Graphics Programming HW 1

Due: 3/21 11:59 PM

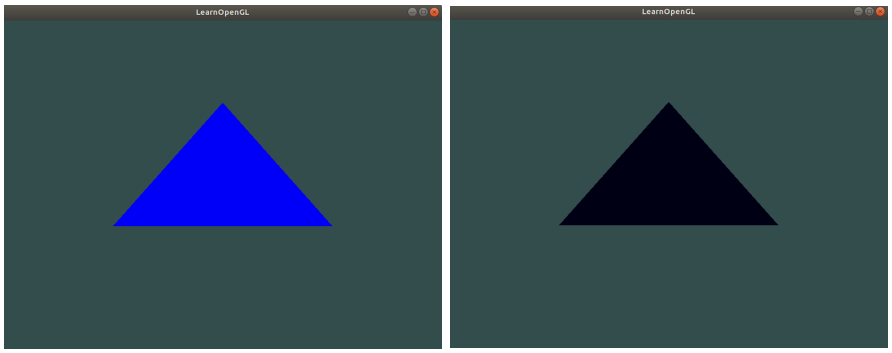
Throughout the all of the homework, it is discouraged to use fixed function pipeline such as `glBegin`, `glVertex`, which is deprecated from OpenGL 3.0. Please use shader instead.

Prob 1. Getting Started with a Triangle (40pt)

In this problem, you will implement a program drawing a triangle using shader. Fill the blanks in the provided code to satisfy following requirements.

1.1 Flickering blue triangle (10pt)

Your task here is to make periodically flickering triangle with blue color. The triangle should be flickering which means that the color of the triangle should be changed from black to blue. The triangle should be flickering periodically, but the period doesn't matter. *Hint*: Use these functions: `glfwGetTime`, `glUniformXX`, `glGetUniformLocation`, `getID` (in Shader Class).



(a) (b)

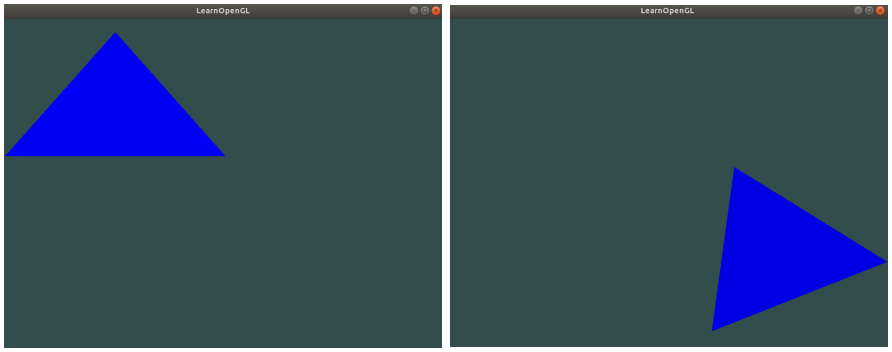
Figure 1: Flickering triangle

1.2 Moving triangle (15pt)

This problem is the continuation of Problem 1.1 above. Your task here is to allow user control to move the triangle. Specifically, the triangle should move in the direction defined by the pressed keys. It should be possible to press several keys to move the triangle diagonally. You can freely choose the keys to assign directions. Please state them in your code. *Hint*: Use these functions: `glfwGetKey`, `setXXX` (in shader Class)

1.3 Rotating triangle (15pt)

This problem builds on Problem 1.2. Your task here is to add a rotation function to the triangle. Specifically, the triangle should rotate counter-clockwise around the center of gravity when you are pressing the key 'r'. Rotating the triangle should simultaneously work with movement defined in Problem 1.2.



(a) 1.2 Moving triangle (b) 1.3 Rotating triangle

Figure 2

Prob 2. Simple 2D Game with OpenGL (45pt)

In this problem, you will make a simple 2D game using OpenGL. Triangles fall from the top of the screen, and a player moves the red bar at the bottom to gather them. The picture of the game is on Fig. 3. We provide a skeleton code and you have to implement the followings.

- For each time interval, new randomly shaped triangle appears at the top of the screen. Its x position is randomly selected. It falls in $-y$ direction with a constant speed, and also rotates with a constant angular velocity. You should update each triangle's position and orientation for each frame. Use uniform variables to update it.
- You then have to make a red rectangular bar near the bottom of the screen. Define additional VAO, VBO to render a quad. The bar should be able to move left and right in response to the user pressing the arrow keys.
- Try to implement simple collision detection. Because it is hard to thoroughly detect intersection between two polygons, you may use simple approximation. Just check whether the center of the polygon is inside the box or not (Fig. 4). Also check if the polygon has disappeared outside of the boundary, and remove it.
- Finally, you will make a simple score board. Draw triangles at the bottom such that the number of triangles represents the current, and also show the score on the top left corner at the screen. Initialize the score if it reaches a certain number, maybe 20, to prevent drawing too many triangles such that they go beyond the screen boundary.

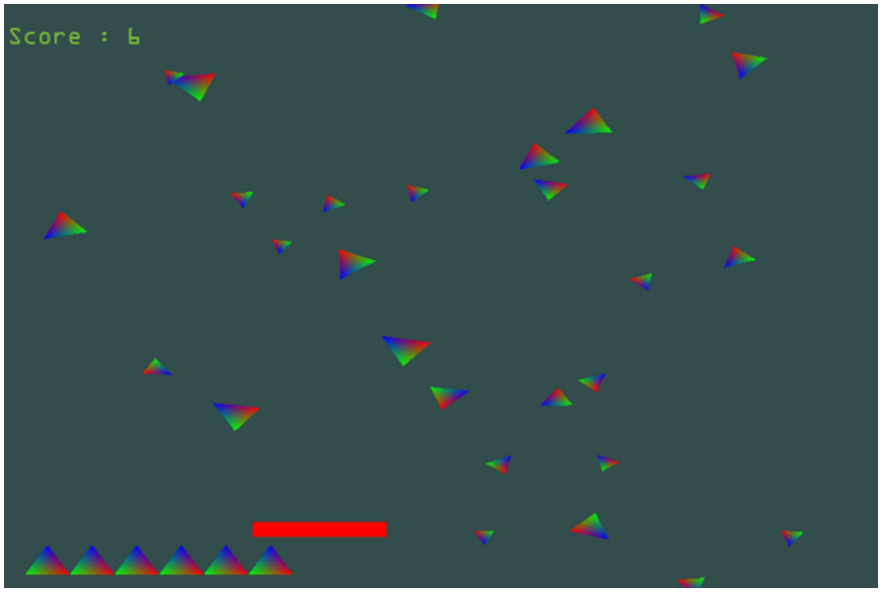


Figure 3: The game you have to implement in Problem 2.

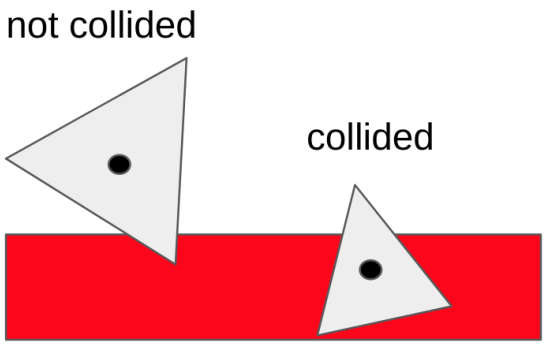


Figure 4: Simple collision detection

Prob 3. Anti-aliasing (15pt)

In this problem, you will implement several anti-aliasing (AA) algorithms. Most of anti-aliasing methods can be categorized into two types; spatial anti-aliasing and post-process anti-aliasing. Spatial anti-aliasing temporarily renders a screen in a higher resolution and shrinks back to the original resolution. Post-process (or screen-space) anti-aliasing blurs the rendered scene. To get more information, please refer here (strongly recommended).

You will implement following anti-aliasing algorithms. SSAA and MSAA belong to spatial anti-aliasing algorithm while simple convolution and FXAA belong to post-processing anti-aliasing algorithm.

- SSAA (Supersample anti-aliasing) : SSAA renders a scene on a temporal framebuffer with a size larger than the screen resolution and then shrinks it to the original resolution. It is the most effective anti-aliasing method, but requires a huge amount of computing power, so is rarely used for practical purpose.
- MSAA (Multisample anti-aliasing) : MSAA is similar to SSAA, but only the edges of the polygons are smoothed out, not the textures or colors that fill the polygons. This reduces computation cost.
- Simple Convolution : Convolution with simple low pass filters, such as a Gaussian or box kernel, can resolve the anti-aliasing effect. Applying such kernels will equally blur the entirely image.
- FXAA (Fast Approximate anti-aliasing) : FXAA finds the edge of a polygon by evaluating the color contrast between two pixels. It then blurs the pixel in the direction perpendicular to the edge direction, with a blend factor calculated from the color contrast. FXAA is the most popular AA algorithm on the market because of its high performance with decent quality. Refer the original NVIDIA white paper for the details.

Here are implementation tips for each algorithm.

- (**3pt**) SSAA : Render a scene into a temporal framebuffer with the size doubled and then shrink it to the original framebuffer. You can use `glBlitFramebuffer` function.
- (**7pt**) MSAA : You can easily implement MSAA with two approaches. One approach is to make use of the `GLFW_SAMPLES` option and enabling `GL_MULTISAMPLE`. However, such approach makes it hard to implement other AA algorithms, since the created window is appropriate only for MSAA. If you want to implement other AA algorithms to gain more points, it is encouraged to implement MSAA with multisampled framebuffer which is well explained in the above link.
- (**3pt**) Simple Convolution: Implement anti-aliasing by applying convolution kernel at the post-processing stage. You can freely set the type (Gaussian, box) and the size of the kernel.
- (**2pt**) FXAA : You can copy FXAA post processing shader code from an existing code repository. Add reference with comments on the shader file.

Make user to be able to switch each AA algorithm by pressing `KEY_1` (no AA) and `KEY_2,3,4,5` for each algorithm.

Students who do not want to spend too much time on this problem can submit their assignment implementing only MSAA with an easy way that uses `GLFW_SAMPLES` option (Fill `TODO1`). However, for students who want to get a full credit, you have to implement additional framebuffers and post-processing (Fill `TODO2` and shaders for post-processing AA algorithms). You can find a useful article related to framebuffers and post-processing here.

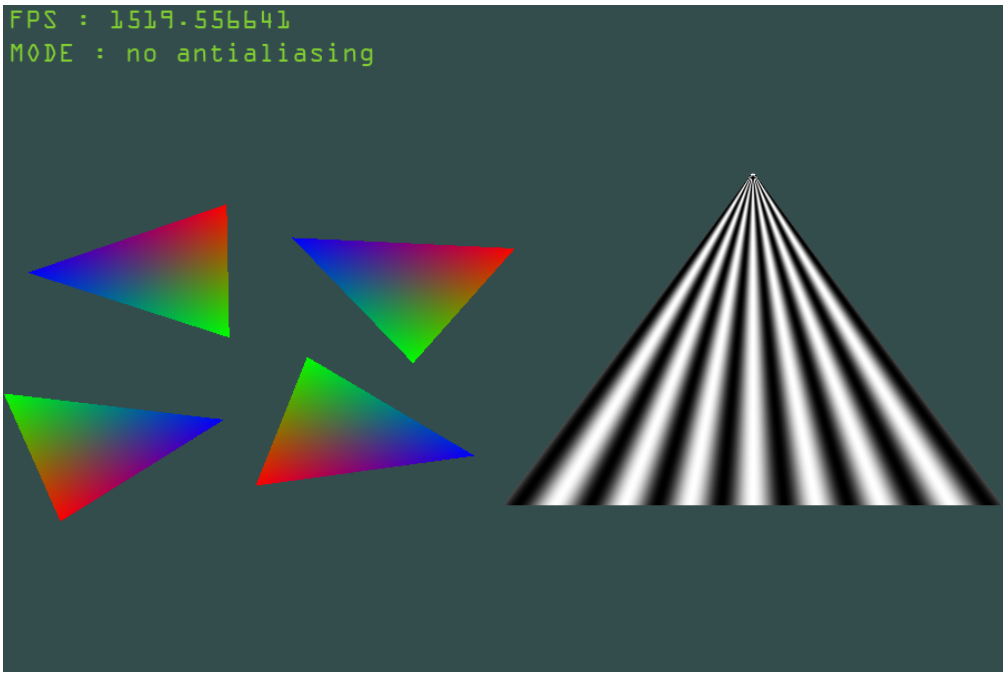


Figure 5: (Key 1) No anti-aliasing

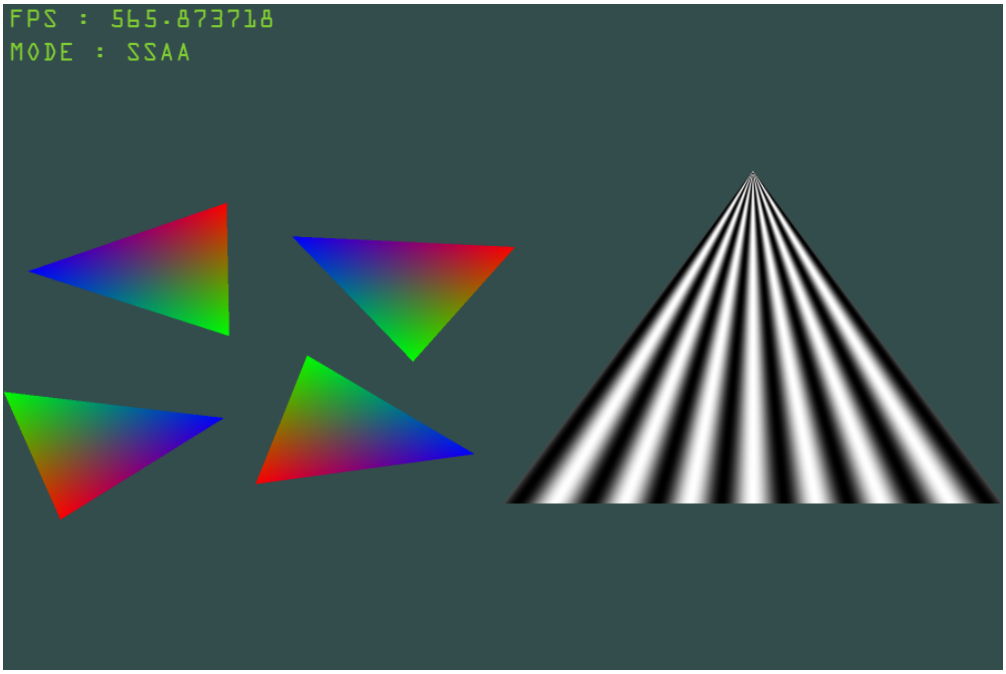


Figure 6: (Key 2) SSAA

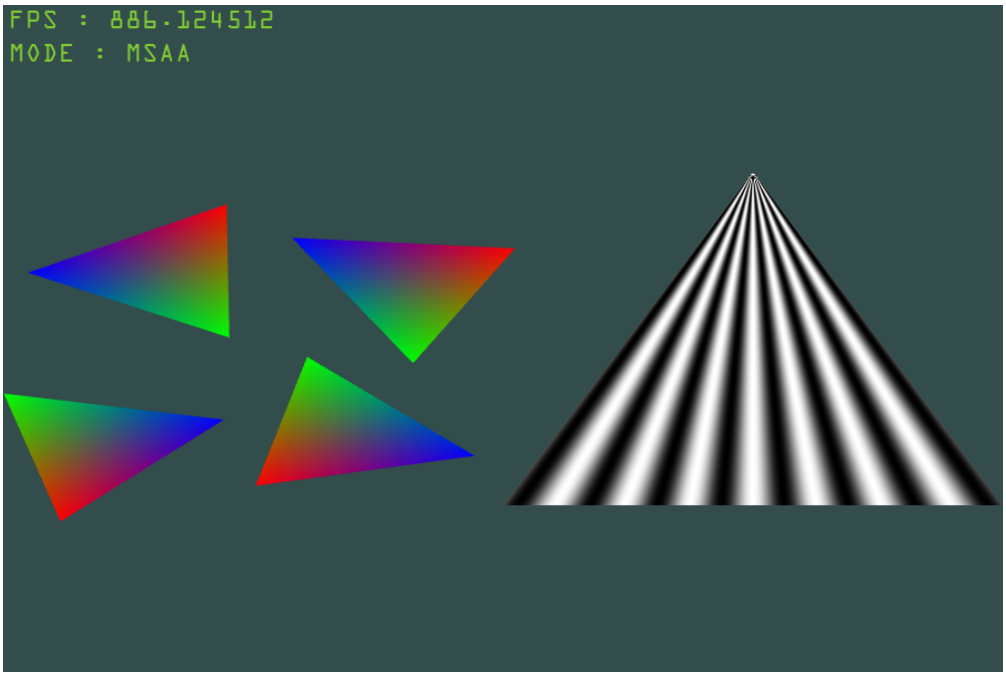


Figure 7: (Key 3) MSAA

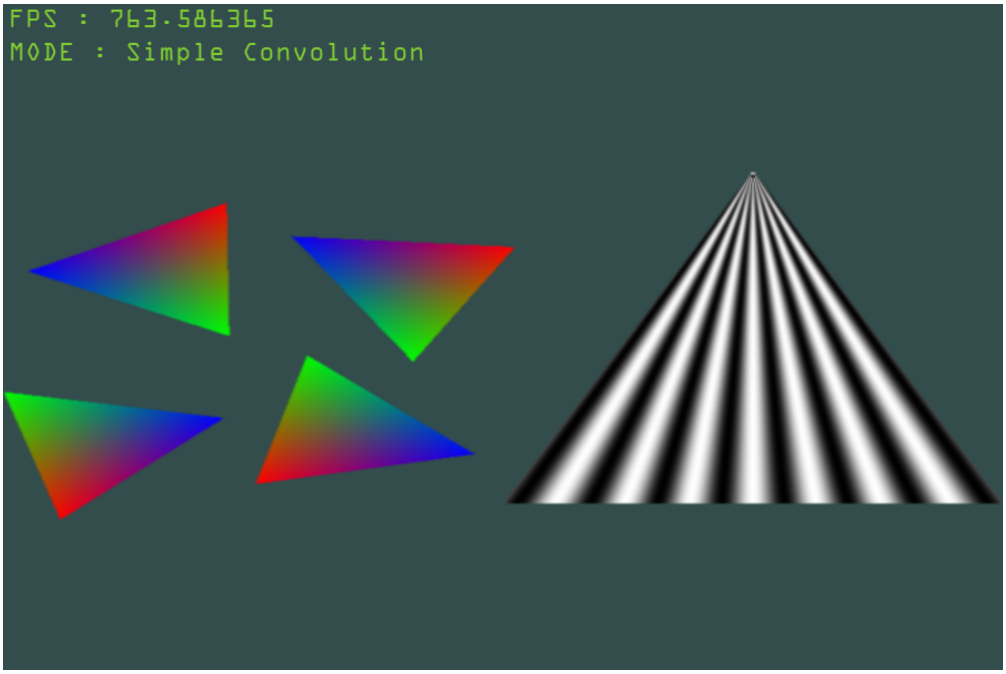


Figure 8: (Key 4) Simple Convolution

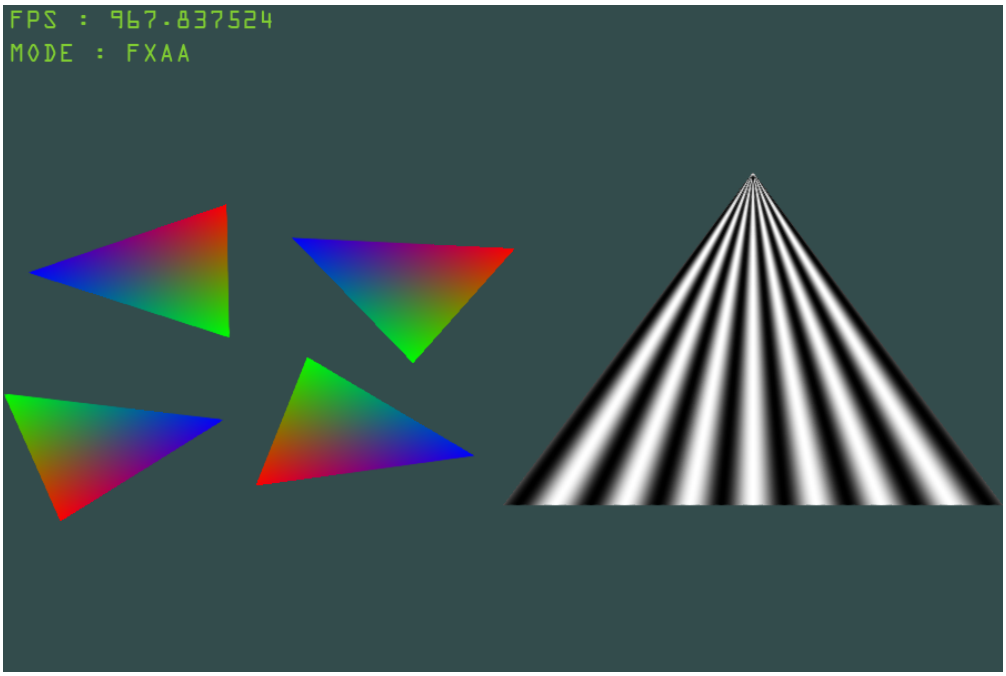


Figure 9: (Key 5) FXAA