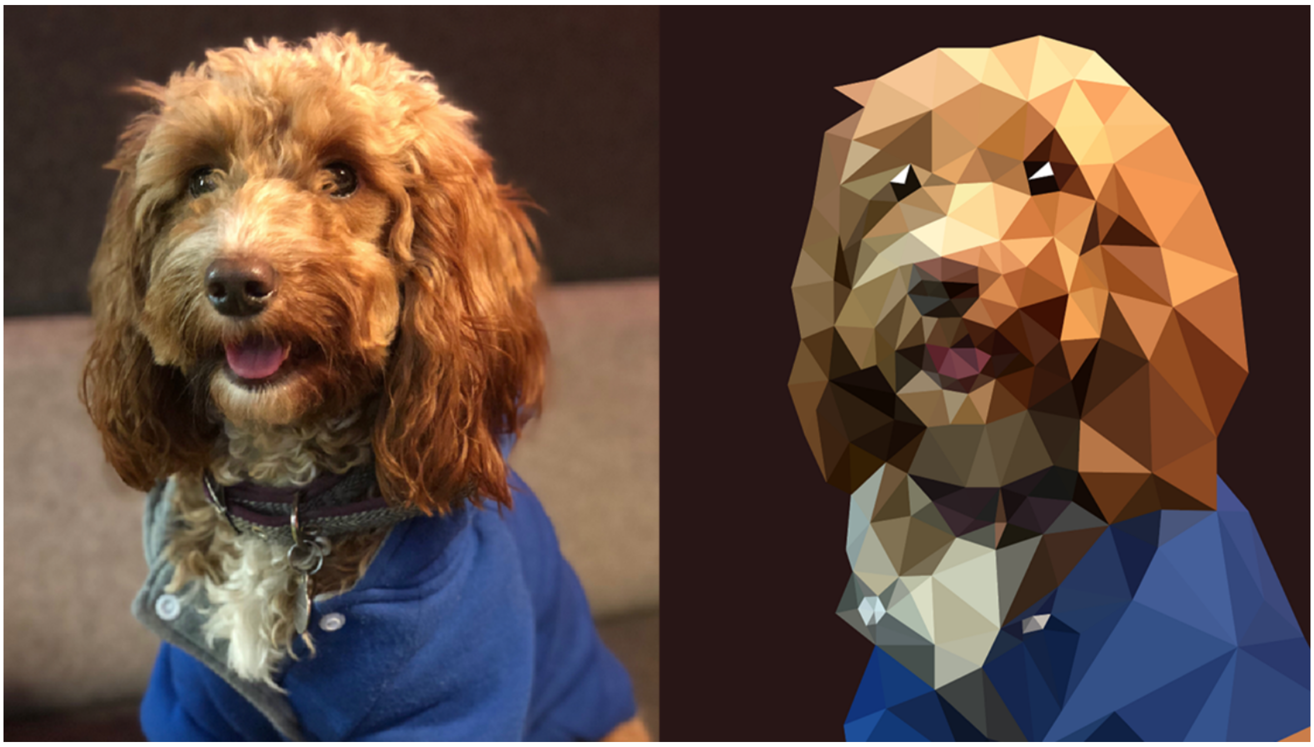
# Parallel Low Poly Style Image/Video Converter

at: https://darkforte.github.io/LowPoly/

## Summary

We are going to implement a parallel low poly style image converter with CUDA. It accepts an input image and transforms it into a low poly style one. We hope it can achieve real-time speed such that it can also do low poly transformation on videos.

## Background



(Image credit: How to create low poly art in Adobe Illustrator)

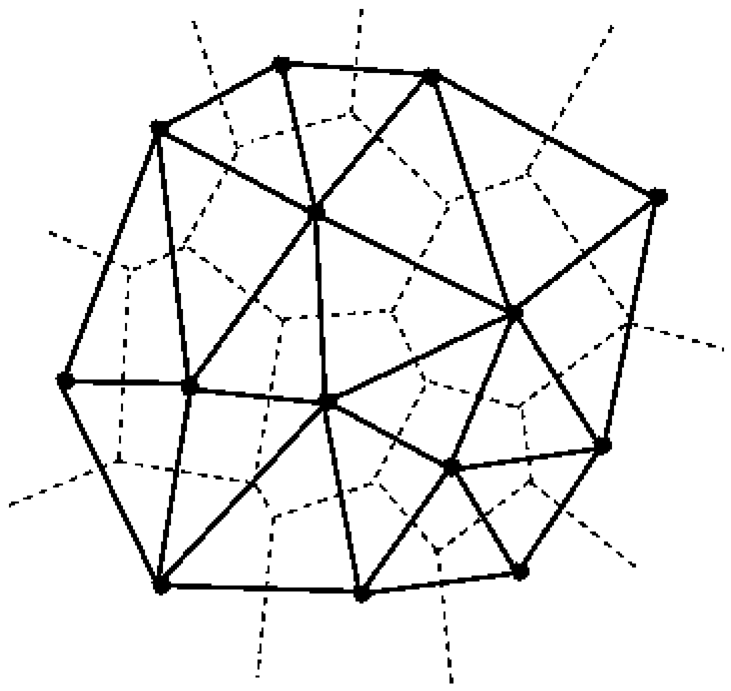
Low Poly Art is an art style that expresses objects with only a limited number of polygons. It was introduced in early stage computer games when the computers were not so powerful as they are today. Nowadays, Low Poly Art becomes a popular style in modern design because it brings an unique abstract and retro-style aesthetic value. There are many converters that can convert an image to low poly style. They are very helpful for designers who need generating low poly style pictures, artists who are looking for new ideas, or ordinary people who do this just for fun.

A popular method to convert a picture to low poly style is called Delaunay Triangulation (DT). Delaunay Triangulation finds a non-overlapping triangulation of a set of points on a plane while making sure that there is no triangle that is too sharp. The main flow of making a low poly style picture is to:

1. Spread a set of points on the picture;

2. Calculate a Delaunay Triangulation of these points;
3. Fill all the triangles with the color of their centers.

Delaunay Triangulation is known to be computational expensive. One way to accelerate it is to parallelize the computation with GPU. The main idea of computing DT for a given set of points is to compute the Voronoi Diagram (VD) of it on the picture (dashed lines), then connect the points of adjacent regions of the diagram to produce the triangulation (solid lines). We can use GPU-friendly Jump Flooding Algorithm to construct the VD.



We are going to achieve the three steps of transforming an image to a low poly art with CUDA. Our plan is as follows:

1. For **picking points on the picture**, we will first try randomly generated points. Then after we have a whole working program, we will implement a Sobel Edge Detector with CUDA and sample with higher probability on edges.
2. For **VD and DT Computation**, we will use the Jump Flooding Algorithm and parallel triangle construction strategy introduced in this slide.
3. For **shading triangles**, this is similar to rendering circles in Assignment 2. We will refer to assignment 2 and do it efficiently with CUDA.

## Challenges

- Implementation of CUDA VD computation is tricky. Every step of computation can actually be parallelized with smart task assignment, and we need to avoid race conditions in the Jump Flooding Algorithm to maximize the performance.
- Some modifications might be necessary after computing the VD to produce better looking pictures, like island removal and special treatment for the edges of the picture. Dealing with them may need the participation of CPU, which may lag performance down. There might be a tradeoff between output quality and computation time here.
- We are unclear how much speedup could be achieved with this algorithm, especially compared to the traditional increment or divide-and-conquer algorithms. Since this algorithm is not sensitive to the

number of points, when there are few points it may be actually worse than the CPU version.
- If we are going to do low poly transformation on video, there may be correlations between adjacent frames so some of the computation may be shared. It remains unclear how leverage this property and make efficient transformation on video.

## Resources

- We are going to start from scratch for this project. There are some former implementations, but they are too complicated. If we have time, we can compare the performance with some of these versions.
- There are some former attempts to parallelize Delaunay Triangulation with CPU, like https://cse.buffalo.edu/faculty/miller/Courses/CSE633/adarsh-prakash-Spring-2017-CSE633.pdf which uses OpenMPI to parallelize the algorithm. We will compare our performance with this version. We expect to achieve higher speedup than it.
- We are referring to http://www.cs.utah.edu/~maljovec/files/DT_on_the_GPU_Print.pdf, http://rykap.com/graphics/skew/2016/02/25/voronoi-diagrams/ and https://www.comp.nus.edu.sg/~tants/delaunay/GPUDT.pdf as guidance to our implementation.

## Goals and Deliverables

### Expected Goals (Plan to achieve):

- Make a program that loads a picture from disk, use Sobel Edge Detector to spread points, triangulates the image, then store and shows the low poly style image.
- Implement the Sobel Edge Detector, Parallel Jump Flooding Algorithm and Triangle Rendering with CUDA.
- Achieve real-time computation speed (30fps), and more than 10x speedup compared to sequential CPU version using the same algorithm. The number comes from the report of MPI version, which is about 5x. Since we don't need the merge step, the number should be higher than the MPI version.

### Minimum Goals:

- Finish the parallel version of Delaunay Triangulation and achieve at least 5x speedup compared to CPU version.
- Complete image input/output features to have displayable demos.

### Ideal Goals (Hope to achieve):

- Make a real-time triangulation converter for video, and achieve better speedup on videos than processing individual frames since the frames in video are correlated.

## Platform Choice

We are planning to use GHC machines with GPU for experiments, and we are planning to use C++ for development and OpenCV for some helpers like reading images and displaying images on the screen.

## Schedule

| Time | Work |
|---|---|
| 10.29 - 11.4 | Write proposal, research for existing work |
| 11.5 - 11.11 | Complete sequential version of Sobel Edge Detector and Jump Flooding |
| **11.12 - 11.18** | Complete CUDA version of Edge Detector, prepare for Checkpoint |
| 11.19 - 11.25 | Complete CUDA version of Jump Flooding and Triangle Rendering |
| 11.26 - 12.2 | Performance tuning and analysis |
| 12.3 - 12.9 | Complete image input and output interface, get ready for demo |
| **12.9 - 12.15** | Final Report |