GPU-Based Research of Highly Efficient Ray Tracing

He Jin, Fang Zhiyi, Ji Liang, Cai Ruicheng, and Chen Lin

College of Computer Science and Technology, Jilin University, Changchun, China
fangzy@jlu.edu.cn

Abstract

By further study of GPU architecture and GPU stream programming model. In this paper, uniform grid acceleration structure implements on the GPU stream programming model of the ray tracing. It has a lot of ray intersection calculations in the whole rendering process, reducing the efficiency of the whole scene rendering. Rendering without compromising the quality of the premise, put forward that use hardware accelerated Z-buffer to determine the main ray grating method to improve the efficiency of the main ray generation and its experimental verification.

Key words: GPU, Streaming Programming Model Ray Tracing, Rasterization

1. Introduction

Ray tracing is an image synthesis technology, which simulate the intersection of ray and surface to achieve the image rendering. Ray tracing is a rendering method which puts three-dimensional image into two-dimensional screen. Ray tracing is now widely used in computer games, TV, DVD production and film products. In ray tracing, each ray path formed by the multiple straight line, almost always includes scenes from the origin to the reflection, refraction, and shadow effects.

Scene in different roles may form a complicated situation increases the difficulty of panoramic three-dimensional rendering of the film. But in fact, Most of the time, many roles are still at rest (or relatively static state), then no necessary role in the implementation of a complete re-rendering. According to the role of these characteristics, Using distributed rendering cluster, which allows the CPU and GPU at the same time play to its advantages. The parallel processing power of GPU acts on the graphics rendering, while the main CPU for general computing capacity and scheduling of communication between nodes.

This paper studies the technology of GPU-based efficient ray tracing, with automatic load balancing panoramic three-dimensional film rendering software in order to solve the problem in rendering dynamic scenes in the panoramic three-dimensional film.

2. GPU architecture stream analysis

2.1. Programmable GPU rendering pipeline

As show in Figure 1, the graphics processor has two programmable pipelines. One is a Vertex Processor, which writes vertex shader to handle vertex buffer in the data. Another pipeline is Fragment Processor; it uses Fragment Shader to process the texture data in pixel buffer. It can also handle the output to pixel buffer with the technology of Render to Texture. The output of Vertex Shader can be used as input of Fragment Processor. Vertex texture provided by VS3.0, could transmit the data in Pixel Buffer to Vertex Buffer, so that the two programmable pipelines can work together.

Figure 1. Programmable GPU rendering pipeline
2.2. The memory architecture of GPU

The storage system of GPU is a branch of modern computer memory architecture. Similar to CPU, GPU has its own cache and register to accelerate the data access in calculation. Besides, the main memory of GPU has its own memory space, which means that before the program runs, the programmer must explicitly copy the data into GPU memory. This is a bottleneck in many applications, but the new PCI Express bus standard that could cause memory between the CPU and GPU sharing becomes feasible in the near future.

![Figure 2. The Memory architecture of CPU and GPU](image)

2.3. Stream type of GPU

Different from CPU memory, GPU memory has some restrictions on usage, and only through abstract graphical programming interface to access. Each of these abstract types can be thought of as different flow, and each flow has its own rules set of access. GPU programmers can see three kinds of stream type, which is vertex stream, frame buffer stream and texture stream. The forth stream is fragment stream, generated and fully consumed inside the GPU. Figure 3 shows a modern GPU pipeline, three users can access the stream, and in the pipeline where they can be used.

2.4. The Workflow of GPU

GPU mainly to complete the 3D graphics processing - the generation of graphics rendering. GPU graphic processing pipeline complete the work as follows [1][2]:

Vertex processing: GPU reads the vertex data which describe the appearance of 3D graphics at this stage, and builds 3D graphics using the shape and the relationship of position. These work completed by the Vertex Shader, if DX8 and DX9 specifications are supported in GPU.

![Figure 3. A modern pipeline of GPU](image)

Rasterization computing: The graphic on monitor consists of pixels. We need to convert the above points and lines on the graph to the corresponding pixel through some algorithms. The process which converts the vector graphics into a series of pixels is called rasterization. For example: A mathematical formula slashes segment, ultimately be converted into a ladder-like pixels.
Texture mapping: Polygons generated by the vertex units constitute only the outline of 3D objects, and texture mapping to complete the mapping of polygon surface. Popular to say, it is put the corresponding images on the surface in the polygon, to produce more realistic graphics. TMU (Texture mapping unit) That is used to complete the work.

Pixel processing: During the rasterizing of pixels, GPU completes the calculation and processing of pixels. Then determine the final properties of each pixel.

In general, GPU’s job is to generate 3D graphics, map the graphical to the corresponding pixel point, and calculate each pixel to determine the final color to output.

3. GPU’s stream programming model

As is known, the CPU’s programming uses a serial programming model, and for many high-powered application programs, the model is not suitable. Because of that if the model is used in these application program, concurrency and communication patterns are not supported. Somebody once put forward several methods that abstract the GPU to data stream processor. To realize the GPU stream programming model, Brook put forward a solution method, which is compiling and running [5]. Another method is to use Sh [6], which is a embedded C++ language library. GPU Meta Programming language also uses stream abstractions. In my thesis, the GPU storage model is analyzed, and deep analyses and researches about GPU [10, 11] stream programming model are taken. The stream model theory is stated in the thesis.

3.1. Stream model

Definition 1: A stream is a set of data. And all data belong to a same type. The data in a stream can only do the operation read-only or write-only.

Definition 2: An array of data is a set of data which belong to a same type. The data in array support random access.

Definition 3: Kernel is a small program which based on stream operation. Kernel gets a stream as input and generates a stream as output. Kernel can also access some static data arrays.

In the stream programming model, the application is built with several kernels in series. For example, In the stream programming model of graphics pipeline needs to write a vertex kernel program, triangle assembly the kernel, and cut the kernel, and then connect the output of a kernel to another kernel an input. Figure 4 shows how the graphics pipeline is mapped to the streaming model. This model defines the communication between the kernels.

3.2. Stream Model Implemented On GPU

(1) Stream: The floating-point textures, support render target

(2) Data Array: A Set of floating-point textures

(3) Kernel: The fragment shader

A GPU fragment program can read the location of any texture, but can not arbitrarily write to memory. The output of fragment program is an image in the global memory, and each segment calculation Correspond a separate image pixel.
Stream Generator

Shown in Figure 5, the view port which likes a quadrilateral is a stream generator. Pixels must be generated used to perform fragment shader program. It can be achieved by following commands on OpenGL.

- Initialize:
  - Set view port: `glViewport(0,0,w,h);`
  - Set texture model: `glMatrixMode (GL_PROJECTION);`
  - Load scene entity: `glLoadIdentity ( );`
  - Set 2D scene: `gluOrtho2D (-1,1,-1,1);`
  - Set Material model: `glMatrixMode (GL_MODELVIEW);`
  - Load view model: `glLoadIdentity ( );`

- Draw the view port:
  ```
  //Begin
  glBegin (GL_QUADS);
  //Draw
  glVertex3f(-1,-1,-0.5f);
  glVertex3f(1,-1,-0.5f);
  glVertex3f(1,1,-0.5f);
  glVertex3f(-1,1,-0.5f);
  glEnd ();
  ```

- View port can also be generated with only a triangle and adjust the view port so that only a quadrilateral to be drawn.

(5) General GPU stream processing program

- Draw Quadrilateral, 1:1 pixel mapping to the texture
- Run a SIMD program on Each fragment
- Generated texture buffer (output stream) as input at a stage of texture (input stream)

4. Highly efficient ray tracing based on GPU

We use a uniform grid structure and hardware accelerated Z-buffer rasterization technology to complete highly efficient ray tracing based on GPU. This study draws on the methods proposed by Purcell [7] , The whole scene pre-stored in memory in the form of textures. With the GPU rendering, the scene in memory is loaded into GPU texture memory. In this experiment, we have improved the generation method of main ray.
4.1. Accelerating Structure

In a uniform grid scene, it is evenly divided into voxels, the voxel contains the triangular facets and triangular patches index [3][4][8].

A simple way to create a uniform grid:

All the triangles (Ti) in scene:
1. Calculate Boundary points (b1, b2) of Ti.
2. Triangle bounding intersection test : Each boundary Cj (b1, b2) of Ti.
3. If the return value is true, add a reference value of Ti to Cj.

4.2. Elimination of Ray Propagation in the Loop

Using the method of eliminating the loop in ray propagation to calculate the color value in the iterations: First, calculate the color of one ray, then two rays, and more. Finally, Proof the results using the complete induction.

\[ c(n) = \sum_{j=1}^{n} M_j \left( \prod_{i=1}^{j-1} r_i \right) (1 - r_j) \]

( Ri: The reflection coefficient of material i; Mi: The color of material i; n: The maximal Iterative coefficient )

Do the calculation using Equation 1 has less memory consumption and higher computing speed, and easy to implement.

4.3. The Storage of 3D rendered scene

Here we use Two-dimensional texture to store scene. Element Indexes and the 32-bit floating point texture grid unit are used to indicate the components, element data is a 32-bit floating point texture with 4 components. Element data must be aligned and the triangle in a 2D texture must in the same row.

For large scenes, it may be necessary to split the scene stored in a different uniform texture.
Theoretically, in a scene will be possible to solve more than 1,600 million triangles. However, current graphics hardware is limited to 256 MB of memory, so the maximum number of triangles will be 100 million or so.

4.4. Improvement of the Main Ray Generated

In traditional, Ray tracing must have eye line, For example, line of sight from the eye cast to the scene through each pixel of the image. For each ray, we need to determine whether it was the first time that it is blocked by the objects in the scene. Then calculate the Intersection vector \((t, u, v)\) \((t\) is the distance), and \((u, v)\) is the coordinate in triangle. In the experiment, using Z-buffer technology and hardware rasterization to determine the first collision ray between the eye sight and the scene.

The traditional method of generating the main ray needs to test all of the bounding boxes for each pixel, including a large number of empty pixels which wasn’t covered by any bounding boxes. Therefore, in the improved method, the rasterization of each bounding box will be more efficient (A main ray is located by the viewpoint and the coordinate of pixel).

First of all, the whole scene is rendered into the frame buffer. Three right-angle vertexes \((1,0,0)\), \((1,0,0)\), \((0,0,1)\) are stored as the RGB color value of the triangle. Record the ID value of this triangle in texture coordinate system. After rasterizating the grid bounding box, we can get a vector \((w, u, v)\) by interpolating each pixel of the original triangle. (Figure 8). The vertex \((w,u,v)\) corresponds to the value of index ID. Intersection \(x\) can be calculated by the following equation [9]:

\[
x = w \cdot v_0 + u \cdot v_1 + v \cdot v_2
\]

\[(2)\]

\((v0, v1\) and \(v2\) are the three vertices of the triangle)

After locating the intersection of the main light, we can trace the secondary ray on GPU. The tracing has three steps: generate secondary ray, traversal the accelerate structure and Shade. It is necessary to create secondary ray, In order to produce global illumination effects. Secondary light can be easily obtained after obtaining the intersection, normal, eye position and light position.

5. Experimental results and analysis

This system supports Direct3D (HLSL) and OpenGL (GLSL) to achieve ray tracing. Specific application model shows as Figure 10.

In this experiment, we use Direct3D Ray Trace software which is written in HLSL, GLSL is also supported. And the parameters of our computer in hardware are as follows, VGA Card: NVIDIA Quadro FX 570, CPU:Intel(R)Pentium(R) Dual, 3.00GHz , RMA:1.00GB.

In order to increase the contrast, we chose two scenes to compare GPU rendering. After loading, as show in Figure 11, (a) is the rendering of scene 1, and (b) is the rendering of scene 2. In figure 12, (a) is the uniform grid of scene 1, and (b) refers to scene 2.
The scene complexity, different grid sizes and different iterations of the rendering time will affect the comparison of rendering time of scene. The following Table 1 shows the rendering time with only different number of grid sizes in different scenes. Table 2 shows the rendering time in different number of iterations.

**Table 1. Rendering time with different number of grids**

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>Rendering scene 1 Improved Pre and Post(sec)</th>
<th>Rendering scene 2 Improved Pre and Post(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x2x2</td>
<td>2.00s/1.46s</td>
<td>1.00s/0.73s</td>
</tr>
<tr>
<td>10x10x10</td>
<td>1.50s/0.89s</td>
<td>0.60s/0.34s</td>
</tr>
<tr>
<td>30x30x30</td>
<td>1.00s/0.46s</td>
<td>0.20s/0.1s</td>
</tr>
</tbody>
</table>

**Table 2. Rendering time with different number of iterations**

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Rendering scene 1 Improved Pre and Post(sec)</th>
<th>Rendering scene 2 Improved Pre and Post(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.80s/0.48s</td>
<td>0.60s/0.42s</td>
</tr>
<tr>
<td>4</td>
<td>2.00s/0.98s</td>
<td>1.60s/0.89s</td>
</tr>
<tr>
<td>10</td>
<td>3.00s/1.67s</td>
<td>2.50s/1.13s</td>
</tr>
</tbody>
</table>
To illustrate the feasibility of GPU accelerated ray tracing, we renders the scene with CPU and GPU. The rendering is shown in figure 14.

Two graphs is not very different from the visual, but 2.00 seconds is used in CPU, but 0.85 seconds in GPU. The latter shows more efficient. The difference of the rendering time between CPU and GPU is not large. Because of that the selected scene is mainly too simple. The ability of parallel computing on GPU is not reflected in calculating the intersections of rays. Its advantage will be reflected in complex scenes.

6. Conclusions

In this paper, we analyze the architecture of GPU, the stream programming model on GPU; understand the principle of stream programming model and how to build a streaming processor.

We archive the stream programming model in C++. Implemented ray tracing based on GPU, using uniform acceleration structure. After analyzing the part of generating main ray, we proposed the method that use Hardware Z-buffering and accelerated rasterization to improve the efficiency of generating the main rays. Finally, we compared the experimental results, and the effect is very satisfactory.

7. References