Parallel Low Poly Style Image/Video Converter

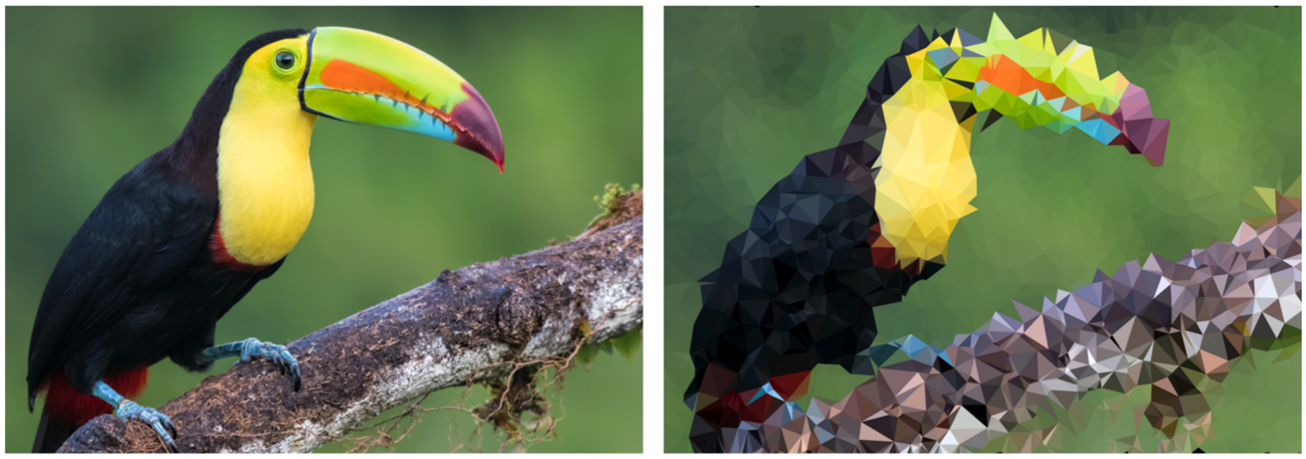
at: https://darkforte.github.io/LowPoly/

Project Checkpoint

Summary

We implemented a parallel low poly style image converter on CUDA. It accepts a picture of any size and converts it to a composition of many single colored triangles. We implemented the workflow of the converter on both CPU and CUDA and tested it on pictures with different sizes. Experiments showed that our CUDA implementation can achieve ~50x speedup compared to the CPU version.

Background and Motivation



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.

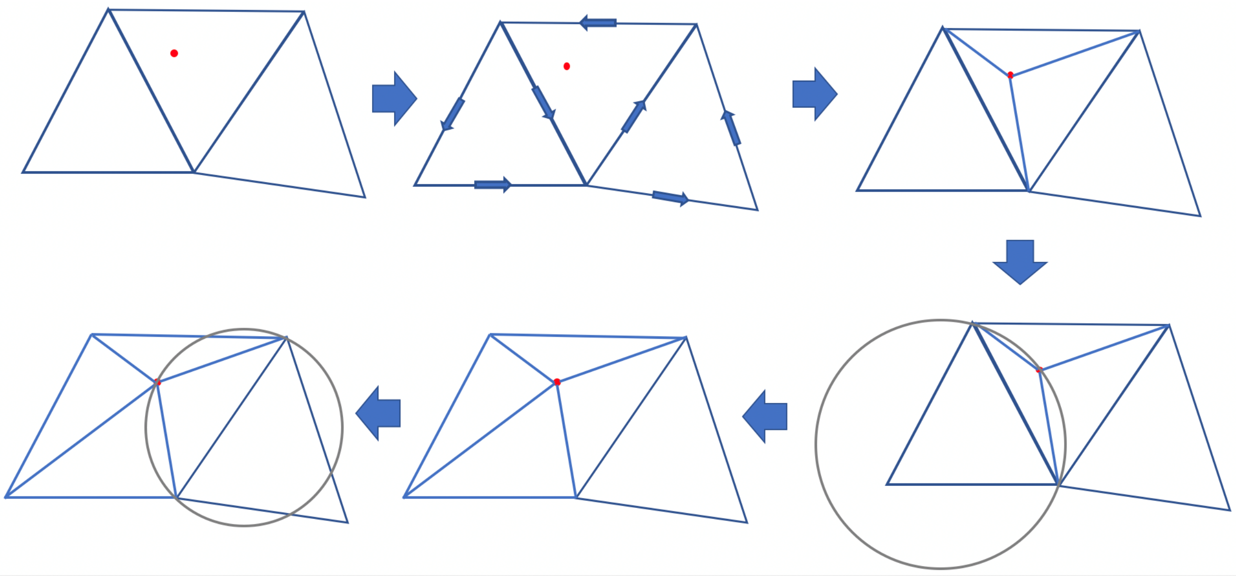
The main workflow of making a low poly style picture involves three steps:

- **Point Selection** spreads a series of points on the picture and preserves the structure of the image. The easiest way to do it is to spread the points uniformly on the picture. However, in order to preserve the picture structure, it would be better to first extract the edges in the picture, then spread more points on the edges than the other parts.
- **Triangulation** connects the points to form a triangle mesh. The most common algorithm is Delaunay Triangulation (DT). This is the most tricky part in the workflow, which will be explained in detail later.

- **Rendering** re-draws the picture using the triangle mesh. Usually it renders each triangle to the color at its center.

All three parts can benefit from parallel execution on GPU. Among them, Delaunay Triangulation (DT) is the most computational expensive part, and it is more tricky to parallelize than the other two. DT refers to a triangulation on a set of points so that no point is inside the circumcircle of another triangle. A common DT algorithm for CPU is the Bowyer–Watson Algorithm. The workflow of this algorithm is:

1. Add a super triangle that includes all the points;
2. Iteratively add points to the current triangle mesh. As shown in the following picture, adding a new point will form three new triangles. Then, it checks whether there is a triangle whose vertex is in the circumcircle of the new triangle (thus violating the Delaunay condition). If yes, then the common edge of the two triangles is flipped.



3. Keep adding the points. After adding all the points, removing the super triangle gives the DT of the points.

This algorithm is straightforward, but it is hard to parallelize. The reason is that this algorithm is essentially iterative, and parallelizing it will introduce huge contention. We cannot add two points simultaneously if they are in the same triangle because both points want to modify it. This restriction severely hampers the potential for parallelism of this method.

Some previous attempts used the divide and conquer algorithm to leverage the parallelism in it. For example, Prakash implemented an OpenMPI version of DT with the divide and conquer algorithm. The basic idea of the algorithm is to split the points into two areas, do DT in each area, then merge the points on the borders. Although natural to parallelize, this algorithm is much harder to implement, and the communication overhead between processors has great impact on the overall speedup. Even if data fits in a single machine, using 32 cores can only bring 5x speedup.

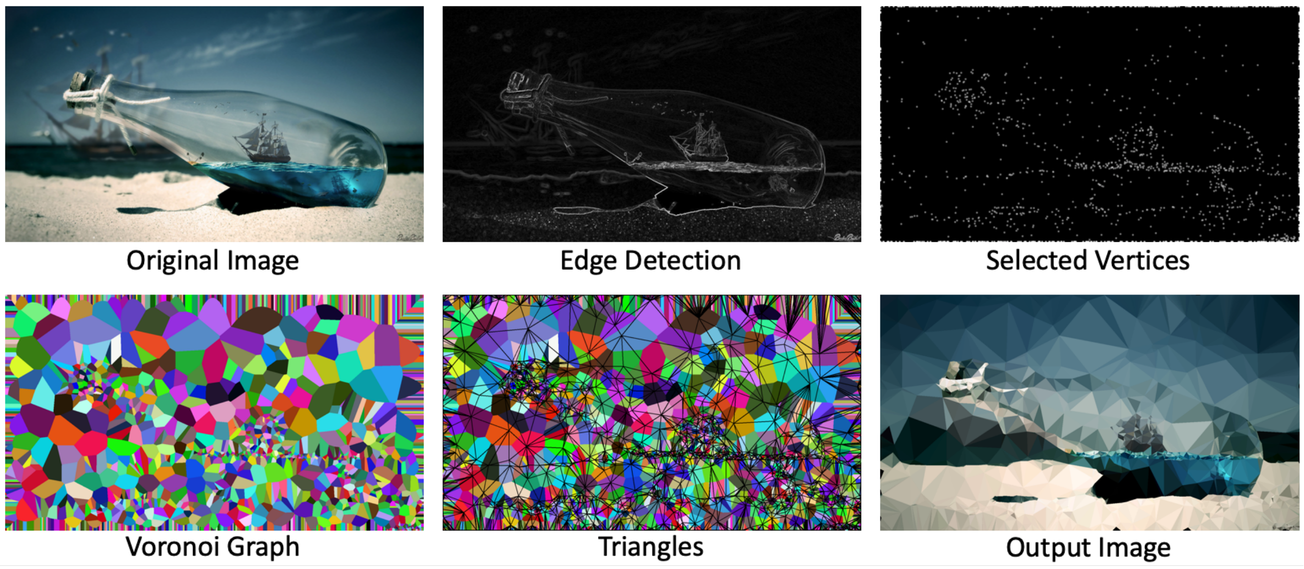
Since these methods are hard to parallelize, we used a third algorithm that is more suitable to parallel architecture. The basic idea is to first compute a Voronoi Graph (VG) of the original picture, which is known to be the dual problem of DT, then obtain the DT with the computed VG. Computing VG is suitable to parallelize on GPU. We will explain the details in the next section.

Methods

Overview

Here is an illustration of our workflow. It consists of several steps:

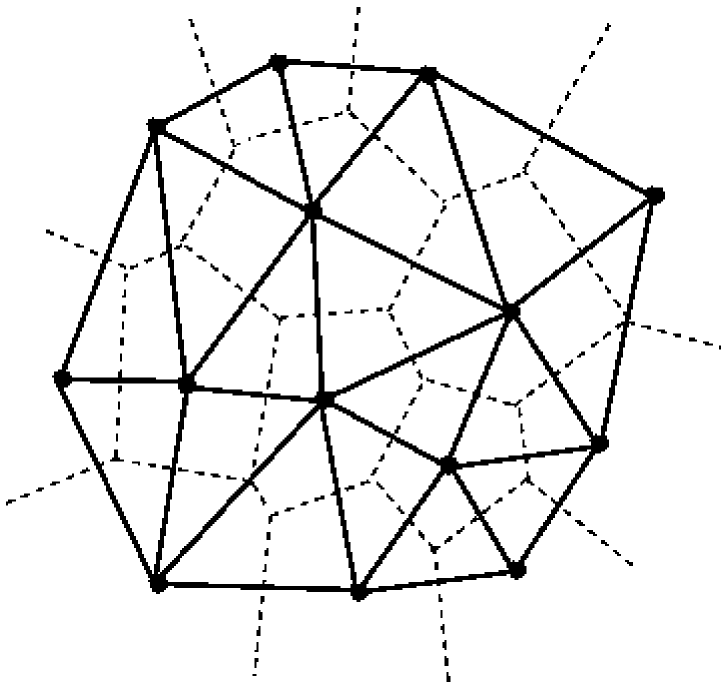
1. Detect the edges of the picture, to better choose the points to preserve the picture structure;
2. Randomly choose points, with more probability on edges than other parts;
3. Compute the VG of the points;
4. Obtain DT mesh from the VG;
5. Render the triangles and produce the final output.



Edge Detection

Voronoi Graph

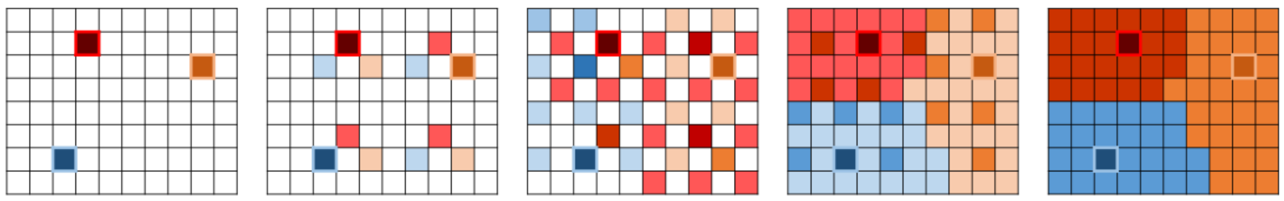
For triangulation, we used an algorithm that is proposed by Rong et al. The basic idea is, instead of directly computing DT, we first compute the Voronoi Graph (VG) of it on the picture. VG is a partition of a plane into regions according to their nearest points. VG is the dual problem of DT, if we obtained a VG (dashed lines), then connect the points of adjacent regions of the diagram gives us the DT (solid lines).



Computing VG on a picture is done by the Jump-Flooding algorithm, which will mark each pixel with its nearest neighbor point. In each iteration with a step size k, a pixel (x, y) will look at its eight neighbors $(x+i, y+j)$ where $i, j \in \{-k, 0, +k\}$, and try to find a closer point to it. The pseudo code for Jump-Flooding algorithm is:

```
owner = {}
step = picture_size / 2
while step>=1:
    for pixel_A in pixels:
        for pixel_B who is (step) away from pixel_A:
            if pixel_A has no owner, or owner[pixel_B] is closer than owner[pixel_A]:
                owner[pixel_A] = owner[pixel_B]
    step /= 2
```

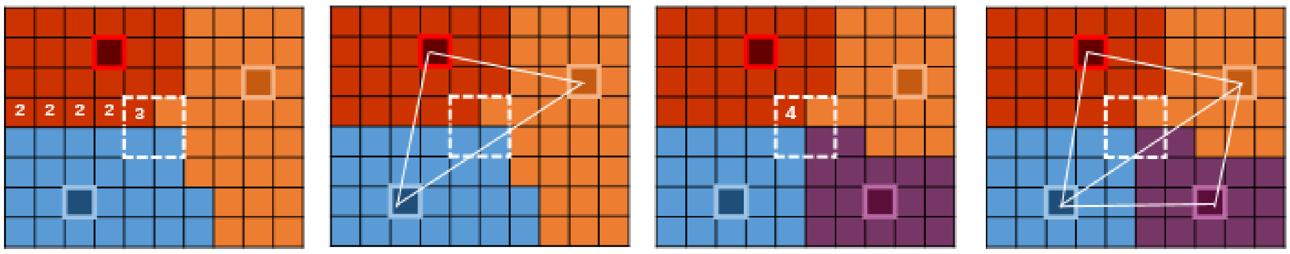
Here is a illustration for the steps for the Jump-Flooding algorithm on three points, with initial step = 4.



This algorithm is very GPU friendly since we can parallel the computation by each pixel. We can map each pixel to a thread on GPU, and each thread looks at the eight neighbors and update their owner point. There will be no contention if we use double buffers to implement this algorithm, since the updating process is fully synchronous.

Generating Triangles

After getting the VG, there is a neat trick to generate the triangle mesh in a fully parallel way. It turns out that the pixel map is sufficient to construct the triangles. Specifically, our task is find a 2x2 square in the pixel map that has 3 or 4 different owners. A square of 3 owners suggests those 3 regions intersect here, so one triangle should be generated to connect the 3 regions. Similarly, a square of 4 owners suggests there should be two triangles to connect the 4 regions. Here is an illustration of this process. The number in the pixel refers to the number of owners in the 2x2 square.



Since we cannot dynamically add triangles in CUDA, constructing the triangle mesh is a two step process.

First, we compute the total number of the triangles, and assign each pixel with their triangle index. This can be done by mapping each pixel to a thread, and each pixel checks himself and the three pixels on its right, bottom and bottom-right. If the total number of different owners is 3, then mark the pixel as 1; or if it is 4, them mark the pixel as 2. Otherwise it is 0.

After that, we produce an exclusive scan to get the prefix sum of the pixel map. The exclusive scan gives the total number of triangles and the index of each pixel. For example, if prefix_sum[pixel_A] = 10, and prefix_sum[pixel_next_to_pixel_A] = 12, we know that pixel_A has two triangles: the 10th and the 11th. We use `thrust::exclusive_scan` to carry out this process.

At last, we allocate an array whose length is equal to the number of triangles, then launch another kernel in which every pixel puts their triangle to the corresponding indices.

The whole process in generating triangles does not involve any contention.

Rendering

References

- 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 close 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.

Preliminary Results

	Edge Detection	Select Vertices	Generate Voronoi	Triangulation	Rendering	Other	Total
CPU - O0	60	50	6710	1430	180	60	8490
CPU - O3	70	80	1300	440	60	40	1990
GPU - O3	0.2	2.4	30	10	NI		350

As shown in the above form, we tested our current algorithm with a 1920x1080 image and 1000 random vertices. NI here means "Not Implemented". We are able to achieve about 4x overall speedup for now, compared to `-O3` compiled CPU code. We have not fine tuned the GPU version performance yet, but we suppose it is due to memory transferring from CPU to GPU. We believe a lot of memory transferring can be saved after we implemented the GPU version of edge detection and triangle rendering, which will improve the performance of our program. Also we are using `std::clock()` for clocking now. We may switch to a higher precision timer, like the `CycleTimer` used in HW2, to get more accurate profiling data.

Schedule

Time	Work	Status	Owner
10.29 - 11.4	Write the proposal, research for existing work	Done!	Both
11.5 - 11.11	Complete sequential version of Sobel Edge Detector and Jump Flooding	Done!	Both
11.12 - 11.18	Complete CUDA version of Delaunay Triangulation, prepare for Checkpoint	Done!	Both
11.19 - 11.25	Complete CUDA version of Sobel Edge Detector and Triangle Rendering	Working	Zhengjia
11.26 - 12.2	Performance tuning and analysis	Working	Weichen
12.3 - 12.6	Solve the current problem on the image boundaries		Weichen
12.7 - 12.9	Complete image input and output interface, get ready for demo		Zhengjia
12.9 - 12.15	Final Report		Both