Parallel 3D Computer Game

E4750_2018Fall_GAME_report

Zian Zhao (zz2558)
Columbia University

Abstract

This project is mainly an engineering problem: to build a computer game from scratch with PyOpenCL. Parallel Computing is widely used in image processing. The core problem is to render a 3D city model into a 2D RGB projection image. After studied 3D modeling and different 3D rendering algorithms, I finally built the game and optimized it with shared memory, improved the time complexity from O(N) to O(sqrt(N)).

1. Overview

1.1 Problem in a Nutshell

The problem can be separated as two parts: How to build a game; How to optimize the code with parallel tools to make it faster.

In the real world, 3D models are object models, allowing users freely zoom in or out without any blur (reference [1] Fundamentals of Computer Graphics). 3D rendering algorithms are about object-models rendering. Due to compatibility reason, I have to render a bitmap model, design algorithm about bitmap rendering on my own (see 2.2.1 3D Modeling section).

Despite the hard work of coding and debugging, it is a lot of fun to develop a colorful game with tools you used in homework! Since I did everything by my own from scratch. There is not much prior work.

2. Description

Below I will discuss project from three parts: 3D Modeling, Projection Algorithm, and Optimization. In each part I will demonstrate the challenges, problem formulation and design, then finally software implementation.

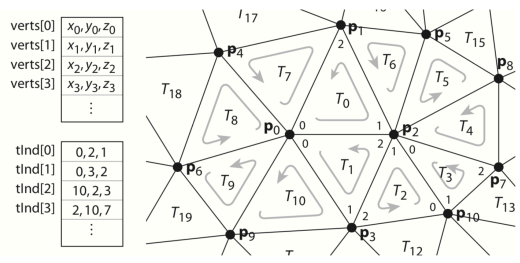


Figure 12.8. A larger triangle mesh, with part of its representation as an indexed triangle mesh.

Object Model
From Fundamentals of Computer Graphics

2.1. Objectives and Technical Challenges

To do the 3D rendering, first we need to have a 3D model. That model must be compatible with NumPy and PyOpenCL. Therefore I have to build a 3D model from scratch. I also need to make sure the model that I build itself is as I expected. The correctness of 3D model is the foundation of projection.

The following challenge is the projection algorithm. I have to calculate the direction of each ray correctly, which needs some works on linear algebra.

The third challenge is about optimization. I send a 3D model in, then get a projection RGB image back. There are a lot of parameters to configure. I have to handle situations like out of boundary etc.

The last challenge is to build a user interface for the game. However I cannot make it because this is a program running on a server using a strange "sbatch" command, making it impossible to use python input function. Besides, UI is nothing to do with parallel computing. So I finally give up the UI and produce a video instead.

2.2. Problem Formulation and Design

Flowchart:

3D Modeling => Projection => Optimization

2.2.1 3D Modeling

The 3D model is built once and used many times. Therefore the code of 3D Modeling is non-parallel. The model is built once and handled by the parallel projection code every time with a set of new projection parameters. To make it compatible with PyOpenCL, its data type is NumPy array.

Object Model vs. Bitmap Model:

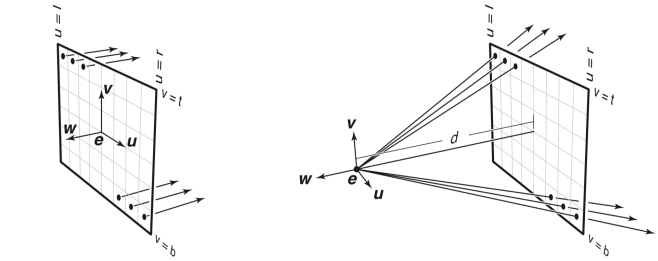
```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 1. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 1. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 1. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Bitmap Model (a slice of 3D matrix where z=0)
1 stands for building block, 0 is air

Although object-oriented triangle mash model is mostly used real game, I chose to construct bitmap model in this project because object-oriented model is:

1. Costly to construct,
2. Difficult to visualize for developers,
3. Difficult for implement in C Language kernel.

These are irrelevant issues with parallel computing. So bitmap is a better choice in this case.



Parallel vs Perspective Projection
From Fundamentals of Computer Graphics

2.2.2 Projection Algorithm

There are many ways to do the projection, which provides different visual effect. You can choose oblige or orthographic, parallel projection or perspective projection. My choice is orthographic perspective projection, which have a better 3D effect.

Linear Algebra

Here we note the index of 3D model as (x, y, z), the index on the projection screen, the blue rectangle in the diagram, as (i, j).

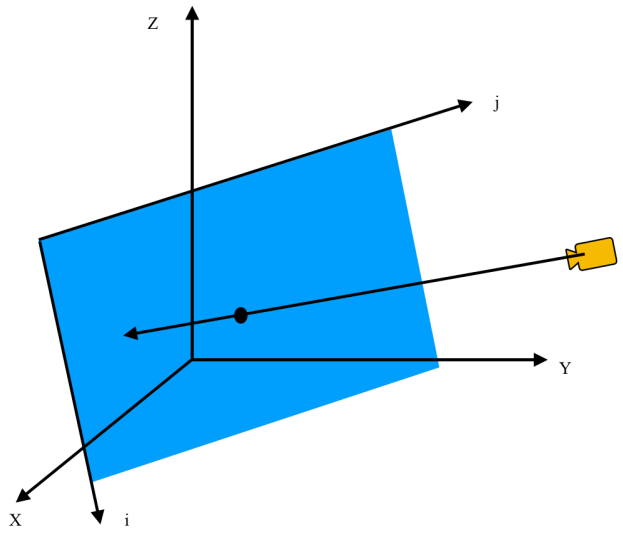


Diagram of Projection

Note $\hat{d}$ as direction of camera, $\hat{p}$ as position of camera,

The center of 3D model is (0, 0, 0).

Let the camera focus to the center of 3D model

$\hat{d} = -\hat{p}/|\hat{p}|$

To get the direction $\widehat{d_{i,j}}$ for each pixel (i, j),

first we need to get direction of $\hat{\imath}$, $\hat{\jmath}$

$\hat{\imath} \perp \hat{\jmath} \perp \hat{d}$; $\hat{\jmath} \parallel xOy$;

$\hat{\jmath} = \frac{\hat{\jmath} \times \hat{z}}{|\hat{\jmath} \times \hat{z}|}$, $\hat{\imath} = \hat{d} \times \hat{\jmath}$;

$(\Delta i, \Delta j)$ is the relative position of (i, j) to the center of screen

$\widehat{d_{i,j}} = \hat{d} + \Delta i\hat{\imath} + \Delta j\hat{\jmath}$;

Then we check the pixel value starting from $\hat{p}$,

direction $\widehat{d_{i,j}}$, all the way to the boundary of the model

Pseudo-code & Complexity

```
for each pixel on the screen:
  compute viewing ray direction: dij;
  find the first object hit by the ray;
  set pixel value on the screen to the hit point;
```

With PyOpenCL, I can have each thread to compute for each pixel and viewing ray.

Assume N is the size of one dimension of the 3D model, Time Complexity for each thread is: O(N), for you check each pixel on the ray.

Assume [Si, Sj] is the size of screen, number of threads we need are: Si*Sj

2.2.3 Optimization

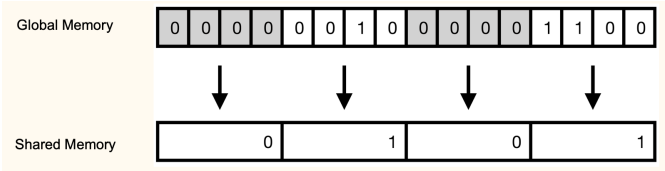
With PyOpenCL, I can have each block to compute for each pixel, and have each thread to compute part of the viewing ray, finally let one thread in the block to summary.

Pseudo-code & Complexity

```
for each pixel on the screen:
  compute viewing ray direction: dij;
  divide the ray to M parts;
  find the first object hit by each part of the ray;
  find the first object hit by ray from all parts;
  set pixel value on the screen to the hit point;
```

N is the size of the 3D model, Let M = sqrt(N), Time Complexity is: O(N^1/2)+O(N^1/2) = O(N^1/2).

We need Si*Sj blocks and M threads in each block, overall Si*Sj*M threads.



blockshape = (1, 1, M)

globalshape = (Si, Sj, M)

We can put the intermediate result to shared memory for optimization.

2.3 Software Design

2.3.1 3D Modeling: City.py

I could build the 3D model as a 3D matrix of RGB value, but that would be costly for storage, also takes a lot more copy time. So I build a 3D matrix of int, and map its value to a set of RGB colors when rendering the model to 2D image.

To build a city model, I first build a function to generate one building. Then I send a lot of random parameters to that function to construct a city.

This program can generate two 3D model in format of NumPy array matrix, a simple one for debug, and a city model.

2.3.2 Projection & Optimization: Projection.py

Using PyOpenCL, this code provide 3 projection computing mode:

naive: Algorithm as 2.2.2 Projection Algorithm. Each thread computes one viewing ray.

optimized: Algorithm as 2.2.3 Optimization. Each block computes one viewing ray, using shared memory.

colorful: Algorithm the same as naive, 2.2.2 Projection Algorithm. The difference is I add some gradient color effect to the result:

```
// x, y, z is the first hit object position
// L is the size of 3D model
value = 3Dmodel[x][y][z]
if (value != 0) {
parameter_r = (x+y)/L/2
result = parameter_r*color[value].start +
        (1-parameter_r)*color[value].end
}
```

2.3.3 Play the Game: Game.py

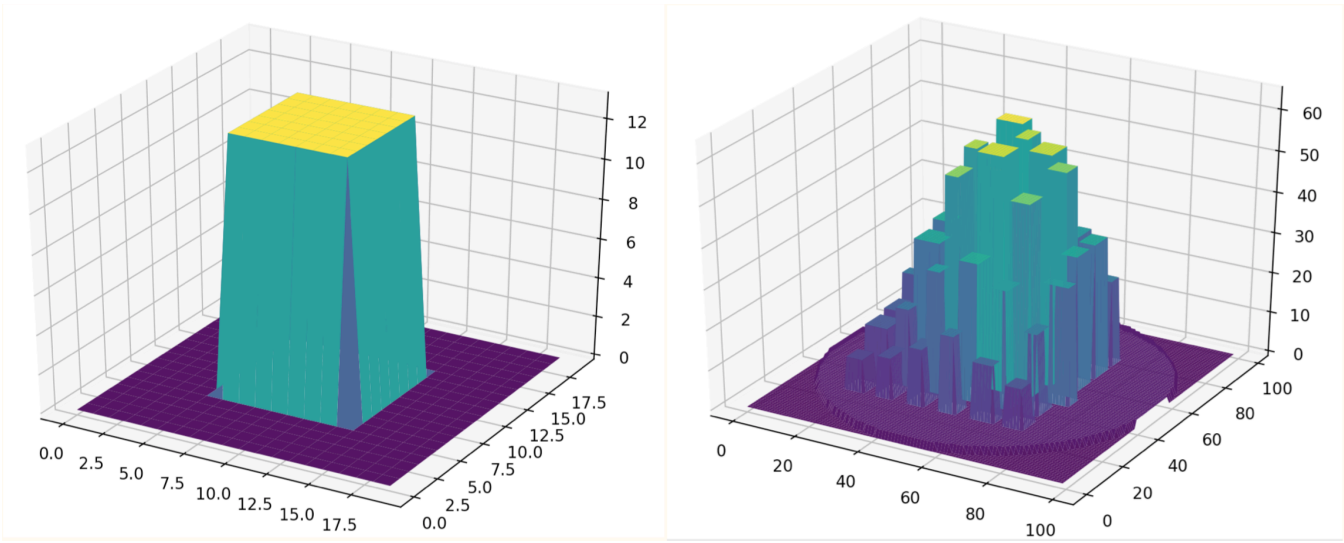
This program imports functions from Projection.py.

It plots the projection images and saves them into view.png. The latter result will cover the former result. To see the result, before running Game.py, open another terminal and input:

$ xdg-open view.png

2.3.4 Optimization Speed: Runtime.py

This program is for comparing the speed of naive version to optimized version. It builds 3D model of various size, computes projection and records the runtime. The output is a runtime curve. It imports functions from City.py and Projection.py.

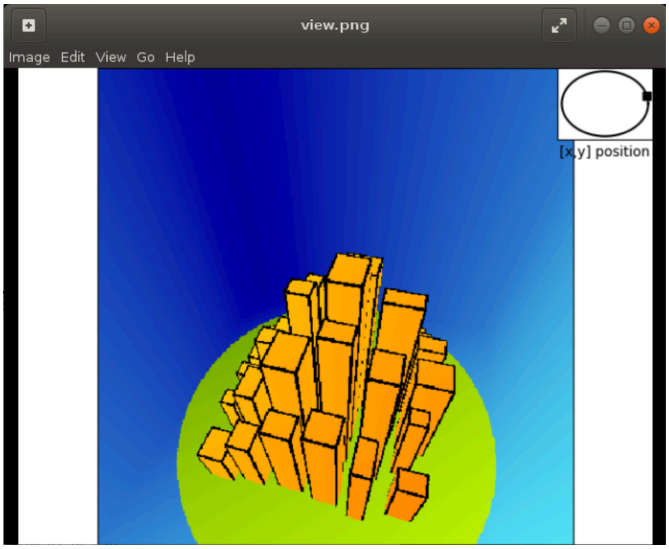


Result of 3D Modeling, Plot of City Model by Matplotlib

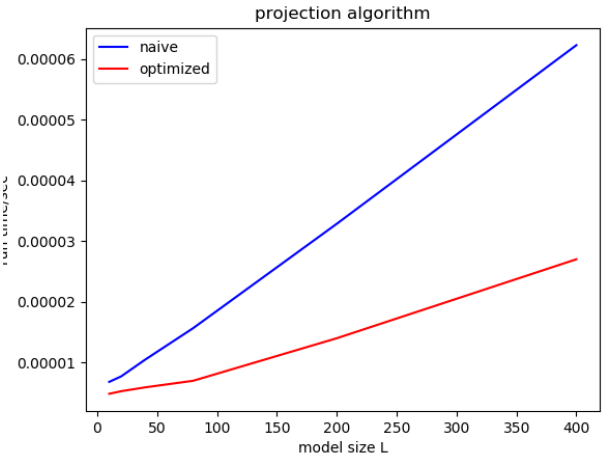
Left: a small simple model for code debug, Right: a model of a city, size 100^3

3. Results

I successfully implemented both naive and optimized versions and plotted out the projection image.



Projection image, Produced by Game.py
in 'colorful' mode



Runtime comparison, Produced by Runtime.py

As expected, the optimized version is faster than naive version. The time complexity of naive version is O(N), N is one dimension of 3D model. In the optimized version, theoretically time complexity can be O(sqrt(N)) if I set M=sqrt(N), as the number of threads in each block. However, in implementation, I set M=8 in case shared memory leaks. Therefore, the runtime is not O(sqrt(N)), but optimized version is still faster than the naive version.

It is amazing that when I have as much as 400*400*8 threads, the code still works!

4. Demonstration

Two demonstration videos are attached in Appendices. These are produced by Game.py

5. Discussion

To have a good looking picture, I also spend a lot of time on 3D modeling and setting color. I found gradient color effect makes the image more elegent, therefore I build a special version named 'colorful'.

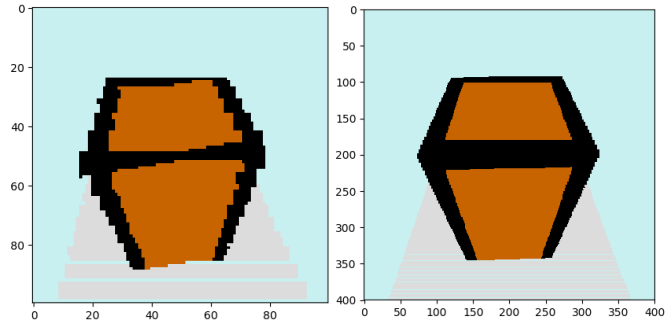
I also found that to have a good image resolution, the size of screen should be correspondence with the 3D model. For example if the screen size is 200^2, the model should be no less than 100^3.

In the real game, developers uses object model, like triangle mash, for modeling. That should be faster than my bitmap method, which could be optimized2 version. That is where I can improve (However it requires me to rewrite my code from the beginning).

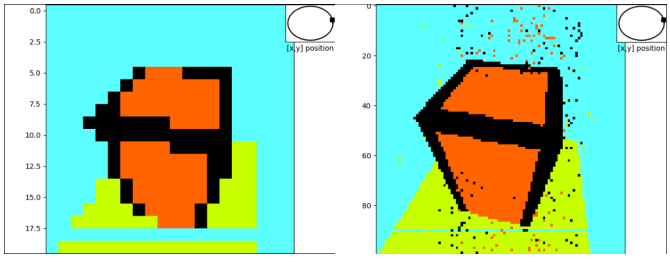
6. Conclusion

The result is simple and beautiful. It was a long and hard journey to reach it. In this project, I started from scratch, writing the code and testing step by step, adding one feature and another, until finally released.

In this project, everything went on well and the results met my expectation (except the UI! I would have more time if I had a partner). I started from 3D modeling, designed optimized algorithm and implimented both naive version and optimized version. It is really fun to build a game on your own.



Early versions of naive code. The size of 3D model and projection screen will affect the resolution.



Early versions of optimized code.
The 'bad points' are due to bugs.

It is a good chance to learn how to use parallel computing tool in real world. Also, I learnt a lot about modeling, software engineering, how to organize code when project grows large.

Improvement: This game will be more interesting if a UI is built. If I had time, I could also try to build object model and compare the speed of bitmap model vs object model.

7. Acknowledgements

Thanks two TAs in 4750: Tianyao Hua and Jeongmin Oh, for their suggestions about how to organize the code and demonstrate the result.

8. References

[1] Fundamentals of Computer Graphics, Peter Shirley

[2] Introduction to Game Development, Coursera

[3] Manual of PyOpenCL, NumPy, Matplotlib

9. Appendices

Source Code Under the directory /Source:

There is a README.md about how to run the code.

Two Demo Videos, Under the directory /Demo:

I recorded the screeenshot as .MOV videos.