# A GRAPHICS ARCHITECTURE FOR RAY TRACING AND PHOTON MAPPING

A Thesis

by

JUNYI LING

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2004

Major Subject: Computer Engineering

# A GRAPHICS ARCHITECTURE FOR RAY TRACING AND PHOTON MAPPING

A Thesis

by

JUNYI LING

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

| | |
|---|---|
| Rabi N. Mahapatra<br>(Chair of Committee) | Ergun Akleman<br>(Member) |
| John Keyser<br>(Member) | Valerie E. Taylor<br>(Head of Department) |

August 2004

Major Subject: Computer Engineering

# ABSTRACT

A Graphics Architecture for
Ray Tracing and Photon Mapping.
(August 2004)
Junyi Ling, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Rabi Mahapatra

Recently, methods were developed to render various global illumination effects with rasterization GPUs. Among those were hardware based ray tracing and photon mapping. However, due to current GPU's inherent architectural limitations, the efficiency and throughput of these methods remained low. In this thesis, we propose a coherent rendering system that addresses these issues. First, we introduce new photon mapping and ray racing acceleration algorithms that facilitate data coherence and spatial locality, as well as eliminating unnecessary random memory accesses. A high level abstraction of the combined ray tracing and photon mapping streaming pipeline is introduced. Based on this abstraction, an efficient ray tracing and photon mapping GPU is designed. Using an event driven simulator, developed for this GPU, we verify and validate the proposed algorithms and architecture. Simulation results have validated better interactive performances compared to the current GPUs.

To

Baldur, son of Odin and Frigg, the god of light

# ACKNOWLEDGMENTS

I would like to thank my parents for tolerating my perpetual studentship and for supporting me both spiritually, emotionally and financially for so many years.

I would like to thank Dr. Mahapatra, Dr. Keyser and Dr. Akleman for teaching me so much in the last couple of years. Without their knowledge and their direction I would not be where I am today.

On a lighter note, I would like to thank my office mates. I would like to thank John for teaching me FPGA related "stuff", with which I have been earning a living for the last couple of years. I would like to thank Brian for "Sam Hall." I would like to thank Ian and Joe for introducing me to "A Hitchhiker's Guide to the Galaxy", which has become my all time favorite book – sorry I read "Lord of the Rings", it didn't take, there was way too much running, I got tired. I would like to thank Praveen for reminding me that I am not "professor material", and his noble, yet ultimately unsuccessful attempt at saving me from academic madness. I would like to thank Anand for reminding me constantly about how inadequate he feels about himself, thereby making me feel even less adequate, because he already has two major publications. I would very much like to thank Nitesh Goyal for stating the obvious so many times and making me feel better about myself. I would like to thank Siddharth for not making me feel bad about following in his footsteps. I would also like to acknowledge my reverence to the first Ethiopian who discovered the potent substance that we know today as coffee, for it has kept me awake for unnaturally long periods of time.

TABLE OF CONTENTS

CHAPTER Page

# LIST OF TABLES

TABLE Page

# LIST OF FIGURES

FIGURE Page

# CHAPTER I

## INTRODUCTION

Three dimensional image synthesis (also referred to as rendering in this thesis) has been one of the mostly studied topics in computer graphics. There are two key approaches to image synthesis: rasterization and ray tracing. Interactive rendering is commonly addressed through rasterization. The visual quality of ray traced images is generally considered to be superior to that of rasterization. This is due to the capability of ray tracing to create indirect illumination effects such as soft shadow, reflection and refraction. Photon mapping has been developed [1] to create realistic diffused inter-reflection, caustics and subsurface scattering effects. Recently, there have been a number of studies with programmable Graphics Processing Units (GPU) that illustrate hardware accelerated ray tracing and photon mapping. However, implementing ray tracing and photon mapping on current generation GPU is inefficient and difficult. Modern rasterization GPU contain a number of Vertex and Fragment processors as well as a fixed rasterization pipeline. Only a subset of fragment processors has the data paths [2] [3] for ray tracing and photon mapping. The vast majority of the rasterization hardware resources cannot be used for ray tracing or photon mapping. We are motivated by the desire to create a better GPU, one that is designed for ray tracing instead of rasterization.

In this thesis, we propose a novel GPU image synthesis system that is specifically designed to realize ray tracing and photon mapping. We introduce a set of new acceleration algorithms to efficiently support this goal. These algorithms create data coherence and enhance multi-nodal stream processing. We propose a GPU architec-

---

The journal model is *IEEE Transactions on Automatic Control.*

ture to process these algorithms efficiently. The processor's data paths are partitioned into individual data streams. Each data stream sequentially passes through separate PMs in order to create the final image. The complete rendering procedure takes place locally within the GPU, without costly CPU read-backs as seen in previous architectures. System-level simulation is setup to evaluate our GPU design and performance. From the results, it is shown that the proposed pipeline rendering architecture can perform ray tracing at interactive frame rates and that photon mapping combined with ray tracing is possible at near interactive frame rates. We also show that using multiple GPU rendering systems further improves performance of our rendering system. In chapter II, we discuss the related works in this area. Chapter III introduces a number of new algorithms for our rendering system. In chapter IV the hardware architecture is introduced. In chapter V we present the results obtained with our system. Chapter VI contains our conclusion and some discussions regarding possible future studies in this area.

# CHAPTER II

# PREVIOUS WORKS

Ray tracing and photon mapping are high-end image synthesis methods in computer graphics. Many recent research efforts on these topics have been focused on the quality of renderings, i.e. how closely the rendered images approximate the perceived reality. A number of studies have also been conducted with hardware accelerated ray tracing and photon mapping. In this section, we briefly introduce the current state of GPU design, discussing existing algorithms for ray tracing and photon mapping, as well as describe previous studies relevant to hardware aceelerated ray tracing and photon mapping.

## A. Background and Fundamentals

In the following section we briefly discribe the basic concepts in image synthesis and graphics rendering hardware. We also define the technical terms used throughout this thesis.

### 1. Rasterization and Rasterization Hardware

In interactive applications, rasterization is the preferred rendering method. Within a GPU, the rasterization pipeline can be abtracted into three distinct stages: Transformation and Lighting, Rasterization, and Per-Fragment/Per-Pixel operations. The Transformation and Lighting operation performs geometric transformation, and assigns per-vertex lighting information according to the position and orientation of the vertices to the light sources. The rasterization stage converts the original 3-D geometry in "world-space" to a 2-D projection plane. During this process, most hardware systems also clip and cull the geometry that is outside of the viewing frustum. The re-

sulting geometry is turned into triangle fragments that can be lit locally on a per-pixel basis. The resulting fragments are tested against the depth buffer. If the fragments' depth values are less than the values stored in the depth buffer they are passed to the frame buffer and rendered.

This has been the main rendering method for interactive image synthesis for many years. This scheme however is not very flexible in terms of providing realistic illumination models. The reflectance function of a surface is abstracted by its BRDF (Bidirectional Reflectance Distribution Function). Introduced by Nicodemus et al in 1977 [4], this function give an accurate approximation of surface reflectance. It has not been implemented by GPUs until very recently.

The recent major innovations in GPU design have been the programmable vertex and fragment shaders. This limited programmability have allowed flexible local shading models to be rendered at interactive frame rates. With multi-pass rendering limited indirect illuminations can also be rendered. Shadows are rendered with either, stencil mapping or shadow mapping algorithms. Physically incorrect reflection and refraction are simulated with environment mapping. Pre-computed diffused inter-reflection can be applied as textures to create the illusion of global illumination.

However, these methods suffer from the limitations of rasterization. The computation of physically accurate shadows is expensive. Accurate reflection and refraction can not be computed for dynamic environments. More advanced global illumination effects such as diffused inter-reflection and caustics can not be generated interactively. Further more, with rasterization hardwares per pixel assypmtotic complexity of rasterization is $O(n)$ time [5], with $n$ denoting the number of polygons in the rendered model.

It should be noted that there are methods that reduce the computational complexity of rasterization. These methods have been implemented in software as a

preprocessing step. There have also been hardware occlusion methods implemented within GPUs recently, these methods do not reduce the asyptotic complexity of rasterization. We only consider hardware-based renderings in this thesis.

## 2. Ray Tracing

Ray tracing has been the preferred method for generating realistic images, because ray tracing produces more global illumination effects. It is known for generating realistic soft shadows, highly specular reflection, and refraction. Ray Tracing has been considered traditionally as the slower of the two rendering methods. However, it has been shown in [5] that with appropriate acceleration algorithms, the per-pixel cost of ray tracing is $O(\log n)$ time, which is faster than rasterization.

The first ray tracing acceleration algorithm implemented is the Oct-tree traversal algorithm proposed by Andrew Glassner [6] in 1984. Since then, many acceleration data structures have been implemented. They can be seperated into four major catagories: Oct-tree based, grid based [7], BSP-tree based [8], and bounding-hierarchy based [9]. The majority of these methods utilizes tree data structures for storing geometric information.

## 3. Global Illumination

Diffused reflection effects can be generated with finite element radiosity methods for Lambertian surfaces. Monte Carlo bidirectional path tracing can also generate diffused inter-reflections. This is described in detail in [10]. Recently the method of photon mapping introduced in [1] and [11] has become the preferred algorithm for producing global illumination effects such as diffused inter-reflection, caustics and subsurface scattering [12]. Both bidirectional path tracing and photon mapping require two-pass ray tracing schemes to perform the reflected radiance estimation

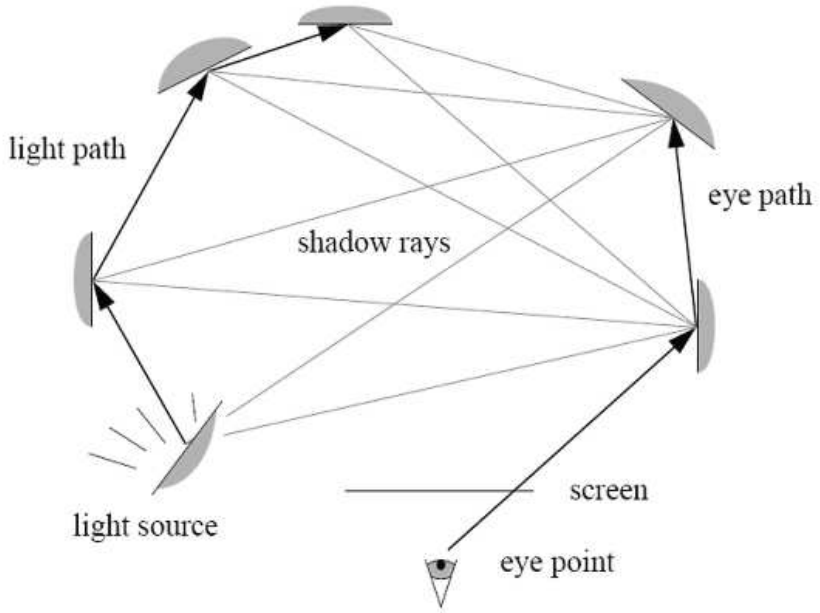


Fig. 1. Bidirectional Path Tracing.

computation.

a. Bi-directional Path Tracing

We call attention to the fact that there is at least two ray tracing passes in both bi-directional path-tracing and photon mapping. In the case of bi-directional path tracing, the light rays $L_l$, s.t. $l \in \{1, c\}$, where $c$ is a constant are stochastically emitted from a light source. A list of rays intersection points $P_l$, that ray $L_l$ intersect is stored. A single eye ray $L_e$, is emitted from the eye point. $L_e$ is reflected and refracted multiple time. For each point of reflection/refraction, a point $P_e$ is stored. The radiance flux at eye point in the direction of ray $L_e$ can be computed by solving the visibility and form factor between all points in $P_l$ to all points in $P_e$, see figure

1. A large number of light rays has to be generated to reduce the visually disturbing variance, created by this stochastic method.

b. Photon Mapping

Photon mapping is considered superior both finite element radiosity and Monte Carlo bidirectional path tracing, because it is faster, conforms to arbitrary geometry, and produces fewer artifacts. Figure 2 demonstrates global illuminstation effects generated with photon mapping.

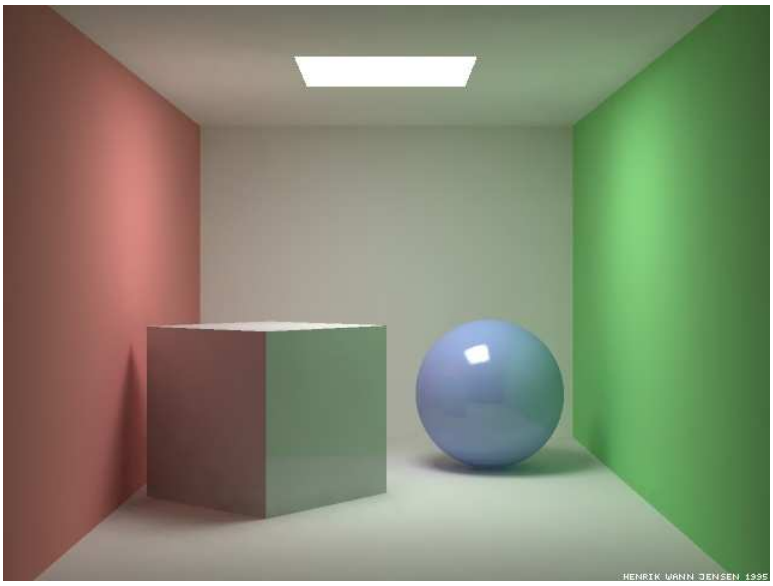


Fig. 2. Photon Mapping. Image Curtosy of H. W. Jensen, UCSD.

In the case of photon mapping all photon rays are emitted in one pass. The photons are reflected or refracted according to the reflectance function of the surfaces in the ray's path. The secondary rays are created via recursion from the primary rays. A photon is saved in the photon list when it encounters a Lambertian surface.

The photon is saved as three vectors: a position vector, a normal vector and a intensity vector. The photon mapping algorithm efficiently stores, searches and filters a 3-D photon space with a balanced Kd-tree data structure. A balanced Kd-tree is essentially a BSP-tree data-structure that use the photons' positions as axis-aligned partitioning points. The left and right subtrees of any node are equal or similar in weight, as illustrated in figure 3. This tree can be constructed in $O(n \log n)$ time and we can find k nearest photons around a point $p$ in $O(\log n + k)$ time.

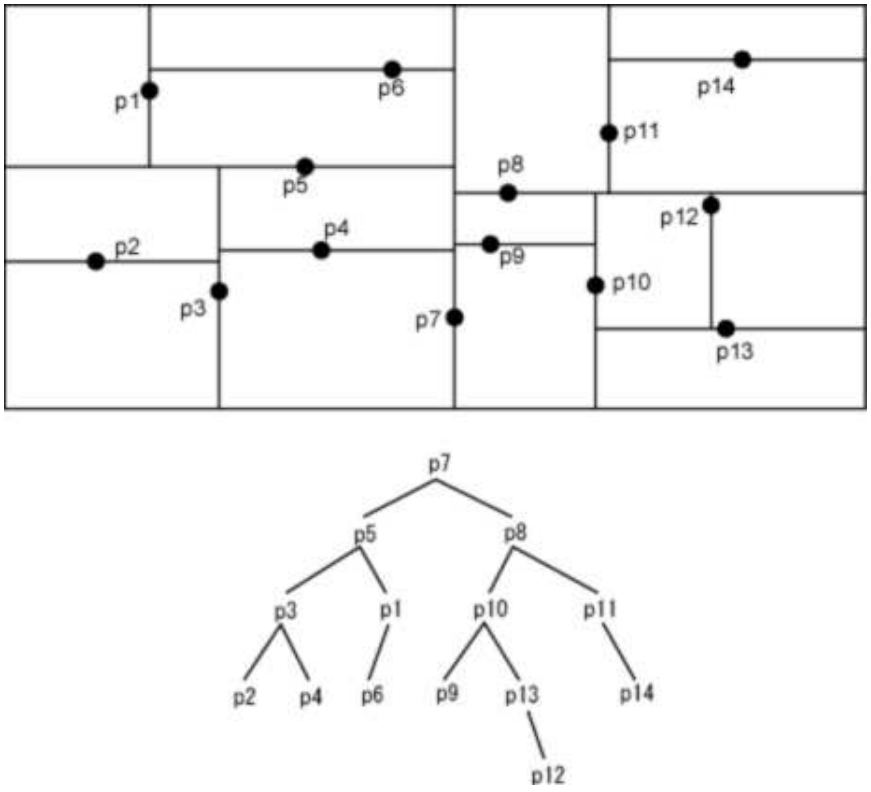


Fig. 3. Kd-tree Data Structure

## B. Hardware Accelerated Ray Tracing

In the past, clustered PC's, Supercomputers and off-line hardware accelerator systems have been developed for ray tracing. These systems are expensive to set up and to maintain. Their communication overhead is usually high and often require complex software coordination.

Recent work presented in [13] [14] [5] [15] demonstrated ray tracing with GPUs. Studies done by Carr et al [13] and Purcell et al [14] have demonstrated ray tracing on the fragment processor of modern GPUs. Saarland University reported in [5] [15] the design and simulation results of a ray tracing GPU, which rendered complex images at interactive frame rates.

## C. Hardware Accelerated Photon Mapping

Purcell et al. have implemented an uniform grid-base BSP tree structure for storing and tracing photons with the fragment processor of commodity GPUs [16] [17]. Limited by the current GPU architecture, this photon mapping implementation is fixed in size and in resolution. Only the fragment processor is used in this scheme and the vast majority of the hardware resources are untapped. Several publications [18] [19] also reported interactive rendering of only caustics with photon maps. In these studies general purpose photon mapping for full global illumination was not tackled. Ma and McCool [20] have proposed a hash table based low latency photon map in 2003, which may be used for hardware-based rendering. All reported hardware-based photon mapping methods to date are less versatile and scalable than the original Kd-tree implementation.

# CHAPTER III

## ALGORITHMS AND RENDERING PIPELINE

For hardware based rendering, the disadvantage of using current software-based algorithms for ray tracing and photon mapping is that it requires instruction sets not present in GPUs. In the near future, GPUs are not likely to support recursion operations and complex memory operations [17]. In fact, it may be counter productive to implement fully CPU-like general processors for graphics processing.

We propose new algorithmic approaches is different. We have designed a comprehensive rendering pipeline with both ray tracing and photon mapping in mind. We propose new sorting and traversal algorithms that promotes parallelism, data spatial locality and multi-stream processing. Similar axis aligned, balanced BSP trees are used for accelerating both operations. Our algorithms are more flexible than the previous hardware based algorithms. Like the balanced Kd-tree, our photon map is de-coupled from geometry. In addition, our photon mapping and ray tracing acceleration algorithms are highly optimized for the proposed hardware architecture.

### A. Photon Mapping

Recently, NVidia and ATI have proposed the architectural trends for their next generation GPUs [3] [2] [17]. Based on these standards, we impose a number of constraints while designing our pipelined architecture. Recursive calls and dynamic memory allocations are forbidden. The memory units are treated as arrays of numbers or vectors, whose sizes are determined at compile time. We allow limited indexing of arrays. However, dynamic and flexible allocation of pointers are not allowed. Random access reads with computed indexing are allowed but main memory can only be written in a sequential fashion.

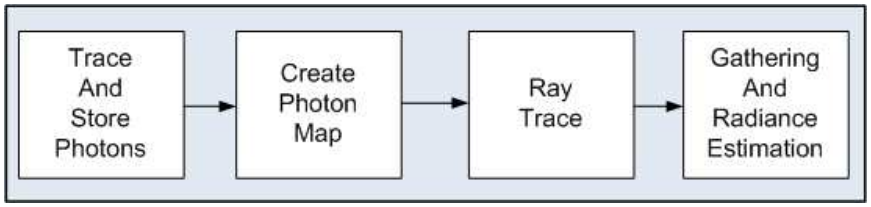


Fig. 4. Photon Mapping Pipeline.

Figure 4 illustrates the pipeline of our photon mapping algorithm. The first step in creating a photon map is backward ray tracing [10], i.e. tracing rays from a light source. When the path of the photon is obstructed by a Lambertian surface, we store that photon. Russian roulett method is used to eliminate photons depending on the absorbance of the surface and construct secondary reflected/refracted photon path arrays. When enough photons have been gathered we construct an acceleration data structure from the initial photon list, so that it can be searched efficiently. Third step is reflected radiance estimation. During this step we search for $m$ photons closest to point $P$, and estimate the reflected radiance of $P$. To better understand photon mapping the reader is referred to several excellent sources [1] [11].

### 1. Sorting Photonmap Algorithm

A balanced axis-aligned Binary Spatial Partitioning (BSP) tree scheme is utilized for our acceleration data structure. We balance the children of each node so that both children contain equal or similar number of photons. We create partitioning points based on density of the photon maps. The resulting BSP tree is not a grid-like structure. The advantage of having this partitioning structure is that: First, There is no empty voxels, hence there is also no need for hash-table based voxel traversal

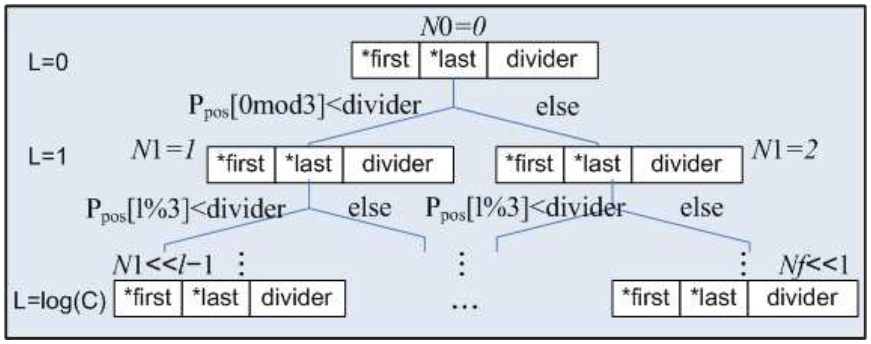


Fig. 5. Photon Map Tree Structure.

algorithm. Second, we guarantee a much more balanced tree structure with our partitioning scheme. Third, we do not require the redistribution of photon powers when a voxel is full. Fourth, a number of quality filtering methods can be applied cheaply to this data structure.

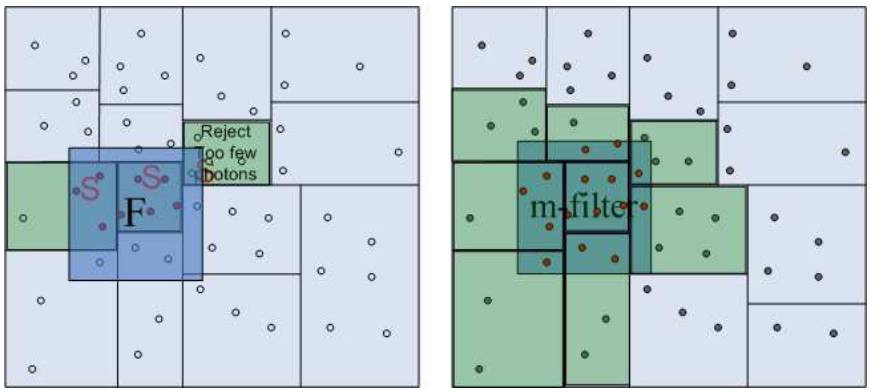


Fig. 6. Photon Map Partitioning.

We construct a photon tree (see figure 5) as a heap of nodes. Each node, $N_i$,

contains three 32-bit values, a memory index to the first photon of a voxel, $*P_\alpha$, a memory index to the last index of a voxel $*P_\omega$, and a floating point representation of the partitioning point dividing that voxel, $X_d$. At compile time, we allocate a set amount of main memory, $M$, for the partitioning node heap. The array contains $C$ leaf nodes, with the last node being $2C-1$. No complex pointer operation is required to index this data structure. For each parent node N with index $N_i$, its left child is indexed by $N_{i<<1+1}$ and its right child is indexed by $N_{i<<1+2}$. ($<<$ indicates left shift, $A << b$ indicates $A$ left shifts by $b$ radix points) The first node, $N_f$, on Level $l$ is indexed as $N_{1<<l-1}$, and the last node, $N_l$, at Level $l$ is indexed as $N_{f<<1}$. With these characteristics, we can compute indices very cheaply with single-cycle integer instructions with integer addition and shifting.

Figure 7 describes the photon map set up algorithm. To create a photon acceleration data structure we use a 2-pass sorting algorithm. The total sorting operation cost $O(n \log n)$ in total time. But since the data accessed in the memory is contiguous, there is virtually no penalty in terms of pipeline data hazards. There is also the added advantage of a non-recursive algorithm - the elimination of the recursion overhead.

The first pass is used to find the partitioning point(s). The partitioning point(s) along an $axis_{j \bmod 3}$ ideally is the median(s), $X'_d$, of photon positions along $axis_{j \bmod 3}$. However, finding the exact median is intractable. We look for the center of mass along $axis_{j \bmod 3}$ instead. All photons are given equal partitioning weights. The center of mass of each axis is the sum of all the photon positions along that axis divided by the number of photons. We find $2^l$ partitions at each level, $l$. The second pass is a storage pass. We store all photons, $P_l$, such that $P_l$ is less than partition point, $X_d$ in the left child. In the third pass, we store all Points $P_r$, such that $P_r$ is equal or greater than $X_d$ to the right child. The array at each level $i$ is exactly the same size as

the array size at level $i+1$. Photon lists P and P' are two static blocks of swappable memories of equal size created at compile time. Refer to Algorithm 1 for pseudocode of acceleration data structure setup algorithm. This operation takes $O(\log n)$ passes to complete. The additional division point creation pass does not increase overall complexity of the algorithm. Further more, division point creation pass allows the creation of balanced photon maps of arbitrary size. Like the balanced Kd-tree, our data structure is also independent of the scene geometry.

## 2. Sampling and Filtering

To find exact k-nearest photons in our data structure is intractable. Instead, we propose two general methods: volume filtering and stochastically reflected radiance sampling. One could also view the photon map's radiance estimation problem as the problem of creating an efficient low-pass filter as indicated by H W Jensen in [11]. The reflected radiance flux at point $x$ can be expressed with equation 3.1 [11]. $L_r$ is the reflected radiance term. $x$ is the point of intersection. $\vec{\omega}'$ denotes direction of reflected radiance. $\vec{\omega}$ is the direction of the incoming radiance. $f_r(x, \vec{\omega}', \vec{\omega})$ is the BRDF of the surface at $x$. $\Phi_i(x, \vec{\omega}')$ is flux at point $x$. $A$ stands for the area.

$$L_r(x, \vec{\omega}) = \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) \frac{d^2\Phi_i(x, \vec{\omega}')}{dA_i} \tag{3.1}$$

$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^{n} f_r(x, \vec{\omega}', \vec{\omega}) \frac{\Delta\Phi_p(x_p, \vec{\omega}'_p)}{\Delta A} \tag{3.2}$$

Equation 3.1 can be approximated in finite element terms with equation 3.2. This equation is the equivalent of placing a simple disk filter on a plane orthogonal to the normal of the surface at $x$. In this expression, $x_p$ are photon positions within the area of $\Delta A$. Substituting A with $\pi r^2$ results an usable rendering equation. This

filtering method works well when we can restrict $r$ to the maximum radius of k-photons. Because of the restrictions we placed on our instruction set, our algorithm has a static search radius, r imposed. Equation 3.3 applies a hyper-cone filter on the radiance estimation equation. $C$ is a scaling constant. This 4-dimensional expression enables the extraction of finer details than the disk filter. However, $n$ in equation 3.3 is not bounded.

$$L_r(x,\vec{\omega}) \approx C\sum_{p=1}^{n} f_r(x,\vec{\omega}',\vec{\omega})\frac{\Delta\Phi_p(x_p,\vec{\omega}_p')}{\Delta A}(1-\frac{|x_p-x|}{\Delta D}) \tag{3.3}$$

We can estimate reflectance using a stochastic sampling method 3.5, such that the number of photons searched for reflected radiance estimation is bounded. In this equation 3.5, $k$ represents the number of leaf node voxels we sample. Expression 3.4 is the radiance estimate of a single stochastically sampled leaf-voxel, we then apply the cone filter and estimate radiance for each voxel. The average radiance of the outer summation loop yields the estimated reflected radiance at point $x$.

$$\sum_{p=1}^{m} f_r(x,\vec{\omega}',\vec{\omega})\frac{\Delta\Phi_p(x_p,\vec{\omega}_p')}{\Delta A_s} \tag{3.4}$$

$$L_r(x,\vec{\omega}) \approx \frac{C}{k}\sum_{s=1}^{k}\sum_{p=1}^{m} f_r(x,\vec{\omega}',\vec{\omega})\frac{\Delta\Phi_p(x_p,\vec{\omega}_p')}{\Delta A_s}(1-\frac{|x_p-x|}{\Delta D_s}) \tag{3.5}$$

### 3. Tracing Photon Map Algorithm

Figure 8 illustrates an iterative algorithm that implements equation 3.3. We refer to this algorithm as the m-voxel filter. It has the complexity of $O(m\log n)$, with $m$ denoting all the photons with in the search radius $r$ and $n$ denoting all the photons inside the photon map. $m$ is not bounded, therefore it cannot be reduced to a constant. Under most circumstances, $m$ is only a minute fraction of $n$. It is also

less expensive to construct a cubic filter than a spherical filter. In a cubic filter only 1-axis has to be tested per pass.

We can use a stochastic sampling algorithm to sample $k$ voxels. In this method, we randomly generate $k-1$-points $x_{k-1}$ around point $x_0$. Also let $x_i$, s.t. $i \in \{0, k-1\}$, be orthogonal to the normal at $x_0$ to increase sampling efficiency. All sample point set $x_i$ are within radius $r$ from point $x_0$. $x_k$ is passed from a similar algorithm to the m-filter algorithm. Only $k$-voxels are searched. Reflected radiance is estimated for each voxel within the set $x_k$. We take the average reflect radiance as the overall value. This algorithm corresponds to equation 3.5. Because we have a constant number of k-voxels, and a constant number of $C$ photons within each balanced voxel node, the complexity of this algorithm is $O(\log n)$.

## B. Ray Tracing Pipeline

We use a multi-pass rendering pipeline similar to that employed by *Renderman*. Multiple rendering passes create images with different effects that are super-imposed to create the final image. In the case of simple ray casting, we have only two steps. One, acceleration data structure setup. Two, ray trace and acceleration data traversal.

We construct an acceleration data structure similar to that used for photon mapping. The geometric data is partitioned in a balanced axis-aligned BSP tree. This gives us equal traversal time to every node in the structure, and similar number of triangles within each leaf nodes. Shadow, reflection and refraction frames are rendered in multiple passes. At the end of the ray-cast operation, we store the intersection points, and normals. Depending on the user-specified surface discreption, shadow, reflection and refraction masking maps can be generated. The shadow, reflective and refactive (indirect illumination) rays are then created and traversed in seperate passes

through the acceleration data structure. These seperate frames are summed for the final rendering output. Figure 9 illustrates this iterative pipeline.

### 1. Geometry Partitioning Algorithm

Our balanced axis-aligned BSP tree, is created via iterative processes. As in the photon map sorting algorithm, this is essentially the same algorithm as algorithm 1. However, we may have triangles, which occupy multiple voxels. An additional expansion pass is added to the photon sorting algorithm. The expansion pass copies the same triangle into 2 child nodes. This expansion of geometry increases the number of triangles in the final data structure. Therefore, at compile time we allocate twenty times the memory of the original triangle list in the main memory. Only the position vectors of each vertex are duplicated. This duplication may seem expensive, however the main memory is relatively inexpensive. It is extremely rare to have models with over a quarter million triangles in any interactive applications. About 120MB of memory is required for storing a accelerated data structure whose original size is 250,000 triangles. Currently, the cost of commodity DDR SDRAM is $\$150 per GB$. The cost of 120MB of main memory is trivial compared to averall cost of a GPU.

### 2. Ray Tracing Algorithm, Traversal

There are 3-directional bits are associated with each ray $R$. Bit-0 corresponds to the direction $D$ of ray $R$ in the x-direction. Bit-1 corresponds to $D$ in the y-direction, etc. An array of leaf-voxels, through which R passes, is constructed. The algorithm is similar to algorithm 2. The directional bits direct the sorted voxel list such that the ray always traverses the voxel nearer to the origin first. Once a minimum intersection point is found for a ray, the current ray traversal loop is terminated and a new traversal list is loaded.

It is trivial to modify algorithms 1 and 2 to support triangle sorting and searching. For brevity, the detailed algorithmic descriptions are not repeated here.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{3.6}$$

C. Ray-Triangle Intersector

Because triangles are the the most commonly used primitives for interactive applications, it is the only geometric primitives supported in our design. There are a number of different methods for ray triangle intersection tests. Ideally, our ray-triangle intersection testing algorithm is one that does not have any conditional branches and answers either "yes" or "no" at the end of the computation. For this purpose we found that the Möller's ray-triangle intersection test [21] to be the most suitable (equation 3.6). In this equation, $V_0$, $V_1$ and $V_2$ are vertices of a triangle, $D$ is the vector of a ray, $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, $T = O - V_0$, $P = (D \times E_2)$ and $Q = (T \times E_1)$. For further details please refer to [21]. We construct an eleven stage pipelined intersection tester based on this algorithm. A status word is passed along with the triangle, and records of the address of the triangle and whether or not that triangle is intersectable by the ray.

input unsorted photon list $P[photonCount]$
this node index $k \leftarrow 0$
**for all** levels $j \in \{0, L-1\}$ **do**
  axis of division $d \leftarrow j \bmod 3$
  First node in level $j+1 \rightarrow kk$, s.t. $(kk \leftarrow (k \ll 1) + 1)$
  **for all** nodes $N \in \{k, kk-1\}$ **do**
    partition point $X_d \leftarrow 0$
    **for all** photons $P[i]$, $i \in \{*N_{first}, *N_{last}\}$ **do**
      center of mass $C_d[p] = \sum \frac{P[i]_d}{p_{last} - p_{first}}$
    **end for**
  **end for**
  output photon map P' index, $ii \leftarrow 0$
  **for all** nodes $N \in \{k, kk-1\}$ **do**
    left child node index $Nl_{*first} \leftarrow N \ll 1 + 1$
    **for all** photons $P[i]$, $i \in \{*N_{first}, *N_{last}\}$ **do**
      **if** $P[i]_d < C_d[p]$ **then**
        $P'[ii_{++}] \leftarrow P[i]$
      **end if**
    **end for**
    $Nl_{*last} \leftarrow ii$;
    right child node index $Nr_{*first} \longleftarrow N \ll 1 + 2$
    **for all** photons $P[i]$, $i \in \{*N_{first}, *N_{last}\}$ **do**
      **if** $P[i]_d \geq C_d[p]$ **then**
        $P'[ii_{++}] \leftarrow P[i]$
      **end if**
    **end for**
    $*Nr_{last} \leftarrow ii$;
  **end for**
  swap $P \Leftrightarrow P'$
**end for**

Fig. 7. Photon Map Setup

Node Count $C_0 \leftarrow 1$
Node list $N_0 \leftarrow \emptyset$ $N_1 \leftarrow \emptyset$
**for all** levels $j \in \{0, L-1\}$ **do**
  axis of division $d \leftarrow j \bmod 3$
  **for all** nodes $N_i$ s.t. $i \in \{0, C_0\}$ **do**
    **if** Division Point, $D_{n0} >$ bbox minima, $BB_{min}$ **then**
      $N_1 \cup (N_0[i] \ll 1) + 1$
      $C_1 \leftarrow C_1 + 1$
    **end if**
    **if** $D_{n0} < BB_{max}$ **then**
      $N_1 \cup (N_0[i] \ll 1) + 2$
      $C_1 \leftarrow C_1 + 1$
    **end if**
  **end for**
  $N_0 \leftarrow N_1$ $N_1 \leftarrow \emptyset$
  $C_0 \leftarrow C_1$ $C_1 \leftarrow 0$
**end for**
**for all** nodes $N_0[0 \rightarrow C_0]$ **do**
  apply filter on $P_0 \leftarrow N_0$
**end for**

Fig. 8. Photon Map Search and Filtering Algorithm

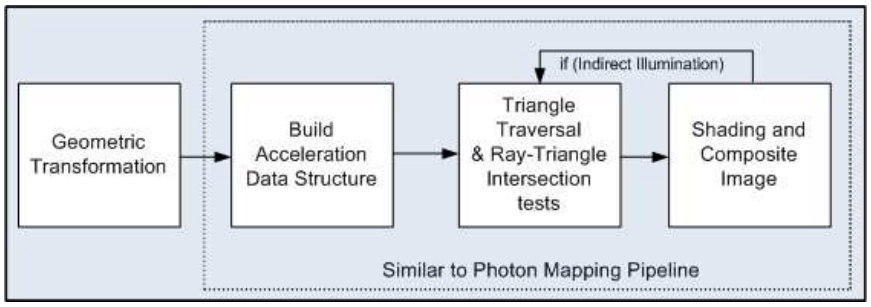


Fig. 9. Ray Tracing Pipeline.

# CHAPTER IV

## SYSTEM ARCHITECTURE

We classify the proposed system as a Multiple Instruction Multiple Data (MIMD) application specific system. Our system can also be abstracted as a fine-grained, multi-nodal stream processing system, with multiple Steaming Processing Elements (SPE) computing different stages of a data stream.

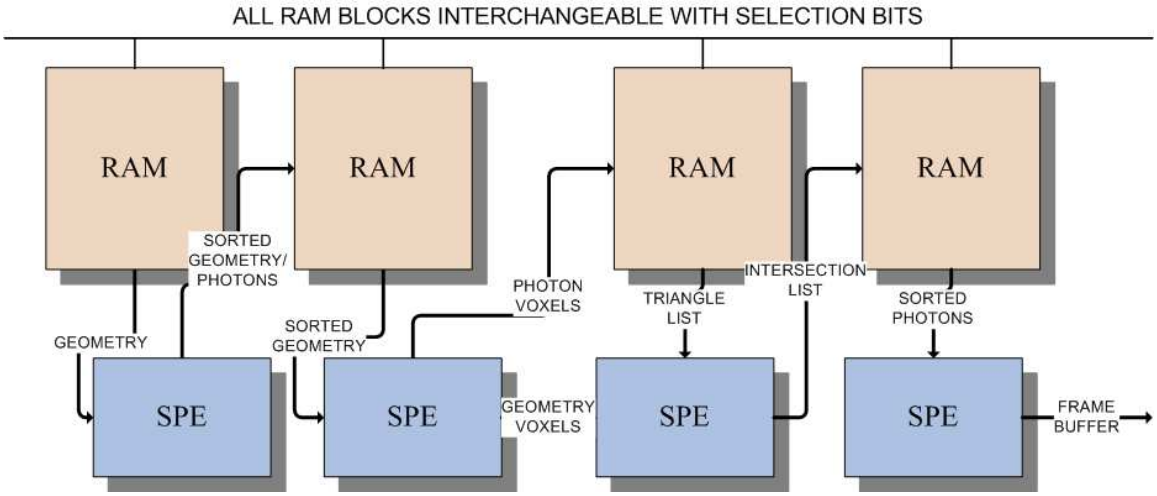


Fig. 10. Multi-Stream Processor Architecture.

Figure 10 shows top-level abstraction of the data stream passing through each of the Streaming Processing Elements (SPE). These abstracted units process local data and pass the results to the next SPE back-to-back in a chain. This processing arrangement is highly effecient because computation is isolated and communications between SPEs are highly predictable.

It is worth noting that the photon mapping and ray tracing share much of the

same data paths. The rendering pipeline's behavior is similar to a normal hardware pipeline. The difference is that the state transitions are asynchronous and the lower-priority stream sequences can be preempted by a higher priority sequence.

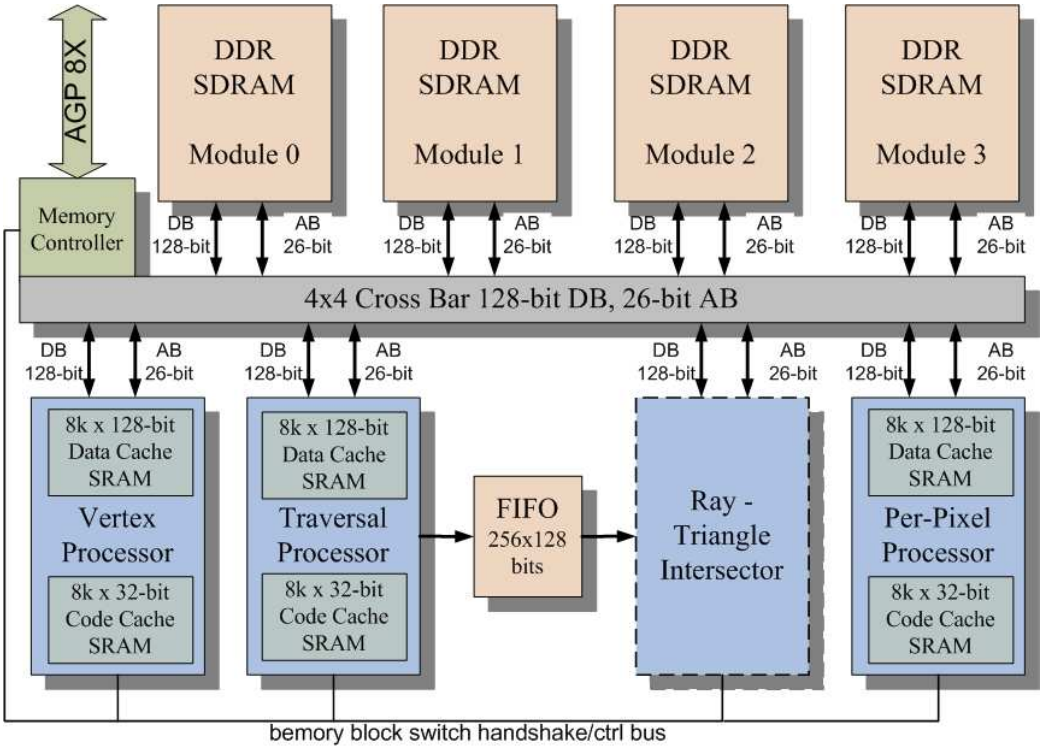


Fig. 11. The Top-Level System Architecture.

## A. Processing Modules

Figure 11 illustrates the top level design based on this paradigm. Each SPE is implemented as a Processing Module (PM) in the proposed architecture. It consists of four PMs: a Vertex PM, a Traversal PM, a Ray-Triangle Intersection PM and a Fragment PM.

### 1. Programmable PMs

The Vertex PM, Traversal PM and Fragment PM are programmable. Vertex and Fragment Processors share essentially the same instruction set and design. The Traversal Processor is slightly different in that it has a direct data path from it to the ray-triangle intersector. This data path allows the Traversal PM to pass memory addresses of triangles directly to the ray-triangle intersection processor via a FIFO. With the exception of the ray triangle intersection PM, all PMs are essentially programmable 4D vector processors. The instruction set share similar requirement to new programmable GPUs from NVIDIA and ATI [16].

Cheap integer arithmetic operations, nested looping and data dependant branching are implemented in the instruction set. We also extend the random-access reads to all three programmable PMs. These features make computed indexing of memory possible. These programmable PMs can not perform random writes because our algorithms do not require such operations.

### 2. Processing Module Caching

Vertex and Fragment Processors contain two banks of cache memory each, the instruction cache and the data cache. In proposed simulation model, we set the data cache for all three programmable processors to blocks the size of 8K×128-bits. We set the instruction cache to an arbitrary number, because it is bounded by the complexity of the application. The Traversal processor also contains a special FIFO bank that is directly connected to the Ray-Triangle Intersection Unit.

We use duel-port SRAM for data cache on the programmable PMs. The duel port RAM allows simultaneous paging and local access. The processor pre-fetches the next memory block on the main memory if the index of the current operation $M_i$

and index of first address of current page $M_i'$ are such that $M_i > M_i' + \frac{BlockSize}{2}$. This happens while the memory from the last page is still being processed. Therefore, each PM can page the Main Memory and execute instructions on its cache simultaneously. Eight blocks of data cache memory are available to each of the Vertex, Traversal and Fragment Processing Modules at any time. They work in pairs: cache block $0 \cup 1, 2 \cup 3...6 \cup 7$. Each pair contains a current memory page and a pre-fetched memory page. The 4-cache block pairs allow multiple points of the main memory to be accessed without stalls. All cache blocks are $1024 \times 128$ - bits in size.

### 3. Ray-Triangle Intersection PM

Ray-Triangle Intersection PM is the only Fixed, non-programmable PM. It is shown in figure 12 as an eleven–stage, pipelined processor. The Po, and D vector is loaded in the PM in two-clock cycles at initialization. In regular mode, one vertex (96-bits) is loaded at a time. Memory management Finite State Machine (FSM) load the three vertices of a triangle in three clock cycles. Each pipelined stage, therefore, takes three clock-cycles to complete. The FIFO, which is filled from the traversal processor with the index pointers vertices controls withch triangles are loaded. A extra 32-bit header is associated with each vertex. The header contains the index for the original normal and texture coordinates associated with each vertex. The PM stores the minimal intersection distance $s$ for all intersected triangles $T_i$ in the intersection list.

## B. Memory Coherence

The proposed architecture is similar to a barrel-shifter at the top level. Each PM is associated with a main-memory module. Every memory module is identical in size so that every PM address every memory module in the same way. Each PM sees

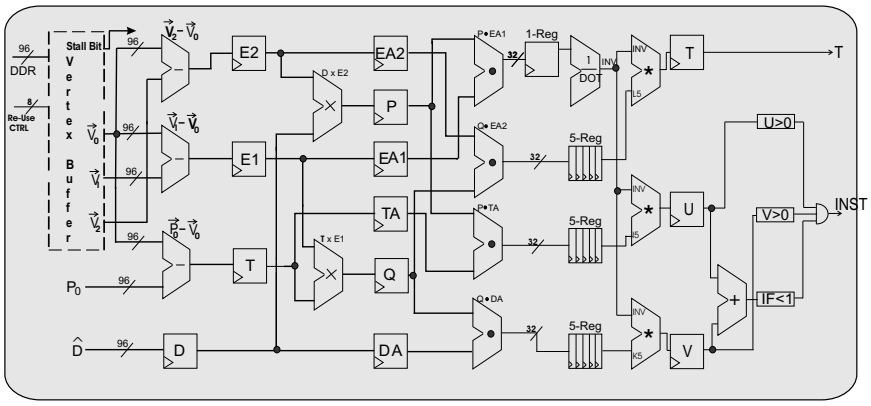


Fig. 12. Ray Triangle Intersection Processor.

only its "partner" memory module via the cross bar. When it is finished, it releases that memory module and requests the next memory module for processing. The arbitration is done with the memory controller, which controls a cross bar. When the PMs request different frames, this top level memory control unit "swap" or "shifts" the memory modules by re-routing the top-two address bits. It can also copy entire blocks of data from one memory module to another.

We model our memory access scheme after commodity DDR Double Data Rate Synchronous Dynamic Random Access Memory (SDRAM). It is relatively inexpensive and it has good burst performance. However, even at 600 MHz access rate, this SDRAM is still the bottle neck for the rendering pipeline. DDR memory also incurs heavier penalties for a page misses than traditional SDRAM. The sequential access of the proposed algorithms nullifies this shortcoming.

The four main memory modules operate in pairs. Memory modules 0&1 contain data associated with image 0. Memory modules 2&3 contain the data stream for image 1. Rendering process of that image is able to interrupt processes of the low-

priority image. If interrupted, the PM finishes the ray/pixel operation it is working on before context switching to the higher priority process. The rendering stream terminates once the final image is produced. Memory controller assigns one of the two images as the higher priority image according to the time of its arrival. The priority of lower priority process is increased by the memory controller when the raw data for the next frame is accepted.

## C. Flow Control

The overall program flow control is directed by the Fragment PM and the control unit. The related memory modules are linked back to the Vertex PM for additional processing if additional rendering passes are required. This determination is given by the assembly instructions written to the Fragment PM. Each rendering pass is similar to the ray casting pass, which we demonstrate in figure 13. The final output image is usually the composite of many seperate rendering passes.

Figure 13 is a state diagram that illustrates the state transitions of a normal ray casting rendering pass. When each processor module is finished with its current frame, it releases the memory module back to the top-level controller. Figure 14 illustrates this process for a simple photon mapping loop. The system has a 1-to-1 match of all memory modules and PMs at any time.

There exists a high degree of regularity within the proposed architecture. 5-stage pipelined instruction architecture are uniformly applied to all programmable PMs. This feature allows the design of each programmable PM to be nearly identical to each other. Hardware granularity is an advantage gained through this design. Further more, each PM and memory module is isolated from one another. Clocking and signal skews between modules are reduced as long as consistency is maintained

locally between a PMs and its current Memory Module.

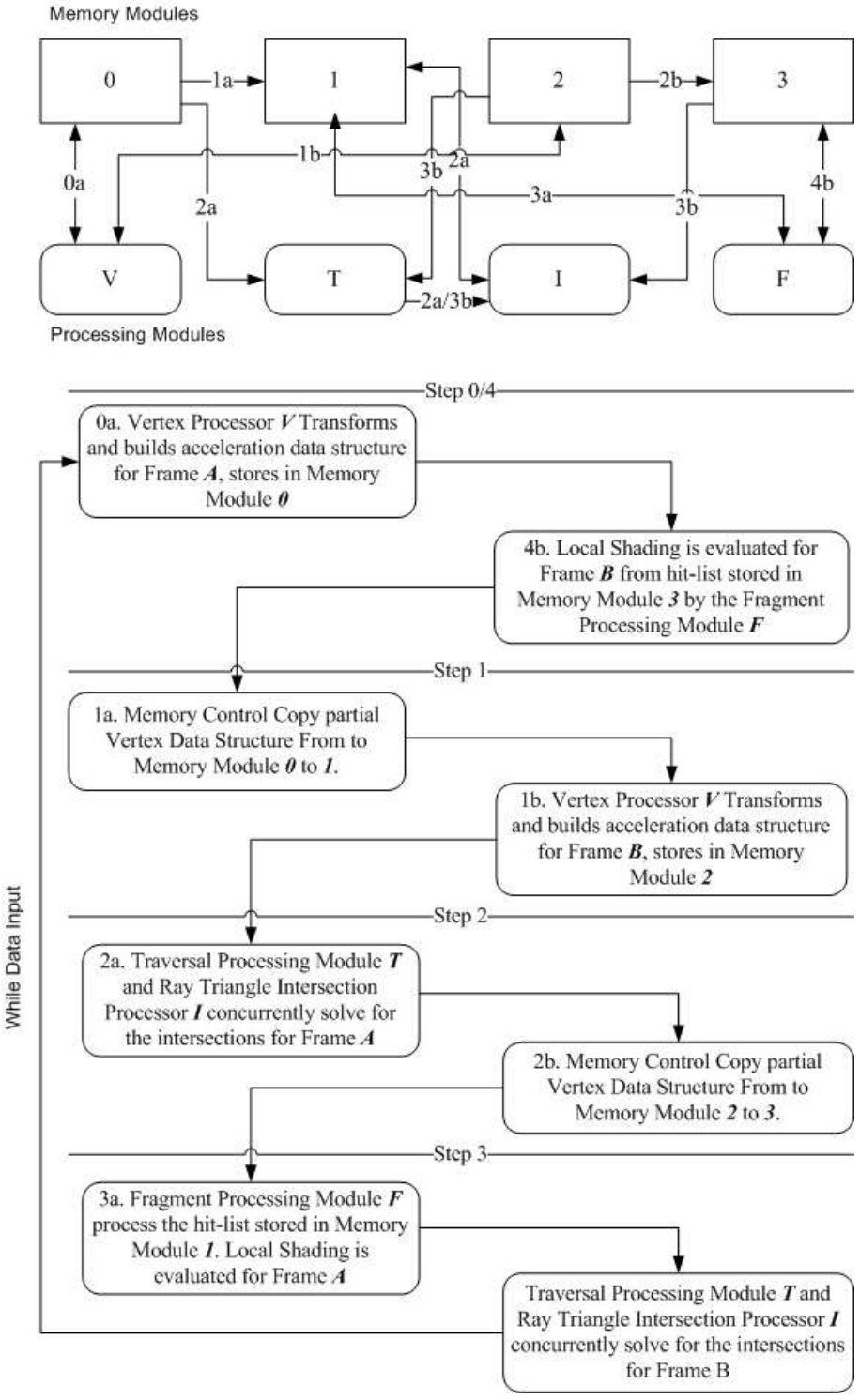


Fig. 13. Ray Casting State Diagram.

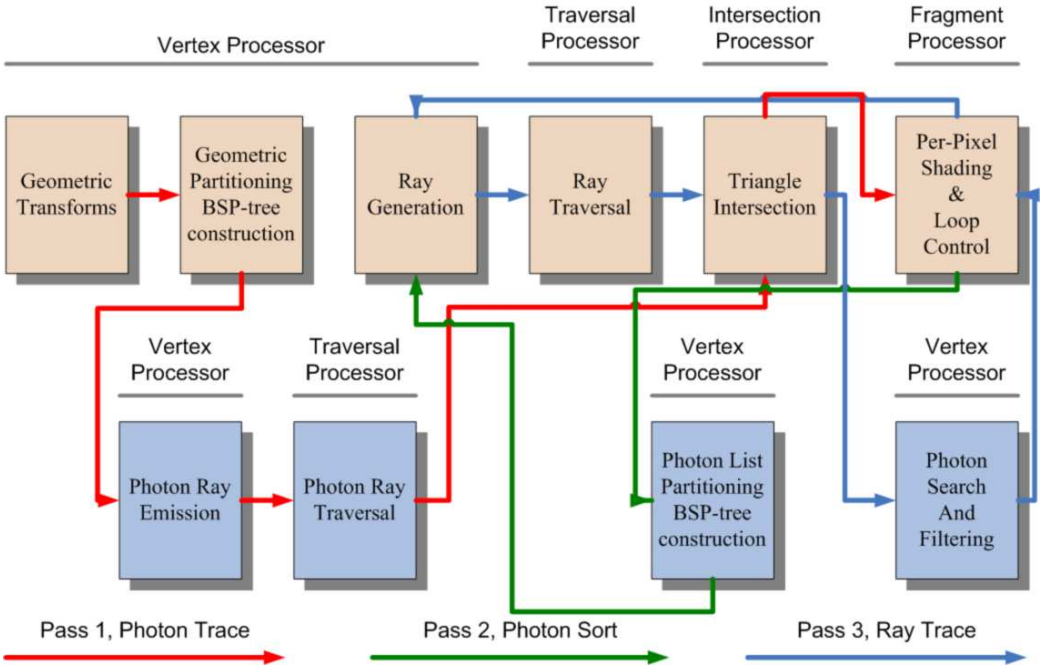


Fig. 14. Photon Mapping Flow Diagram.

# CHAPTER V

# RESULTS AND ANALYSIS

We report and analyse the experimental setup and benchmarks, pipeline throughput, latency, image quality, and filtering in this section.

## A. Experimental Setup and Benchmarks

A simulator for this architecture is written in C++. The simulator contains four primary processing kernels, corresponding to each PM. Top-level data streams are pushed through these kernels to simulate system performance. With this method, accurate per-clock cycle delays and overall throughput is measured. We also record the overhead of copying data from one memory bank to the other and context switching. Detailed rendering data are recorded by the simulation engine. We report relavent results pertaining to the throughput of our pipeline and its latency in this chapter. This simulator also allows the finetuning of algorithms to achieve optimal load balance.

In the first experiment, we render three bench mark images with direct illuminations and shadows. In the subsequent renderings, we apply reflection, refraction and photon mapping. The results of these trial runs are shown in Figure 15, 16 and 17 for these test cases.

To convert our clock-cycle based results to frames-per-second benchmark, a core clock frequency is assigned. Main memory access time is assumed as the critical path in the system. Many new commodity GPUs are connected to DDR SDRAM modules rated at speeds greater than or equal to 600-Mhz. Therefore, we model our main system frequency at 600-Mhz. Per Processing Module timing is computed using the following equation for each of the processors.

$$T_{total} = \sum T_{norm} + \sum T_{ms} + \sum T_{ps} + \sum T_{ds} \tag{5.1}$$

The simulator accounts for the timing cost separately in each PM kernel. When the individual PMs execute instructions from their local data cache, they are capable of exicuting one pipelined, 4-D vector instruction, $T_{norm}$, per-system clock cycle. However, if there is a page-miss memory access then there is a penalty in terms of the Row and Column access latencies, $T_{ms}$. We model this delay to be 12 system clock cycles. In the case of a conditional branch there is a chance for a 4-cycle pipeline flush penalty, $T_{ps}$. We estimate the per-processor timing by counting the number system clock cycles on each processor. The overall timing also accounts for the data stalls, $T_{ds}$ caused by memory swaps, data copies between memory modules, and Processing Module handshakes. These handshake stalls exist because one PM requires the data processed by another PM to start its job. These handshakes stalls are artificially introduced by semaphores within the memory control module that prevents the commencement of any process before the necessary data arrives. Each kernel measures its own performance and the top module keeps track of the overall throughput of the GPU.

## B. Results and System Performance

To further analyse the system performance of proposed architecture, we compare our results with other contemporary work. Schmittler et al. reported in [5], that SaarCOR's ray tracing performance at frame rates of 7.52 fps to 28.88 fps with a single ray tracing core (RTC). While the throughput of the SaarCOR is impressive, with a fixed rendering pipeline, this architecture is not designed for photon mapping. It is also less flexible in terms of its shading options. Purcell et al's commodity

Table I. Scene With Ring Statistics.

646 triangles, $320 \times 240$ Pixels, $20,000$ photons

| type | RCast | Reflect | Refract | PMap |
|---|---|---|---|---|
| TVert | .378 | .378 | .378 | 1.34 |
| TTrav | 24.4 | 31.6 | 68.6 | 64.4 |
| Tisct | 10.8 | 15.4 | 26.1 | 27.8 |
| TFrag | 4.12 | 4.42 | 6.28 | 16.9 |
| TThru | 24.8 | 32.0 | 68.9 | 64.9 |
| FPS | 24.2fps | 18.8fps | 8.71fps | 9.24fps |

GPU based stream ray tracing studies reported throughput of 1.8 fps to 10.5 fps rendering simple ray-casting images without indirect illumination effects. Photon mapping without accelerated ray tracing takes as long as 8.1 seconds for the Ring scene and 64.3 seconds per frame [16] for the Cornell Box scene similar to ours [fig 17]. This is primarily due to the fact that most rasterization GPU resources can not be utilized by the process stream.

Ray-traced scenes are rendered at frame rates as high as 24 fps with our architecture. This performance is comparable to the SaarCOR's ray tracing performance. Photon mapped images are rendered at near interactive frame rates. Photon Mapping performance of our GPU ranges from 3.037 fps for the room scene to 9.24 fps for the ring scene. This is very close to interactive frame rates and on average above ten-times faster than the commodity GPU-based rendering time. Details are show in

Table II. Room Scene Statistics.

7812 triangles, $320 \times 240$ Pixels, $80,000$ photons

| type | RayCast | Reflect | Refract | PMap |
|---|---|---|---|---|
| TVert | 4.57 | 4.57 | 4.57 | 8.47 |
| TTrav | 41.2 | 43.8 | 91.4 | 113 |
| Tisct | 47.7 | 51.9 | 110 | 123 |
| TFrag | 5.96 | 6.28 | 79.0 | 178 |
| TThru | 47.9 | 52.1 | 110.4 | 178 |
| FPS | 12.5fps | 11.5fps | 5.43fps | 3.037fps |

figures 15, 16 and 17 and table I, II and III.

We determined empirically that the load between different PMs is not entirely balanced. However, we are able balance the load of Traversal and Ray-triangle intersection tester by altering the depth of our BSP-tree. We discovered that $\lfloor \ln n \rfloor$ levels, with $n$ denoting the number of photons and/or triangles, yields good performance balance for both ray tracing and photon mapping.

## C. Quality of Images

Tables 1-3 lists the performance of proposed rendering system. Each column from left to right represents the following: images with ray casting and shadows (RayCast), images with added reflection pass (Reflect), images with the addition of both reflection and refraction (Refrat), and images with direct illumination, shadow reflection,

Table III. Cornell Box Scene Statistics.

2604 triangles, $320 \times 320$ Pixels, $120,000$ photons

| type | RCast | Reflect | Refract | PMap |
|---|---|---|---|---|
| TVert | 1.52 | 1.52 | 1.52 | 8.73 |
| TTrav | 42.7 | 46.3 | 75.8 | 107 |
| Tisct | 37.9 | 39.6 | 64.3 | 102 |
| TFrag | 8.14 | 8.49 | 10.3 | 210 |
| TThru | 43.0 | 46.7 | 76.1 | 210 |
| FPS | 14.0fps | 12.9fps | 7.88fps | 2.86fps |

refraction and photon mapping (PMap). System performances are measured in Millions of Clock Cycles. TVert is the processing time of the Vertex processor. TTrav = time of the Traversal Processor, Tisct = time of the intersect process. TFrag = Fragment processor time. Images are rendered at throughput rate of TThru. The bottom row is the rendering throughput in terms of number of frames per second. The four images in each set illustrate the difference between ray-casting with shadows, reflection, refraction with Reflection and Photon Mapped Global Illumination.

Regarding photon mapping filtering methods, the stochastic sampling filter has a heavier noise level than the m-filter. However, the m-filter algorithm is not bounded in time, i.e. the worst search time for $m$ photons within radius $R$ of the sampling point $x$, can not be reduced to a constant. However the stochastic sampling filter time is bounded. If K-random photons are found within the radius R, we simply terminate

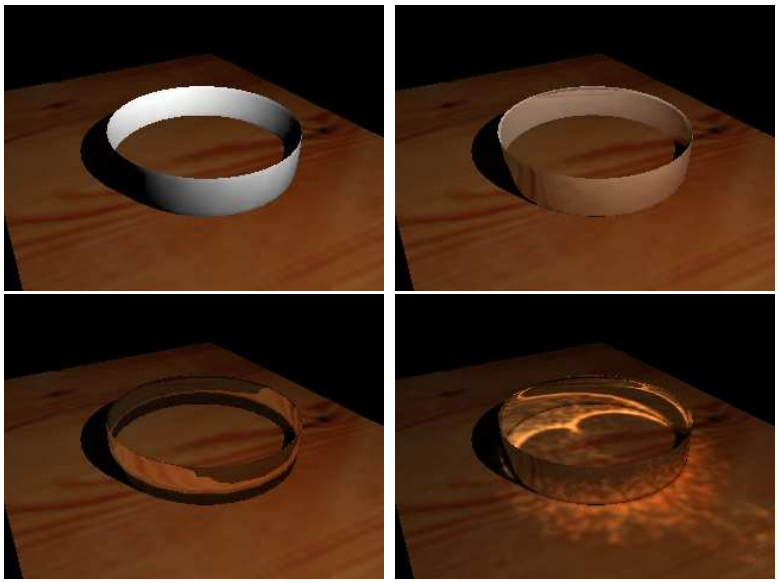

Fig. 15. Scene With Ring. 646 triangles, $320 \times 240$ Pixels, $20,000$ photons.

the search sequence. We discovered that the m-filter algorithm is efficient for rendering caustics. The k-voxel stochastic sampling algorithm is faster for rendering defused inter-reflection, with minimal increase of noise. We discovered that when rendering with photon mapping, the gathering and filtering step is usually the critical path. In the ring scene the traversal stage is costlier because many photon and eye-ray paths do not yield any intersection.

## D. Parallelism and Scalability

The entire rendering architecture is scaled-up cheaply by adding additional PMs. The only scaling overhead is the additional memory frames associated with each duplicated PM as well as the routing complexity and fan-out of the memory cross-bar. The Traversal, Ray-triangle intersection testing and Fragment processing pipeline

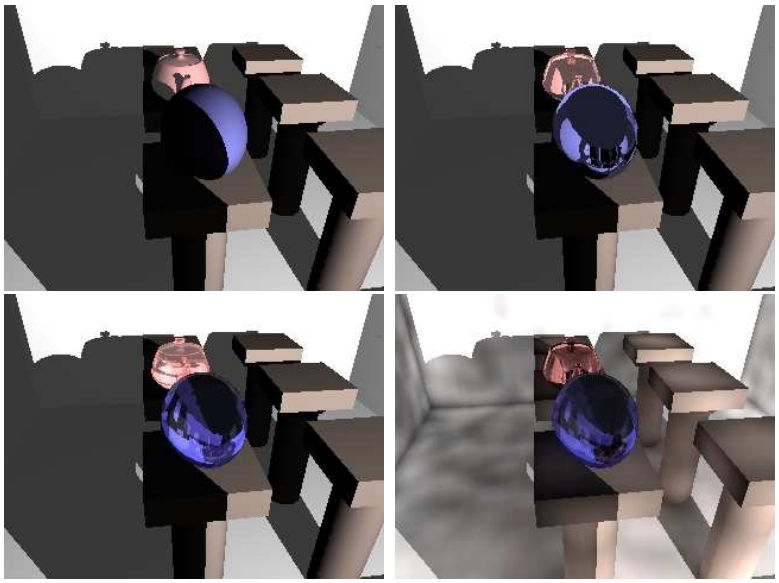

Fig. 16. Room Scene. 7812 triangles, $320 \times 240$ Pixels, $80,000$ photons.

through-put scales non-linearly with each additional parallel streaming pipeline we install. The overhead of creating the common acceleration data structure, memory management and routing is constant. However, since each of the programmable processing nodes are nearly identical, each PM is programmed to work on another PM's job when it is idle. This balances the load and nullifies the cost of the acceleration data structure construction. This however, requires dynamic load-balancing control scheme that is highly non-trivial and beyond the scope our current architecture.

### 1. Analytical Model for Multi-Pipeline Streams

Analytical models have been created to analyse the system performance of different parallelization schemes. We parallelize our rendering system by duplicating the entire rendering pipeline. We partition the image into $q$ quadrants. Each pipeline render

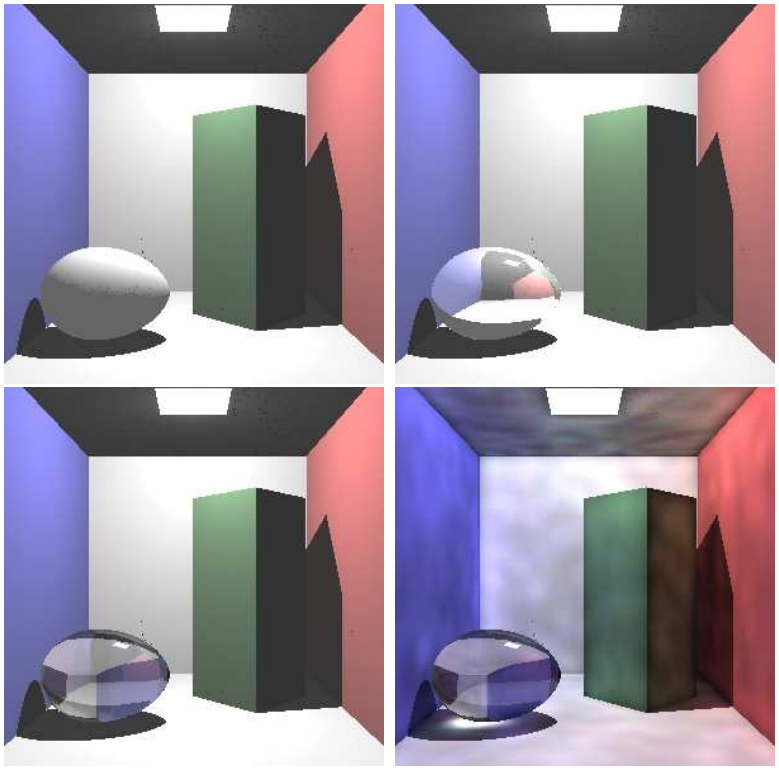

Fig. 17. Cornell Box Scene. 2604 triangles, $320 \times 320$ Pixels, $120,000$ photons.

a quadrant $Q_i$, $i \in \{1, q\}$, at a time to compose the final image. The initial geometry/photon list are partitioned via a single Vertex PM. All of other operations can be parallelized on a per-quadrant basis. Identical memory block can be copied to each pipeline. This would also require a simple top level control unit to control the final composition of the image. However, non-trivial algorithms have been designed by previous studies for cluster rendering systems that duplicates acceleration data structures on a "need-to-know" basis. These schemes can be applied to reduce load of each rendering pipeline.

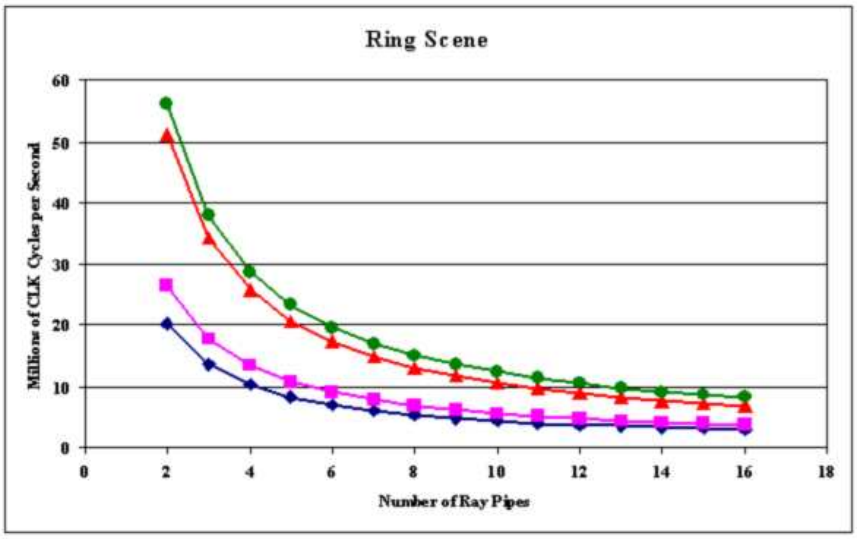


Fig. 18. Multi Pipeline Rendering, Ring Scene.

$$T_{tot} \approx A \times T_{vet0} + \frac{\sum_{i=1}^{N} B \times \max(T_{ti}, T_{ri}, T_{fi})}{N} \tag{5.2}$$

Equation 5.2 is an estimation of speed-up for the rendering process with additional processing pipelines. $T_{tot}$ is the total time. $A, B$ are constants, such that, if $N = 1$, $T_{tot}$ = acquired simulation time. $T_t$ denotes the timing cost of Traversal PM. $T_r$ denotes ray triangle intersection cost. $T_f$ is the Fragment processor time. $N$ denotes the number of parallel rendering pipes. The results of these rendering times are illustrated in figure 20 in terms of clock cycles.

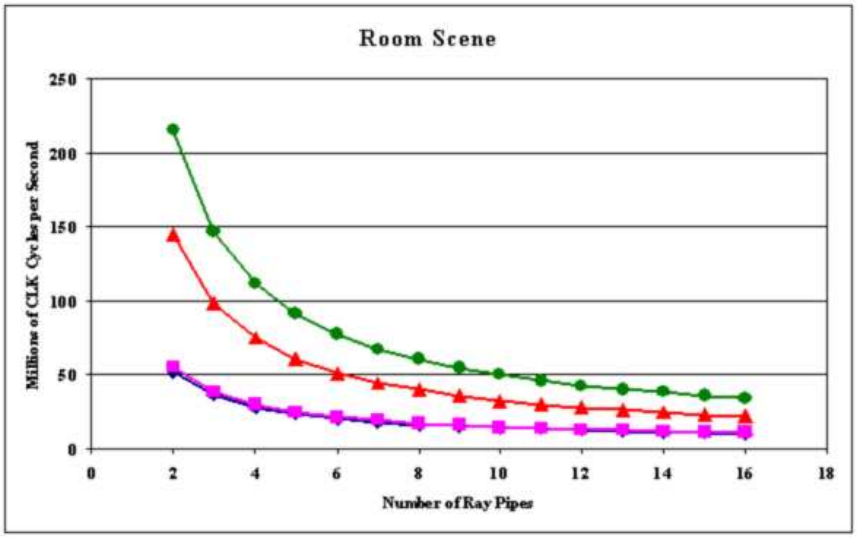


Fig. 19. Multi Pipeline Rendering, Room Scene.

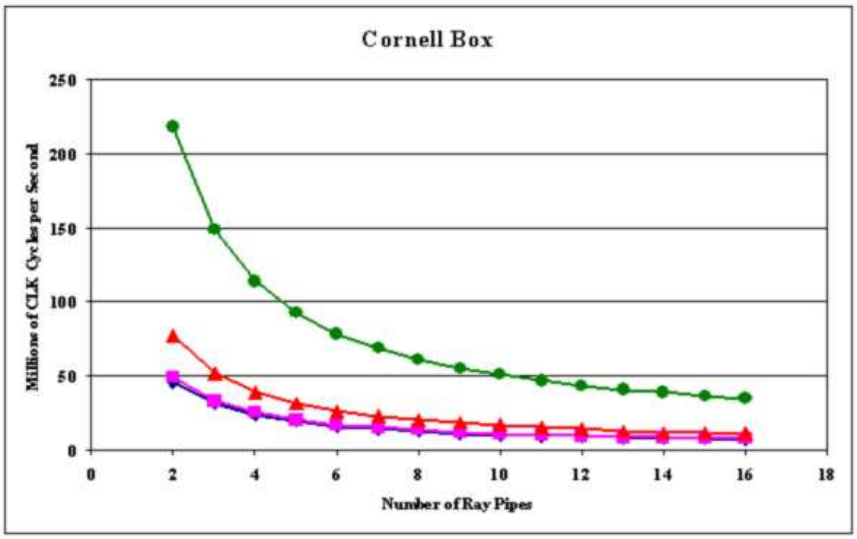


Fig. 20. Multi Pipeline Rendering, Cornell-box Scene.

# CHAPTER VI

## CONCLUSION AND FUTURE WORK

In this paper we presented a new multi-nodal streaming processing paradigm and architecture for ray tracing and photon mapping. Algorithms are designed to create and exploit coherence within the spatial locality of the acceleration data structure. These algorithms form the rendering pipeline that is mapped to our multi-stream architecture. In simulation, the proposed architecture has demonstrated interactive rendering with ray-tracing and near-interactive rendering with photonmapping.

This new architecture also has broader applications in computer science. As the speed gap between Computational Units and Memory Units continues to increase, the parallel stream processor's ability to exploit memory bandwidth is going to be increasingly advantageous. Many graphics applications rely on tree data structures to accelerate their algorithms. Most of these data structures are currently created and traversed recursively. These algorithms can be converted into iterative algorithms using pointer-less data structures similar to our's and benefit from similar hardware architectures. Further more, ray tracing a complex scene is relevant beyond the application of image synthesis. It is particularly useful in collision detection in physically based simulations. The same acceleration data structure can be used for both rendering and physically based simulations.

With only four processing nodes in our design, we pre-schedule our kernels statically with a pre-defined, application specific rendering pipeline. There are inherent inefficiencies in this scheduling scheme. Experiments have show that the proposed architecture suffer as much as 50% PM idle time due to data starvation. A dynamic scheduler can be implemented within the control module and may be considered as a future research topic.

The transition from rasterization GPUs to ray tracing GPUs would likely be a gradual one. Issues of downward compatibility in drivers and Application Programming Interfaces (API) will not allow an abrupt conversion. Our architecture is ideal for this gradual mode of conversion. Rasterization hardware could remain vestigially in parallel with our Traversal processor and Ray-triangle tester. The rasterization unit can process the ray-casting portion of the rendering process. The ray tracing specific Processing Modules can render the indirect illumination effect for which ray tracing is better suited. This scheme allows the traditional APIs to be used on an as-is basis, while gradually introducing the more advanced features to the software developers.

# REFERENCES

[1] H. W. Jensen, "Global illumination using photon maps," in *Proc. Eurographics Workshop on Rendering Techniques '96*, Porto, Portugal, 1996, pp. 21–30.

[2] A. Medvedev, "NVIDIA GeForce FX, or 'Cinema show started'," http://www.digit-life.com/articles2/gffx/, 2002.

[3] M. Segal, "Graphics hardware and graphics programming interfaces," http://artis.imag.fr/Ñicolas.Holzschuch/06102003/MSegal.pdf, 2003.

[4] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis, *Geometrical Considerations and Nomenclature for Reflectance.* NBS Monograph 160, National Bureau of Standards, U.S. Department of Commerce, Washington, DC, October 1977.

[5] J. Schmittler, I. Wald, and P. Slusallek, "Saarcor: a hardware architecture for ray tracing," in *Proc. ACM Siggraph/Eurographics Conference on Graphics Hardware*, Saarbrucken, Germany, 2002, pp. 27–36.

[6] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 15–22, Oct. 1984.

[7] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Proc. Eurographics '87*, Amsterdam, North-Holland, 1987, pp. 3–10.

[8] B. F. Naylor and W. C. Thibault, "Applications of BSP trees to ray-tracing and CSG evalutation," School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Tech. Rep. GIT-ICS 86/03, Feb. 1986.

[9] J. Arvo and D. Kirk, "Fast ray tracing by ray classification," in *Proc. 14th Annual Conference on Computer Graphics and Interactive Techniques*, Anaheim, CA, 1987, pp. 55–64.

[10] J. Arvo, "Backward ray tracing," in *Proc. Siggraph '86 Developments in Ray Tracing Seminar Notes*, vol. 12, New Orleans, LA, August 1986.

[11] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping*. Wellesley, MA: A. K. Peters, Ltd., 2001.

[12] H. W. Jensen, S. R. Marschner, M. Levoy, and P. Hanrahan, "A practical model for subsurface light transport," in *Proc. 28th Annual Conference on Computer Graphics and Interactive Techniques*, Los Angeles, CA, 2001, pp. 511–518.

[13] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *Proc. ACM Siggraph/Eurographics Conference on Graphics Hardware*, Saarbrucken, Germany, 2002, pp. 37–46.

[14] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proc. 29th Annual Conference on Computer Graphics and Interactive Techniques*, San Antonio, TX, 2002, pp. 703–712.

[15] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," in *Proc. Eurographics*, Manchster, United Kingdom, 2001, pp. 153–164.

[16] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proc. ACM Siggraph/Eurographics Conference on Graphics Hardware*, San Diego, CA, 2003, pp. 41–50.

[17] T. J. Purcell, "Ray tracing on a stream processor," Ph.D. dissertation, Stanford University, Stanford, CA, March 2004.

[18] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek, "Realtime ray tracing and its use for interactive global illumination," in *Proc. Eurographics State of the Art Reports*, Granada, Spain, 2003, pp. 65–78.

[19] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek, "Interactive global illumination using fast ray tracing," in *Proc. 13th Eurographics Workshop on Rendering*, Pisa, Italy, 2002, pp. 15–24.

[20] V. C. H. Ma and M. D. McCool, "Low latency photon mapping using block hashing," in *Proc. ACM Siggraph/Eurographics Conference on Graphics Hardware*, Saarbrucken, Germany, 2002, pp. 89–99.

[21] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *J. Graph. Tools*, vol. 2, no. 1, pp. 21–28, 1997.

# VITA

Junyi Ling was born in Shanghai, China. He is the son of Shitao Ling and Mingjie Zhou. He graduated from Canyon High School, Texas in 1996. That year he enrolled in West Texas A&M University and then transferred to Texas A&M University in 1999, where he graduated with a Bachelor of Science in Computer Engineering in the fall of 2001.

He can be contacted at the following permanent address: 7545 Katella Ave. Apt. 27, Stanton, CA 90680.