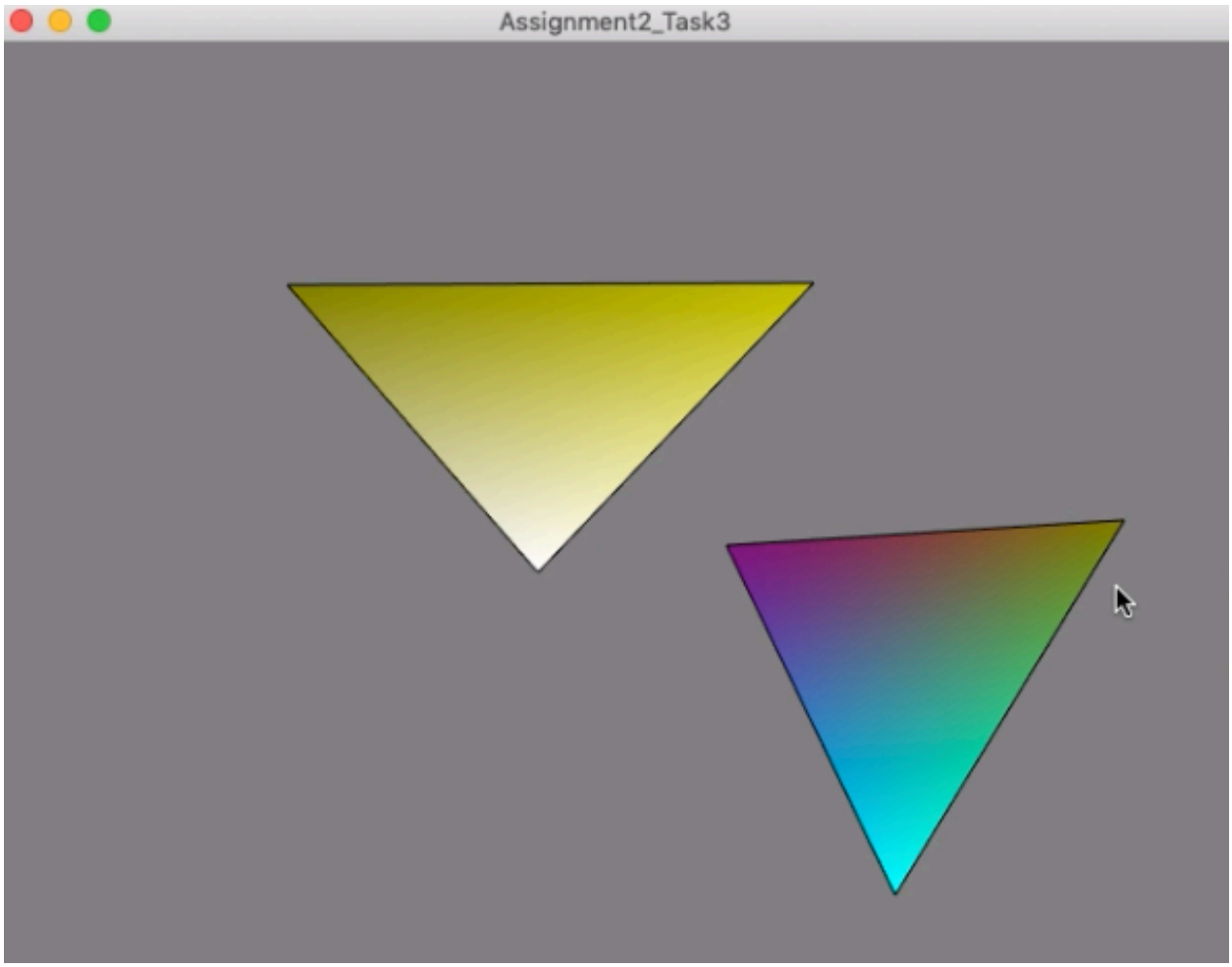
Chose a vertex and applied a color code:



Chose another vertex and applied a different color code:



Applied color codes to many vertices in succedent steps:



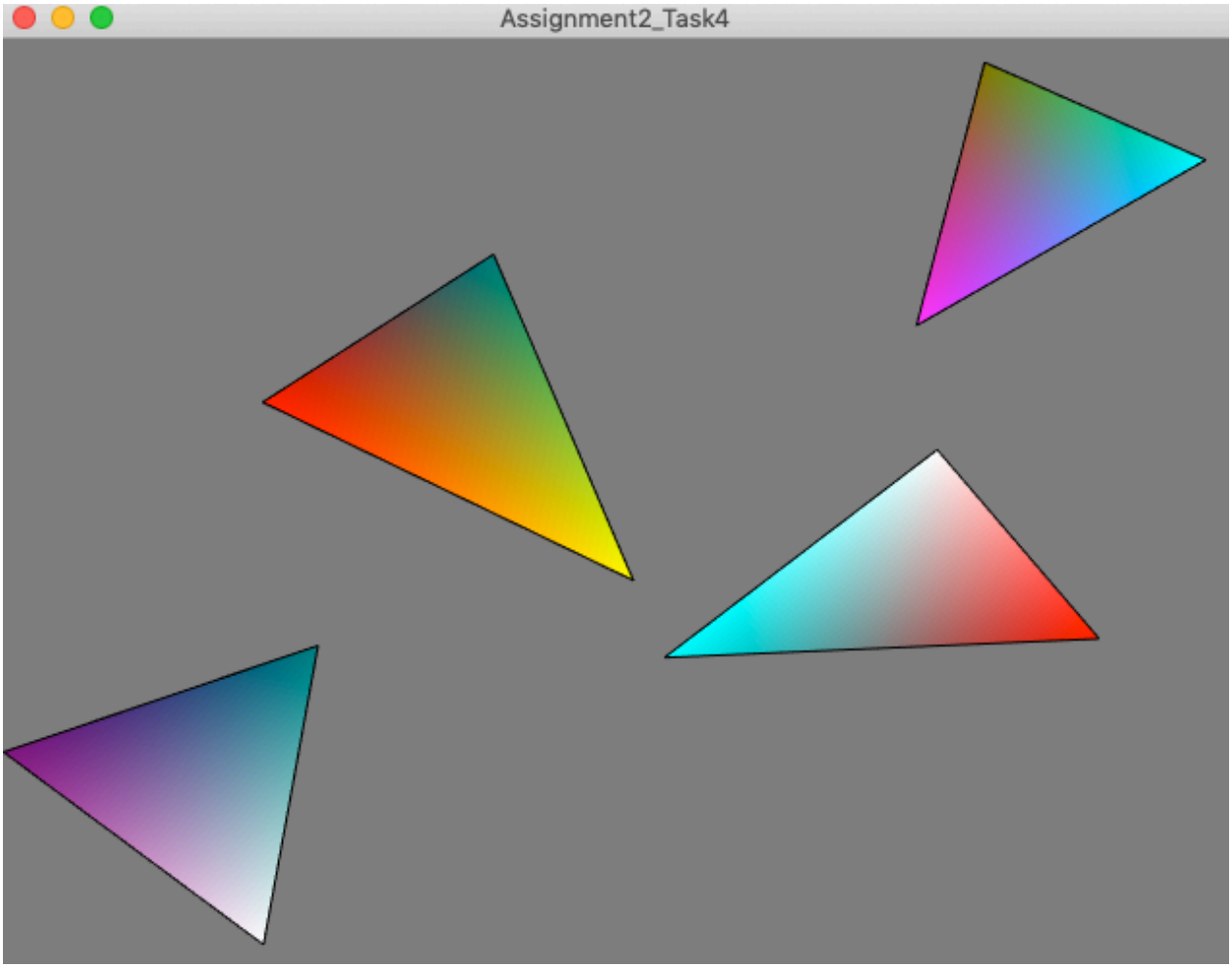
Task4: View control

Implementation:

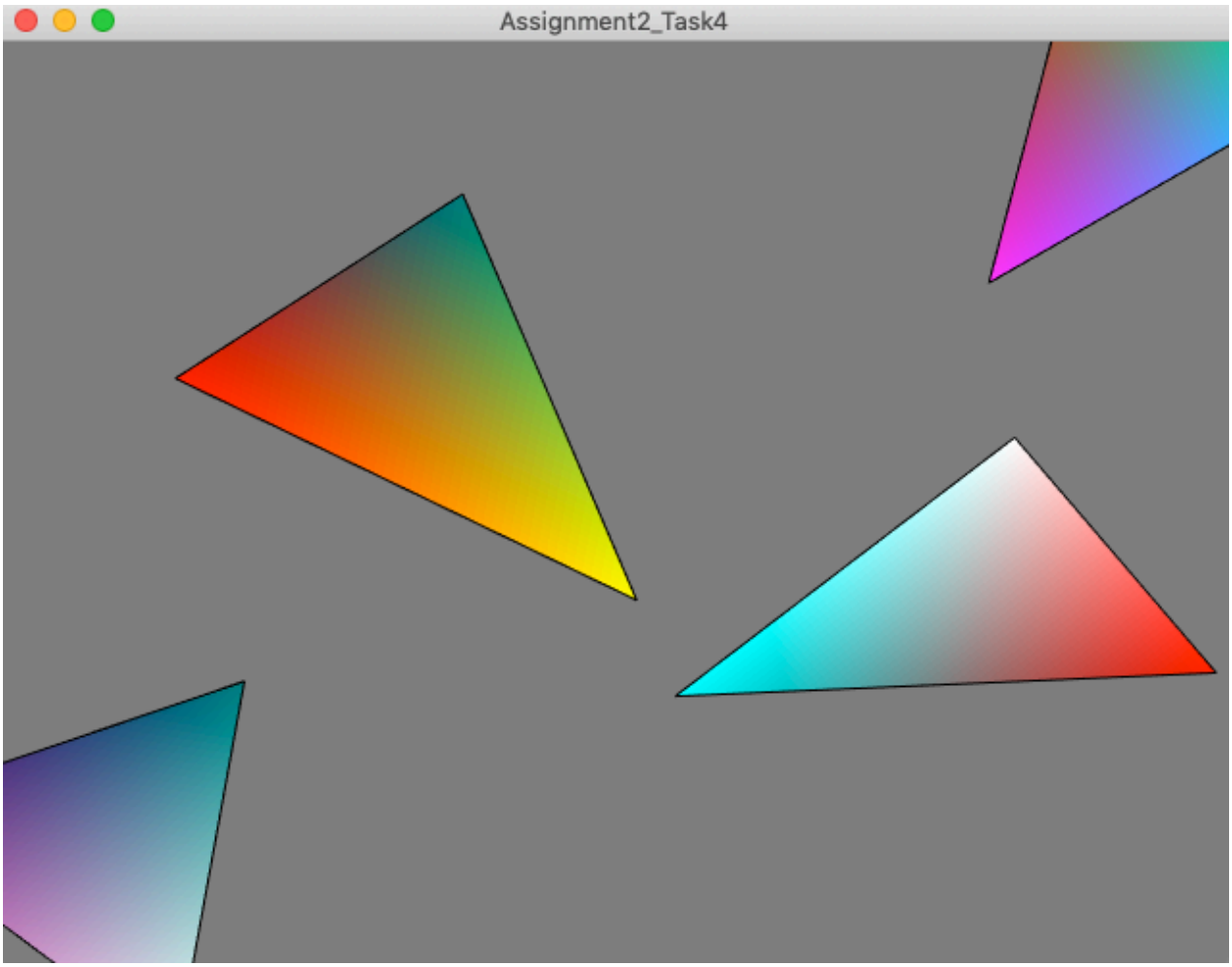
1. Includes most of the steps from task3
2. A matrix is created named ‘totalView ‘ to hold view control for the whole screen.
3. Keys ‘+’, ‘-‘, ‘w’, ‘a’, ‘s’, and ‘d’ are detected using key_callback and used to control totalview values as per the instructions from requirements document.
4. Like task2, view matrix is changed but not the vertices values of the objects. So, shaders take the responsibility of transforming the screen according to the user request.
5. screen coordinates are multiplied with the inverse of the the view matrix to ensure that the user interaction will adapt to the current view.

Result:

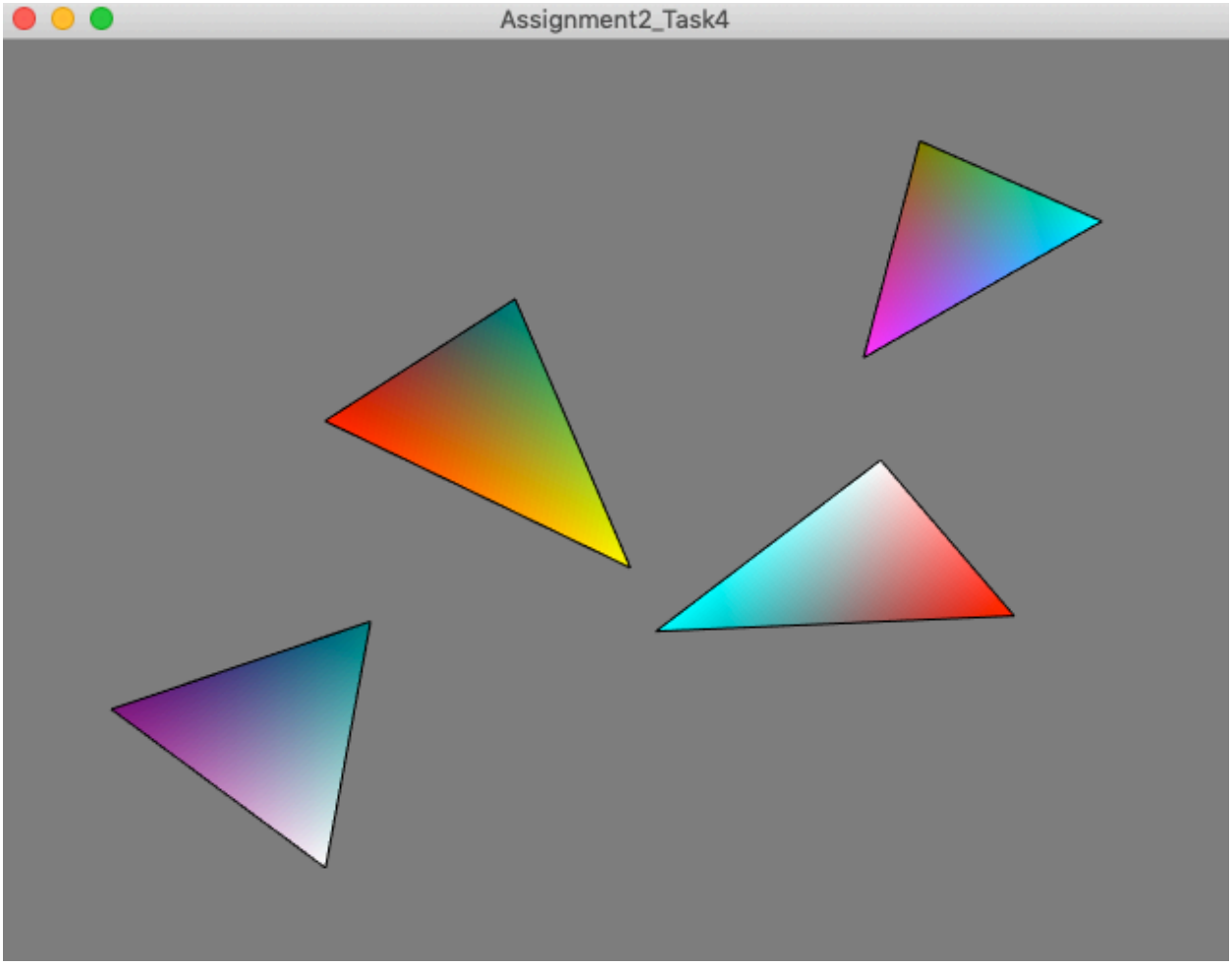
Original (before applying any tranformations):



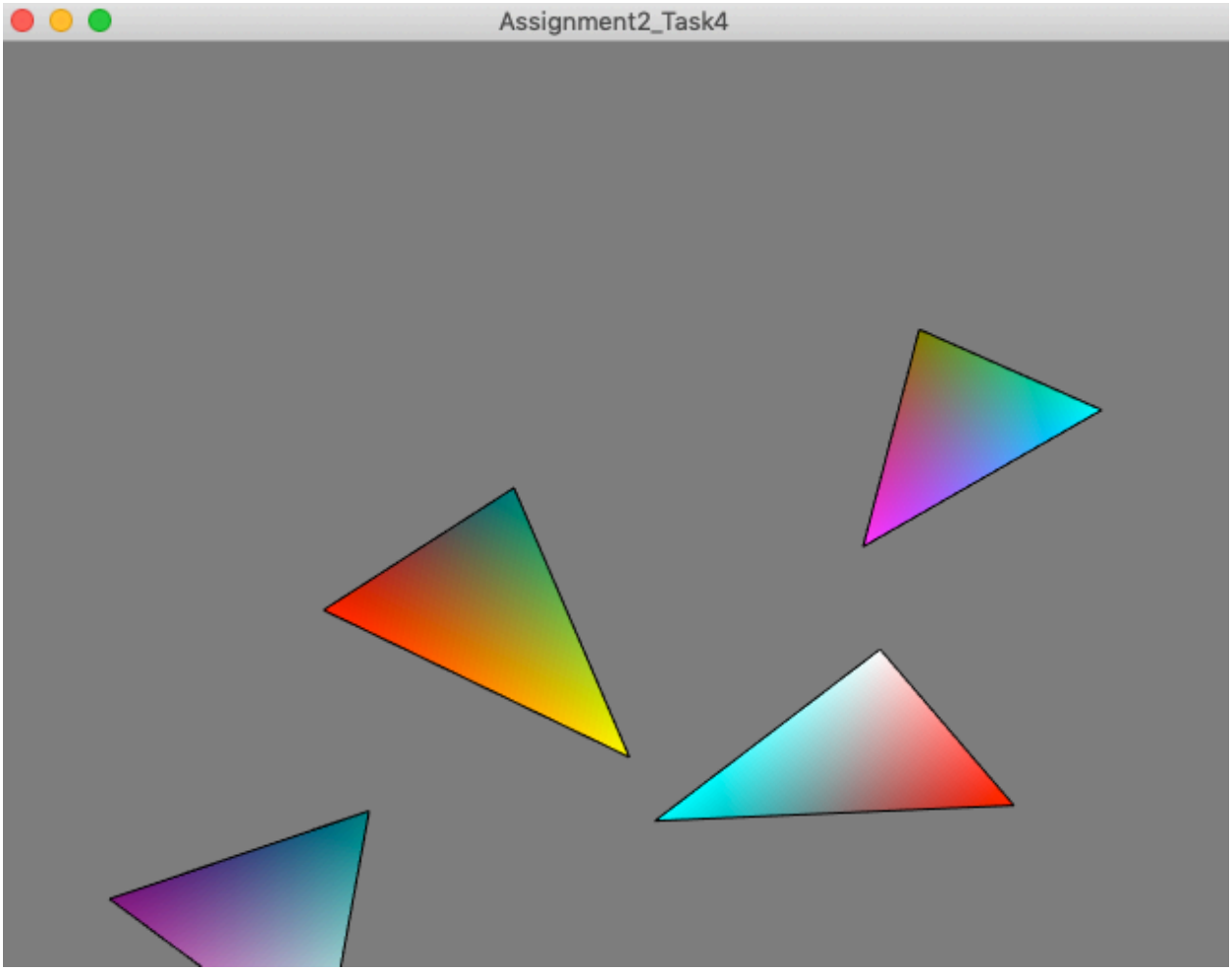
Applied 20% zoom increase in the center of the screen (press '+' once):



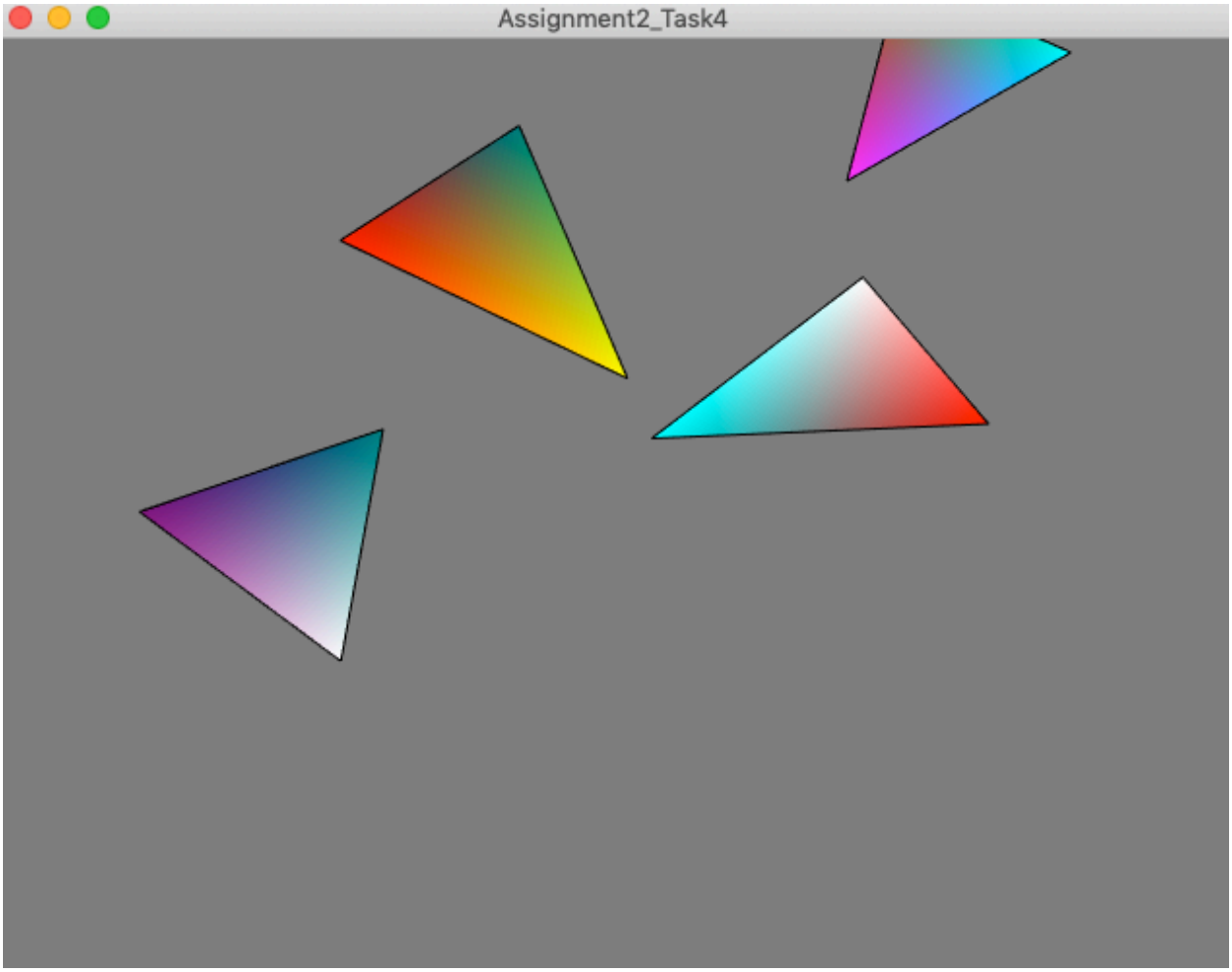
Applied 20% zoom decrease in the center of the screen from the original (i.e., press '-' twice from the above state): *for a better understanding*



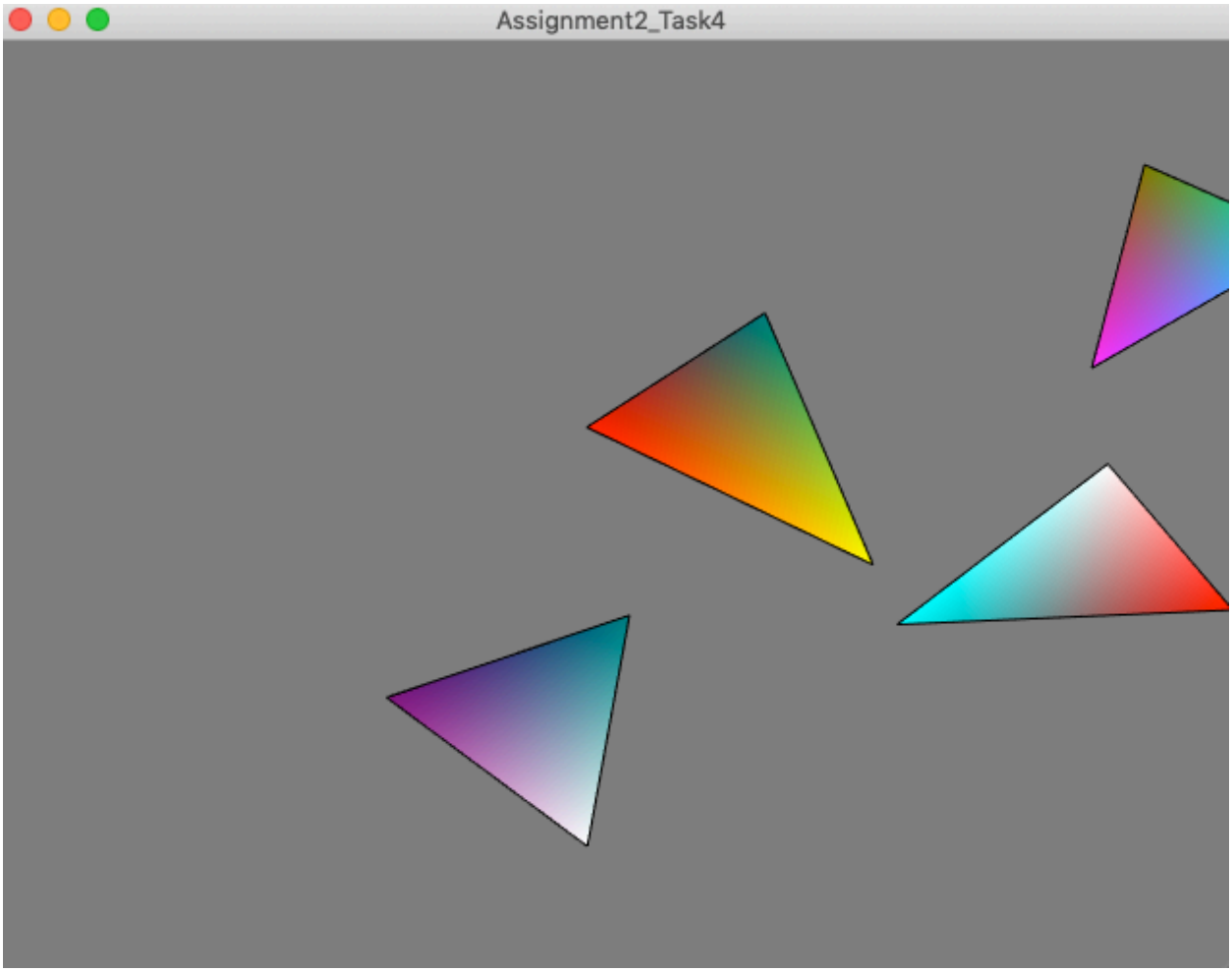
Applied 20% pan down to the visible portion of the screen from the previous state after zoom down by 20% (i.e., press ‘w’ once):



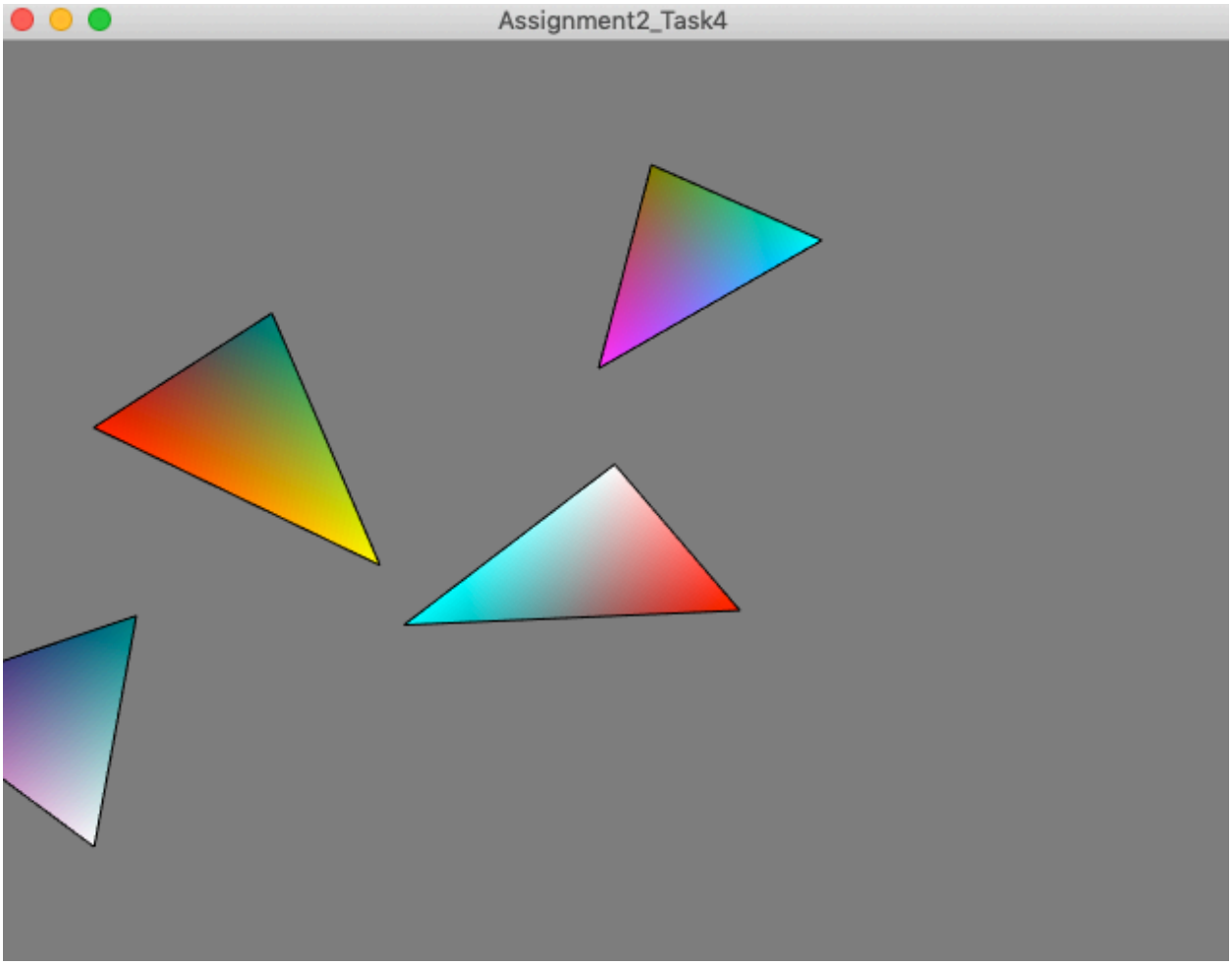
Applied 20% pan down to the visible portion of the screen from the original position after zoom down by 20% (i.e., press ‘s’ twice from the previous state): *for a better understanding*



Applied 20% pan right to the visible portion of the screen from the original position after zoom down by 20% (i.e., press ‘w’ and press ‘a’ from previous state): *for a better understanding*



Applied 20% pan left to the visible portion of the screen from the original place after zoom down by 20% (i.e., press ‘d’ twice from the previous state): *for a better understanding*



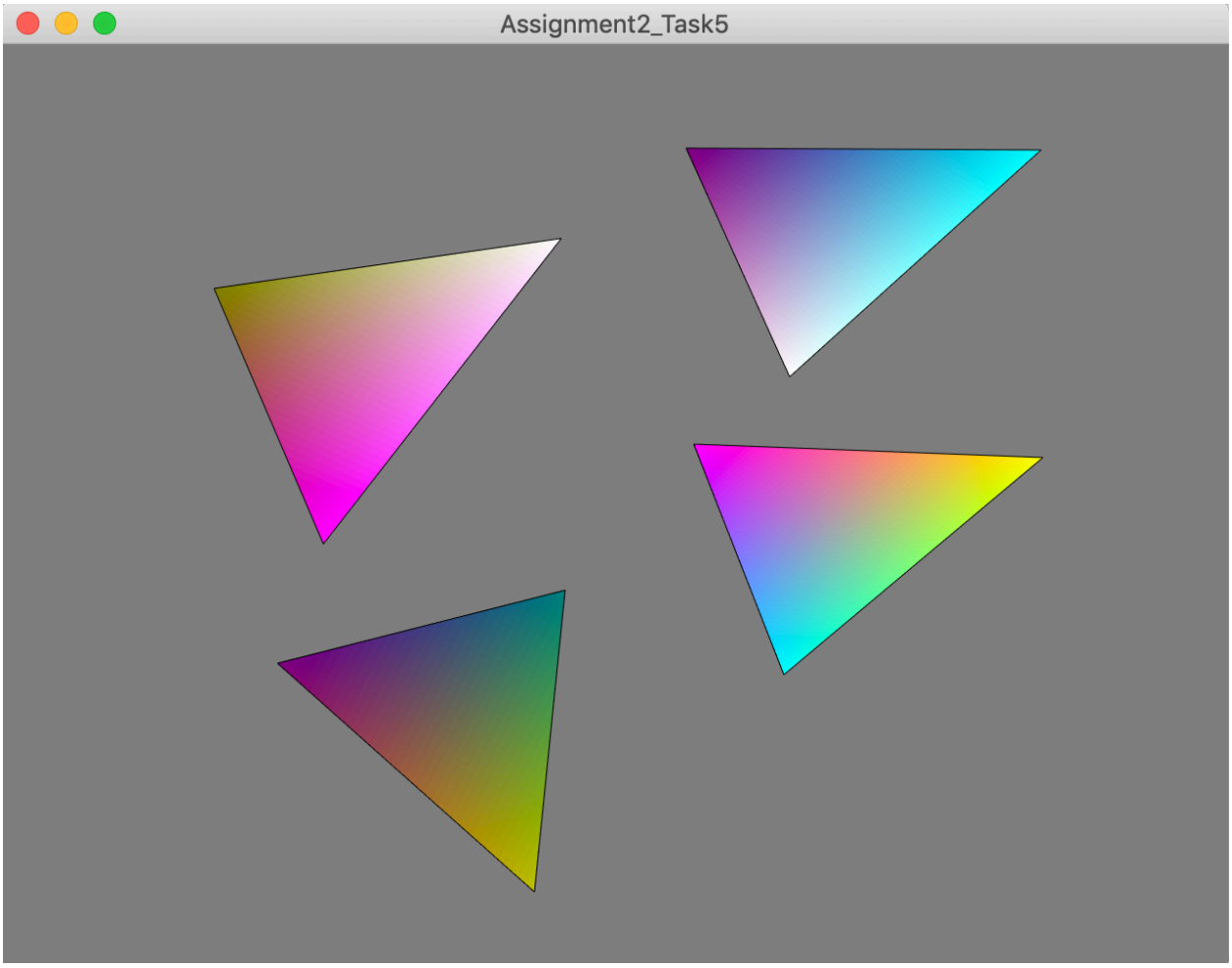
Task5: Add keyframing

Implementation:

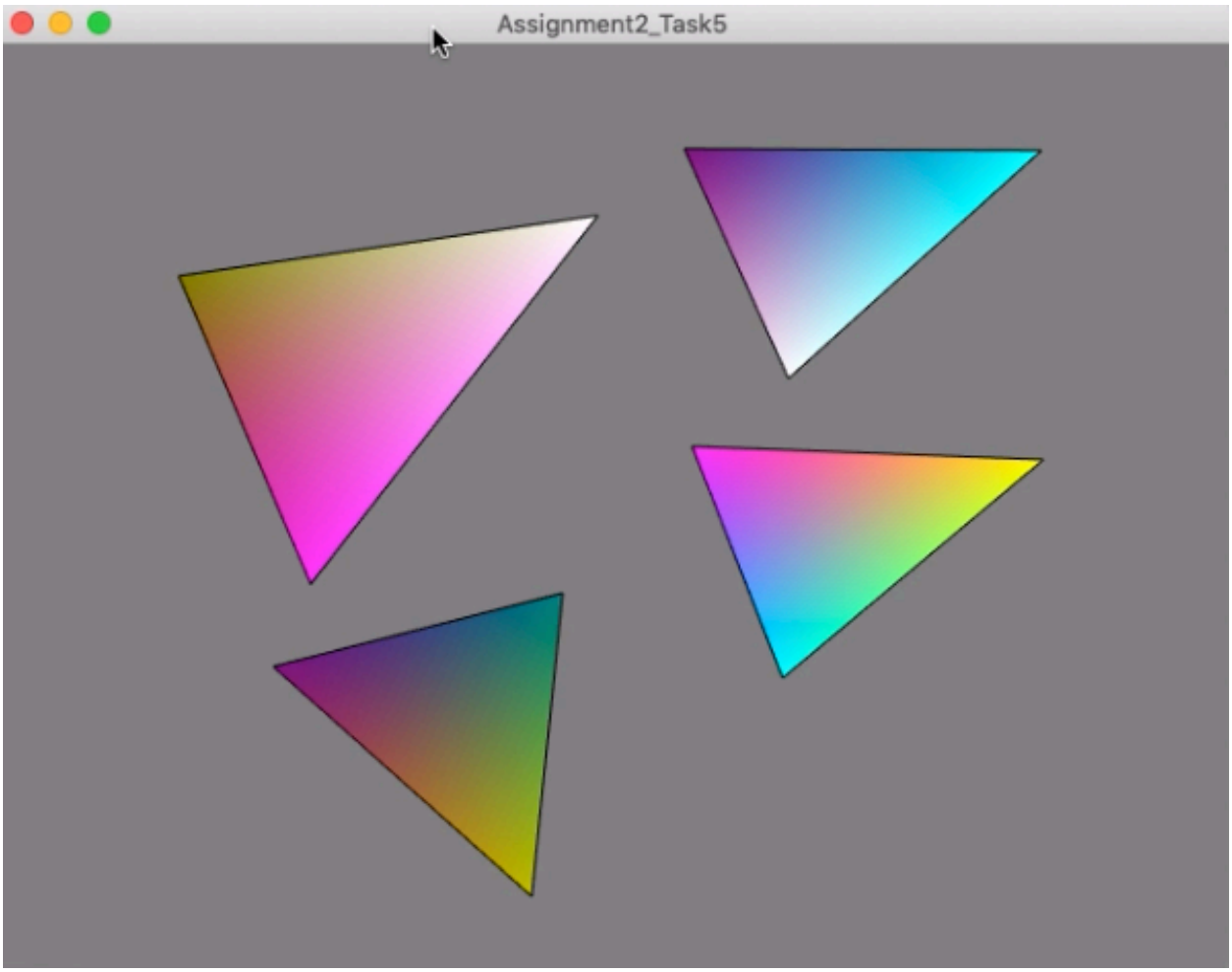
1. Includes most of the steps from task4 and user has to press key ‘z’ to apply ‘scale’ animation or press key ‘x’ to apply ‘rotation and zoom in/out’ animation.
2. An algorithm(`interpolateKeyframe`) for interpolation of vertices is created.
3. Two animations logic is created:
    a. `initiateScaleKeyframe` will create key frames for the objects with a 100% scale of the original
    b. `initiateRotationKeyframe` will create key frames for the objects with a 180-degree rotation and a zoom in/out animation.
4. A timer (`std::chrono::high_resolution_clock`) is used to create animation effect. Keyframes are updated every 0.1 s with the interpolate values to the vertices. Matrix V and VBO are updated with interpolated vertices values.
5. It is also easy and follows the same pattern to update the view instead of the vertices so that shaders take the responsibility of updating gl_Position with the same framework as discussed above.
6. Animation is applied to objects one after the other in their created sequence. And it automatically repeats in the same sequence forever until the window is closed or any other key is pressed.

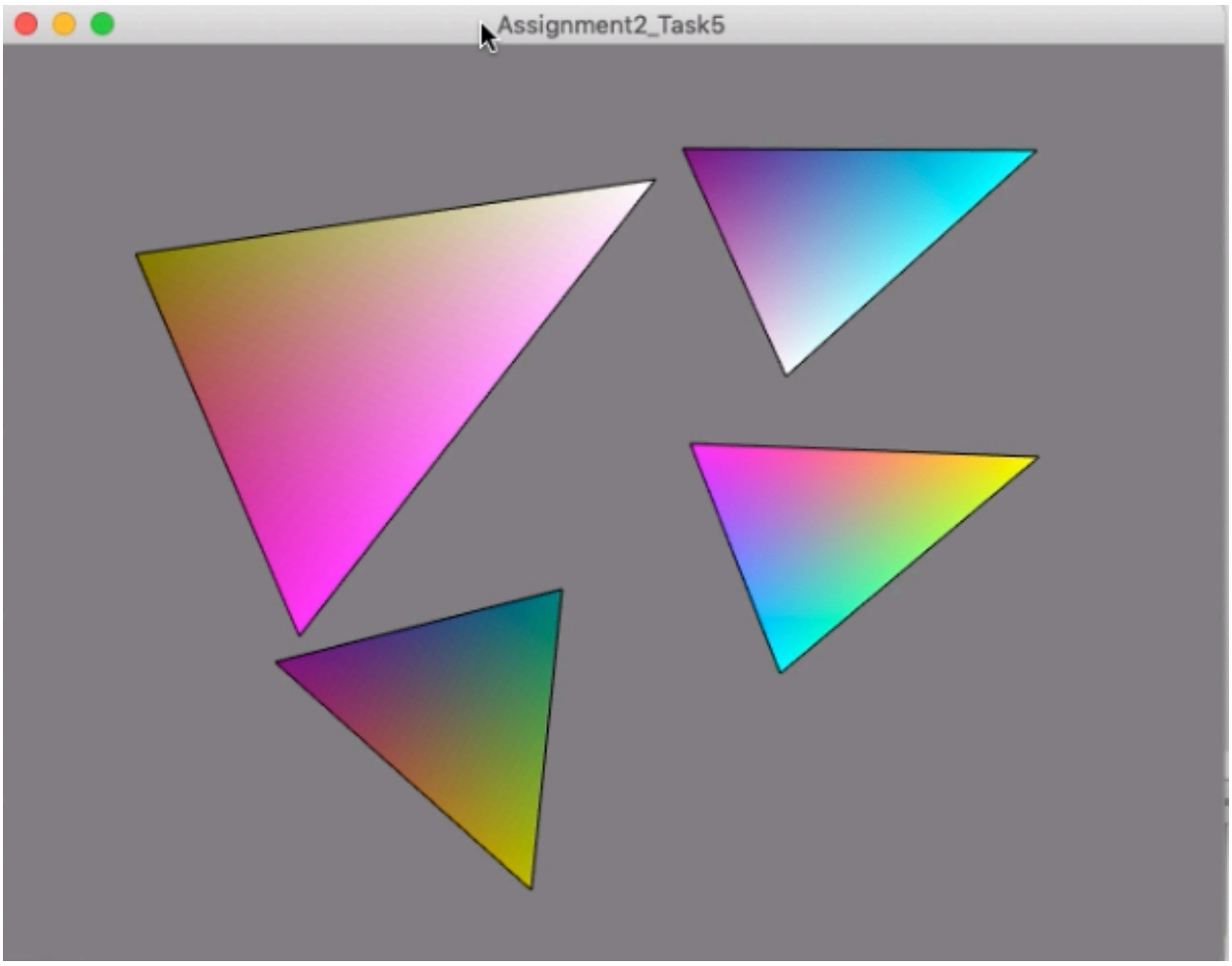
Result:

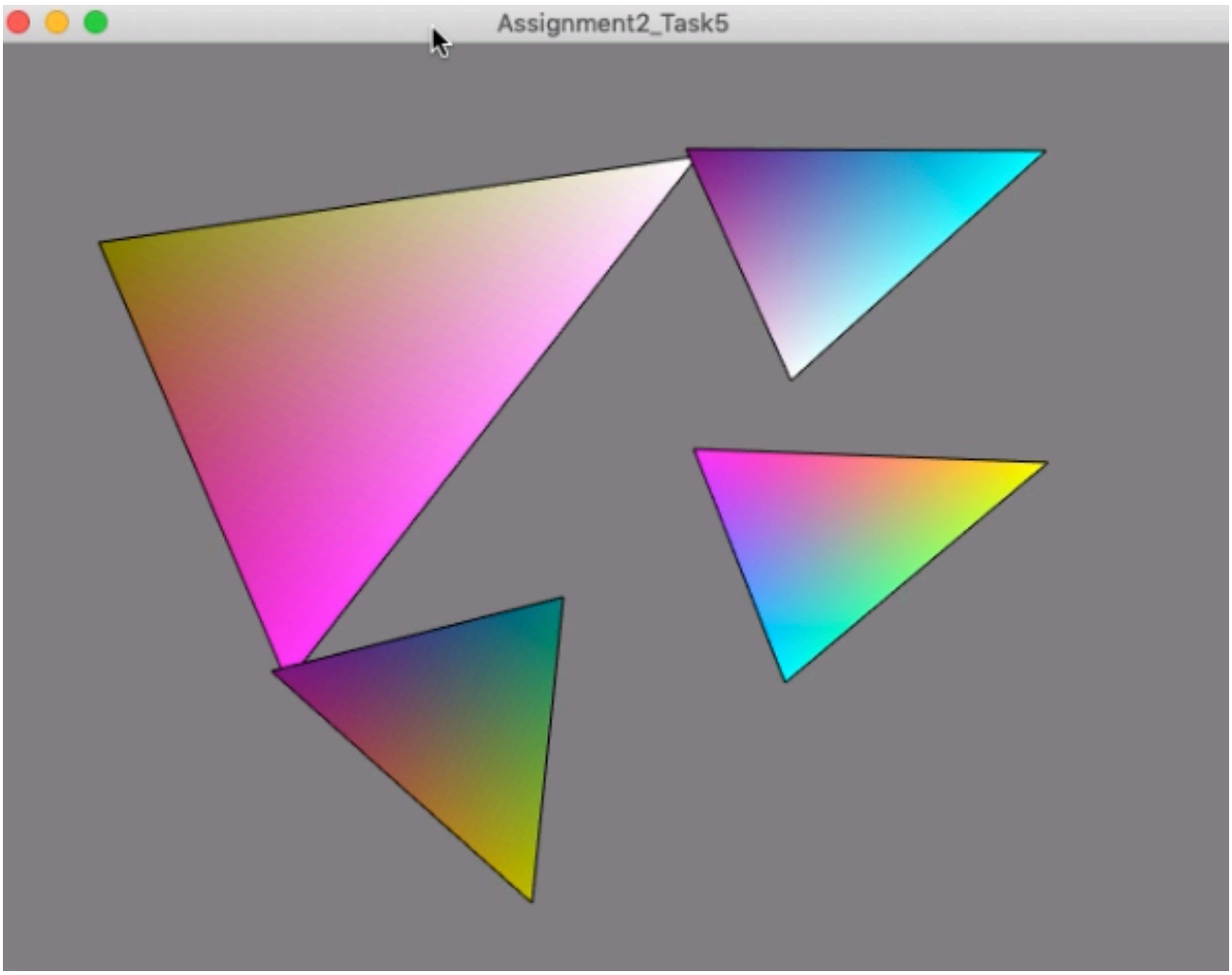
Original(before applying animations):



Scale animation (press ‘z’):

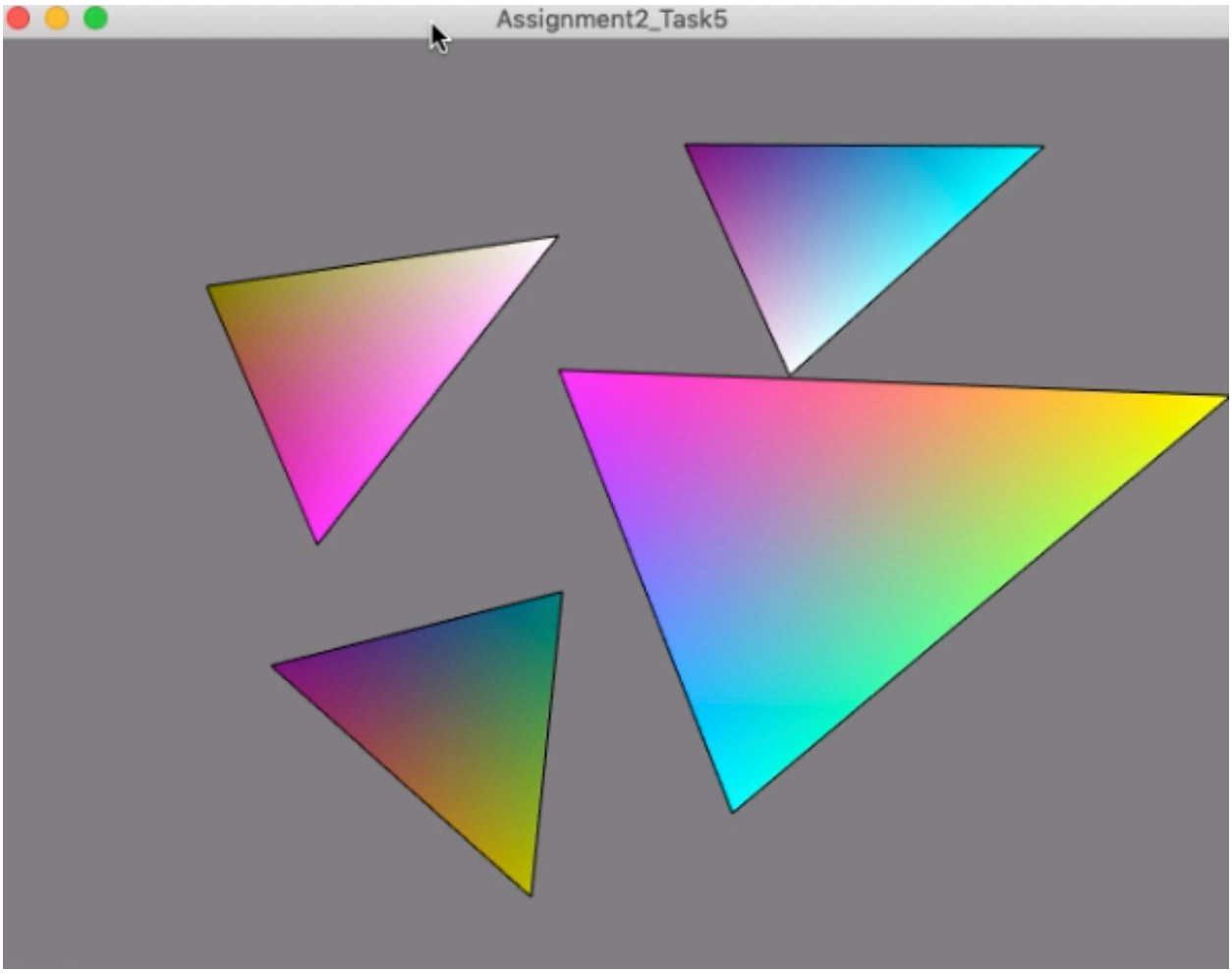



Assignment2_Task5

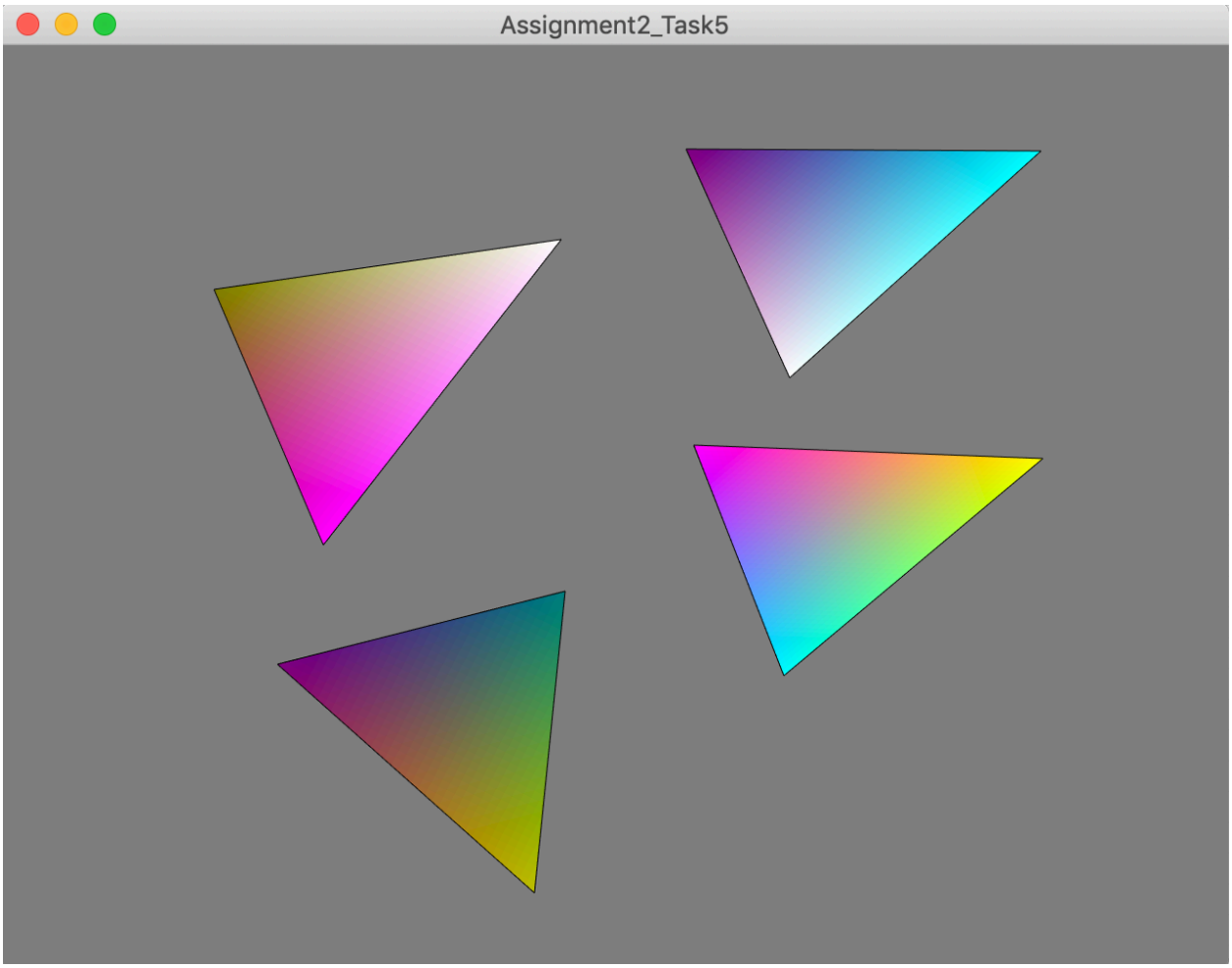


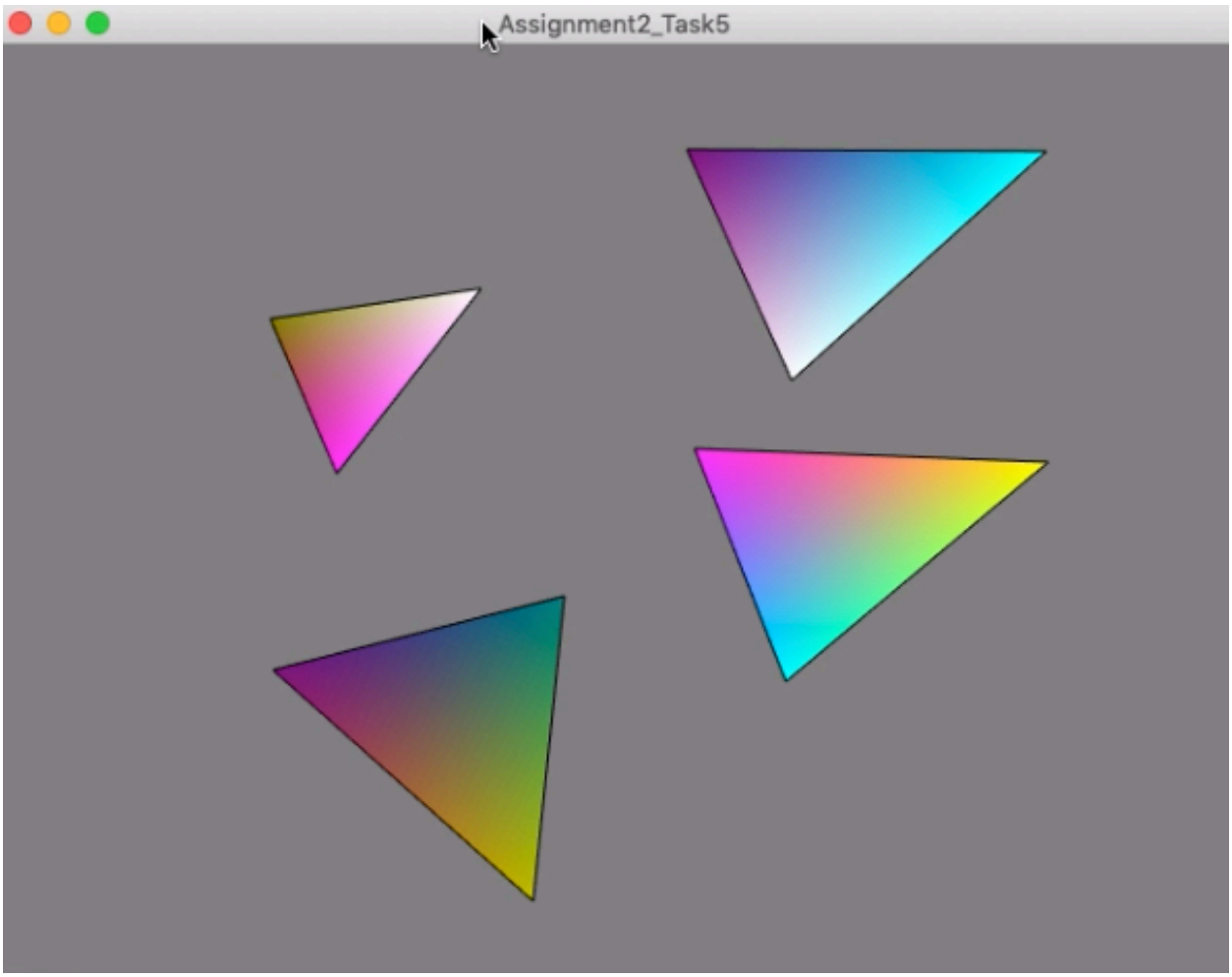
It’s tough to capture pictures in between the animation effect as it changes a frame at 0.1s rate.

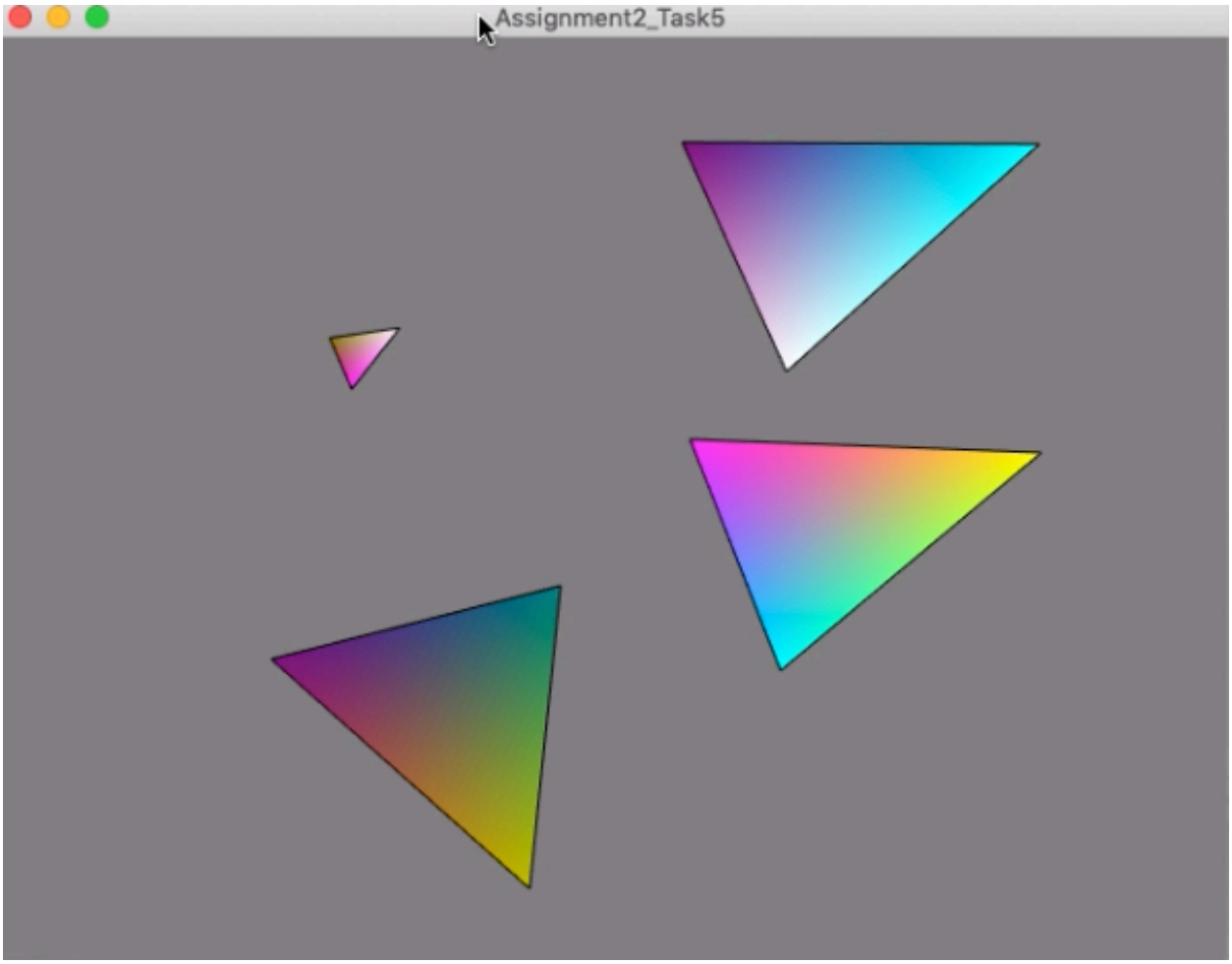
It continues for other objects in the screen:

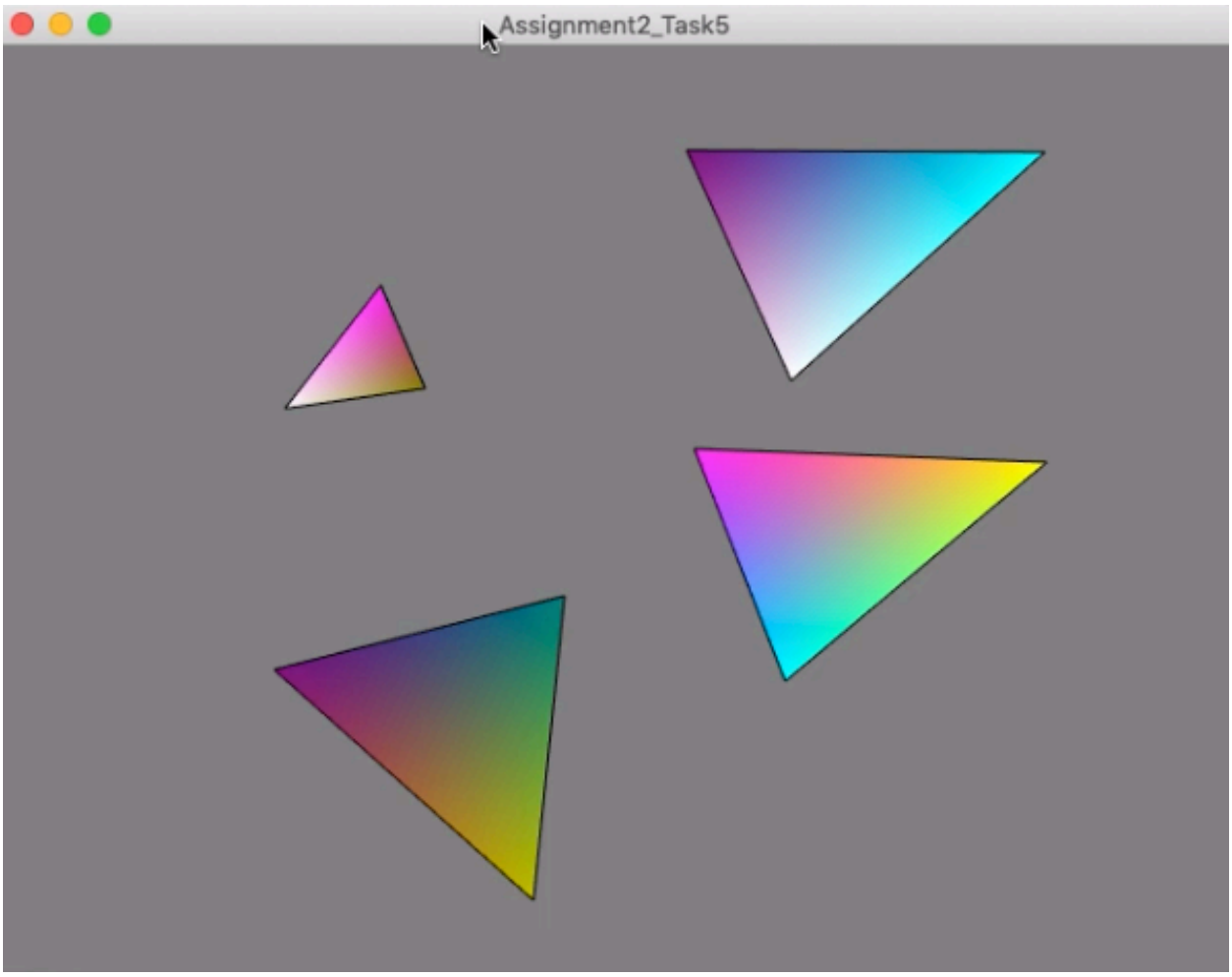

Assignment2_Task5

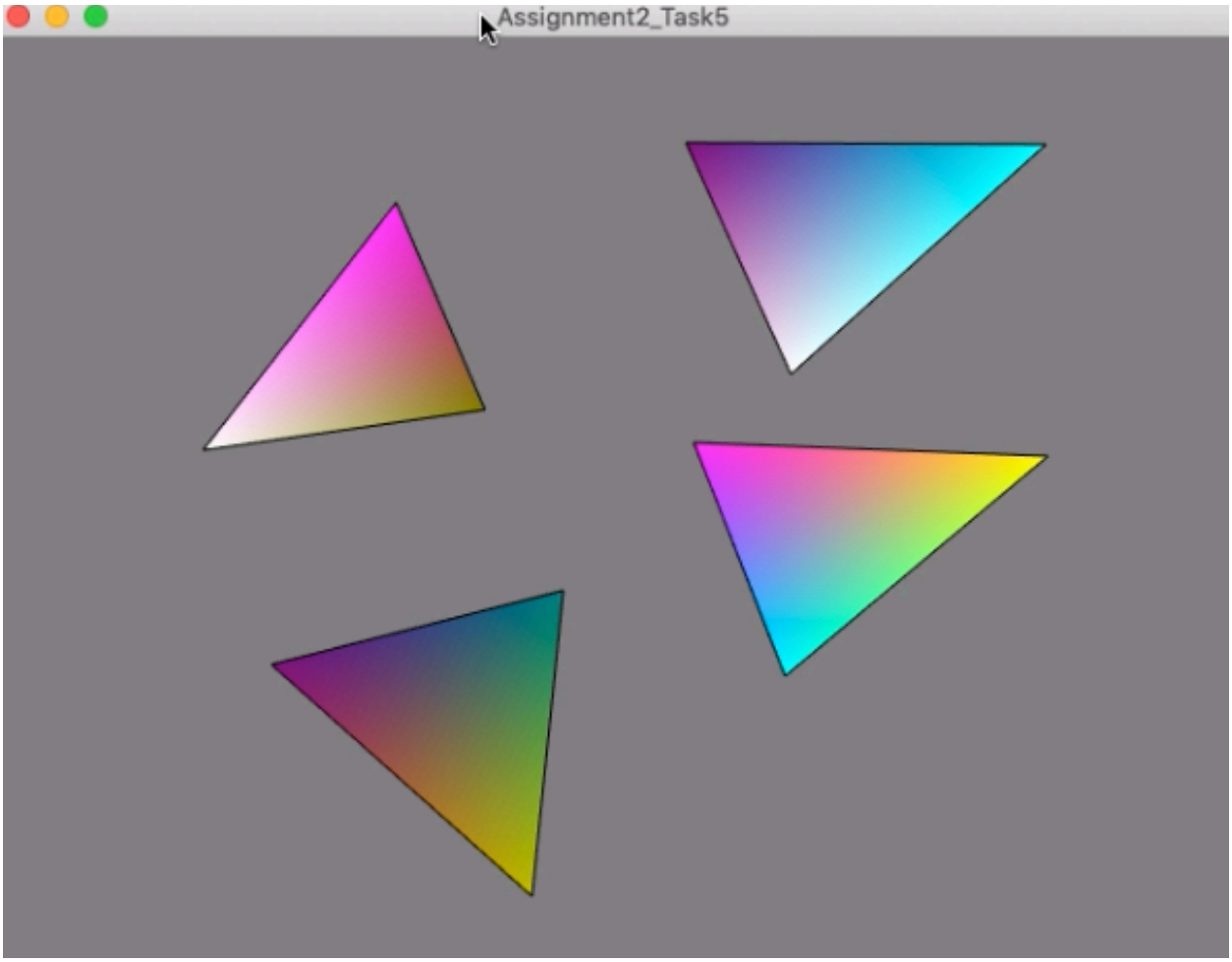
Rotation and zoom in/out animation (press ‘x’):




Assignment2_Task5


Assignment2_Task5


Assignment2_Task5


Assignment2_Task5