Golden Monkey

A Realistic Renderer on the Cell Processor

Si Yin

Faculty of Science

Vrije Universiteit Amsterdam
1544047

A thesis submitted for the degree of

Master of Science (M.Sc.) in Computer Science

2008

First Reader: ***Thilo Kielmann***

Second Reader: ***Kees van Reeuwijk***

Third Reader: ***Maik Nijhuis***

To my passion for ray tracing.

Acknowledgements

I have been passionate for graphics rendering since the start of my master education. Eventually after two and half years, I finished my degree with a renderer from my hand. I would like to thank my supervisor Thilo Kielmann for his organizing and guidance for my work. Without his supervising, I couldn't finish my thesis in such great quality. I also would like to thank Maik Nijhuis for his persistent patience on correcting my writing and also his great work of the Gordon Library. Finally I would like to thank Kees van Reeuwijk for his valuable advice on the ray tracing techniques. I appreciate his remarks on my thesis as well.

Things in life may not go straight, but the support from the following people always push me ahead. I would like to thank my parents for their consistent concern about my studying progress. I also would like to thank my girlfriend Fujin and my friend Liu, Wei for their help in the past time. My former mentor Jasper and Barry in IBM give a lot of concern about my thesis as well. In the end, I want to bless the people suffering in the winter disaster in China in 2008. I hope they are getting much better now.

Abstract

Ray tracing is one of the image synthesis techniques in the computer graphics world. It is famous for its realistic image quality and also for the overwhelmed cost in time and computing resources. With the development of modern computers, there is potential value in both economy and academic research to perform the ray tracing in reasonably short time. In this thesis we present a ray tracing implementation on the cutting-edge processor, Cell Broadband Engine. Unlike the standard ray tracing algorithm that uses recursive function calls, our ray tracing algorithm uses a master/worker model, which is capable of tracing multiple rays in parallel. In addition, we make use of the SIMD instruction set to speedup the tracing. Instead of tracing one ray, we are tracing four rays at a time. We also abstract the shading algorithm to apply to the Map/Reduce working model. By using the map function on the SPE and reduce function on the PPE, we well balance the workload and reduce the communication cost among workers. At the end of this thesis, we evaluate and analyze our ray tracing program compared to other ray tracing applications. The result shows that our application performs better in speed than the Intel based ray tracing application but also lacks enough features and needs to be improved in the future.

Contents

List of Figures

Chapter 1

Introduction

The beauty of the universe always fascinates humans. We can enjoy climbing the mountains, jogging in the park, and drinking beers under the fancy light beams of favorite clubs. It is not long before all those things can be shown in the virtual world and today even the most experienced photographer cannot distinguish computer generated images and the real shots in a Hollywood movie.

The process of generating virtual images is called rendering in terms of computer graphics. The whole rendering algorithm is complicated and can be divided into physical based and non-physical based categories. Physical-based rendering, also named unbiased rendering, generates images based on simulation of nature and physics. Conversely, non-physical based rendering is biased on special purpose but not the nature physics. There are some reasons for biased rendering to exist. For example, some users like artists want to make images opposed to the nature (e.g., the special effects in a fiction film). But one of the most important reasons is our hardware limitation. Modern computers are not powerful enough to follow nature easily. To find a balance between speed and realism, developers prefer to use all kinds of hacks to cheat the human eyes and improve the speed of the program. In the past decades, rasterization rendering, which is in concept a hack to the eye, dominate the entertainment industry like animation, electronic games, and education. Physical-based rendering is hardly used because of the high cost in both rendering time and hardware expenses.

The computing world today is becoming more and more diverse and unpredictable. The

calculation speed of commodity computers is almost comparable to supercomputers made several years ago. In 2006 Christmas, STI, a partnership between Sony, Toshiba and IBM, launched a 9 core micro-processor, with code name Cell in Sony's game console PlayStation 3. The Cell processor is quite different from its competitors because of the heterogeneous architecture, which results in an over 230 Giga Flops monster. Such a cutting-edge processor inspires us to think whether it can balance the complexity between physical based features and acceptable rendering time.

1.1 The Ray Tracing Problem

Physical-based rendering usually boils down to a ray tracing algorithm, which imitates the human eye system by tracing the rays from the view point to the light source. Figure 1.1 illustrates the concept of ray tracing. The algorithm follows the path of

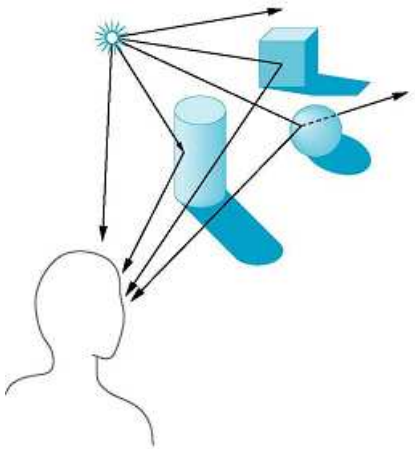


Figure 1.1: The Ray tracing illustration.

the rays, and builds up the whole image by the illumination color of the first hitting objects in the scene. The advantage of the ray tracing algorithm is the ease of getting shadows and reflections. When the program shoots the ray and finds the hitting object, the algorithm can, based on the material of the object, detect the shadow by shadow rays and generate the reflection rays (also called secondary rays). This procedure will keep going until the ray diminishes in space or hits the light source. A good ray tracer is hard to construct, but the essential implementation is quite elegant and simple as shown in Listing 1.1.

Listing 1.1: Pseudocode for a standard ray tracer

```
render:
  for each pixel:
    sample one position inside of the pixel
    generate the ray by transformation
    pixel color = tracing( ray )

tracing( ray ):
  find the nearest intersection point in the scene
  color = shade( point )
  return color

shade( point ):
  get the normal at the point
  color = 0
  for each light source:
    trace shadow ray to the light source
    if shadow ray intersects the light source without any obstacle
    color = color + direct illumination( point )
  color = color + tracing( reflection ray/specular ray )
  return color
```

The main disadvantage of ray tracing is the low performance. When the number of objects increases, the complexity of searching the nearest intersection point becomes exponentially complicated as each ray potentially generates new rays when doing the reflection. This thesis discusses a parallel approach to make ray tracing fast enough even for complex scenes.

1.2 Introduction to the Cell Processor

The Cell processor is a microprocessor developed by a joint team from Sony, Toshiba and IBM, known as 'STI'. The Cell is the shorthand for Cell Broadband Engine Architecture and it combines a general-purpose Power Architecture core with 8 streamlined coprocessors called Synergistic Processing Element (SPE). The first commodity product of Cell is Sony's game console PlayStation 3 launched in 2006, and since then lots of researchers have been impressed by the performance of Cell. On September 16, 2007, a distributed computing project called Folding@Home, has been recognized by the Guinness World Records organization as the first petaflops distributed network

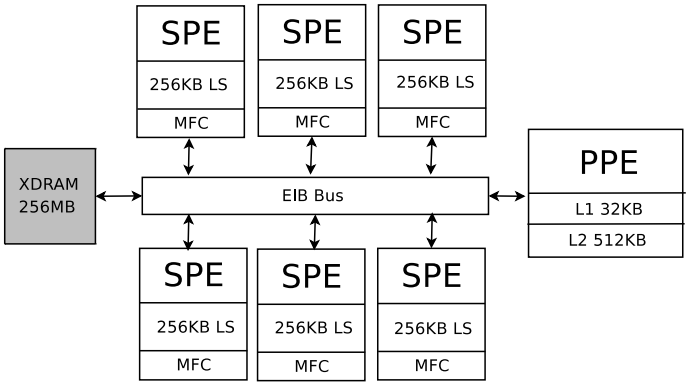


Figure 1.2: The Architecture of Cell Processor

in the world[10]. According to web-site's report, over 70% number of CPUs used in Folding@home are contributed by the Cell processor in PlayStation3 console.

1.2.1 The Architecture of Cell Processor

A brief overview of the architecture of Cell processor is shown in Figure 1.2. As mentioned before, the Cell processor has 8 coprocessors (2 cannot be used in PlayStation3) called SPE and one PowerPC processor called PPE. The 6 cores are connected by the central high speed bus Element Interconnect Bus (EIB). The PPE is a general purpose 64-bit PowerPC processor and the SPE is a 32-bit RISC processor specialized for compute intensive tasks like video decoding and graphics rendering. We can see from Figure 1.2 that each SPE has a separate 256KB Local Store for saving data and code, which requires developers to really take care of. The SPE communicates with the main memory via the Memory Flow Controller(MFC), which in principle sends and receives data by DMA commands. The peak computing capability of Cell processor is over 230 GFlops, although the average computing capability varies a lot based on implementations.

1.2.2 The Programming Model

Because of the heterogeneous architecture of the Cell Processor, lots of programming models have been introduced[4]. Our application 'Golden Monkey Renderer' uses the

master/worker model, which takes the 6 SPEs as individual workers and the PPE as the master. The master keeps dispatching new rays as jobs to each worker and integrating the result when any task is accomplished. One reason to use such a model is because of the independence of each ray. Rays from each pixel are independent, which allows us to easily split the whole rendering task into a sequence of ray jobs. The function of each job is to compute the shading value of each ray. One big difference between the typical master/worker model and our implementation is that a job may generate new jobs via reflections. One solution to deal with it is that the master extracts the reflection ray from the result of the worker, treats them indifferently as normal rays and puts them in the ray queue. Another solution would be that the workers process their own reflection rays within their work and send back the result as a whole. We have adapted both solutions in our implementation for research purposes and analyze the pros and cons in the chapter 4.

1.2.3 SIMD Programming

SIMD programming stands for Single Instruction Multiple Data programming, which sometimes is also called vector programming. A vector is an instruction operand containing multiple data elements as one single operand. The type of elements can be 32-bit integers or floating point values based on the length of the vector register. A vector instruction, the same as a scalar instruction, manipulates vectors within the same clock time. Both the PPE and the SPE in the Cell processor are 128-bit wide

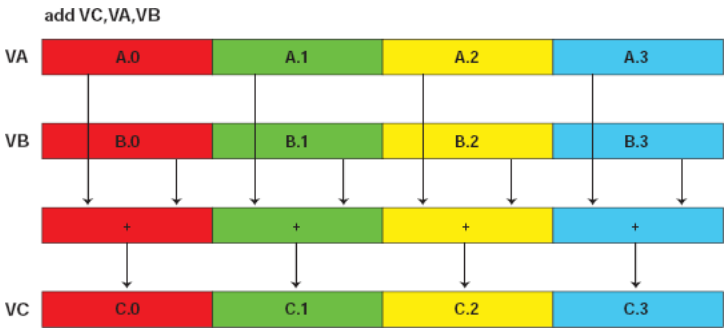


Figure 1.3: A vector *add* operation on two vectors *va* and *vb*. Both vectors are 128-bit wide and contain four 32-bit elements inside. The result vector *vc* is the sum result of each element from *va* and *vb*.

vector processors. Developers can do the vector programming by using the VMX and SPE instruction set from the SDK[4].

A basic example of a vector *add* operation is shown in Figure 1.3. In the figure, vector *va* is added to vector *vb* and results in vector *vc*. As both the PPE and the SPE use 128-bit vector registers, they can compute four 32-bit integers or single precision floating point values at the same time. The result vector *vc* contains the adding result of the 4 elements in both vector *va* and *vb*. As seen from the example, SIMD programming exploits data-level parallelism, which requires the programmer to transpose the data structure from Array of Structure (AOS) to Structure of Array (SOA)[4]. Figure 1.4 presents both kinds of structures and the transportation to SOA is called vectorization or SIMDization. Normally the transportation is automatically done by compilers, but it can also be specified manually by developers. The Golden Monkey Renderer uses manual vectorization optimizations in order to gain the best performance.

Listing 1.2: AOS

```
struct {
  float a;
  float b;
  float c;
} A;

A v[LENGTH];
```

Listing 1.3: SOA

```
struct {
  float a[LENGTH];
  float b[LENGTH];
  float c[LENGTH];
} A;

A v;
```

Figure 1.4: Comparison between *Array of Structure (AOS)* and *Structure of Array (SOA)*. It is recommended to transpose AOS data to SOA to do vector programming.

1.3 Goal and overview of this thesis

In this thesis, the author explores a full-fledged ray tracing renderer on the Cell processor and evaluates the capability and performance with the popular ray tracing application POV-RAY. Our main goal for this project is to design a next generation ray tracer using multi-core technology. Our contributions in this paper are mainly as follows:

- A master/worker working model ray tracer is implemented on the Cell processor

to achieve parallel ray tracing. The PPE in the Cell processor plays the master that integrates the work results from workers while the others play the worker role to do the intensive tracing work.

- A map/reduce model is applied to the ray tracing application to optimize the work balance for each core. The working model also enhances the application's scalability and flexibility.
- A detailed evaluation of the productivity and performance is compared with the POV-RAY ray tracer. The application achieves around 1.2x speedup in the final performance benchmark compared to the POV-RAY ray tracer, which even runs on a more expensive Intel PC.

We conclude that the Cell processor has the potential to solve the ray tracing problem. The difficulties lie in having efficient data communication and smart work balance management between multiple cores.

The remainder of this thesis is organized as follows. In Chapter 2, we present design details like the pipeline design and the format of our job packages. Later on we introduce the XML scene description files. Chapter 3 is about the implementation details of the Golden Monkey Renderer. The Golden Monkey Renderer is composed of several components and each component has different implementations for different algorithms. We also introduce our component design, which makes it easy for developers to expand new features. Chapter 4 presents our evaluation results of our renderer. The evaluation mainly contains two comparisons. The first one is a feature comparison with the popular ray tracing application POV-RAY. The second evaluation is about performance comparison, which measures the runtime speed for our implementations. In chapter 4, we also discuss the bottleneck and implementation pitfalls and propose our possible improvements in the future. At the end, we wrap up the whole thesis in chapter 5.

Chapter 2

Design

The mission of a renderer is to build up virtual images from a scene description. The rendering process is usually not a single algorithm but consists of several stages or processes. For example, a modern graphics pipeline contains more than 8 stages. In each stage there may be more than one algorithm to choose from for different purposes. A good renderer has the capability to adjust to different qualities and also has an open interface for other design and modeling software. The Golden Monkey Renderer aims to achieve such a high production level and uses the standard XML file format as the scene description for the ease of the configuration and readability.

As a research project on the Cell platform, the Golden Monkey Renderer has a well organized class hierarchy to adapt different algorithms at different stages. One of the purposes for the project is to construct a flexible framework so that researchers can try their new goodies on the fly.

2.1 The Pipeline Architecture

In chapter 1, we mentioned that the Golden Monkey Renderer uses a master/worker architecture as the core. The PPE acts as the master, which keeps sending jobs and integrating work results accomplished by the workers. The SPEs act as the workers, which receive tasks and do the intensive computing work. When the master is working

with multiple workers, asynchronous communication is required to save the time slices on waiting jobs. We illustrate the concept in Figure 2.1. When the master sends out a job to one worker, it will not wait until the job is finished but will fetch the next job and send it to another available worker. The only moment of waiting is when all pending jobs are sent to the workers, and the master needs a result from a worker to continue. When this model comes to practical design, a straightforward solution is using multi-threading. Each worker and master runs in a separate thread so that they can manage their work and communications on their own. As the whole rendering process is not only about tracing rays but also includes sampling and shading, we decide to use a pipeline design to tap as much computing resources as possible and minimize the waiting time.

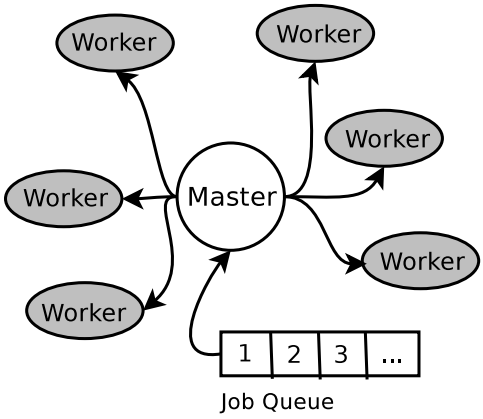


Figure 2.1: The master/worker working mode. The master keeps fetching new jobs and sending to available workers. It will get into waiting status if no worker is free.

The rendering pipeline is illustrated in Figure 2.2. It is composed of 5 components and 4 threads. The first two components Sampling and Camera are located in the main thread while the other 3 threads are the Accelerator, Jobber, and Integrator. The spheres in the figure are the global variables shared by all the threads and the arrows show the operation relations in the work flow.

The whole pipeline starts from sampling points on the screen in the main thread. The Camera component transforms the sampling points into rays in the world view and pushes the data into the ray queue. The Accelerator thread keeps fetching rays from the queue and package them with scene data as jobs and puts them in another queue

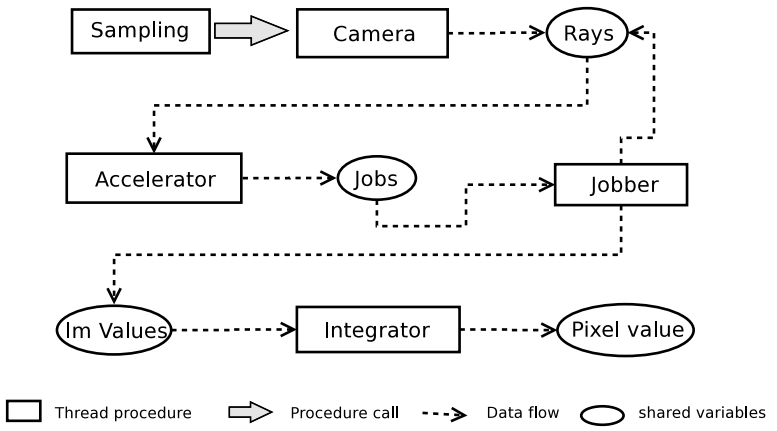


Figure 2.2: The rendering pipeline.

for the Jobber. The Jobber thread plays the master in Figure 2.1 and sends jobs to the SPE workers. The Integrator thread will process the job result in the *Im value* queue and do the final shading for each ray. The final result of the Integrator is the pixel value of the image. In the remainder of this section, we start to introduce the details of each component in the pipeline.

Sampling: Sampling in ray tracing is to choose the points from the eye in order to generate rays. One common approach of sampling is to choose one point per pixel, which may cause aliasing since one pixel may have multiple color values. For example, a pixel on the display is colored as red but in the real picture, the point of the pixel may contains more colors other than red. It is also called *jagging* for the edges inside one pixel. Our sampling approach uses stratified sampling, which gives one random sampling point within one pixel to reduce the aliasing. More information about sampling can be found in [8].

Camera: The purpose of Camera component is to generate rays based on the camera view and the sampling points. There are two kinds of camera in the Golden Monkey Renderer. One is called Orthogonal Camera, which generates parallel rays with the same direction. The other one is called Perspective Camera, which shots rays from the projection center to the view plane. Perspective Camera, which works similarly to the human eyes, is able to present the depth of each object in the image.

Accelerator: Accelerator is a data structure presenting the scene in order to accelerate

the intersection test of each ray. When there are millions of rays and enormous objects to render, it would be a big cost to test each ray against each object. Data structures like KD-Tree, Uniform Grid, and Bounding Volume Hierarchy (BVH) are designed to avoid redundant tests based on the known space relations[14]. Another task of the accelerator in our design is to package rays and scene information to jobs. Each job is self-described with all the information needed and a global queue job is allocated for storing and fetching.

Jobber: The Jobber component plays the exact role of master described in Figure 2.1. It keeps fetching jobs from the job queue and sending them to the workers. Asynchronous communication and processing the work results are the main tasks of Jobber. Communications between master and worker on the Cell platform are done by DMA commands and we use a handy tool called Gordon to do that. Additionally, our design of the Jobber component is not Cell specific but has an abstract interface to make it easy to expand to other platforms. For example, the Jobber component can be ported to a normal PC by implementing the worker as thread.

In Figure 2.2, there is a flow from the Jobber to the ray queue. This flow contains the reflection rays mentioned in chapter 1.2. The master extracts the information of reflection rays and processes them the same as the primary rays. However, in our optimization we let the worker handle the reflections within its own thread.

Integrator: The Integrator component computes the shading values of each pixel from the work results. The shading algorithm we are using is the Whitted improved illumination model[13]. Since we unroll the recursive tracing by sequential jobs, our reflection shading is quite limited. Higher reflection level will distort the image quality by inaccurate colors. More information about reflection can be found in the next chapter. The result of the Integrator component are the pixel values of the image. At the end of the pipeline, we use the OpenEXR library to write the pixel array into an image file.

2.2 The Gordon Library

The Gordon Library, currently being developed at VU University Amsterdam, is a small handy library for data communications between the SPEs and the PPE on the Cell Processor. It constructs an abstract framework, which uses the master/worker model, to easily send and receive jobs. With the Gordon library, application developers can avoid implementing nasty DMA communications and speed up their product development. The principle of the Gordon library is to package all the required data into a data structure and transfer it to SPEs by statically linking to the function object file. Listing 2.1 gives the definition of the job structure of a Gordon job.

Listing 2.1: The data structure of the worker job

```
typedef struct {
  int index;
  int nin;
  int nout;
  void* buffers[MAX_BUFFER_NUM];
  int ss[MAX_BUFFER_NUM];
} gordon_job;
```

The actual implementation has more fields but here we only list the most used ones. The index field is the index number of the function to be called on the SPE. Functions are compiled into pre-compiled object files. nin and nout specify the number of the input and output buffers. Supposing you are calling a sum function with two integers, then the value of nin is 2 and nout is 1. The buffers field stores the actual data and the size of each buffer is in the ss array. After filling in the structure, the job is ready to be sent. The sending function is the same as in Listing 2.2.

Listing 2.2: The sending function of Gordon library

```
void gordon_pushjob(gordon_job_t *job,
                    void (*callback)(void *),
                    void *callarg);
```

The callback is a pointer to the callback function which is called once the job is finished. The callarg argument is the input argument for the callback function. In the Golden Monkey Renderer, the callback function puts the job results into the global *Im value* buffer for the Integrator component. There are other features in the Gordon library

but they are beyond the scope of this thesis.

2.3 Job Definition

In the previous sections, we introduced the pipeline design and the job structure used in the Golden Monkey Renderer. In this section we will present how we use this framework and what kind of data is assigned to the structure fields. As illustrated in Figure 1.1, the principle of ray tracing is tracing rays from the viewpoint and finding the nearest intersections for each ray in the scene. Specifically, two basic data structures are required to do the work. One is the ray data and the other is the scene description. The rays are an array of floating point values, which represent the location and direction of each ray. The scene information is based on the geometry features of the object, e.g., a sphere surface requires the radius and the location of the central point. Other data structures are required on demand of features. For example, light positions are required by shadow rays and shading. Considering the limited memory on each worker SPE, our

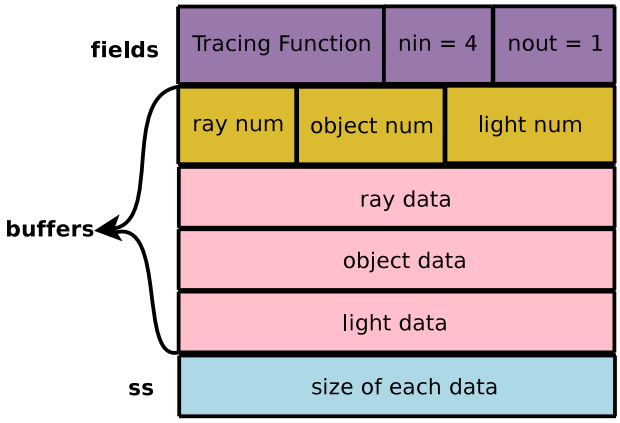


Figure 2.3: The job structure used in the Golden Monkey Renderer

job structure omits the material description and lets the PPE do the shading. Figure 2.3 illustrates the fields needed in our Golden Monkey Renderer. The information of the ray, object, and light is stored in a flat array. Workers first extract the number of each object from the fields array in Figure 2.3 and then reconstruct each object from the serialized flat buffer. The exact implementation tricks of marshaling and deserialization are introduced in the next chapter. As a demonstration of packaging job structures, we

list the source code of a simple job packaging in Listing 2.3. In the listing the pointer

Listing 2.3: Serialization of a simple job

```
1  void SimpleJob::serialize(gordon_job_t* p) {
2      p -> index  = jobType;
3      p -> nin    = 4;
4      p -> ninout = 0;
5      p -> nout   = 1;
6      p -> next   = 0;
7      p -> nalloc = 0;
8
9      p -> buffers[0] = (uint64_t)scene;
10     p -> buffers[1] = (uint64_t)rays;
11     p -> buffers[2] = (uint64_t)shapes;
12     p -> buffers[3] = (uint64_t)lights;
13     p -> buffers[4] = (uint64_t)out;
14
15     int nobj = scene[1];
16     int nray = scene[2];
17     int nlgt = scene[3];
18
19     p -> ss[0] = _round16(4 * sizeof(uint64_t));
20     p -> ss[1] = _round16(nray * sizeof(spu_ray_t));
21     p -> ss[2] = _round16(nobj * sizeof(spu_shape_t));
22     p -> ss[3] = _round16(nlgt * sizeof(spu_light_t));
23 }
```

p is the job structure to be filled in. Variables scene, rays, shapes, lights and out are the actual data arrays with the respective data. At the end of the function, the ss array is filled with corresponding memory size of each buffer. The _round16 is a macro function to round the byte number into a multiple of 16 for the address alignment of the DMA command[4].

2.4 Rendering Equation

The rendering equation is the formula to calculate the shading value of each ray. A realistic rendering equation usually takes long time in rendering. For instance, global illumination algorithms like Monte Carlo integration and Photon Mapping cost lots of memory and time on the random sampling points, which are used to estimate the

intensity of illumination[8]. The roadmap of the Golden Monkey Renderer project is first to show reasonable image quality on the Cell platform and after the concept is proven, the algorithm will be refined to target highly realistic pictures. Upon the time of writing, we use the traditional Whitted style illumination model[13] as the rendering equation.

The original paper of the Whitted style model is published by Turner Whitted in 1988 and it greatly improves the specular shading effects from the older Phong model[11]. Though the model is about 20 years old, most rendering engines today are still using it(e.g. the OpenGL specification uses it as a quick and dirty way of local illumination). The rendering equation of Whitted-Style shading is shown in the following formula.

$$I = I_a + k_d(N \bullet L_j) + k_s S + k_r R \tag{2.1}$$

where

I_a = the ambient illumination

S = the intensity of the light incident from the R direction

N = the normal for the intersection point

L_j = the incident ray from the jth light source

k_d = the diffuse coefficient

k_s = the specular coefficient

k_r = the transmission coefficient

R = the intensity of light from the reflected direction

In the equation the I_a, k_d, k_s, and k_r are constant coefficients specified in the material. The normal N and vector L_j are all normalized vectors and their dot product indicates the proportion of the light flux coming from the incident ray. We assume that all the objects have perfect specular material so that reflection ray is identical to the incident ray and vice verse. More implementation information on each shading coefficient is introduced in chapter 3.

2.5 The XML Scene Description

In a rendering application, the scene file is used to describe the 3D world in a certain format. A basic scene file should contain the description of each object, light, and also the camera. The BSP format, for example, is a well-known scene format which describes the 3D world in a Binary Space Partition tree. In modern rendering applications, the scene file is also coupled with modeling tools, which generate optimized and pre-processed file for better rendering speed. Since one of the goals in the Golden Monkey Renderer is to design a framework for experiments on different algorithms, the scene description requires high configurability and readability for both users and computers. Our implementation uses the XML(Extensible Markup Language) format as our scene description language. The XML format is one of the most widely used and recognizable markup languages in the world. We use the existing open source libraries (libXML) to facilitate parsing the format, which makes our developing process more efficient.

Listing 2.4: The skeleton of the scene format

```
<scene attributes = 'value'>
  <object>
  ...
  </object>
  <camera>
  ...
  </camera>
  <light>
  ...
  <light>
</scene>
```

The skeleton of the description file is illustrated in Listing 2.4. Our scene description starts from a <scene> tag which tells the renderer to start rendering a new scene. Several attributes are available to configure the scene like the number of SPE used, the number of the sampling rays per pixel, and the camera type. Within the scene tag, the user can set up their objects, lighting sources, and the camera views. We list the scene file of a simple scene in Listing B.2 as an example. Details of tags and available attributes are listed in Appendix B.

2.6 Developing Platform

The Golden Money Renderer is developed on Sony's PlayStation 3 game console. The console is equipped with a 3.2GHz Cell Processor and 256MB XDR main memory. It is officially supported by Sony to run a Linux kernel on the console. The program running on the PPE is coded in C++ whereas the SPE program is coded in C. Table 2.1 shows the details of our development environment and libraries we are using.

Developing Platform	
Hardware Platform	Playstation 3 game console with 3.2GHz Cell Processor and 256MB Main Memory
OS	Linux Kernel 2.6.23 ppc64
Compiler	GCC 4.1.2 and spu-elf-gcc 4.3.0
Libraries	Cell SDK 2.1 openEXR 1.4.0 libxml2 2.6.27 gordon

Table 2.1: The developing platform of the Golden Monkey Renderer

2.7 Summary

In this chapter, we present the key issues in the design of the Golden Monkey Renderer. The Golden Monkey Renderer uses a pipeline architecture as the backbone of the whole rendering process. By the structure of the heterogeneous architecture, it is straightforward to use the master/worker programming model for the development on the Cell processor. Since each of the worker SPE only has limited local memory, the job it tackles needs to be as compact as possible. Our approach is nothing dramatic but packaging the required raw data like rays, objects, and lights by a serialization method. After that we discuss the Whitted style rendering equation we choose to use . For performance and time reasons, we give some assumptions and changes to solve the equation (e.g., we avoid the ambient illumination and assume that all the objects in the world are perfectly specular reflected). Finally we describe our XML file format as

the scene file format. The XML file format is very easy to write and modify, both by human beings and computers. The developing platform is built upon a normal Linux on the PlayStation3 and we also involve several open source libraries to speed up the development.

Chapter 3

Implementation

The Golden Monkey Renderer aims to achieve two goals. The first goal is to have certain rendering performance that results in a usable application. The second goal is to build a configurable framework for investigating new algorithms. We take serious consideration on both goals and decided to use C++ as the main developing language. One reason to use C++ is its object-oriented features, which can easily manage the second goal by the concept of polymorphism. Another reason to use C++ is the compatibility to C, which is the only supported high level programming language by the SDK. The structure of the source code follows the pipeline architecture that each component has separate derived classes for different implementations(e.g., the Camera class is just an abstract class for the Camera component and the Perspective class inheriting it do the actual work). In this chapter, we will first review the whole source code, then present the work flow of the program and after that we will introduce each component in detail. In the end, we introduce our significant optimizations used in the implementation.

3.1 Overview of the source code

It is a convention in the Golden Monkey Renderer that different components are located in different directories. The Golden Monkey Renderer uses static linking to glue compiled components together and save redundant compiling time. We give the source tree of the Golden Monkey Renderer in Figure 3.1. On top of the tree is the *main.cpp*

file, which starts the whole program. Following the root, there are directories storing the code for respective components. The *camera*, *accelerator*, *jobber*, *integrator* and

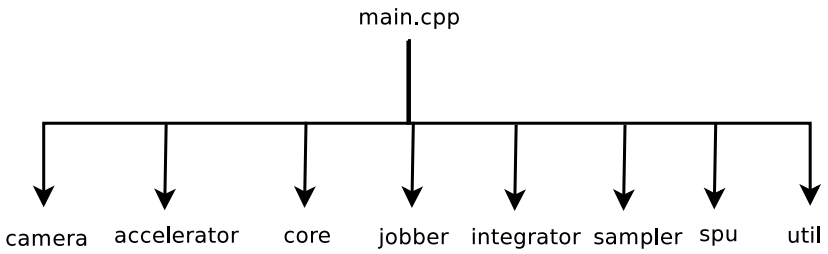


Figure 3.1: The source code tree of Golden Monkey Renderer

sampler directories are the components mentioned in the pipeline. The *core* directory stores all kinds of definitions of primitives used in the renderer such as the definition of sphere and the definition of ray. The *spu* directory contains the source code running on the SPE and the *util* directory contains tools like memory management and global variables.

Figure 3.2 gives the work flow of a component to illustrate how it works during execution. Each threaded component in the pipeline is controlled by a conditional variable.

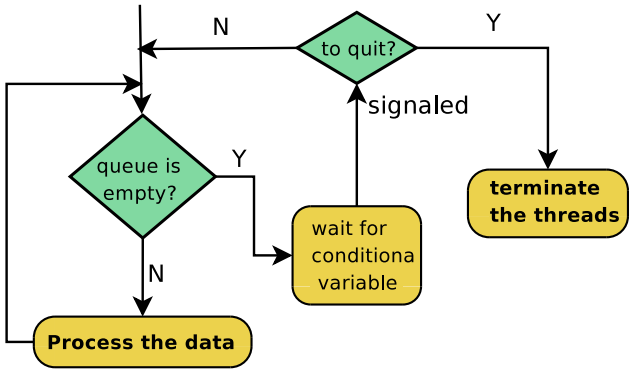


Figure 3.2: The work flow of a component

When the waiting queues are empty, the thread stalls and waits for a signal. The main thread will broadcast the signal when the queue is filled again. We observed during our development that using conditional variable is more efficient than an infinite loop and also avoids race conditions when multiple threads access the same data queue. Besides the conditional variable for each component, there is a global conditional variable

called *cond_stop* for controlling the whole pipeline. When all the components are in the waiting status, *cond_stop* will be signaled and the main function will decide whether the application is ready to exit.

Listing 3.1 illustrates the start point of the source code. At the beginning of the application, the main function first initializes a *GMRender* object, which encloses the components of the pipeline and is initialized during parsing the scene file. Later on it starts to render the scene via the render() method and write the results back into an image. The render() method is the core of the whole program in Listing 3.2.

Listing 3.1: The main file of the renderer

```
int main(int argc, char **argv) {
    GMRender *renderer;
    renderer = <parsing the file>;
    renderer -> render();
    renderer -> writeToImage(argv[2]);
    return 0;
}
```

It first initializes the global variables at line 2 followed by the camera object generating rays at line 3. Line 7 to 9 are the starting points of the threads. Variables *integrator*, *jobber*, and *accel* are all the component objects in the pipeline. One thing to notice is that components should start in the reversed order (from right to left) in the pipeline so that later ones will be ready before the data comes in. From line 10, the pipeline is already working and the program goes into an infinite loop to wait for the conditional variable *cond_stop*. The code from line 11 to line 18 are to confirm that there is no data left in the pipeline and it is ready to exit.

Listing 3.2: The initialization of the pipeline

```
1  int GMrender::render() {
2    Global::global_initialize();
3    camera -> generateRay();
4    accel -> warm_up();
5
6    Thread::thread_mutex_lock(&Global::mutex_stop);
7    integrator -> start();
8    jobber -> start();
9    accel -> start();
10
11   while(1) {
```

```
12      Thread::thread_cond_wait(&Global::cond_stop, &Global::mutex_stop);
13      Thread::thread_mutex_unlock(&Global::mutex_stop);
14      if (Global::rayQueue.size()) {
15        Thread::thread_cond_signal(&Global::cond_accel);
16        continue;
17      } else if (jobber -> isFree())
18        break;
19
20      Thread::thread_mutex_lock(&Global::mutex_stop);
21      gordon_wait(1);
22    }
23    accel -> stop(); integrator -> stop(); jobber -> stop();
24    return 0;
25  }
```

3.2 Job Serialization

As shown in the architecture of the Cell processor in Figure 1.2, the PPE and the SPE use individual memories for data storage. When objects are transferred from the PPE to a SPE, a serialization process is required to re-create the objects from a class instance to a flat array. Since the SPE only supports C programs with limited 256KB local store, our deserialization process is required to be as compact as possible.

In Figure 2.3 we present that the data transferred to a SPE are the rays, the scene objects, and the light sources. The rays and light sources are easy to serialize as they only contain unifying data fields. A ray is serialized into 6 floating values, where the first 3 numbers are the origin position and the others are the direction vector. The light source is serialized into 4 floating values, where the first 3 numbers are the position of the light source and the 4th is the light. It is becoming harder when serializing the scene objects as object is an abstract class with multiple derived classes. It is impractical to use one unifying representation for all geometry objects(this can be done by tessellation meshes, but meshes require much more memory than parametric surface and is yet not supported by the Golden Monkey Renderer). Fortunately, C++ supports polymorphism so that different objects can implement their own serialization method to smoothly solve the marshaling problem. Conversely, the SPE only supports C which is problematic to do virtual function tricks. We solve this problem by wrapping

up all the classes inside one union structure in Listing 3.3 and use an index variable to determine which type it is. At the moment only sphere and plane are supported in the structure. The type field is to determine which structure the data stores and acts as an index for the intersection functions against the rays.

Listing 3.3: The union structure of shapes.

```
typedef struct {
  int type;
  union {
    spu_sphere_t sphere;
    spu_plane_t plane;
  } shape;
} spu_shape_t;
```

3.3 Reflection Unrolling

As the pseudocode shown in Listing 1.1, the reflection ray is normally implemented as a recursive function. There are two reasons for us to unroll this recursion in order to apply in our ray tracer. First, the recursion has too much data dependence and some are out-of-core. As it is shown in Listing 1.1, reflection ray will be traced as the same as the primary ray, which does both shadow testing and material looking-up. Since we store the material data on the PPE, the SPE needs to send the reflection ray back and let the PPE to do the material looking-up. The second reason is for the design of the programming models. In both the master/worker and map/reduce programming models, we treat the rays as independent units. Keeping the order of the rays in the queue and integrating them would be a disaster to the performance.

Our solution of the unrolling is based on the assumption that the radiance K of the ray will lower to $K_r * K$ after intersection. The K_r is the reflection parameter specified by the material of the object. The illustration in Figure 3.3 shows that a ray is shot with the initial parameter $K = 1$. When the ray hits the object 1, the parameter K becomes to $K_r[1]$ which is the reflection parameter of object 1. Then the ray hits the object 2 and the parameter goes to $K_r[1] * K_r[2]$. The Integration process at the end multiplies the parameter K with the object color and get the final reflected color.

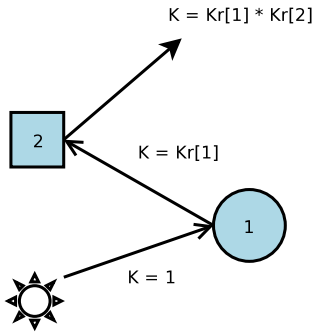


Figure 3.3: The multiply of the reflection parameters.

3.4 Program Optimization

Our optimization tricks are mainly based on SIMD instruction support, which can manipulate multiple scalar values at once. For example, the *add* operation mentioned in 1.3 can be achieved in the following code in SPE:

Listing 3.4: C code for the vector add operation.

```
vector float va = (vector float){a1, a2, a3, a4};
vector float vb = (vector float){b1, b2, b3, b4};
vector float vc = spu_add(va, vb);
```

There are several cases in which individual variables inside the vector must be treated separately. For example, when a vector stores an array of indexes to operate on, the program needs to extract every index out of the vector and then access the memory one at a time. To make this switch from vector to array smooth, we use a union structure shown in Listing 3.5 to allow a vector to be treated as an array or vice verse. In the remainder of the section, we present our optimization approaches based on the

Listing 3.5: Union structure of vector and array.

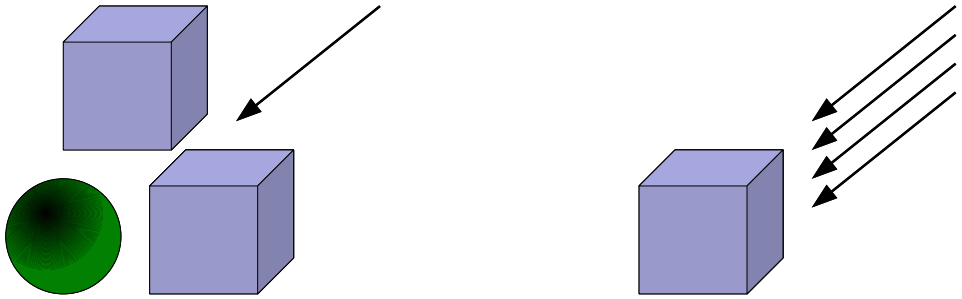
```
typedef union {
  float a[4];
  vector float v;
} mix_float4;
```

rendering algorithms. We first introduce our packaged ray intersection test, which aims to test multiple rays at one time. After that we introduce our original Map/Reduce

integration, which aims to offload the integration work on the SPE and relieve the work of the PPE.

3.4.1 Packaged ray intersection

A ray intersection test finds the nearest intersection point in the scene. It is shown that intersection tests takes around 80% computing time in the whole application[14], especially with reflections as the number of rays grows exponentially[8]. Research on speeding up the intersection test is mainly focusing on constructing the scene hierarchically and intelligently. For example, the BSP tree, which is a binary tree representing the geometry space, avoids testing the objects locating in one subtree when another branch gets intersected. Another breakthrough on fast intersection is due to the development of modern hardware like GPU and multi-core chips. With the power of SIMD programming, it is possible to do multiple intersection tests at one time. This principle can be applied using two models. The first model illustrated in Figure 3.4(a) shows that testing multiple objects with one ray and the second model in Figure 3.4(b) shows testing multiple rays with one object. The former has a disadvantage that different geometry objects have different intersection functions and it is hard to use one equation to solve them all.



(a) (b)

Figure 3.4: a) Multiple objects test with one ray. b) Multiple rays test with one object.

As an example of how SIMD programming works, we are going to present a ray-sphere intersection test. The implicit representation of a sphere is listed in formula 3.1, where

the $\begin{pmatrix} O_x \\ O_y \\ O_z \end{pmatrix}$ is the origin of the sphere and R is the radius.

$$(x - O_x)^2 + (y - O_y)^2 + (z - O_z)^2 - R^2 = 0 \tag{3.1}$$

To obtain $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ which is the location of the intersection point, we first reduce the vector representation into a scalar variable t by using ray's implicit function in formula 3.2.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} o_x \\ o_y \\ o_z \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} * t \tag{3.2}$$

The $\begin{pmatrix} o_x \\ o_y \\ o_z \end{pmatrix}$ is the position of the ray and $\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$ is the direction vector. When replacing $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ with functions over t, we get a quadratic equation of t in 3.3 where we can easily get the solution.

$$\begin{aligned}&(d_x^2 + d_y^2 + d_z^2) * t^2 + 2 * [d_x * (o_x - O_x) + d_y * (o_y - O_y) + d_z * (o_z - O_z)] * t + \\ &(o_x - O_x)^2 + (o_y - O_y)^2 + (o_z - O_z)^2 - R^2 = 0\end{aligned} \tag{3.3}$$

In Listing 3.6, we give the source code of the ray-sphere intersection function with SIMD optimizations. Functions with a prefix *spu_* are the SPE SIMD instructions and as we can see, the whole function is pretty short and elegant.

Listing 3.6: Ray-Sphere intersection function with SIMD optimization.

```
1  vec_float4 simd_intersect_sphere(vec_float4 ox, vec_float4 oy,
2                  vec_float4 oz, vec_float4 dx,
3                  vec_float4 dy, vec_float4 dz,
4                  spu_shape_t shape) {
5    vec_float4 e, t, delta, rs;
6    vec_uint4 flag;
7    // vectorize the sphere structure
8    vec_float4 r   = spu_splats(shape.shape.sphere.radius);
9    vec_float4 sox = spu_splats(shape.shape.sphere.o.x);
10   vec_float4 soy = spu_splats(shape.shape.sphere.o.y);
11   vec_float4 soz = spu_splats(shape.shape.sphere.o.z);
12
13   vec_float4 a = spu_madd(dx, dx, spu_madd(dy, dy, spu_mul(dz, dz)));
14   vec_float4 b = spu_madd(ox, dx, spu_madd(oy, dy, spu_mul(oz, dz)));
```

```
15    b = spu_sub(b, spu_madd(sox, dx, spu_madd(soy, dy, spu_mul(soz, dz))));
16    b = spu_mul(spu_splats(2.0f), b);
17
18    t = spu_sub(ox, sox);
19    e = spu_mul(t, t);
20    t = spu_sub(oy, soy);
21    e = spu_madd(t, t, e);
22    t = spu_sub(oz, soz);
23    e = spu_madd(t, t, e);
24    e = spu_sub(e, spu_mul(r, r));
25
26    t = spu_mul(spu_splats(4.0f), spu_mul(a, e));
27    delta = spu_msub(b, b, t);
28    flag = spu_cmpgt(delta, spu_splats(0.0f));
29    rs = spu_add(b,  _sqrtf4(delta));
30    rs = spu_mul(rs, spu_re(spu_mul(spu_splats(-2.0f), a)));
31    rs = spu_sel(spu_splats(0.0f), rs, flag);
32
33    return rs;
34  }
```

For streaming processors like the SPEs, it is very costy to do condition predictions[3]. One trick to reduce predication statement like if is using spu_sel, which selects elements from two vectors. At line 31, we select the final result depending on the discriminant of the quadratic equation delta. If delta is smaller than zero, the equation has no root and the result will be zero.

3.4.2 Map/Reduce integration

The Integration process computes the shading value of each pixel from the SPE workers. Our shading implementation introduced in chapter 2 has the following four steps when a ray finds the nearest intersection point.

1. **Test the shadow rays from the light source.**
2. **Get the material data of the object intersected.**
3. **Compute the shading value by mutiplying the material data with the light intensity.**

4. **Generate reflection rays and assign the color.**

As we can see from step 1-2, the integration work has many data dependencies like lighting sources and material descriptions. Because of the memory limitation on the SPE, our initial prototype leaves PPE run the total integration work. During the testing we discovered that the performance shrinks sharply when the number of jobs grows and the PPE is heavily overloaded to be the bottleneck of the whole program. To fix this bottleneck, optimization is needed to offload the shading work onto the SPE without occupying additional memory cost. Our algorithm is inspired by the Map/Reduce programming model published by Google[2]. The Map/Reduce programming model has only two functions: one is the *Map* function, which emits large associated data with input keys. The other is the *Reduce* function, which merges all the associated data with the same key into the final value. The beauty of Map/Reduce is the elegant representation of computing tasks, which can easily be computed in parallel. Many real world tasks fall into this programming model and ray tracing is no exception.

To use the Map/Reduce programming model in ray tracing, we first need to find the input *Key* and the *Map/Reduce* functions. It is straightforward to use the unique pixel position as the input key and make the result of *Reduce* function be <pixel position, RGB value>. The *Reduce* function itself can be obtained by the rendering equation explained in chapter 2 as below.

$$reduce(x,y) = \begin{pmatrix} r_i \\ g_i \\ b_i \end{pmatrix} * (k_d + k_s) + \sum_r \begin{pmatrix} r_r \\ g_r \\ b_r \end{pmatrix} * k_r \tag{3.4}$$

The $\begin{pmatrix} r_i \\ g_i \\ b_i \end{pmatrix}$ and $\begin{pmatrix} r_r \\ g_r \\ b_r \end{pmatrix}$ are the material colors of the intersected object hit by the primary and reflection ray. Scalar k_d, k_s and k_r are the computed diffuse, specular, and reflection coefficients. So here comes our *Map* function, which is to compute those coefficients and emit the paired data sets with the pixel position in formula 3.5.

$$map(x,y) \rightarrow < (x,y), i, r, k_d, k_s, k_r > \tag{3.5}$$

The i and r are the object indices for the diffuse and reflection object in formula 3.4. We list the algorithm to compute k_d, k_s, and k_r in the following items.

Diffuse Shading: The equation to compute the diffuse coefficient is

$$k_d = \sum_i \frac{intensiy_i}{|(O_i - p)|^2} \tag{3.6}$$

The O_i is the location of the ith light source and p is the intersection point. The principle behind this is that the radiance arriving at a space point is proportional to the square of the distance between the light and the point.

Specular Shading: For the specular coefficient, we are not using a physical based way to approximate it. Instead, we fake our eyes by heavily increasing the intensity of light when the incident ray is almost parallel to the tangent plane of the surface. In equation 3.7, N is the normal of the intersection point and I is the

$$k_s = \left(\frac{N \cdot I}{|(N \cdot I)|} \right)^{20} \tag{3.7}$$

incident ray. We make an exponential function to the normalized dot product between N an I so that the result will increase dramatically when the value approximate 1.0.

Reflection: As we put the material data on the PPE instead of the SPE, we can not obtain the reflection parameter K inside the SPE. In our implementation, we set the k_r equals to 0.5 for all the materials which assumes that every object emits the same radiance as half as the irradiance it gets from the rays.

We now have defined both *Map* and *Reduce* functions. We leave the *Reduce* function running on the PPE because it is simple and has the right memory access to the material. The *Mapping* function is working on the SPE to generate immediate data set. One more thing in our *Map/Reduce* is that we omit the sorting process of the immediate key because one pixel position only has one paired value in our reduce design. Another good advantage of map/reduce is that it is quite easy to expand. The paired values are independent with each other so it is possible to expand the map function on multiple Cell processors in the future.

3.5 Summary

Building a ray tracing renderer from scratch is not a trivial task, especially on the Cell B.E architecture. In this chapter, we discuss the most important issues in our

implementation of the Golden Monkey Renderer. First we introduce our source code structure to demonstrate the component driven design in our development. Later on we review the state cycle of a component and the initialization code of the program. Since the PPE and the SPE have separate memory storage, objects are required to be serialized into flat arrays to make them transferable using DMA commands. We show that serialization on different objects can be done in both C++ and C by using virtual functions and union structures. In the last part of this chapter, we reveal our most important optimization strategies by using SIMD programming. We mainly optimize two things compared to the original algorithm. The first one is to change the number of rays in the intersection test from one to four, which is ideally 4 times faster. The other modification is our original Map/Reduce integration algorithm. The motivation of this algorithm is to offload more work on the SPE without any additional memory burden. Noticing the simplicity of equation 3.4, we relieve the work on PPE to be a very small summing operation whereas leaves the intensive functions to the SPE. This design not only solves the bottleneck in the program but also presents a high scalability in the future improvement.

Chapter 4

Evaluation

In this chapter, we present our evaluation results of the Golden Monkey Renderer. The evaluation is composed of two parts. The first part is a feature comparison to a mature ray tracer. The comparison shows what basic features our renderer lacks and what advantages we have based on the platform. The second part is about performance. We characterize the performance by a set of parameters, which captures the quality of the image. For different parameters, we compare different values using our metrics. In the end, we conclude our evaluation with a detailed analysis of the whole application.

4.1 Tools and Metrics

As we mentioned in chapter 1, our benchmark is done on a PlayStation 3 with Linux as the OS. The source code on both PPE and SPE is compiled using GCC at the optimization level 3. The performance evaluation is by measuring the execution time. We do not use the Cell B.E. simulator[5] in our benchmark as it lacks accuracy in time. Instead, we record the CPU cycles inside the program and measure the time on the time bases. For example, to measure the total execution time of the whole program, we add the following code in the main file: The `mftb` in our assembly code reads the specific Time Base register, which is incremented periodically by the hardware frequency (in the SPE, we use the decrementer to read the elapsed cycles[3]). On the Cell Processor in a PlayStation 3, the frequency is 79.8Mhz, by which we divide to get the time

Listing 4.1: Measurement of Time

```
int main(int argc, char **argv) {
    uint32_t time_start, time_end;
    __asm__ volatile ("mftb %0" : "=r"(time_start));
    // actual program
    ...
    __asm__ volatile ("mftb %0" : "=r"(time_end));
    printf("Elapsed time: %f", (time_end - time_start) / 79800000.0);
    return 0;
}
```

in milliseconds. Timing on the SPE is more tricky because the SPE does not have direct access to the main memory so that it is hard to sum the time without any extra resource cost. We tackle this problem by including time data in our last job result for the following two reasons. First, there is no additional memory and computing costs in our code because our last job result is nothing but an empty array indicating the end of the buffer. Second, collecting time data from job result avoids the synchronization work because it is already synchronized in the Integrator component in the pipeline. It is a pity that we cannot separate which time data belongs to which SPE from the Gordon library and we use the *Average SPE Time* in our evaluation.

We analyze runtime and application performance using the following three metrics: total execution time, runtime rendering time, and the scalability as the number of SPE is increased from 1 to 6. The runtime rendering time here accounts for the real rendering performance since initialization and cleaning up process may cost a lot in the total execution time. We also measure the percentage of idle time SPEs have, to indicates the efficiency of workers.

4.2 Parameter Characterization

Table 4.1 shows the different parameters used to characterize the Golden Monkey Renderer program. The parameters influence the quality of the final image and the complexity of the rendering process. We have developed several scripts to generate scene files and do automatic benchmarks. From the next section, we are going to discuss our

Parameter	Description	Configurations
Resolution	The resolution of the final image	640x480, 1024x768, 1280x720
Reflection	Reflection visible in the final image	no reflection, 1 level reflection
Scene	The rendering scene	simple scene medium scene complex scene
Optimization	Our optimization using SIMD and Map/Reduce	With optimization, Without optimization

Table 4.1: The characterization model in the Golden Monkey

result with different configurations shown in the table.

4.3 Feature Comparison

In this section, we are going to study the feature capability compared to the POV-Ray ray tracing program. Since we only have relatively very limited developing time, our Golden Monkey Renderer lacks many features as a production application. However,

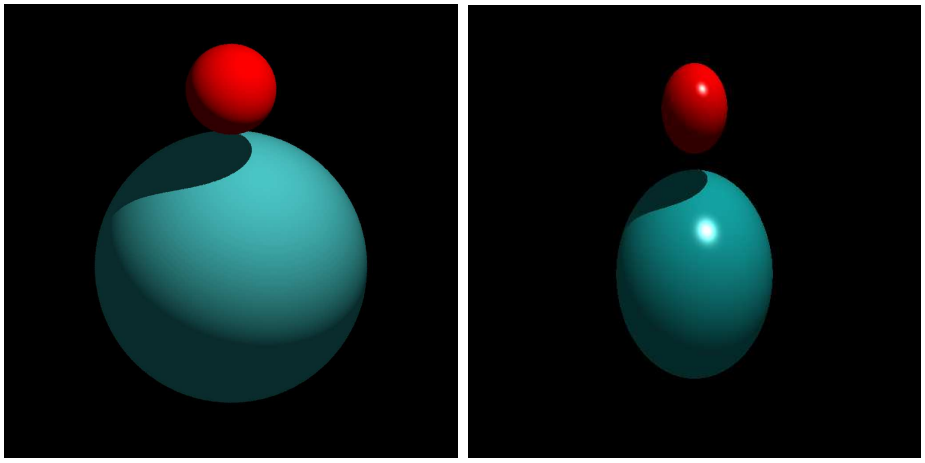


Figure 4.1: Image rendered by two ray tracers.

comparing to another ray tracing application is still a good guidance for future development. The POV-Ray program has been developed since 1980s as an open source

software by David Kirk Buck[9]. It is now widely accepted and used by many designers and producers. In Table 4.2, we list several key features to compare with our Golden Monkey Renderer.

Feature	Golden Money Renderer	POV-Ray
Scene description	XML standard.	New language
Output Image File	EXR file	PNG, TGA, BMP
Geometry Primitive	only sphere and plane	all kinds of primitives used today
Lighting	only point light	simple lighting and area light
Particle	no support	support with atmospheric effects
Shading	local illumination	global illumination
Texture	no support	supported
Platform	Cell B.E	x86
Multicore	supported	no support
OS	Linux	Windows, Mac, Linux

Table 4.2: Feature comparison between Golden Monkey and POV-RAY.

4.4 Total Execution Time

Figure 4.2 shows the total execution time of the Golden Monkey Renderer compared to the POV-RAY ray tracer, which runs on a PC with Intel Duo2 core 1.8Ghz and 1GB memory. Every scene benched here is rendered in 720p resolution using 6 SPEs and with 1 level reflection. In general, our optimized renderer adapts very well to different scenes as the time differences are small. Conversely, the unoptimized renderer and the POV ray tracer scale linearly with the complexity of scenes.

Scene	Optimized	Un-optimized	Optimization speedup	POV-RAY	POV-RAY VS Optimized
simple	2.75s	2.86s	4.5%	2.0s	-37.3%
medium	2.86s	3.30s	13.3%	3.0s	4.7%
complex	3.29s	4.52s	27.2%	4.0s	17.8%

Table 4.3: Benchmark table of the total execution time(s).

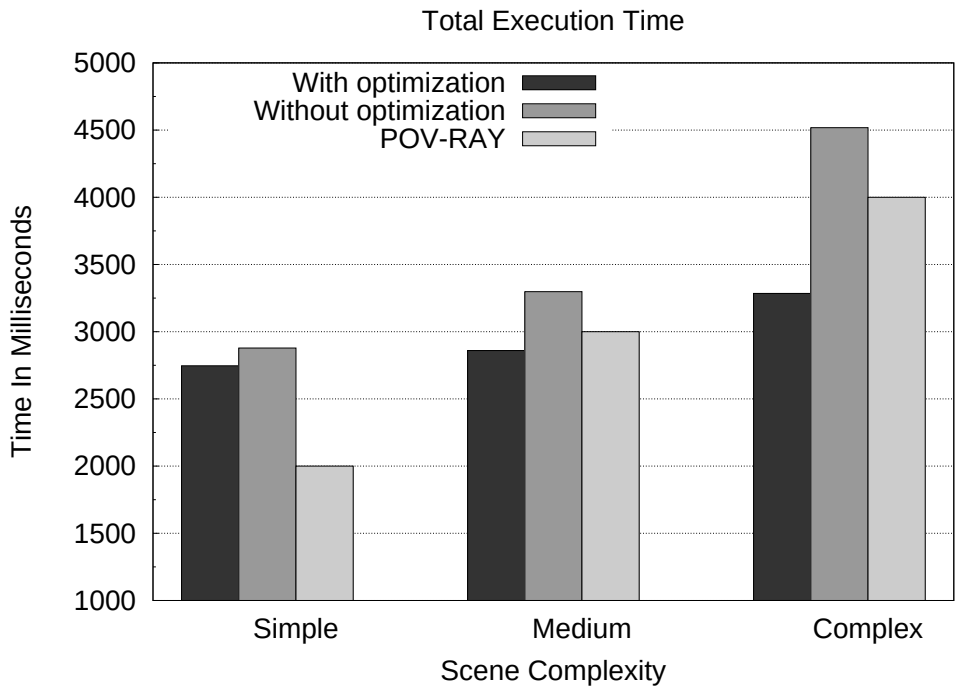


Figure 4.2: Total Execution Time in rendering.

Table 4.3 shows the exact time of each rendering task. In the first simple scene benchmark, our renderer only has 4.5% speedup after optimization and is 37.3% slower than the POV-RAY ray tracer. In the medium scene benchmark, the optimization speedup to unoptimized version triples to 13.3% and a positive 4.7% speedup to the POV-RAY. In the final complex scene benchmark, the speedup to unoptimized version boosts to 27.2% and triples to 17.8% relative to the POV-RAY.

As we can see from Table 4.3, our lowest rendering time for the simplest scene is around 3 seconds, which is still too slow for such simple scene. One reason could be that the initialization and deconstruction procedures are very expensive in time. In Figure 4.3, we annotate our optimized program to show the time partition of the whole execution.

The annotation is measured using different resolutions and scenes. As the resolution goes up, the preparation time increases but treats the same for all the scenes. Our annotation proves that the initialization time grows depending on the resolution, which could be explained by the procedure of sampling the initial position of each ray. For

Figure 4.3: Annotation of the execution time on the optimized version. The benchmark uses variety of resolutions from 640x480 to 1280x720 for different scenes.

example, a 1280x720 image has 921600 pixels and for each pixel, the random number generator is called twice to get the pixel position and the position is transformed from screen view to world view. It is reasonable even for a 3.2Ghz PPE to take this amount of time to perform this. Apart from the initialization time, the runtime performance grows near linearly for different scenes, which is logical due to the increase of the scene complexity.

4.5 Runtime Performance

Our runtime benchmark is to evaluate how the performance grows as the number of SPEs grows. Figure 4.4 shows the frame per second (FPS) rendered using the simple scene as the number of SPEs grows from 1 to 6. The rendering speed initially has a rapid increase when the number of SPEs grows from 1 to 3, and gradually becomes stable from 4 to 6. With the increasing number of SPEs, the power of workers is ideally

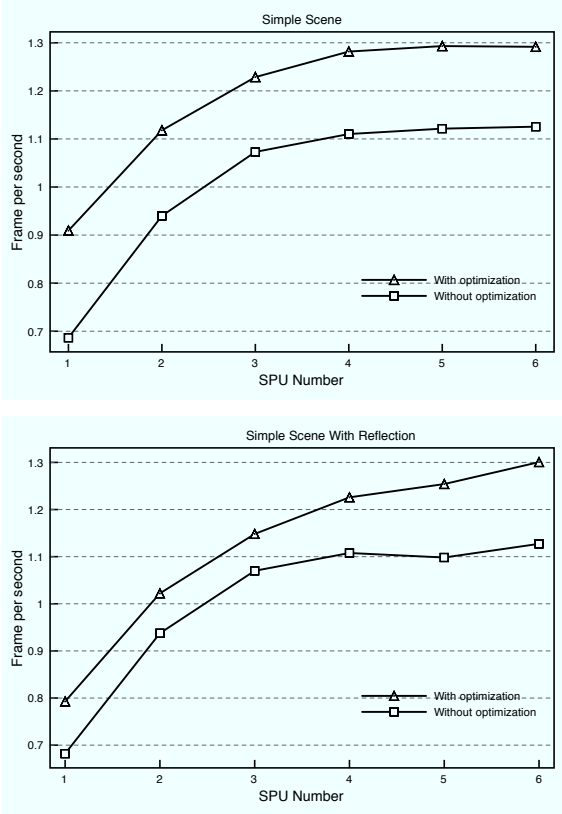


Figure 4.4: Runtime simple scene benchmark using resolution of 1280x720.

multiplied, which explains the first rapid growth. However, when the number of SPEs is enough to handle the current rendering tasks, more SPE workers will not give more improvement as the workforce is already enough. When we compare both benchmarks using the configuration of reflection, the main difference is that the gap between two lines is not identical in the latter benchmark. When the reflection is enabled, the unoptimized program needs to send the reflection ray back and forth in the pipeline, which has more communication work than our Map/Reduce algorithm. When the number of SPE increases, it soon alleviates their work intensity and obtains better speedup. The rendering speed of unoptimized program slightly degrades at number of 5, which shows the overloading work though the speed slightly moves higher at 6.

In Figure 4.5, we continue to benchmark both programs using medium and complex scenes. The graph of the medium scene looks similar to the previous figure. However, the gap between both lines is growing bigger, which indicates that the map/reduce

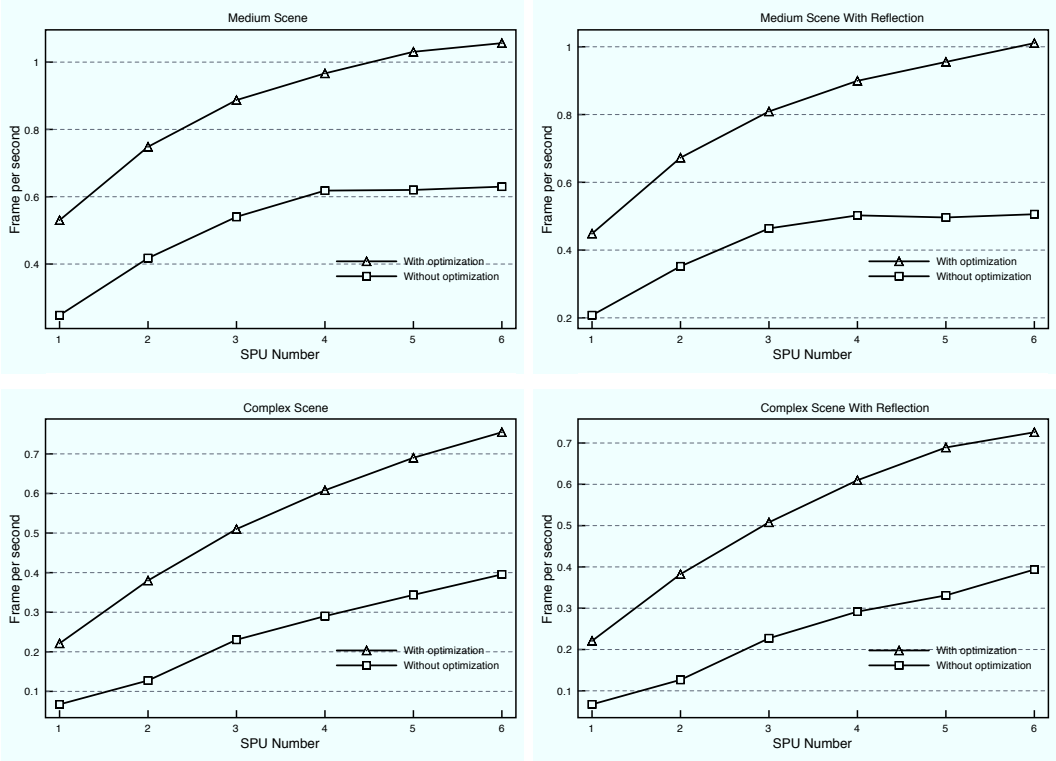


Figure 4.5: Runtime benchmark on medium and complex scene using the resolution of 1280x720.

model works more efficiently in complicated scenes. Despite of that, our best rendering speed is below 1fps, which is still far from interactive speed. Hence, we are going to study the efficiency of workers in the next section.

4.6 SPE Efficiency

The SPE efficiency benchmark measures the idle time of SPE workers. Our measurement is based on the total computing time running on the workers and divided by the number of SPE, called *Average SPE Execution time*. Figure 4.6 presents the percentage of the idle time with different SPE counts. As the SPE count grows, the idle time per SPE almost increases linearly and reaches 45% in the optimized program. Our workers in the optimized program are more than two times as idle as in the unoptimized ones. There are two reasons to account for this inefficiency. First, the optimized workers

are much faster than the unoptimized ones because of the vectorized optimizations. Secondly, the bandwidth of job transferring cannot sustain the workload of the SPE

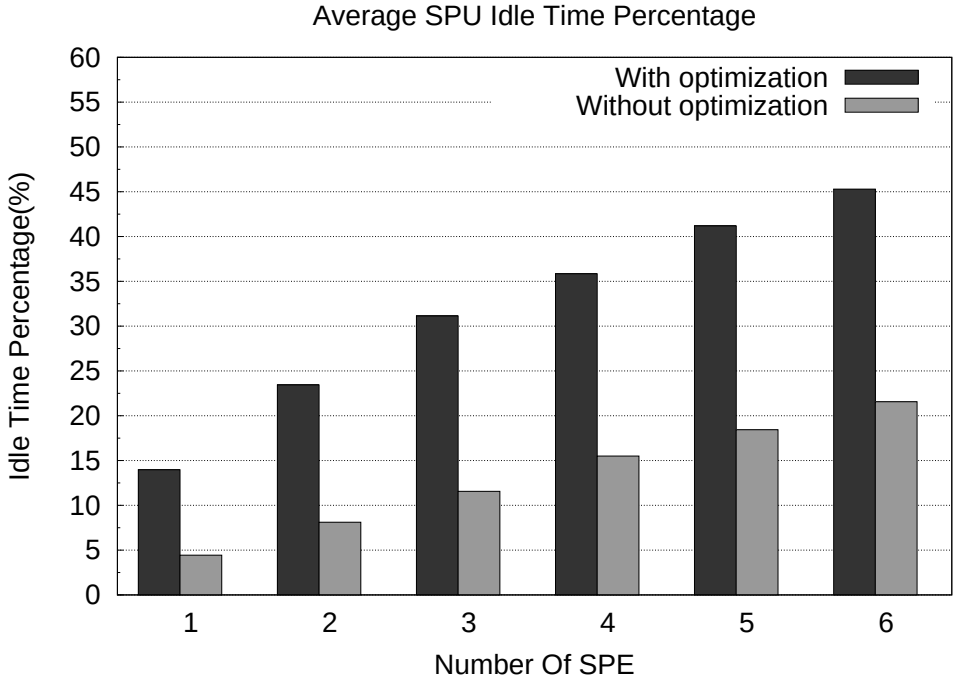


Figure 4.6: The benchmark uses the complex scene with resolution of 1280x720.

workers, which makes them wait for new jobs. This bottleneck can be caused by the overload of the PPE. As we introduced in Chapter 3, the tasks of the PPE during runtime are mainly integration and synchronization. We exempt the overload of the integration because in the reduce integration, the PPE only has a summing operation which is trivial to effect the performance. On the other hand, the synchronization is comprised of both SPE synchronization and pipeline synchronization. As there are only two threads in the pipeline running in realtime (the jobber and integrator), we conclude that the main bottleneck is on SPE synchronization, aka the job management and DMA transferring.

In Table 4.4, we profile the benchmark and list the top four time consuming functions. The first function lives in the Linux kernel vmlinux which takes around 62% time and the function is .restore_user_regs. The .restore_user_regs is to restore registers in the context switching between threads (the SPE is also treated as a thread in the Cell processor[4]). The only context switching places in our application are the multi-

samples	%	app name	symbol name
1429	62.0226	vmlinux	.restore_user_regs
185	8.0295	libc-2.6.1.so	(no symbols)
132	5.7292	libIlmImf.so.4.0.0	(no symbols)
125	5.4253	gm	.gordon_poll_core

Table 4.4: The top 4 time consuming functions in the profiling report.

threading pipeline and the SPE workers. As the PPE core has two hardware threads[3], the .restore_user_regs would be used for the hyper-threading among the multi-threads. The second and third item in the table are the regular library one time calls. The fourth item .gordon_poll_core is called when the PPE is waiting for available SPE workers. So in total, there is around 70% time used in synchronization of multiple cores or workers, which proves the conclusion we give above. In the next section, we discuss the bottleneck and present possible improvements.

4.7 Discussion and Future Improvement

Our feature comparison shows that the Golden Money Renderer is a full fledged ray tracing application with flexible component design. Since the development time was limited, the program is still an initial prototype, lacking necessary features like triangle objects and textures. The simple, flat accelerator we use prevents us from rendering sophisticated geometry objects as big geometry can easily surpass the small memory in the SPE. Our shading equation is designed for local illumination so it still needs more work to implement the global illumination algorithm. However, the component design is quite successful, which leaves lots of expanding spaces for new features without large modification of the source code. As future work, we propose the following two major improvements that can greatly enhance the quality and capability of the renderer.

Hashable and Hierarchical accelerator: The problem of processing complicated geometry is the limitation of local memory on the SPE. Our flat accelerator requires every object in the scene to be located in the local buffer, which is a waste when objects are in the space where rays will never reach (e.g., a object is obstructed by another one in front of it). One ideal approach is to have a hash function, which takes the ray

as keys and results in the exact testing data needed. To reach such a goal, one can use hierarchical partition trees like BVH and BSP Tree[14]. As the memory size of a space partition tree is small, the worker program can persistently keep it for indexing and locating the required geometry data on demand. In additional to the benefit of saving memory, the search complexity also reduces to a logarithmic scale, which is fast enough to tackle any sophisticated scenes. Once the workers indicate which part of data is needed, they can start a DMA transfer on the fly, making the communication more efficient with their master.

Photon Mapping: Photon Mapping is recognized as one of the most important breakthrough in computer graphics in the past decade. It is developed by Henrik Jensen in 1995 as an efficient bidirectional technique to approximate global illuminations[7]. Besides the highly realistic image quality, Photon Mapping can also be easily parallelized. The rendering process of Photon Mapping is not to recursively trace rays, but to estimate the radiance from the nearest photons, which can be bounded with our hash function mentioned above.

The performance benchmark shows that the rendering procedure scales badly when using a master/worker pattern. However, the intensive tracing work gains a remarkable speedup after optimizations and the Map/Reduce algorithm shows high potential for future research and exploration. Nevertheless, our rendering performance is still poor compared to others. As future work, we propose the following two improvements to enhance the performance.

SPE centric working pattern: One of the main performance bottlenecks we discovered is the heavy overload on the centralized master process, which keeps synchronizing jobs and integrating results. We also observed during the benchmark that multi-threading pipeline is not an optimal design on Cell B.E. since the PPE suffers a lot from the overhead of switching threads. The relatively large amount of mutex locking/unlocking and conditional variables' signaling/waiting are primarily responsible for the big idle percentage time of the SPE. Thus, the solution we propose is using decentralized design, in which each SPE autonomously works on certain part of the image and the PPE reduces the emitted paired values into pixels. The difference is that the master/worker model is PPE centric where the master talks to the workers. Whereas the SPE centric pattern is decentralized and the workers talk to the master.

In addition to better data communications, the kernel in each SPE can also manage its software caches more efficiently. One hurdle in such design is the load balance between workers. It would be another waste if one worker finishes his work far ahead than others. One possible solution concerning this is to tile the image into several parts and assign iteratively each part to the SPE.

Enhanced Map/Reduce algorithm: The Map/Reduce algorithm in the design shows high scalability and flexibility in parallel processing. It would be valuable to expand the reduce level on multiple hierarchical models. As an example in our implementation, it is possible to use another PlayStation 3 as a post-processing processor, which processes our immediate values to apply pixel effects and texture mapping.

4.8 Related Work

Our related work discusses current research and applications on ray tracing, and specifically on the Cell B.E platform. The first Ray tracer experiment on the Cell processor is by Carsten et al[1] from inTrace, which is a start-up originally from the OpenRT project. Their approach uses the SPE centric model to let each SPE render a different part of image. They especially take care of the cache management in the SPE by using software-hyperthreading. Topics including index structure and BVH traversal are also described in this paper.

Another trail about raytracing on the Cell processor is by the students from MIT at the Multi-core programming course. The course is mainly about multi-core programming on the Cell processor and one of the team projects called Blue-Steel aims to develop an interactive ray tracing program. Their approach is almost the same as the Golden Monkey Renderer which encloses the scene inside the local cache but they use manual DMA transfers[12] for geometry data.

Upon the time of writing this thesis, the most impressive ray tracing program on the Cell processor is developed by IBM itself. IBM released their interactive ray tracing called iRT[6] to the public to show the strength of their own chip. Since it is a commercial application, we are not able to get more information in depth. Nevertheless, the performance of iRT is remarkable with interactive speed even in rendering complex

scenes.

4.9 Summary

In this chapter, we present our evaluation result of the Golden Monkey Renderer from both function point of view and performance point of view. We first evaluate the supported functions by comparing to the popular POV-Ray ray tracer. It is shown that the Golden Monkey Renderer is a full fledged functional application, providing simple XML style language for scene description. However, the weakness of little geometry support and simple local illumination still make our implementation far away from mature. After that we unveiled our performance evaluation results by rendering different scenes with different configurations. The settings of different configurations indicate different image quality and rendering complexity. The Golden Monkey renderer shows a linear speedup when scaling the number of SPE but performs poorly in the total execution time. We discover that the main cost lives in the initialization procedures during execution. Another bottleneck we found in runtime benchmark is the overhead of data communication and multi-threading synchronization. When the number of SPEs grows, the work of synchronization overloads the application. In the last part of this chapter, we analyze possible future improvements and one of which would be to change centralized master/worker pattern to decentralized SPE centric pattern, which alleviates the work on the PPE and offloads more work to the SPEs.

Chapter 5

Conclusion

Ray tracing is one of the oldest image synthesis techniques in the computer graphics field. By the simplicity and elegance of copying the human eye system, ray tracing is capable of generating the most realistic virtual images, especially for certain effects like shadows and mirrors. However, ray tracing also has a reputation of huge computation cost, which is not overcome in the past 20 years. With the booming speed of modern computers and specially the emerging multi-core technology, it is possible and tempting to do ray tracing in real time, replacing the traditional rasterization approach. The Cell processor developed by STI provides such a cutting-edge and innovative hardware environment with vast performance potentials, but it is also a challenging platform. This thesis presents a ray tracing program called Golden Monkey Renderer for the Cell processor. The application provides a simple description language and a multi-threading pipeline to process the whole tracing algorithm.

We did an evaluation in our application both in capability and performance. Our application shows quite preliminary features comparing to the mature ray tracing application POV-RAY. The limitations on shading and rich geometry are mainly due to the complexity of hardware and relatively constrained development time. On the other hand, the speed of the Golden Monkey Renderer outperforms the POV-RAY during our benchmarks, even though the latter runs on a more expensive Intel machine. Our benchmarks show that in a more complicated scene, the performance of the Cell B.E can achieve around 1.2X speedup compared to POV-RAY and we believe there is more

after the future development.

The optimizations of the Golden Monkey Renderer are done by writing vectorized instructions and using a Map/Reduce model to offload more work on the SPE. Though we obtain a remarkable speedup for processing rendering jobs, we discovered that the main bottleneck during rendering is actually the overhead between the PPE and SPE data communication and multi-threading synchronization. Future development should change the centralized master/worker pattern to a decentralized pattern. Looking forward, this application has the potential to expand to multiple Cell processors and to have more functionality and flexibility.

Appendix A

Render Showcase

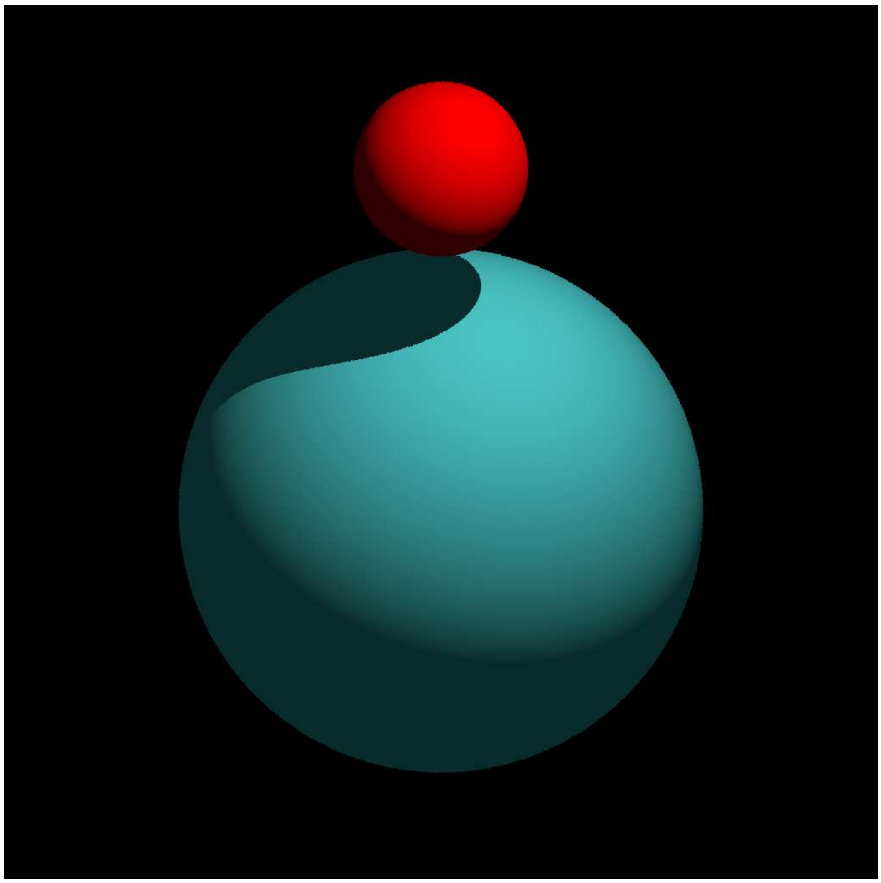


Figure A.1: Simple Scene: Two spheres with one light source

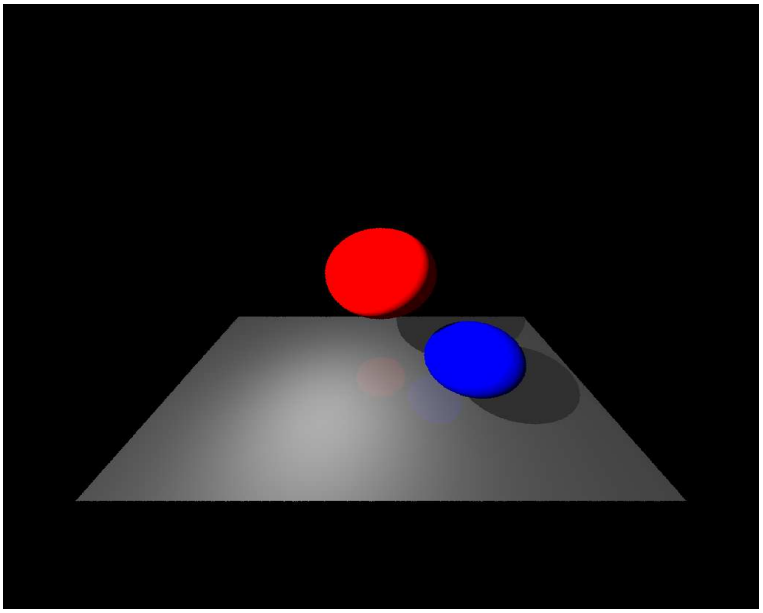


Figure A.2: Sphere and Plane: Two spheres and one plane, one light source with reflection enabled.

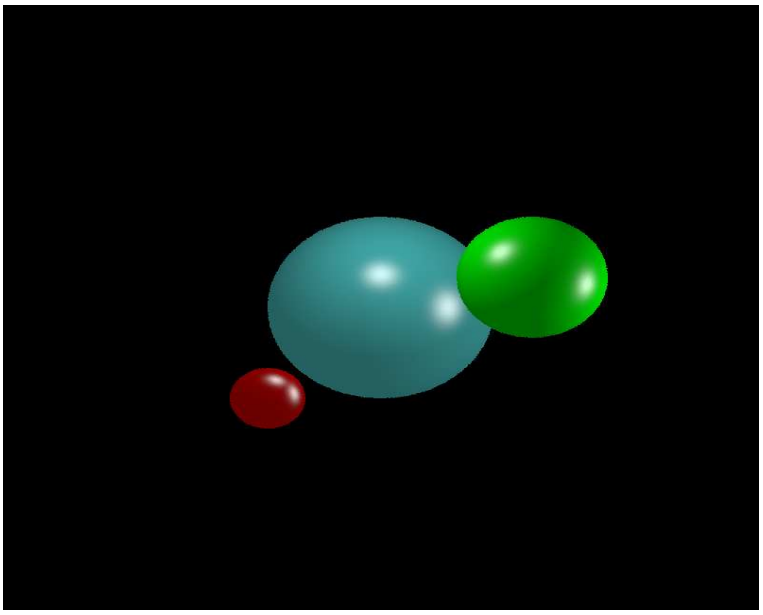


Figure A.3: Specular lighting: Three spheres and two light sources with specular lighting enabled.

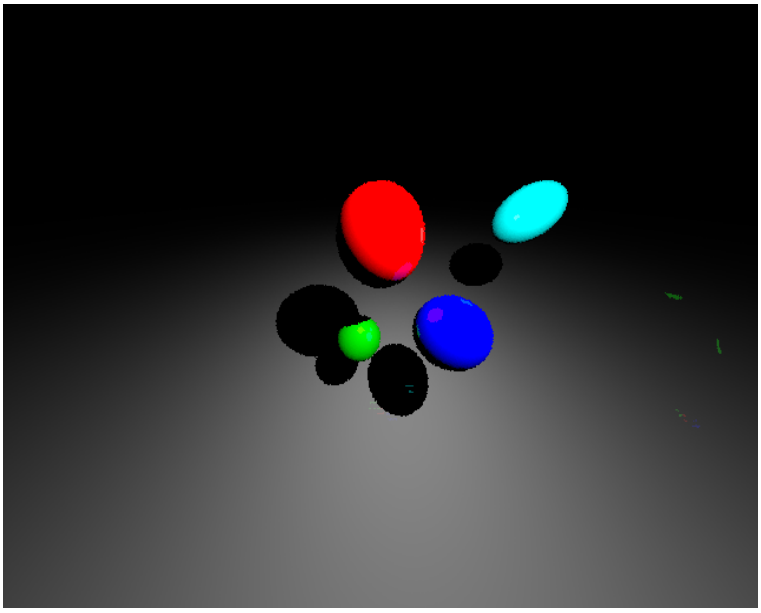


Figure A.4: Medium Scene: Four spheres and one infinite plane, one light source with reflection enabled.

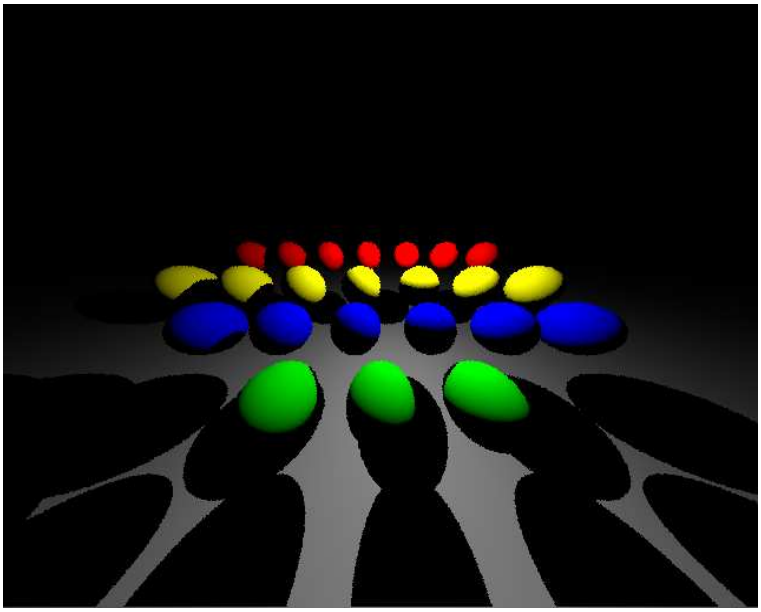


Figure A.5: Complex scene: Twenty-four spheres on a plane, with three light sources.

Appendix B

Scene Description Language

This appendix section describes the scene description language used in the Golden Monkey renderer. Files are created in plain ASCII text and the extension is .xml. The overall syntax of a scene is shown in Listing B.1.

Listing B.1: The syntax of the description format

```
SCENE:
<scene>
 OBJECT        : = {SPHERE | PLANE} MATERIAL | OBJECT
 LIGHT
 CAMERA
</scene>
```

A complete scene is consist of three items, which are the object, light, and the camera. In the rest of this section, we are going to introduce each item supported in our renderer. To demonstrate the overview of a complete scene file, we list the source code of the simple scene file in Table A.1 at last.

Table B.1: The <plane>tag description

Option	Description
point0 point1 point2	three unequal points on the plane

Table B.2: The <scene>tag description

Option	Description
resx	the width of the image
resy	the height of the image
spp	sample per pixel
accel	accelerator type, only *simple* avaliable
camera	the camera type, *perspective* or *orthogonal*
jobber	the jobber type, only *cell* available
rays_per_job	the ray number in job package
max_ray_round	the reflection allowance for the ray
simd	enable the SIMD optimization

Table B.3: The <material>tag description

Option	Description
color	the RGB value of the material
diffuse	the diffuse coefficient of the material
specular	the specular coefficient of the material
reflection	the reflection coefficient of the material

Table B.4: The <light>tag description

Option	Description
point	the origin point of the light source
intensity	the raidance of the light source

Table B.5: The <camera>tag description

Option	Description
point	the location of the camera
lookat	the the lookat direction of the camera
up	the up boundary of the view point
left	the left boundary of the viewpoint
right	the right boundary of the viewpoint
bottom	the down boundary of the viewpoint
near	the nearest focus of the camera
far	the farest focus of the camera

Listing B.2: Simple scene description file for Figure A.1

```
<scene resx="1280" resy="720"
       nspu="6"
       spp="1"
       accel="simple"
       camera="orthogonal"
       jobber="cell"
       rays_per_job="1024"
       max_ray_round="2"
       simd="1">
  <sphere>
    <point>0 -3 -7</point>
    <radius>3.0</radius>
    <material>
      <color>0.1 0.8 0.8</color>
      <diffuse>0.6</diffuse>
      <specular>0.4</specular>
      <reflection>0.0</reflection>
    </material>
  </sphere>

  <sphere>
    <point>0 2 -7</point>
    <radius>1</radius>
    <material>
      <color>1.0 0.0 0.0</color>
      <diffuse>0.8</diffuse>
      <specular>0.2</specular>
      <reflection>0.0</reflection>
    </material>
  </sphere>

  <light>
    <point>3 9 -5</point>
    <intensity>40</intensity>
  </light>

  <camera>
    <point>0 6 3</point>
    <lookat>0 -.8 -1</lookat>
    <up>0 1 0</up>
    <left>-5</left>
    <right>5</right>
    <top>5</top>
    <bottom>-5</bottom>
    <near>0</near>
    <far>10</far>
  </camera>

</scene>
```

Table B.6: The <sphere>tag description

Option	Description
point	the origin point of the sphere
radius	the radius of the sphere

Appendix C

Program Usage Guide

The program package can be obtained from http://www.few.vu.nl/šyn200/gm/. The package contains both the source code and sample scene description files. One can extract the files using the following command.

```
tar xfv gm07.tar
```

Compilation: One can easily compile the whole application by typing the make command in the ./raytracer/src/ directory.

Rendering: The command to render the scene should be as follows:

```
gm [input file] [output file]
```

The **input file** is the input scene description file while the **output file** is the rendering image with the extension of EXR. One can view the image by using any EXR supported image viewers (e.g. the command exrdisplay included in the libexr library).

Bibliography

[1] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the cell processor. *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 15–23, 2006. 42

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, 2004. 28

[3] IBM, editor. *Cell Broadband Engine Programming Handbook, Edition 3.0.* IBM, 2007. 27, 31, 40

[4] IBM, editor. *Cell Broadband Engine Programming Tutorial, Edition 3.0.* IBM, 2007. 4, 6, 14, 39

[5] IBM, editor. *IBM Full-System Simulator User's Guide, Edition 3.0.* IBM, 2007. 31

[6] IBM. An interactive ray tracer for the cell/b.e. processor. Press article, 2007. http://www.alphaworks.ibm.com/tech/irt. 42

[7] Henrik Wann Jensen. *Realistic image synthesis using photon mapping.* A. K. Peters, Ltd., Natick, MA, USA, 2001. 41

[8] Greg Humphreys Matt Pharr, editor. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann, 2004. 10, 15, 25

[9] Persistence of Vision Raytracer Pty. Ltd. Povray, the persistence of vision raytracer. Website, 2007. http://www.povray.org. 34

[10] Vijay Pande and Stanford University. Folding@home distributed computing. Website, 2007. http://fah-web.stanford.edu. 4

[11] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975. 15

[12] Blue-Steel team. Blue-steel ray tracer. Multicore Programming Primer course, MIT, 2007. http://cag.csail.mit.edu/ps3/blue-steel.shtml. 42

[13] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. 11, 15

[14] Si Yin. Acceleration structure in ray tracing. Course report, Virje Universiteit, 2007. http://www.few.vu.nl/~syn200/paper.pdf. 11, 25, 41