Parallel Low-Poly Style Image Processing

106061203 電機21 徐浩宇

Introduction

Low-poly style image

Original Image

Low-Poly Image

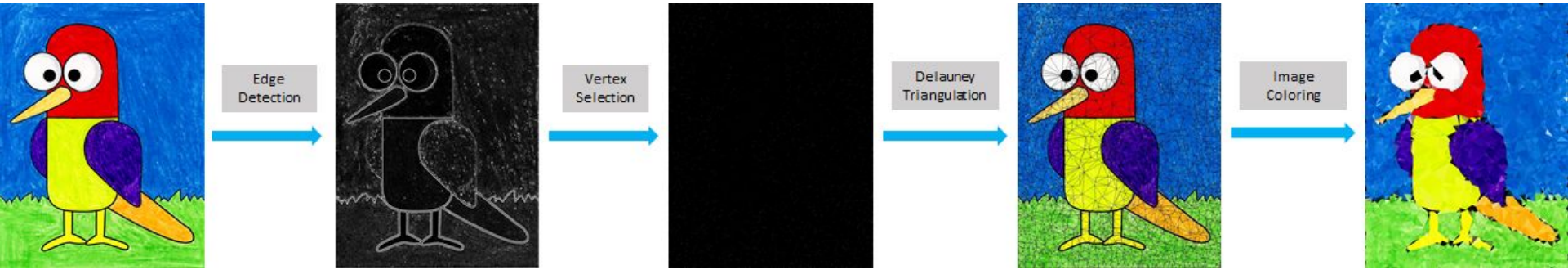
Pipeline Design

Low-Poly Style Image and Video Processing

Wenli Zhang [1], Shuangjiu Xiao [1], Xin Shi [1]
[1] Shanghai Jiao Tong University, 800 Dongchuan RD. Minhang District, Shanghai, China
lilyperfect@sjtu.edu.cn, xsjiu99@cs.sjtu.edu.cn, shinshi@sjtu.edu.cn

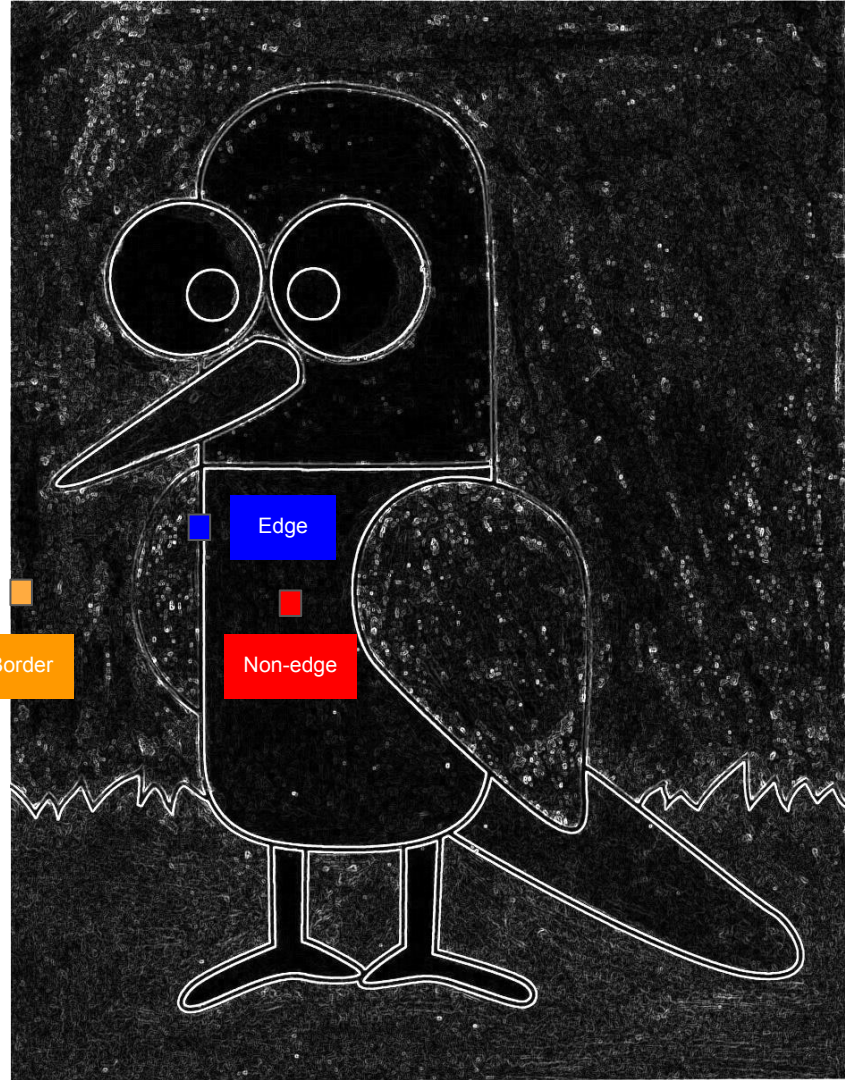
Zhang, W., Xiao, S., & Shi, X. (2015). Low-poly style image and video processing. 2015 International Conference on Systems, Signals and Image Processing (IWSSIP)

Pipeline


Edge Detection
Vertex Selection
Delauney Triangulation
Image Coloring

Edge Detection

$$filter_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad filter_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



Vertex Selection

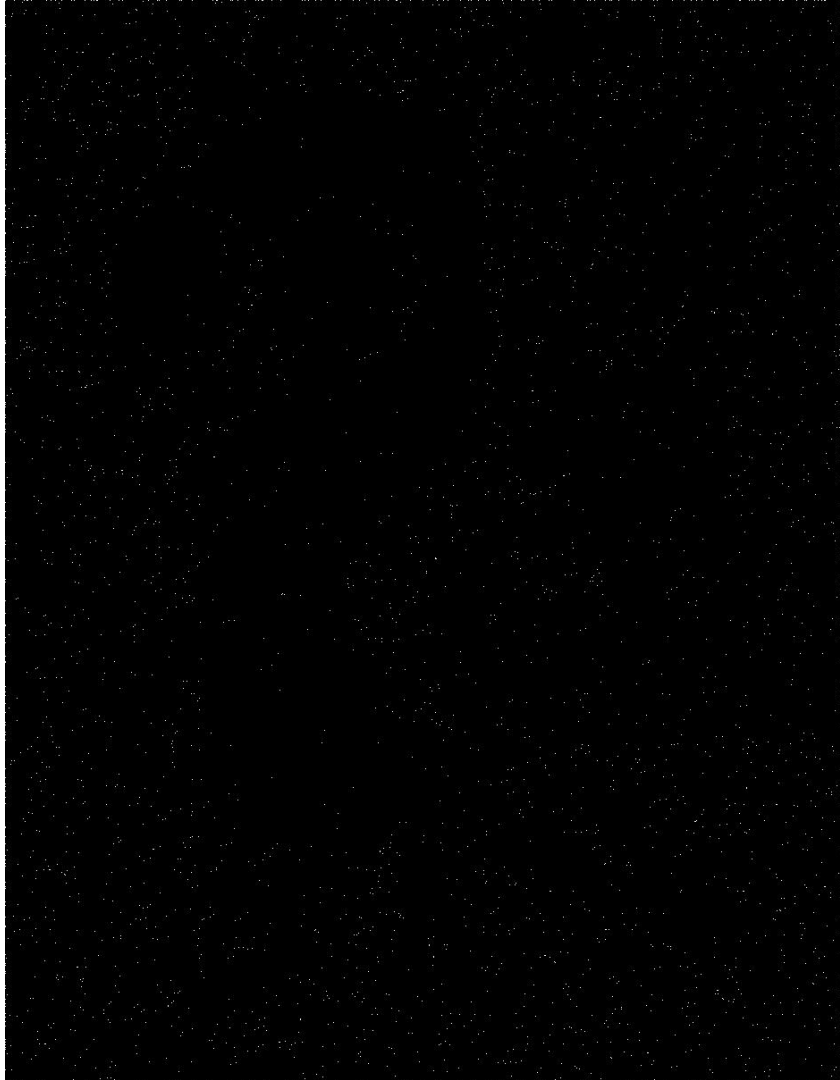
Vertex selection 各參數的設定:

<1> edge threshold: 20

<2> edge probability: 0.005

<3> non-edge probability: 0.0001

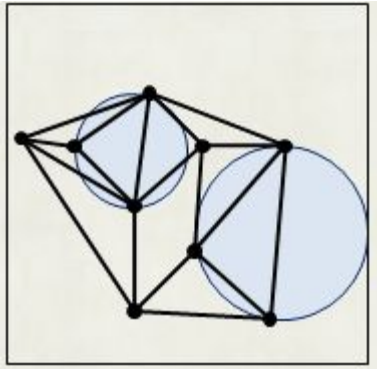
<4> border probability: 0.1

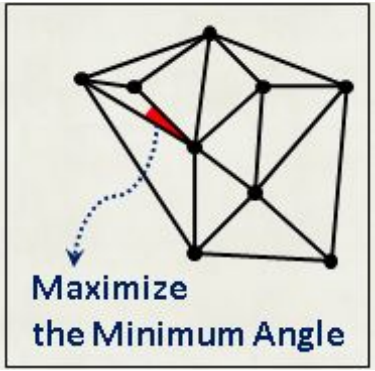


Delauney Triangulation

Delauney Triangulation滿足兩個性質:

- Maxmin Angle Triangulation (最小角度最大化) -> 避免生成狹長的三角形
- Empty Cirucumcircle Property (三角形外接圓內部沒有任何點)





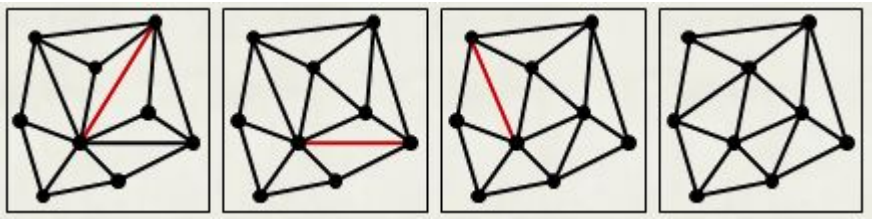
Delauney Triangulation

Delauney Triangulation常見演算法:

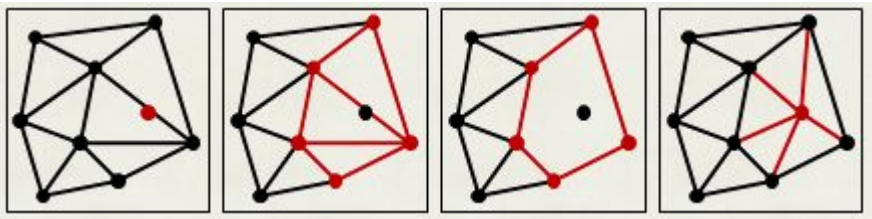
- Lawson's Flip Algorithm: O(N^2)
- Bowyer-Watson Algorithm: O(N^2)

Drawbacks: 兩種演算法不具有 parallel的特性, 前後會是彼此相關的。

Lawson's Flip Algorithm

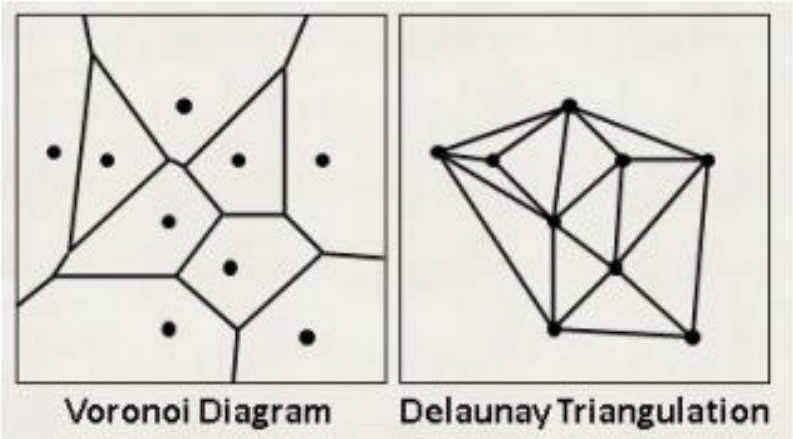


Bowyer-Watson Algorithm



Delauney Triangulation

The Delaunay triangulation of a discrete point set P in general position corresponds to the dual graph of the Voronoi diagram for P.

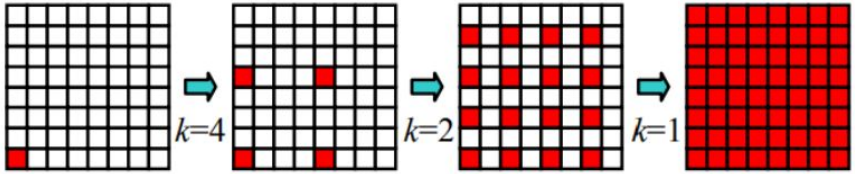


Delauney Triangulation

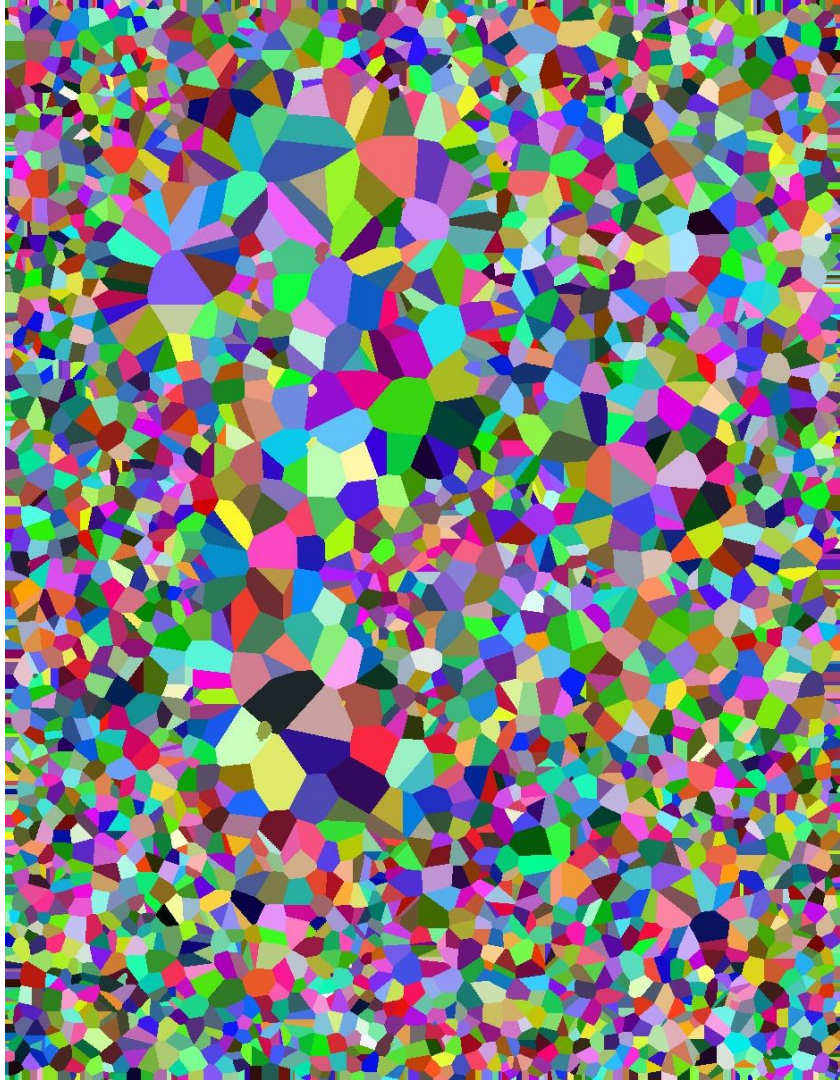
因此，可以將Delauney Triangulation分成兩個步驟：

- Voronoi diagram generation (via Jump Flooding Algorithm)
- Voronoi to Delauney conversion

Voronoi Diagram Generation



Jump Flooding Algorithm



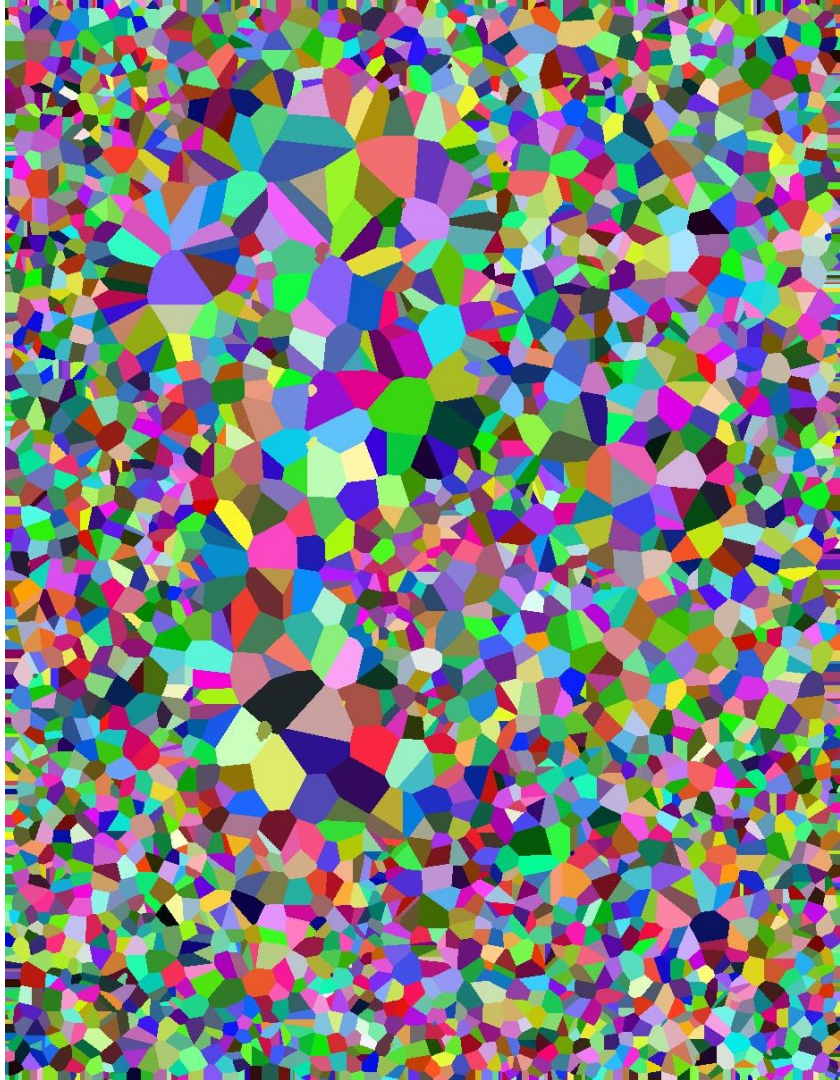
Guodong Rong, Tiow-Seng Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. Proceedings of the 2006 symposium on Interactive 3D graphics and games (ACM)

Voronoi Diagram Generation

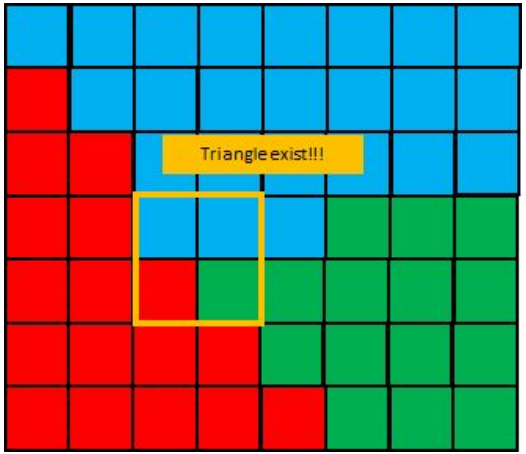
```
Procedure jump_flooding
Begin
  // flooding with decreasing step size
  For step=min(H, W)/2 to 1
    // iterating all pixels
    For each pixel p
      // iterating all 8 directions
      For each direction dir
        ni := neighbor pixel (p + dir * step)
        // get the nearest vertex (owner) of p
        If p not owned or distance(p, ni's owner) < distance(p, p's owner)
          Update p's owner with ni's owner
        End if
      End for
    End for
    step /= 2    // update step size
  End for
End
```

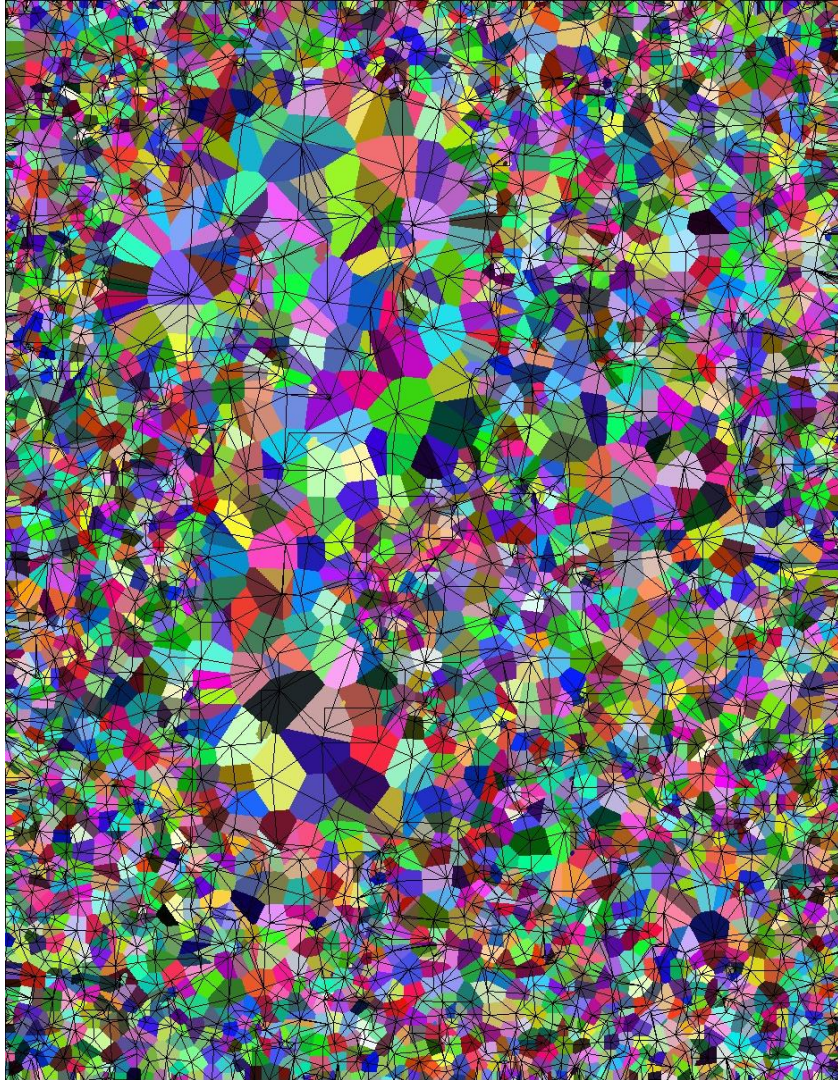
Not faster, but can be parallelize

$$Time\ Complexity = log_2 N * N^2 * 8 = O(N^2 log_2 N)$$



Voronoi to Delauney Conversion





Delauney Triangulation

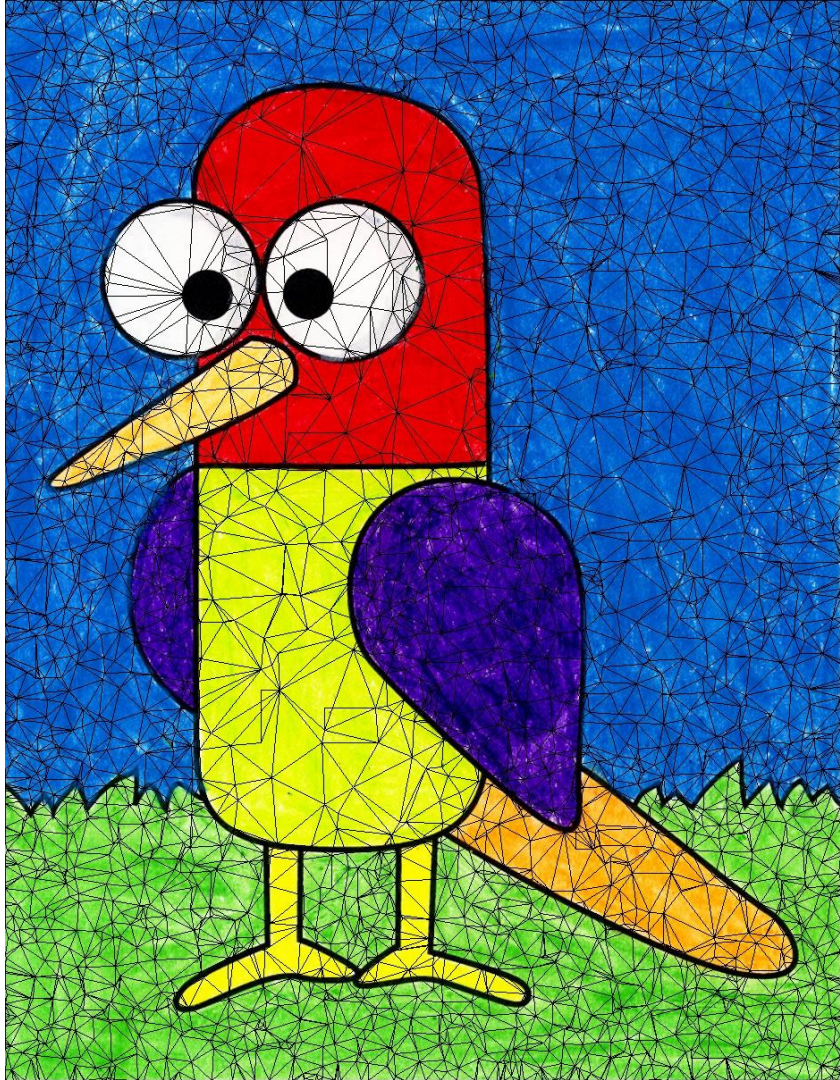
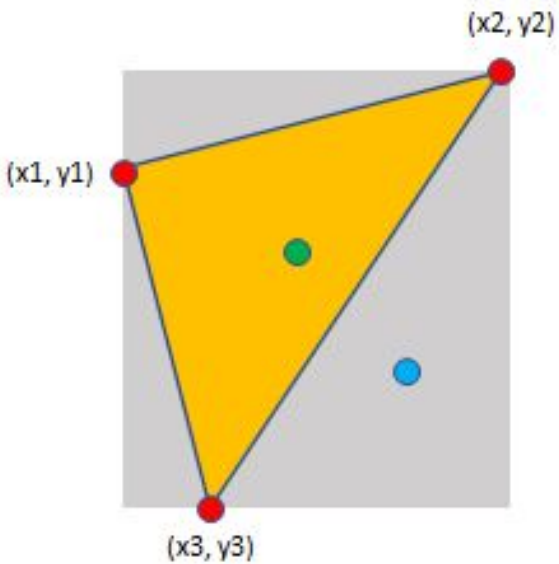
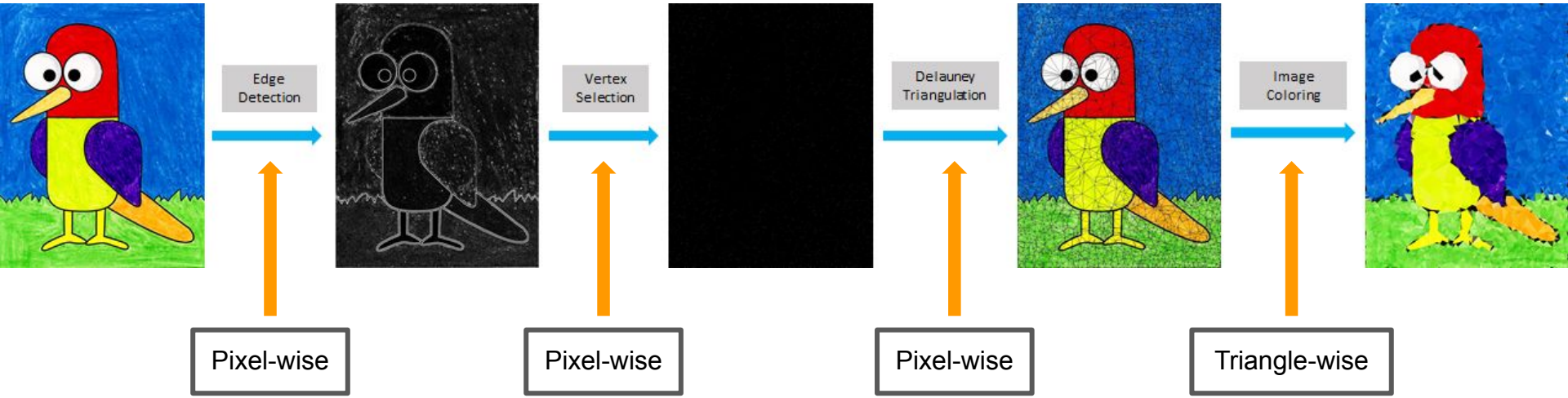


Image Coloring



Parallel Strategy

Parallel Strategy



Parallel Strategy

1. Edge Detection: **pixel-wise** parallelism, each thread computes a sum of filtering value
2. Vertex Selection: **pixel-wise** parallelism, each thread **initialize a random state** then generate a random value from uniform distribution to determine the pixel would be a vertex or not
3. Delauney Triangulation:
    a. Voronoi Diagram: for loop iterates all possible step size, in each iteration, perform **pixel-wise** parallelization, each thread updates the owner of a pixel given the neighbor pixel's information from all 8 directions.
    b. V to D Conversion: **pixel-wise** parallelism, each thread checks for a pixel, if this pixel is at the intersection point of multiple voronoi regions, then form a triangle.
4. Image Coloring: **triangle-wise** parallelism, each thread fills the pixel with color inside a triangle

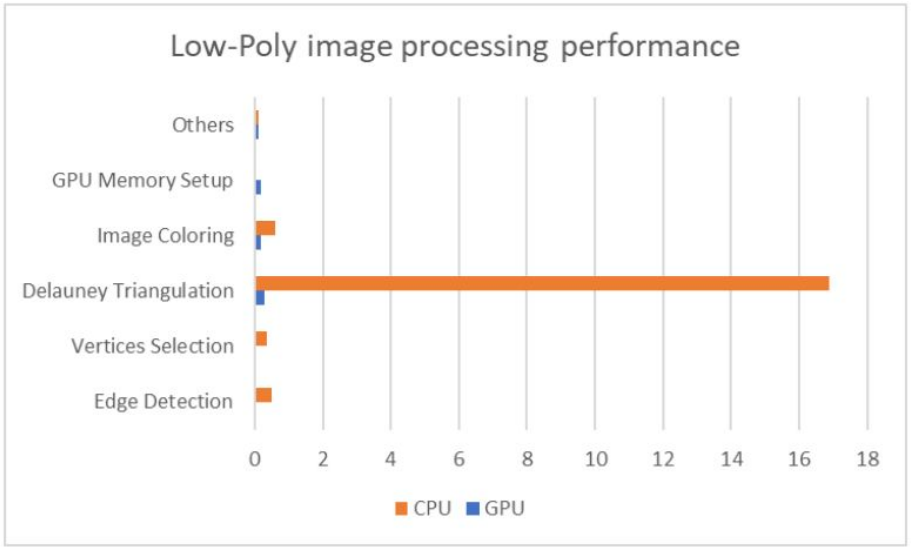
Performance & Optimization

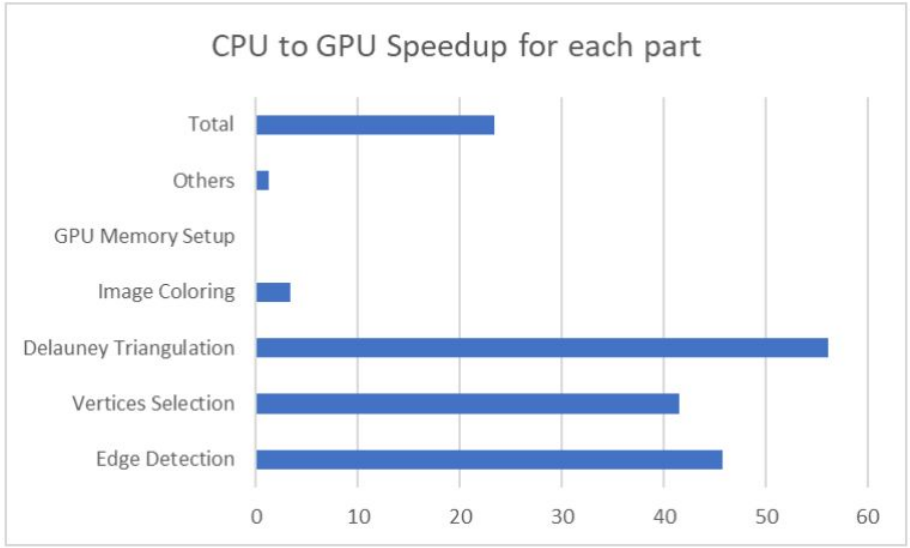
Performance

全部的測試都以burger.jpg為主:

- 長寬: 3247 x 4330
- 三角形數量: 24836

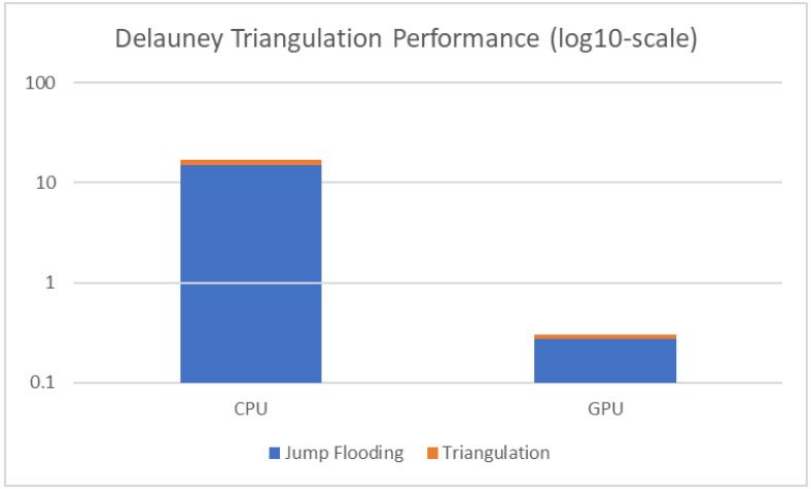
Performance

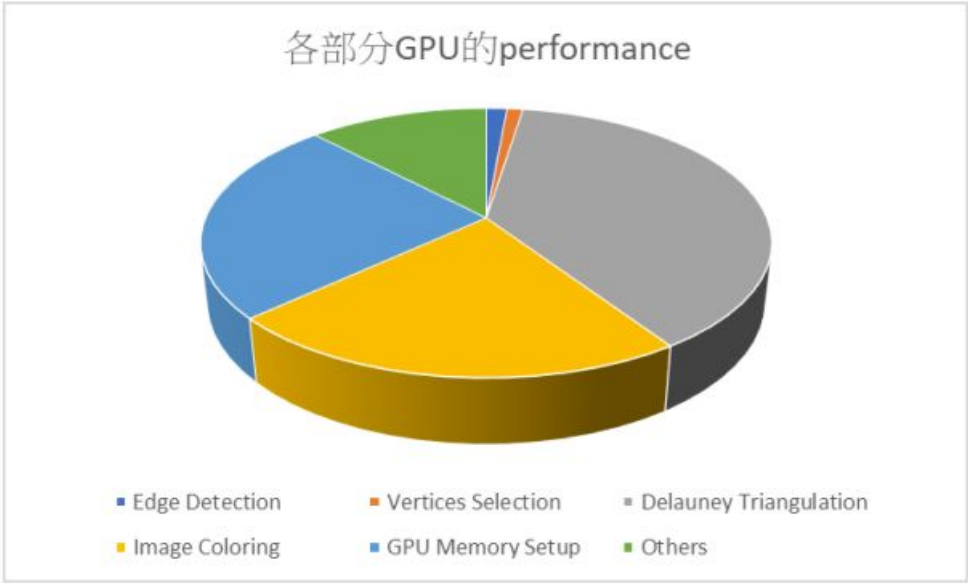




Pixel-wise parallelism acheives higher speedup than triangle-wise parallelism.

Performance





Jump flooding algorithm takes up most of the time in Delauney Triangulation.

Optimization

```
curand_init ( long long seed, long long sequence, long long offset, curandState_t *state)
```

- Random state initialization:

通常在做平行化亂數初始化時，會固定 seed 並輸入不同的 sequence number 以得到亂數值(如下圖)。雖然這樣的方式可以有比較好的 random 性質，即267次後才會重複 loop，但是在 CURAND 會花費大量的時間。

因此，我修改了初始化的方式，改成使用不同的 seed 加上相同的 sequence number (0)，有效地將時間從 115s 縮減為 0.0052s。通常不同的 seed 之間產生的數值不太會 statistically correlated，而且這次的 task 對於亂數品質的要求不用這麼完美，所以這樣的加速是可行的。

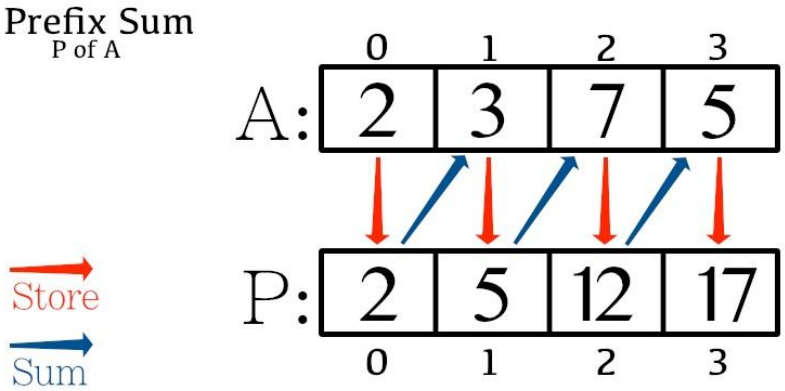
```
curand_init(SEED, id, 0, &state[id]);
```

```
curand_init( (SEED << 20) + id, 0, 0, &state[id])
```

Optimization

- Use “Thrust” for prefix sum:

原本我的方法是先將 GPU 的資料搬到 CPU，在 CPU 端做 prefix sum 後再搬回去 GPU；之後，我改使用 Thrust library 直接在 GPU 做 prefix sum，可以把時間從 0.0412s 降至 0.0008s，省去資料搬運產生的 latency。



Optimization

- 2D jump flooding:

  使用 2D 的 execution configuration，在 step size 小的時候會有 locality 特性，能夠更 consistent 地去跟新每一個 pixel 的 owner，有效減少 noise pattern。

Voronoi 1D

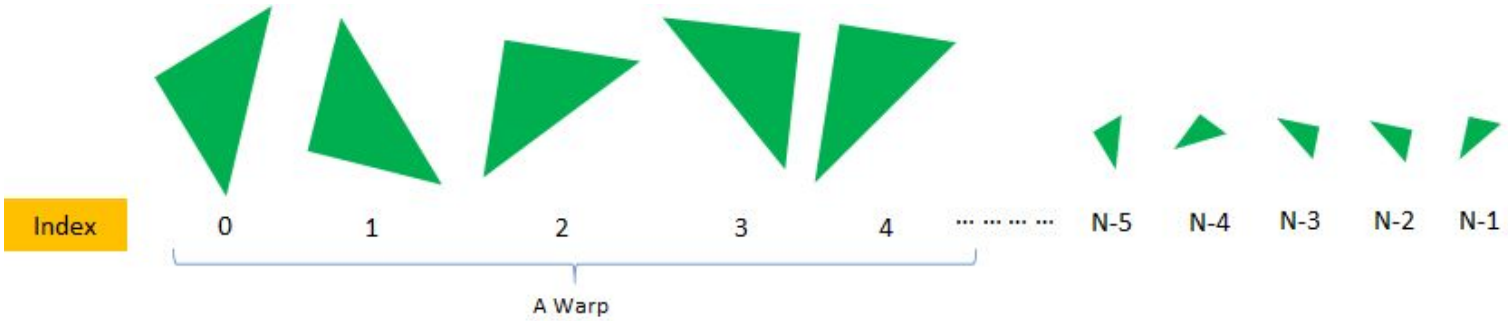
Voronoi 2D

Optimization

- Sort-by-key on triangle's area:

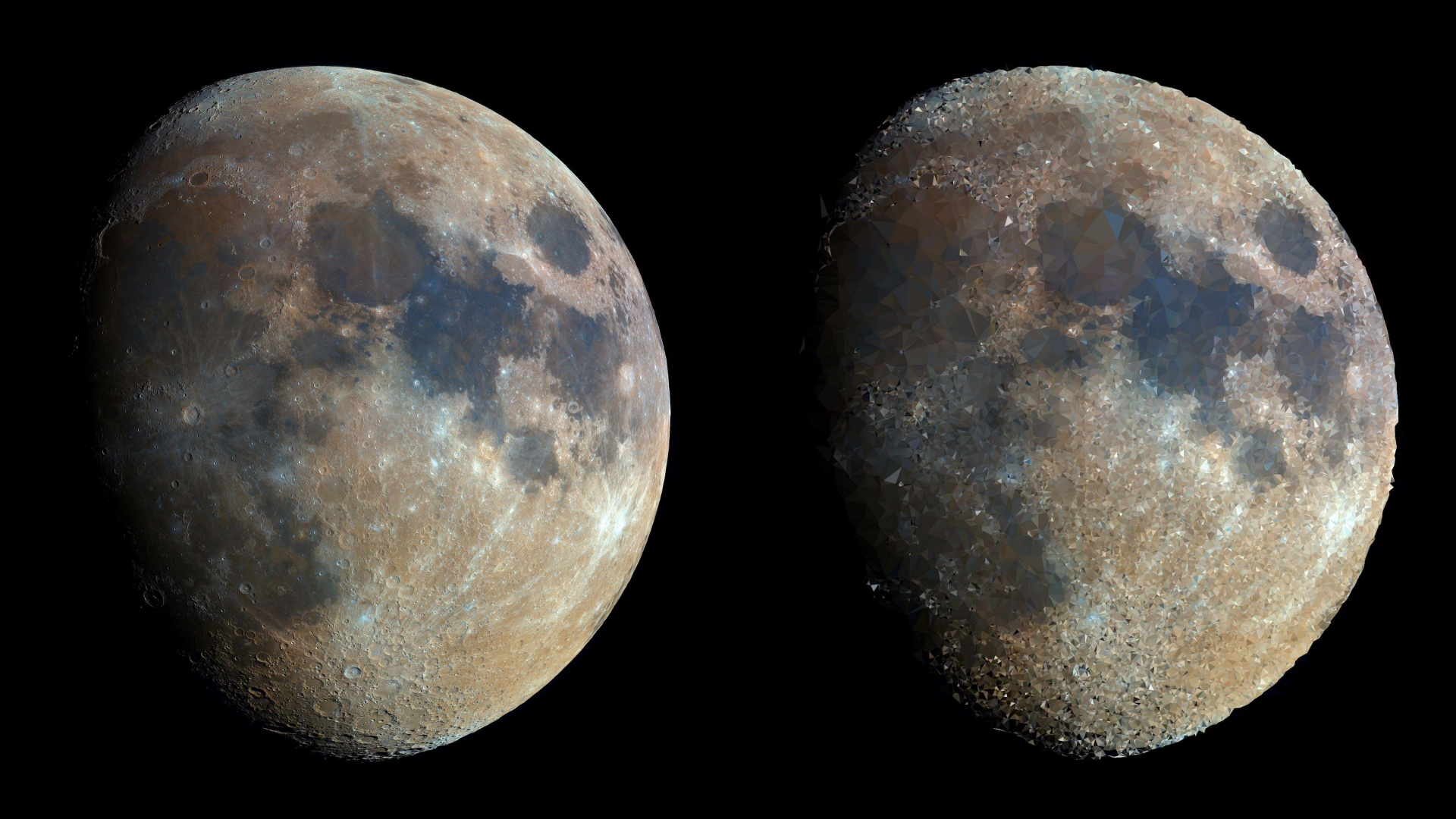
存在著 load imbalance 的問題，因為每一個三角形的大小不同，所以每一個 block 的運行時間會被裡面最大的三角形 dominate，導致運算效率不夠高。最直覺的方法便是依照面積大小 sorting，如此一來，GPU 分配的同一個 block 裡面的三角形大小會更接近。

我先平行計算出每一個三角形外接 bounding box 的面積，接著利用 Thrust library 裡面的 sort-by-key 函式，讓三角形依照其 bounding box 面積去做排序。經過測試發現，運行時間有效從0.268s降低至 0.179s！

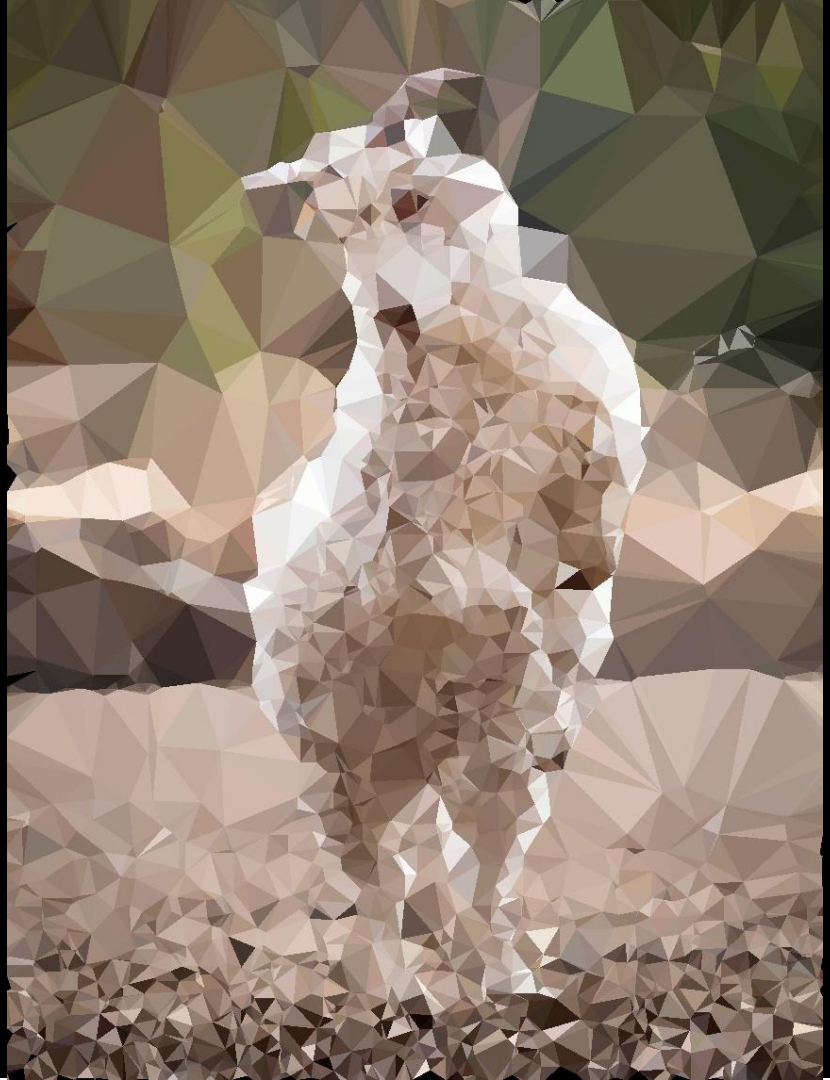


Qualitative Results

High-resolution Image



Low-resolution Image


LAKERS
23


23


23


imgflip.com

The End

Thanks for your listening!