Low Poly Effect Parallel Renderer

Author: Felicity Xu (hanyux), Otto Wang (aow)

Web Page: Low-Poly-Effect-Parallel-Renderer

1 SUMMARY

We are going to develop a low-poly art effect renderer for images using C++ and CUDA on GPUs, focusing on achieving high-quality effect while enhancing computational efficiency through parallel processing.

2 BACKGROUND

Our project aims to parallelize the process of rendering low-poly art images, which is inherently compute intensive due to the multiple transformation steps an image undergoes. The goal is to develop an algorithm that leverages the parallel computation power of modern GPUs.

Step 1: Gaussian Blur: We sill start with preprocessing the image using Gaussian blur. This involves convolving each pixel with a Gaussian kernel to smooth the image. The independent nature of pixel processing here should make it ideal for parallelization.

Pseudocode for Gaussian Blur:

```
GaussianBlur(image, kernelSize, sigma):
    kernel = GenerateGaussianKernel(kernelSize, sigma)
    for each pixel in image:
        sum = 0
        for each kernelValue, offsetX, offsetY in kernel:
            neighborPixel = GetPixel(image, pixel.x + offsetX, pixel.y +
  offsetY)
            sum += neighborPixel * kernelValue
            SetPixel(newImage, pixel.x, pixel.y, sum / kernelSum)
    return newImage
```

Step 2: Edge Drawing and Vertex Extraction: (Topal 2012) This step is critical for defining the structure of the low-poly art. Edge drawing in essence involves the following steps:

- Computation of the gradient magnitude and edge direction maps. Using the smoothed image, compute the horizontal and vertical gradients, G_x and G_y, respectively. Then calculate gradient magnitude $G = \sqrt{G_x^2 + G_y^2}$.
- Extraction of the anchors. Key points, or anchors, are identified based on local gradient maxima. These points serve as vertices for edge tracing.
- Connecting the anchors by smart routing. Use a smart heuristic edge tracing procesure to map edges. In this step, for each anchor, it proceeds in four directions to detect whether neighbors are edge points until a point in the neighboring direction is not an edge point.
- For all edge point, use a sampling method to select ones that are important to use for triangulation.

For the first two steps, we can divide the image into regions and assign one region to each thread for proecessing. For the next two steps, one possible parallelization is to let each thread take a group of anchor points to determine edge points. Overall, edge drawing algorithm shows great potential for speedup through parallelization.

Step 3: Delaunay Triangulation: The heart of low-poly art rendering lies in Delaunay triangulation, where we construct a mesh of triangles using the vertices obtained from the edge drawing step. There are many ways to compute Delaunay Triangulation, however, research has shown that Voronoi Diagram based approaches present best opportunities for parallelization (Lo 2012).

Step 4: Final Color Adjustment: The last simple step is to color each triangle based on the pixels beneath it in the original image. This process is highly parallelizable as it involves no inter-triangle dependencies, allowing each triangle to be colored in isolation on separate processing threads.

Parallelism is beneficial in our project due to the high volume of independent calculations required for each pixel or vertex. The Gaussian blur, for instance, involves convolving a kernel over each pixel, while color adjustment entails computing average or dominant colors for individual triangles. These are tasks that can be distributed across multiple cores in a GPU, significantly accelerating the processing time. Delaunay triangulation is more challenging to parallelize due to its global nature and dependencies but we can still benefit from data-parallelism in Voronoi-based DT algorithms.

3 CHALLENGE

This problem is challenging not only because to generate good low poly effect, it requires good edge detection algorithm. But also because to achieve efficient computation, it requires careful consideration at each step.

The main challenges of rendering come from the two steps edge drawing and Delaunay triangulation.

Edge Drawing Step:

- Load balancing: images that have regions with varying edge complexities can lead to an uneven distribution of workload among the threads. How to distribute work among the threads evenly would be something to take care of.
- Divergent Execution: GPU performance can be optimal when threads execute the same operation concurrently. However, when we are tracing an edge path, the result can be indeterminate, leading to threads following different edge paths, causing an performance decrease.

Delaunay Triangulation Step: While Delaunay triangulation has many algorithms, such as divide and conquer or Bowyer-Watson incremental algorithm, most of them are not suitable for parallelization. Many of these algorithms involves steps that are inherently sequential. Take Bowyer-Watson for example, points in the triangulation are added one at a time depending on the current status of the triangulation. This sequential logic greatly hinders the possibility of parallelization. Therefore, after some research, we have decide that Voronoi Algorithm should be the best a lgorithm so far for Delaunay triangulation due to its great data parallelism. (L0 2012) It takes a different approach from many other basic algorithms and uses the dual of the Delaunay triangulation – the Voronoi diagram to help construct the Delaunay triangulation, which allows us to exploit the data parallelism of the Voronoi diagram.

Apart from algorithm logic seuqential dependencies, there are also many other things about Delaunay triangulation that makes it hard for parallelization:

- **Large Memory Requirements:** For large images, the algorithm requires significant memory resources, which can be a limiting factor in parallel execution.
- **False Sharing:** When threads write to pixels that are close to each other on the image, it may lead to false sharing over a single cache line, resulting in inefficient memory usage and potential performance degradation.

4 RESOURCES

We will develop our low-poly effect parallel renderer from scratch based on C++ and CUDA using GHC machines containing NVIDIA GeForce RTX 2080B GPUs.

For the overall steps of rendering, we are referencing the paper "Artistic low poly rendering for images" [2]. Meanwhile, we aim to delve deep into each individual step. For the edge drawing algorithm, we will be relying on the paper "Edge drawing: a combined real-time edge and segment detector" by Chihan Topal [5]. And for the Delaunay triangulation step, we are still researching on the different algorithms for Delaunay triangulation. Currently, we are considering using the algorithm described in this paper "Computing two-dimensional Delaunay triangulation using graphics hardware" by G Rong [5]. This paper discusses an implementation of using Voronoi diagram to obtain Delaunay Triangulation on GPU.

5 GOALS AND DELIVERABLES

Goals Plan to Achieve

- **CPU Implementation**: Develop an initial version of the low-poly art effect renderer on the CPU, utilizing existing software packages without specifically focusing on CPU optimizations like SIMD or multi-threading, except where such optimizations are inherently supported by the chosen packages. This version will act as a baseline for our project, enabling the conversion of images of any resolution into low-poly art. It will adhere to the four essential steps outlined in our methodology, providing a standardized approach to creating low-poly images.
- **CUDA Enhancement**: Transit the renderer over to CUDA, aiming to make it at least 5 times faster for 1920x1080 images compared to our initial CPU version.

  Justification: Our early investigations show that easier tasks, like Gaussian Blur and some parts of Edge Drawing, work really well with parallel processing. This means we can speed these parts up a lot (more than 10 times faster) because they fit well with GPU acceleration. Since these simpler tasks are a big part of the whole process, focusing on speeding them up can help us achieve our overall goal of making everything 5 times faster. Even the more complex parts can get a bit of a speed boost from using multiple processors.
- **Performance Optimization**: Optimize the renderer by experimenting with various parallel algorithms for the critical steps, aiming for at least a 50% speed increase over the initial CUDA version.

  Justification: While more intricate stages of the rendering process may not straightforwardly reach

such theoretical maxima, our preliminary assessments and available methodologies indicate feasible strategies to amplify their efficiency by at least 50%. We're planning to test these out to see how much more we can speed up the process.

Goals Hope to Achieve

- **Video Rendering**: Adapt the algorithm to enable video processing, while eliminating possible jittering artifact.

  Justification: Directly applying the algorithm designed for individual images to each frame of a video can introduce noticeable jittering artifacts, as vertices may be selected inconsistently across frames, resulting in significant shifts in triangle positions and coloration from one frame to the next. This issue, highlighted in [7], stems from the random selection of vertices in each frame. We plan to overcome this challenge by implementing strategies that include the use of inherited parameters or other methods to ensure consistency in vertex selection and triangle formation across consecutive frames, thereby reducing jitter and enhancing visual continuity.

- **Video Performance Optimization**: Leverage temporal coherence between frames, aiming for a performance boost of more than 20% over the initial CUDA version.

  Justification: The potential for performance improvement lies in leveraging the sequential continuity of video frames, a concept supported by findings in [7]. By exploiting this temporal coherence, we aim to implement optimizations that are not applicable to single-image processing. These optimizations could include more efficient memory usage, reduced computational redundancy, and the anticipation of vertex positions based on previous frames, thereby significantly boosting rendering performance for video content.

Demo to Show

- **Methodology and Performance**: Present a detailed exposition of our approach, accompanied by a gallery of transformed images and performance metrics across various resolutions.
- **Interactive Platform**: Launch an accessible web platform allowing users to upload images (and potentially videos) for rendering. This platform will offer the option to select between CPU and GPU processing modes, providing insights into performance differences based on user-specific inputs.

6 PLATFORM CHOICE

We will develop our low-poly effect parallel renderer based on C++ and CUDA using GHC machines containing NVIDIA GeForce RTX 2080B GPUs. This decision is driven by the inherent advantages of GPUs for our rendering workload. GPUs are specifically designed for image processing tasks. The architecture of GPUs has large shared memory and high throughput, which is optimal for our rendering task as threads are accessing large amount of data. The low-poly rendering process involves several steps, such as Gaussian blur, edge detection and Delaunay triangulation. All of which can be effectively parallelized on GPU with CUDA which have the ability to manage thousands of threads simultaneously.

Another important point is that as we are hoping to add rendering feature for video (if things go well). In light of this, GPUs can easily scale to accommodate the increased workload of processing video frames while maintaining high performance and efficiency.

7 SCHEDULE

Week 1: March 25 - March 30

- Start the basic structure of the renderer in C++ and CUDA. Finish loading and outputing images, viewing images, etc.
- Research any relevant and existing implementations of low-poly rendering and parallel processing techniques.

Week 2: March 31 - April 6

- For first half of the week, outline and finish Gaussian Blur step for the image rendering.
- For second half of the week, start coding on the edge drawing algorithm, aim to finish extraction of anchors and edge routing.

Week 3: April 7 - April 13

- Wrap up edge drawing algorithm, including vertices finding. This is the basic version of our edge drawing step.
- Attempt a basic version of delaunay algorithm and test the baseline performance.

Week 4: April 14 - April 20

- Refine the edge drawing step parallelization to increase speedup. achieve goal speedup
- Refine the final color applying step, this part should be straightforward.

Week 5: April 21 - April 27

- Refine the Delaunay triangulation step parallelization to increase speedup to our goal.

Week 6: April 28 - May 4

8 REFERENCES

[1] Chen, Long, and Jin-chao Xu. ”Optimal delaunay triangulations.” Journal of Computational Mathematics (2004): 299-308.

[2] Gai, Meng, and Guoping Wang. ”Artistic low poly rendering for images.” The visual computer 32 (2016): 491-500.

[3] Lo, S. H. ”Parallel Delaunay triangulation in three dimensions.” Computer Methods in Applied Mechanics and Engineering 237 (2012): 88-106.

[4] Ng, Ruisheng, Lai-Kuan Wong, and John See. ”Pic2Geom: A fast rendering algorithm for low-poly geometric art.” Advances in Multimedia Information Processing–PCM 2017: 18th Pacific-Rim

Conference on Multimedia, Harbin, China, September 28-29, 2017, Revised Selected Papers, Part II 18. Springer International Publishing, 2018.

[5] Rong, Guodong, et al. "Computing two-dimensional Delaunay triangulation using graphics hardware." Proceedings of the 2008 symposium on Interactive 3D graphics and games. 2008.

[6] Topal, Cihan, and Cuneyt Akinlar. "Edge drawing: a combined real-time edge and segment detector." Journal of Visual Communication and Image Representation 23.6 (2012): 862-872.

[7] W. Zhang, S. Xiao and X. Shi, "Low-poly style image and video processing," 2015 International Conference on Systems, Signals and Image Processing (IWSSIP), London, UK, 2015, pp. 97-100, doi: 10.1109/IWSSIP.2015.7314186.