Coursework II: Rasterization

COMP0027 Team

Tobias Ritschel, Michael Fischer, Siddhant Prakash, Chen Liu

November 16, 2023

We have shown you the framework for solving the coursework at `https://uclcg.github.io/uclcg/`.

You should start by extending the respective example there. The programming language is WebGL and the OpenGL ES Shading Language (GLSL). Here is a quick reference for GLSL functions that you will commonly use in your coursework, e.g., `reflect` and `sin`.

This should run in any browser, but we formerly experienced problems with Safari and thus recommend using a different browser such as Chrome. Do not write any answers in any other programming language, in paper, or pseudo code. Do not write code outside the `#define` blocks.

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

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

Please refer to Moodle for the due dates.

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.

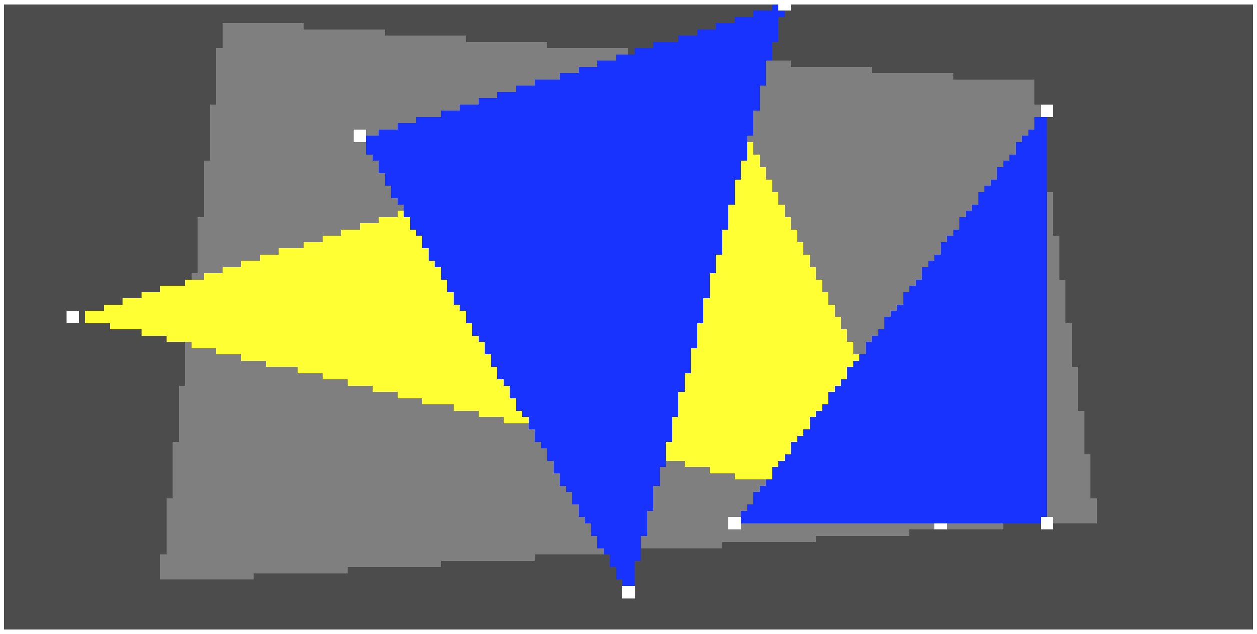
We have defined a simple scene of *two* triangles and want you to program several key steps of a rasterization pipeline: Projection, Clipping, Rasterization, Interpolation, z-buffering, anti-aliasing and blending.

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 rasterization, you can uncomment `//#define SOLUTION_RASTERIZATION` to see the result.

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 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.

2 Clipping (30 points)

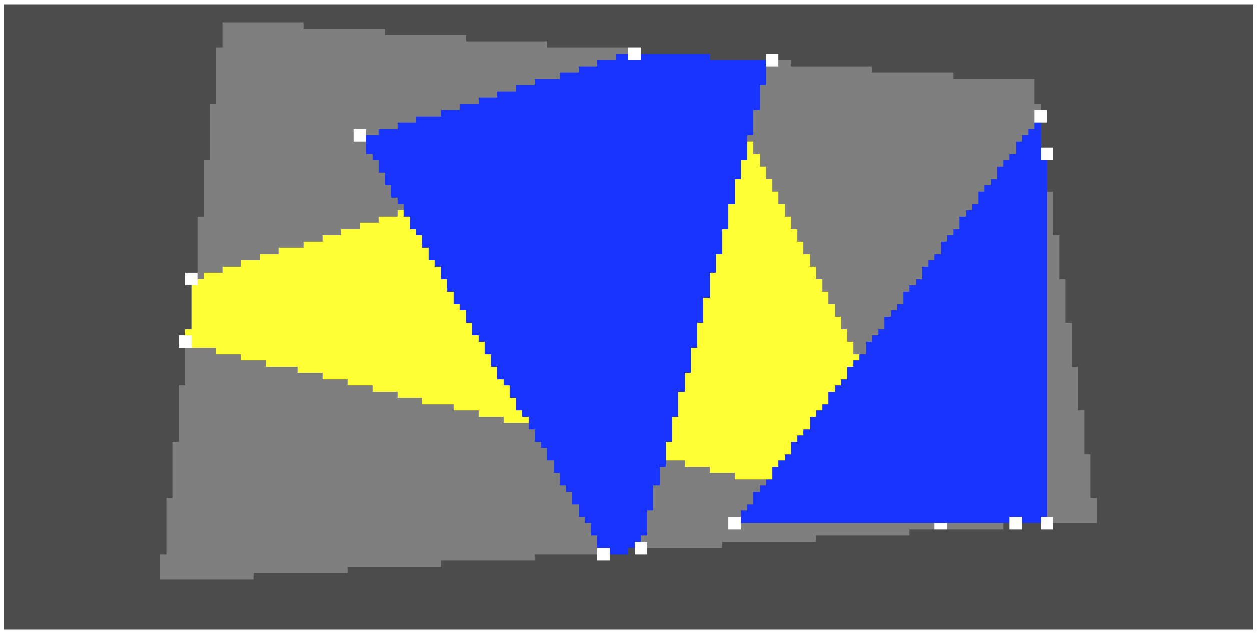
Next, all polygons need to be clipped. To this end, complete the partial Sutherland-Hodgman algorithm we provide.

In the function `sutherlandHodgmanClip`, 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 (**10 points**).

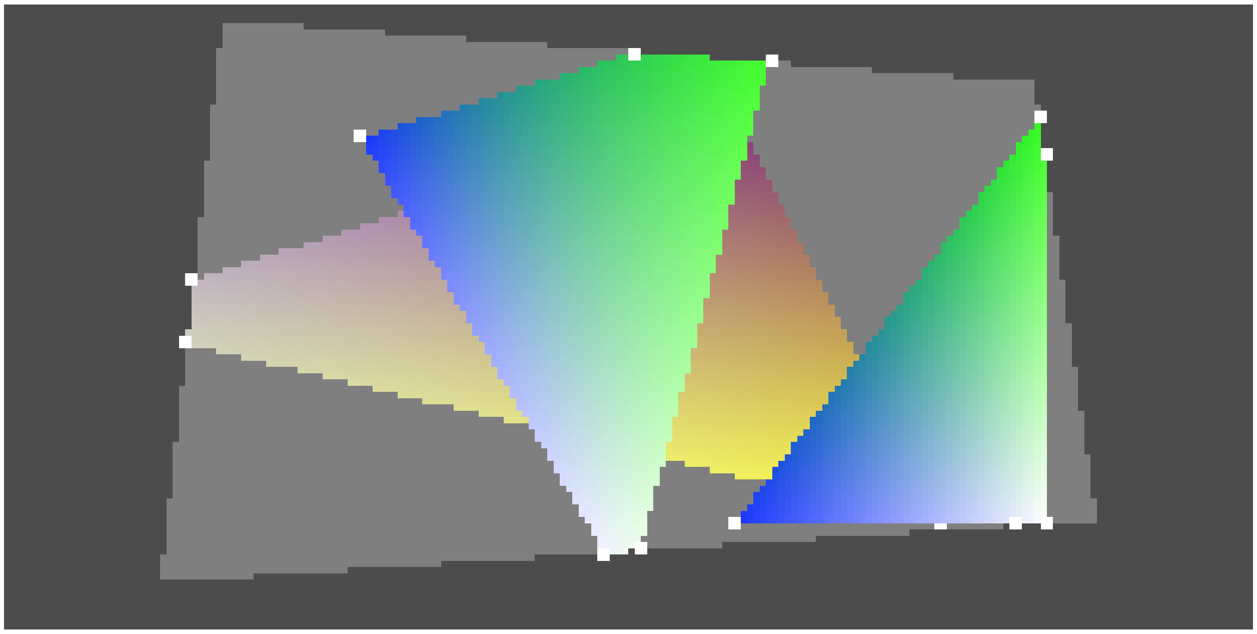
`intersect2D` returns the intersection point between two lines, again defined as pairs of points (**10 points**). Make sure to also change the color variables at each vertex appropriately. Also remember that when interpolating a 3D property like position in `intersect2D`, it needs to be perspectively-correct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

3 Interpolation (20 points)

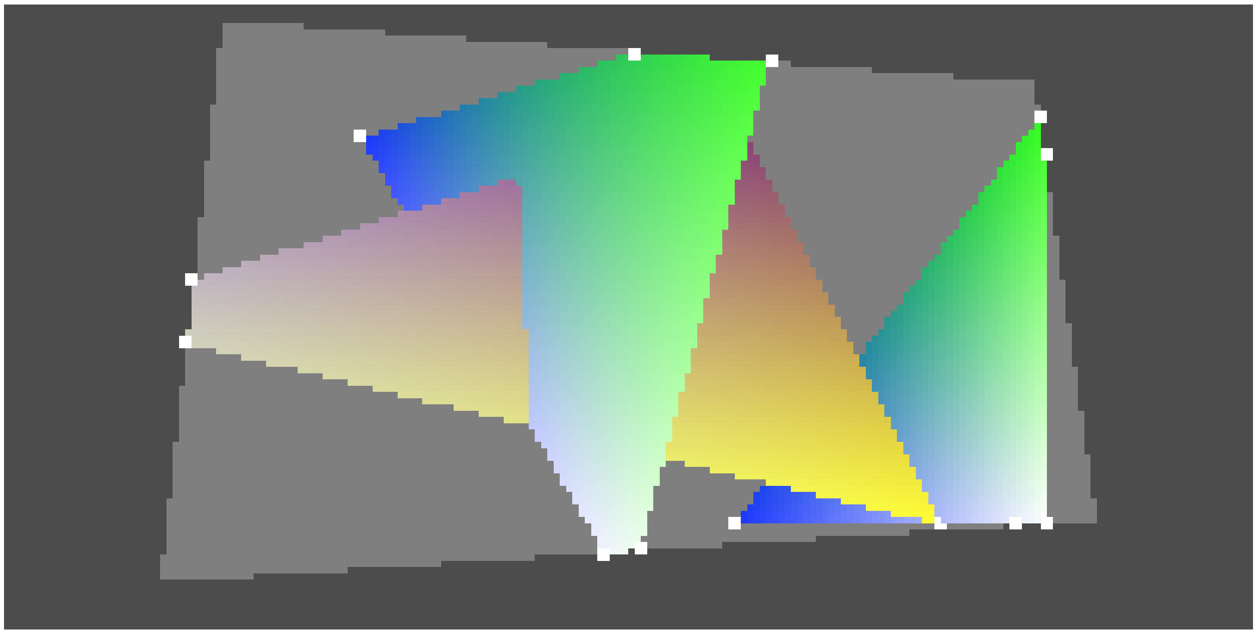
Use barycentric coordinates to interpolate the three-dimensional coordinate and the color from every vertex at every pixel. For this, complete the function `interpolateVertex`, that returns a interpolated position and color information at a when provided a polygon (**20 points**). We have seen in the 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.

4 *z*-buffering (10 points)

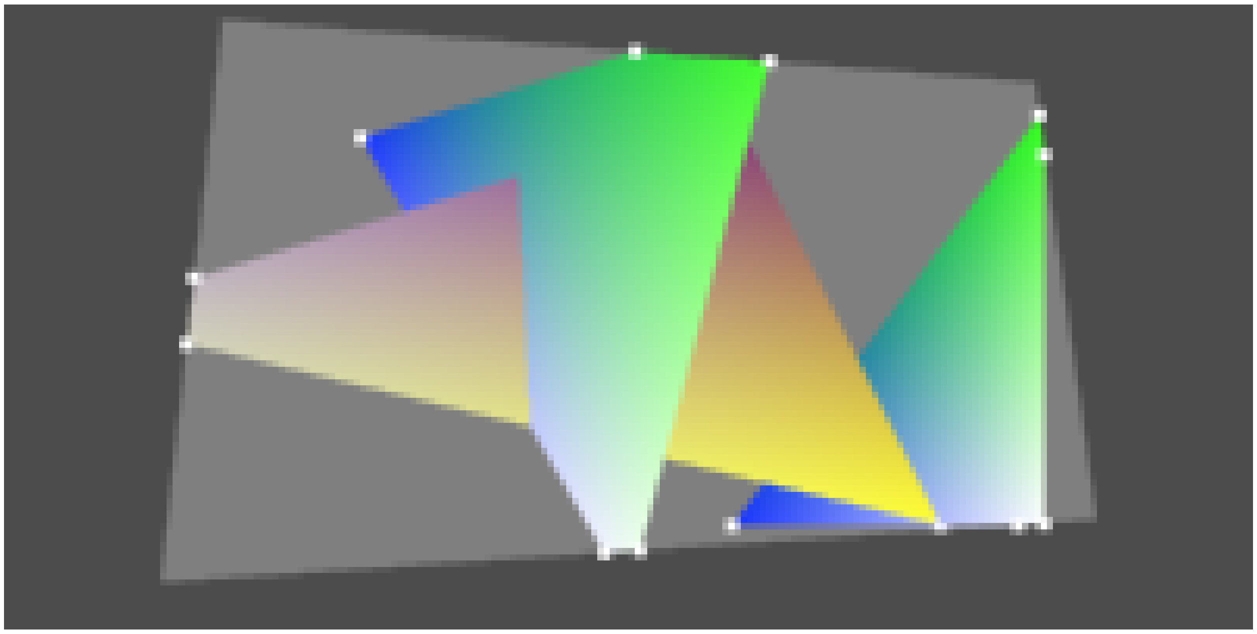
Now, we want you to add z-buffering to the rasterization code (**10 points**). Please remember that the positions need to be interpolated correctly fro the z test to work.



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.

5 Anti-aliasing (15 points)

Next, we would like to add anti-aliasing to this primitive (**13 points**). This is an advanced feature that was not taught at the course (we will get back to it later in the context of Monte Carlo ray-tracing). For now we would ask you to read up on it in your own research and think how this can be applied to the rasterization setting.



When done successfully, a result can look like the above picture, but also other solutions are accepted. It is however important they are not just blurry versions of the high-resolution image, but properly anti-aliased.

This can be a slow process (on Tobias' PC it particularly takes long to compile the shader, not even so much to run). Please offer an option in your code to trade quality and speed (**2 points**).

6 Blending (15 points)

Finally, we ask you to add blending to the rasterization pipeline. We ask you to support four blend modes `BLEND_ZERO`, `BLEND_GL_ONE`, `BLEND_GL_SRC_ALPHA`, and `BLEND_GL_ONE_MINUS_SRC_ALPHA`. Their behavior should be same as their OpenGL counterparts.

The blend mode used is defined by the variables `src_blend_mode` and `dst_blend_mode`.

We will evaluate your solution on three combinations of the blend modes for source factor and destination factor as follows: i) srcFac: `BLEND_GL_ONE`, dstFac: `BLEND_GL_ZERO` (**4 points**) ii) srcFac: `BLEND_GL_ONE`, dstFac: `BLEND_GL_ONE` (**4 points**) iii) srcFac: `BLEND_GL_SRC_ALPHA`, dstFac: `BLEND_GL_ONE_MINUS_SRC_ALPHA` (**4 points**).

We ask for compact and correctly factored code here (**3 points**).

We withhold the reference image for this part of the coursework.