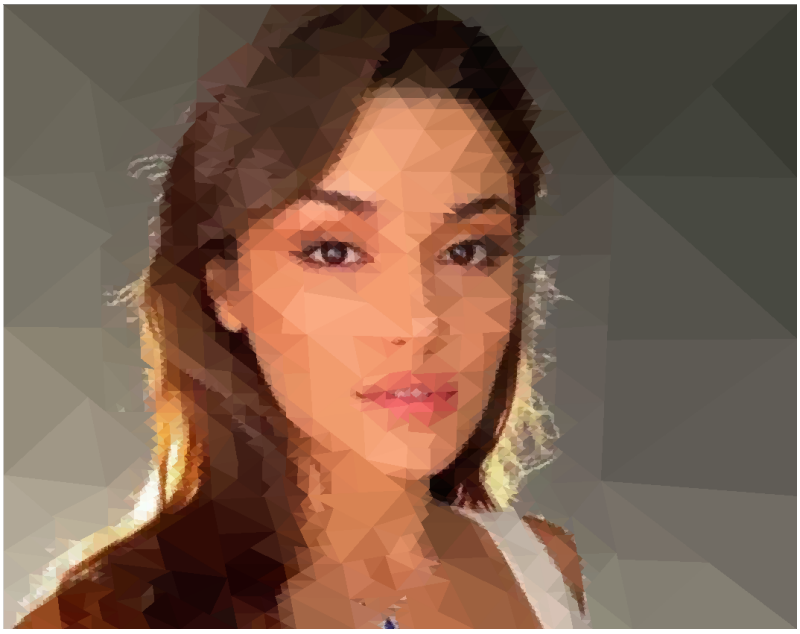
Low Poly Image Generator

Matthew Reed



Introduction The purpose of this project was to create an extension of our SVG renderer that could convert an image into a low polygon version. This would mostly be used for artistic effect, although depending on how many triangles you use, it could also use up less storage. I didn't want to look up any existing algorithms for this kind of functionality, so I created my own from scratch as I will describe below.

I modified *main.cpp, drawsvg.h/.cpp,* and *software_renderer.h/.cpp,* as well as created the files *image_simplifier_settings.h, image_simplifier.h, and image_simplifier.cpp*.

image_simplifier.h/.cpp is where I implemented the class *ImageSimplifier* which simplifies an image into triangles based on the parameters specified in an *ImageSimplifierSettings*. I also implemented a command-line interface that automatically constructs this *ImageSimplifierSettings* struct based on the following command-line arguments:

Argument	Arg type	Functionality
`--triangulate`	`null`	Adding this flag will turn on triangulation of all images for the SVG renderer. Without it, none of the other flags will do anything.
`--num_triangles`	`int`	How many triangles to divide the image into.
`--num_samples`	`int`	How many sample points are used in each triangle to determine the color and amount of detail in a triangle.
`--save_to_path`	`string`	If a file path is specified, a polygon version of the image will be saved as an SVG file using just <polygon/> objects.
`--amount_chaos`	`float`	Degree to which the place where triangles are split is optimized to maximize detail. A value of 0 will result in perfectly regular

		triangles. A value of 0.5 is recommended.
`--area_priority`	`float`	Degree to which larger triangles are preferentially split more than smaller triangles. A value of 0 will split triangles only based on how much detail is in that triangle. A value of 0.5 is recommended.
`--show_pos`	`null`	Instead of plotting triangles, the centroid of each triangle is plotted. This makes it very easy to visualize the distribution of triangles and where the algorithm thinks there is more detail in the image.

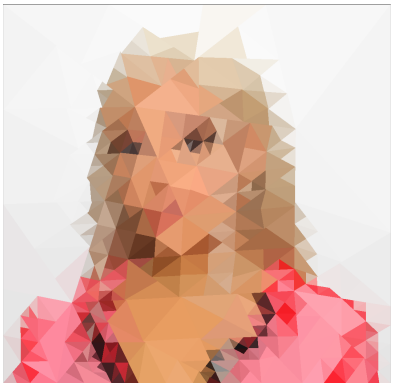
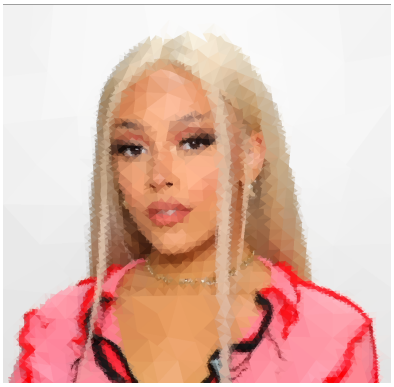
Algorithm The algorithm I created starts with 14 randomly placed non-overlapping triangles which cover the entire image (example shown on right). These triangles are placed into a priority queue which is sorted based on the benefit we would receive from splitting any given triangle into two triangles. This benefit is calculated based on both the size of the triangle and the standard deviation of the various color channels at different randomly sampled points within the triangle. Essentially we are finding the triangle that is the worst approximation of its underlying data.

When splitting a triangle, we pick a point along its longest side which is calculated by using the average location of the detail within the image (a weighted average of the sample points' locations using their standard deviations). These two smaller triangles are then placed back in the priority queue.

Once we have enough triangles, we pop each triangle, one at a time, and fill the corresponding part of the sample buffer with the correct color. This color is determined by averaging the color from each randomly selected sample point from within the triangle.

Examples I will demonstrate the capabilities of my SVG renderer extension by showing examples of modulating some of the algorithm's parameters.

--num_triangles

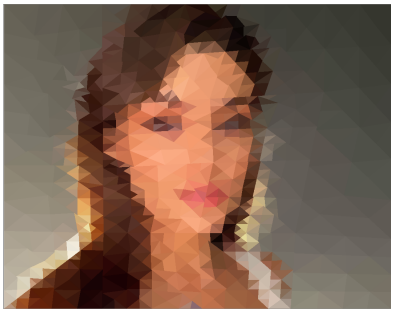




500 triangles 5,000 triangles 50,000 triangles

As you increase the number of triangles, there is more detail preserved in the final image. With a few hundred triangles, you get a cool mosaic pattern, with a few thousand you get a stylized low-poly version of the original photo, and with tens of thousands of triangles, you get a pretty close approximation of the original image.

--area_priority

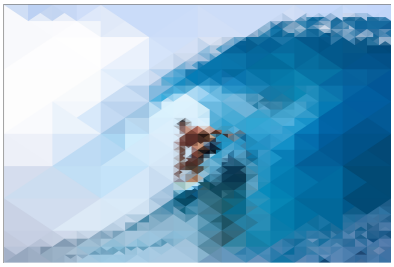
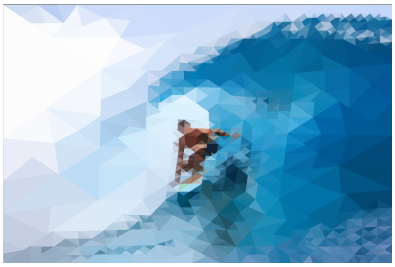
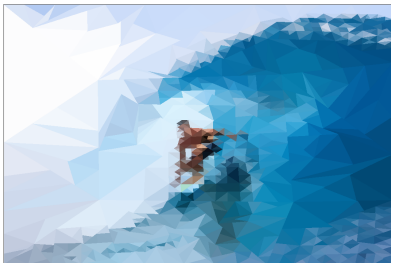




0 0.5 1.5

Area priority affects which triangles are chosen to be split. With a value of 0, the algorithm always chooses the triangle that is covering the most detailed part of the image. That's why most of the triangles are used near the eyes but almost none are used for the gray background. As the value is increased, larger triangles will be prioritized to be split even if they don't cover as much detail. I found 0.5 to be a good compromise between prioritizing detail and maintaining a homogenous look over the whole image.

--amount_chaos

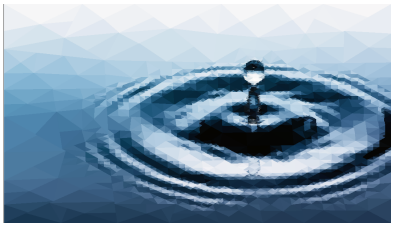




0 0.5 1

The amount of chaos is the degree to which triangles are split unevenly to maximize detail. With a value of 0, the triangles are all similar and occur at 90-degree angles to the x and y-axis. As it's increased, the image has a more natural feel and ends up looking better in the end.

--num_samples

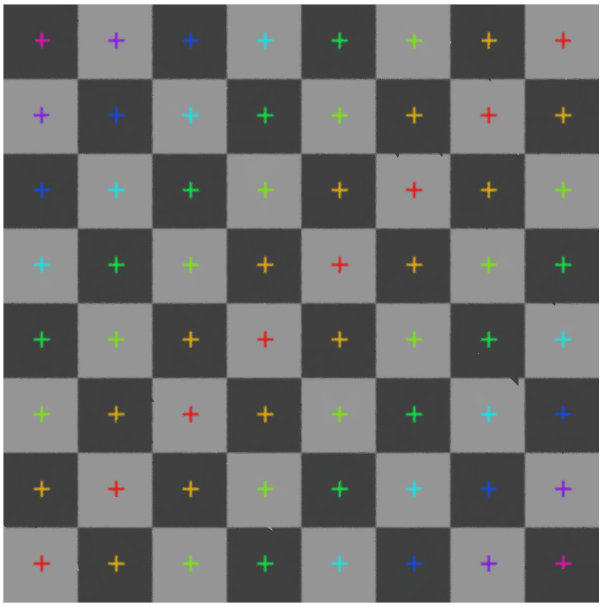




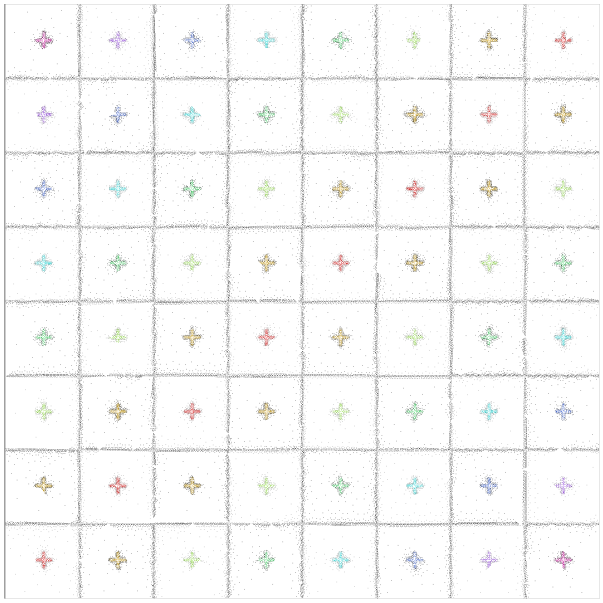
1 10 100

The number of samples determines how many samples are used per triangle to check the amount of detail for that triangle and to calculate the average color of the triangle. Increasing the number accomplishes the same thing as supersampling with the added benefit of giving the algorithm a more accurate idea of detail in the image.

--num_triangles 50,000

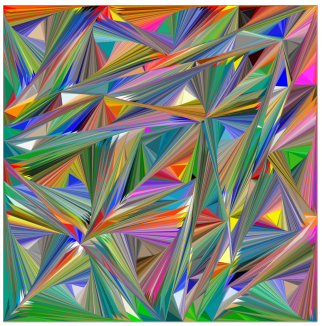


--num_triangles 50,000 --show_pos



This section is simply to demonstrate that the algorithm correctly places more triangles in places where there is higher frequency data (more detail). The left picture is the result of calling my algorithm with 50,000 triangles. The right image is what the image looks like if we plot just the centroid of each of the triangles.

Conclusion



I'm actually really pleased with how well this project turned out. I wasn't sure if I would be able to pull it off or if the results would end up looking good. Some early results (shown to the right) had me pessimistic that my algorithm would even work. But after weeks of fine-tuning the code and trying different computations, I think the results are pretty cool! On the last page, I show the same image simplified with 5,000 triangles twice. The first one is a naive approach where we uniformly place triangles over the image and sample each triangle at its centroid for its color. The second one is using my fully functioning algorithm. I think it has a really nice balance of keeping the details we care about (the hair and the eyes) while still keeping the background simple and artistic.

I've included in my zip file all of the images I was testing with (in svg/test/) along with some triangle versions of them (in the same folder, but ending with _triangle.svg).

Thanks for checking out my project!