Coursework II: Rasterization

COMPGV3080 Team

November 7, 2016

We have shown you the framework for solving the coursework at `cg.cs.ucl.ac.uk/`. You should start by extending the example `Coursework2`. The programming language is OpenGL ES Shading Language (GLSL) `www.khronos.org/files/opengles_shading_language.pdf`. Do not write any answers in any other programming language, in paper, or pseudo code.

Remember to save your solution often enough to a `.js` file. In the end, hand in that file via Moodle.

The total points for this exercise is **100**.

The answers are due on **Friday, November 25th 2016, 23:59 UK time**.

Introduction We will reimplement all important steps of the rasterization pipeline here. The coursework makes use of the struct `Polygon`, that holds a sequence of three-dimensional vectors. The functions `appendVertexToPolygon`, `copyPolygon`, `getWrappedPolygonVertex` and `makeEmptyPolygon` should be used to manipulate polygons.

We have defined a simple scene of *two* triangles and want you to program the key steps of a rasterization pipeline: Projection, Clipping, Rasterization, Interpolation and z-buffering.

At the top of the file, you will find five `#define` statements that have been commented out, one for each of the previously mentioned steps. In the rest of the file, for each of the steps we have provided you with `#if/#else` blocks that depend on these defines. You should solve each step by writing in the `#if` section of each of these code blocks, where it says `// Put your code here`. You should uncomment the defines at the top of the file as you go. For example, when you have replaced the `// Put your code here` lines for projection, you can uncomment `//#define PROJECTION` to see the result.

You will find a separate tab called "Resolution settings". This is made for computers with slow GPU's. If you increase the `SCALING` variable, the rendering will be done at lower resolution.

It is advised to proceed with the tasks in the order listed here. We will do rasterization before clipping for didactical reasons and to ease the visualization process for you.

1 Projection (15 points)

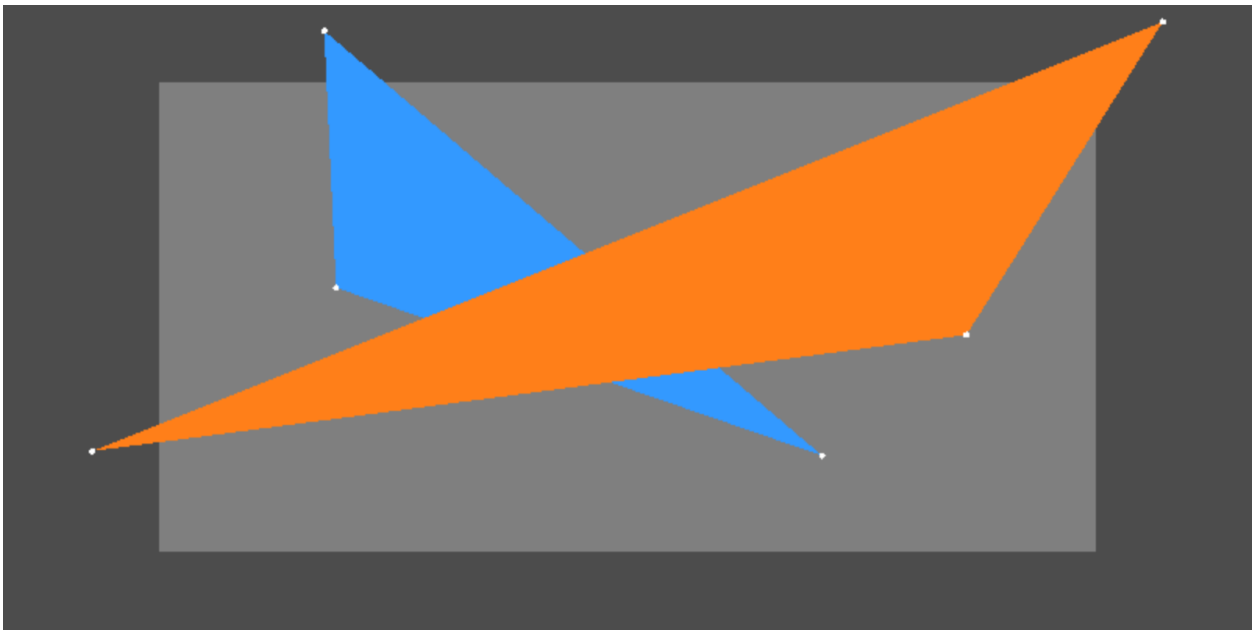
There are two world-space polygons with 3D positions and RGB colors at each vertex in `drawScene`. You are asked to write code to project them. Implement the function `project` to perform the perspective projection making use of a view and projection matrix (**5 points**). Remember the proper use of 4-vectors, the requirement to also have a proper z coordinate and where perspective division occurs exactly. To get the view and camera

matrices, use the function `getViewMatrix` to convert into 3D camera space (**5 points**) and the method `getProjectionMatrix` to convert into image space (**5 points**).

After you finished this part, the image should show the corners of two triangles as shown in the figure above: Six dots, all inside the framebuffer.

2 Rasterization (10 points)

For rasterization we have already prepared a binary test if the current pixel coordinate is inside the polygon: `isPointInPolygon`. You would be asked to complete the inner loop of that function (**10 points**). You might find useful implementing and employing the function `edge`, which should tell you whether a point is on the *inner* or *outer* side of an edge (only consistent on edges of a polygon with the same "clockwisedness").



After you finished this part, the image should look like the one shown in figure above.

3 Clipping (40 points)

Next, all polygons need to be clipped. To this end, complete the partial Sutherland-Hodgman algorithm we provide.

In the function `clipPolygon`, implement the code to detect and handle the crossing type. To do this, make use of the functions `getCrossType` and `intersect2D` you will need to implement. The functions `getWrappedPolygonVertex` and `appendVertexToPolygon` can be used to manipulate polygons.

The function `getCrossType` should take two lines defined by two pairs of points and returns `ENTERING, LEAVING, OUTSIDE` or `INSIDE`, depending on the configuration (**15 points**).

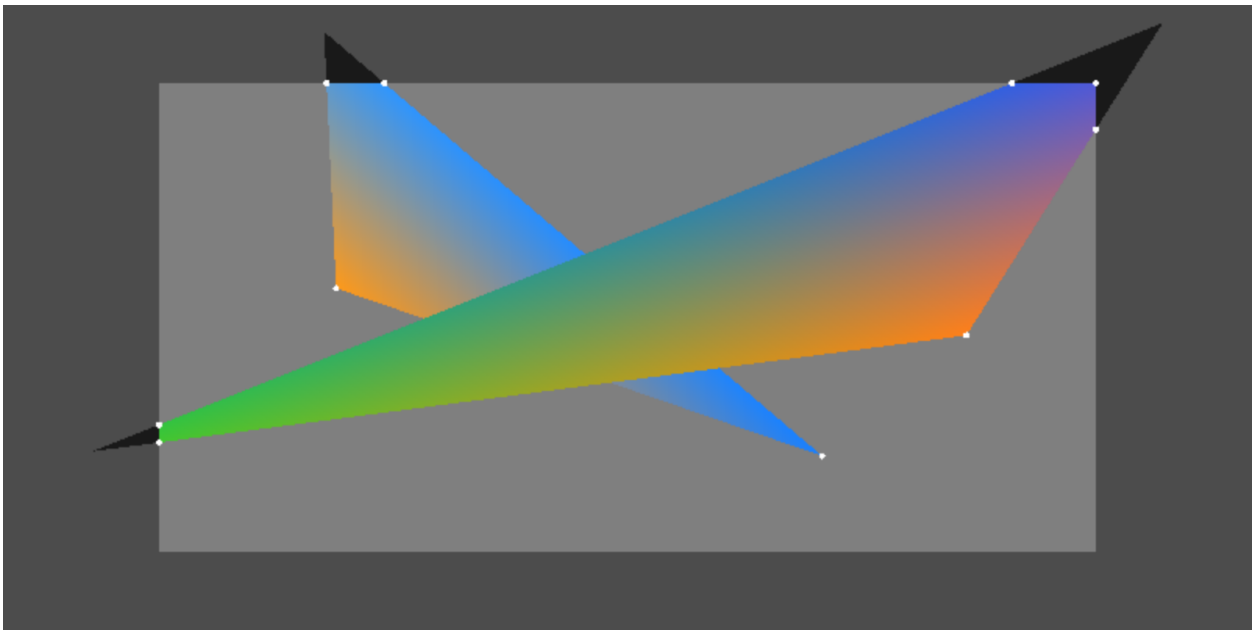
`intersect2D` returns the intersection point between two lines, again defined as pairs of points (**15 points**). Make sure to also change the color variables at each vertex appropriately. Also remember that when interpolating a 3D property like position in `intersect2D`, it needs to be perspectively-correct (**10 points**).

As a result of clipping, polygons might become empty, and the code further down the pipeline has to handle this situation.

After you finished this part, the image should show the corners of triangles, but clipped to polygons as shown in figure above.

4 Interpolation (20 points)

Use barycentric coordinates to interpolate the three-dimensional coordinate and the color from every vertex at every pixel. For this, complete the function `interpolateVertex`, that returns a interpolated position and color information at a when provided a polygon (**20 points**). We have seen in the lecture how to do this for triangles, for the polygons, the same principle can be applied.

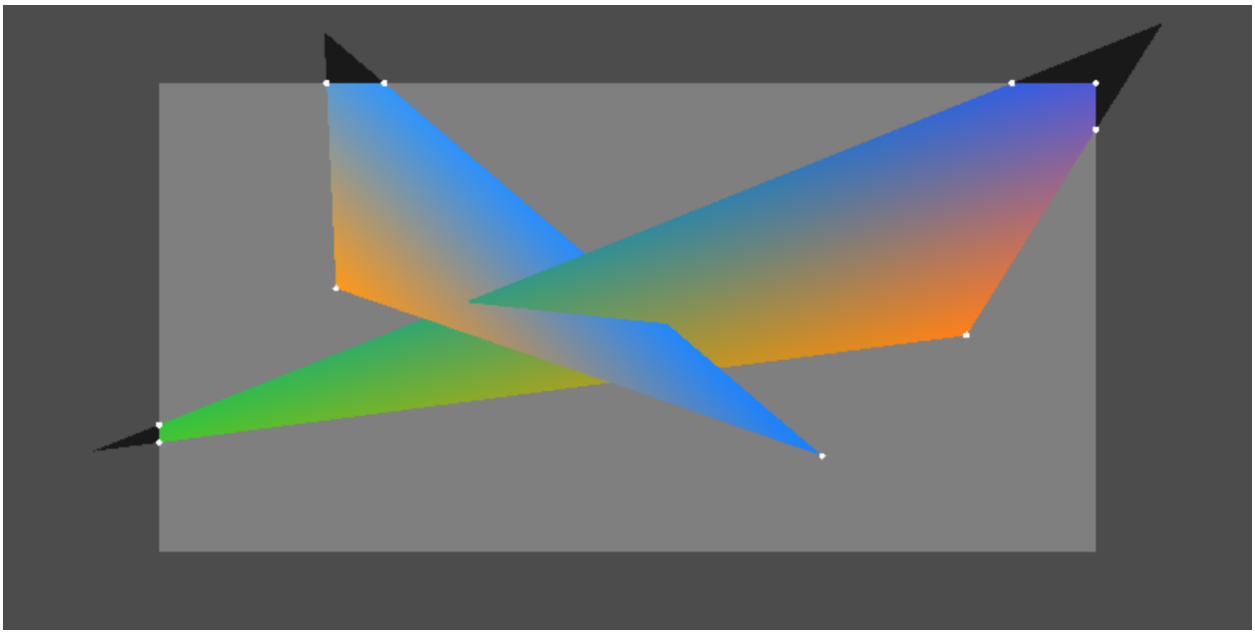


After you finished this part, the image should look like the one shown above: The constant colors have turned

into a smooth color gradients.

5 z-buffering (15 points)

Finally, we want you to add z-buffering to the rasterization code (**15 points**). Please remember that the positions need to be interpolated correctly fro the z test to work.



After you finished this part, the overlapping triangles should mutually have resolved their visibility, i.e., in some pixels where both happen to fall the first is visible, in some part the second. This behaviour is shown in figure above. As per the figure, gradients and clipping should still be working as before.