Universität Karlsruhe (TH)
Forschungsuniversität - gegründet 1825
Fakultät für Informatik
Institut für Betriebs- und Dialogsysteme

Studienarbeit

Accelerating the bidirectional path tracing algorithm using a dedicated intersection processor

Wenzel Jakob

30.07.2007

Supervised by Prof. em. Alfred Schmitt
Dipl. inform. Stefan Preuß

Abstract

Even with the processing power of today's computers, the calculation of global illumination solutions for general scenes remains a lengthy undertaking. This project investigates the possibility of accelerating such calculations using a dedicated hardware co-processor. The algorithm of choice, bidirectional path tracing, presents an additional challenge, since its three-dimensional random walks make the use of coherent ray tracing techniques impracticable. A parallel and deeply pipelined ray-triangle intersection processor including a kd-tree traversal unit was developed during the course of the project, and numerous benchmarks were performed.

Contents

List of Figures

List of Tables

Listings

1 Overview

1.1 Motivation

The main motivation of this project is the inherent parallelism of global illumination algorithms. Large numbers of rays need to be shot through virtual scenes – an operation, which seems very appropriate for acceleration using dedicated hardware.

On the other hand, hardware development platforms capable of prototyping such designs have only recently become available. However, successes by other groups, such as the graphics group of the University of Saarland, encourage further investigation. While most of the current focus is directed towards real-time Whitted-style ray tracing, little has been done in the direction of global illumination acceleration using hardware. One reason for this might be the sheer computational complexity of these algorithms in comparison to plain ray tracing. This project investigates the possibility of accelerating these calculations using a dedicated hardware co-processor. The used algorithm, bidirectional path tracing, poses an additional problem, since its random walks make the use of any coherent ray tracing techniques[1] impractical. To achieve a significant speedup despite this drawback, the algorithm was analyzed, and a new *bipartite mutual visibility query* instruction was implemented in silicon. Each such instruction consists of several intersection tests using hierarchical data structures. The final hardware implementation is able to execute up to 96 of these intersections tests in parallel.

1.2 Architecture

The project consists of three major parts: A global illumination renderer called *beam* (Discussed in Chapter 8), a Linux kernel driver, and the FPGA[2] processor core. The scope of this project was to plan and develop these components. Finally, the intersection architecture was to be implemented on actual hardware.

The FPGA core can be divided into five major blocks: The USB host interface, a DDR memory controller, a multi-ported direct-mapped cache, a ray scheduler, and several combined intersection and kd-tree traversal units. Figure 1.2 contains all FPGA components and their source code locations.

When beam renders an image, the following operations occur in sequence:

[1]Coherent ray tracing techniques exploit the statistical dependence of rays to trace several rays simultaneously with negligible overhead

[2] *Field Programmable Gate Array* - A reconfigurable microchip consisting of programmable logic and programmable interconnect.

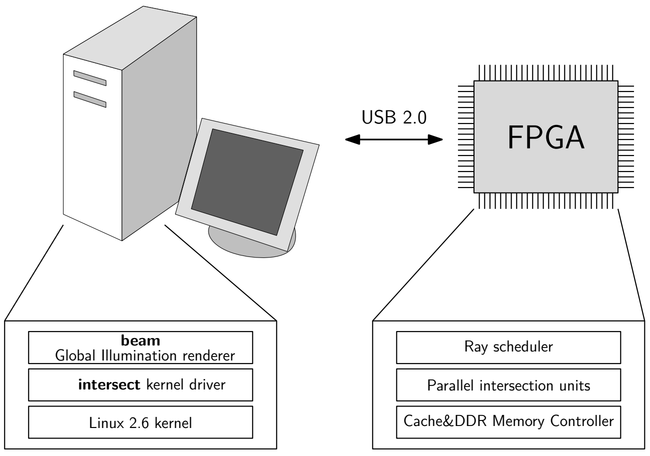

USB 2.0
FPGA
beam
Global Illumination renderer
intersect kernel driver
Linux 2.6 kernel
Ray scheduler
Parallel intersection units
Cache&DDR Memory Controller

Figure 1.1: Architecture overview

(i) beam parses the XML scene description and loads all referenced plugins such as material types, light sources and sampling techniques.

(ii) A kd-tree acceleration data structure is recursively constructed from the scene's geometry using heuristical methods.

(iii) The renderer converts this kd-tree data structure into a memory image that is compatible with the FPGA circuitry. This involves number format conversion and transforming triangle data into a special redundant storage format.

(iv) This memory image is uploaded to a DDR SDRAM chip connected to the FPGA. Afterwards, the FPGA cache and intersection units are initialized.

(v) The renderer starts tracing rays using the intersection processors. Multithreading support is provided by the hardware in order to process packets of rays at the same time, which helps to reduce the effects of memory fetch and pipeline latencies. Furthermore, the beam renderer uses a software pipeline to keep both the FPGA and the computer's CPU busy at all times.

(vi) The resulting bitmap consisting of spectral samples is developed into a RGB-based format. Instead of writing one output image, beam creates several partial images and one combined image to visualize the inner workings of the bidirectional path tracing algorithm.

The following chapters introduce the relevant foundations in the fields of Global Illumination, Monte Carlo integration, bidirectional path tracing and FPGA architecture. The processor implementation is discussed in chapters 6 and 7 and the software implementation is explained in chapter 8. Chapter 9 concludes with benchmark results and ideas for further work.

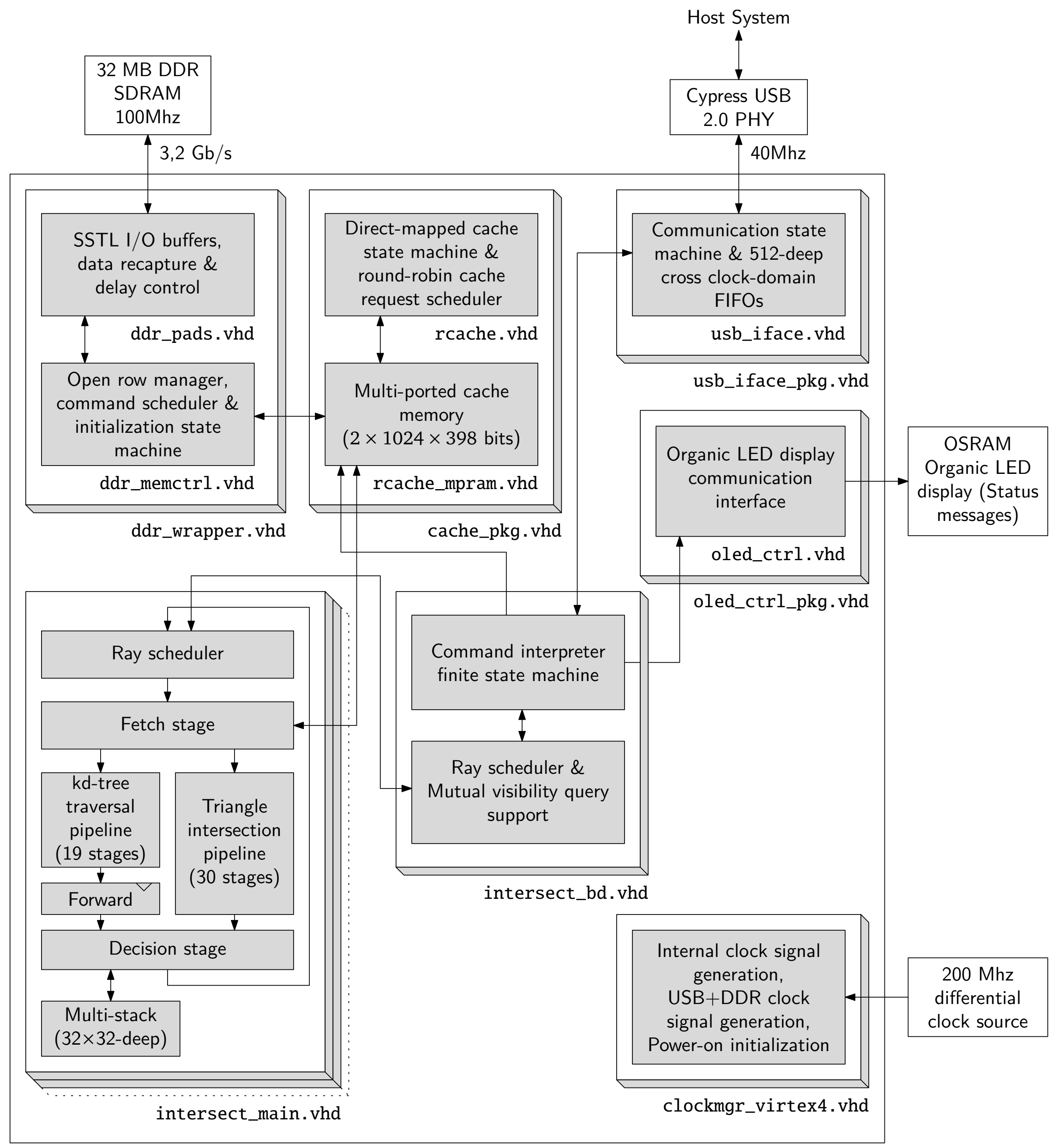

Host System
Cypress USB 2.0 PHY
40Mhz
32 MB DDR SDRAM 100Mhz
3,2 Gb/s
SSTL I/O buffers, data recapture & delay control
ddr_pads.vhd
Open row manager, command scheduler & initialization state machine
ddr_memctrl.vhd
ddr_wrapper.vhd
Direct-mapped cache state machine & round-robin cache request scheduler
rcache.vhd
Multi-ported cache memory (2 × 1024 × 398 bits)
rcache_mpram.vhd
cache_pkg.vhd
Communication state machine & 512-deep cross clock-domain FIFOs
usb_iface.vhd
usb_iface_pkg.vhd
Organic LED display communication interface
oled_ctrl.vhd
oled_ctrl_pkg.vhd
OSRAM Organic LED display (Status messages)
Ray scheduler
Fetch stage
kd-tree traversal pipeline (19 stages)
Triangle intersection pipeline (30 stages)
Forward
Decision stage
Multi-stack (32×32-deep)
intersect_main.vhd
Command interpreter finite state machine
Ray scheduler & Mutual visibility query support
intersect_bd.vhd
Internal clock signal generation, USB+DDR clock signal generation, Power-on initialization
clockmgr_virtex4.vhd
200 Mhz differential clock source

Figure 1.2: FPGA architecture overview

1.3 License

All source code, including the hardware description code, is available under an open source license and may be distributed accordingly. This document is placed into the public domain.

2 Global Illumination

Global illumination (GI) tries to visualize a three-dimensional scene in a way that the appearance of a single object can be influenced by any other object present in the scene. This behavior makes it possible to account for complex illumination features such as indirect lighting or caustics. Images rendered using global illumination look amazingly photo-realistic – but they also are very costly to create in terms of required computational power. This chapter discusses some of the theoretical foundations of global illumination.

2.1 Solid angles

A sound theoretical description of global illumination algorithms requires the integration of functions over the hemisphere above a point. Thus, a measure for the contents of a hemispherical area is needed in order to define a meaningful integral. This is called a *solid angle* and is defined as the surface area on the hemisphere divided by the squared hemisphere radius (usually $r = 1$). Its unit is named *steradian* (sr).

Using this measure and a transformation to spherical coordinates, a solid angle is represented as $\Theta = (\varphi, \theta)$, with φ and θ being the azimuth and elevation, respectively. We then define a differential solid angle

$$d\omega_\Theta := \sin(\theta)\, d\theta\, d\varphi \tag{2.1}$$

where $d\theta$ and $d\varphi$ are the differential elevation and azimuth. The decrease in hemispherical area near the horizon is the reason for the additional sine factor. We define

$$\Omega^+ := [0, 2\pi] \times [0, \frac{\pi}{2}] \tag{2.2}$$

as the set comprising the entire hemisphere. The integral can now be expressed:

$$\int_{\Omega^+} f(\Theta)\, d\omega_\Theta = \int_0^{2\pi} \int_0^{\pi/2} f(\varphi, \theta) \cdot \sin(\theta)\, d\theta\, d\varphi \tag{2.3}$$

2.2 Radiometric quantities

Global illumination algorithms attempt to converge to a stable, physically correct distribution of light – therefore, the development of such an algorithm requires an understanding of the underlying physical quantities. *Radiometry* is the measurement of optical radiation, including the visible range between 400 and 700nm. This section will not go into the complex physical details and instead only mention the parts that are important for computer graphics. A thorough introduction can be found in [2]. As an additional simplification, this section will leave out the concept of wavelengths and spectral radiance.

The most fundamental radiometric quantity is called *radiant power* or *flux* and is usually denoted by the letter Φ. It expresses how much total energy flows through a subset of $\mathbb{R}^3$ per unit time and is measured in Watt (Joules per second).

2.2.1 Irradiance

The *irradiance* with respect to a location x denotes the radiant power per unit surface area and is measured in Watt/m^2. It describes the density of radiant power arriving at a surface point.

$$E(x) := \frac{\mathrm{d}\Phi}{\mathrm{d}A_x} \tag{2.4}$$

where A_x is an infinitesimal surface patch containing x. The word irradiance usually signifies that only incident radiant power is considered. Otherwise, the word *radiosity* is used.

2.2.2 Radiance

The *radiance* for a given point x and a direction of interest Θ is defined as the flux per projected area per unit solid angle. The projected area $A_x^\perp$ is a differential surface area projected perpendicularly to the direction of interest and ω_Θ is a differential solid angle around Θ. For an illustration, refer to figure 2.1. Radiance is measured in Watt/$(sr \cdot m^2)$.

$$L(x,\Theta) := \frac{\mathrm{d}^2\Phi}{\mathrm{d}A_x^\perp\,\mathrm{d}\omega_\Theta} \tag{2.5}$$

If we want to describe the radiance that is incident to or exitant[1] from a given differential surface A_x with the surface normal N_x at x, the following equality holds:

$$L(x,\Theta) = \frac{\mathrm{d}^2\Phi}{\cos(N_x,\Theta)\,\mathrm{d}A_x\,\mathrm{d}\omega_\Theta} \tag{2.6}$$

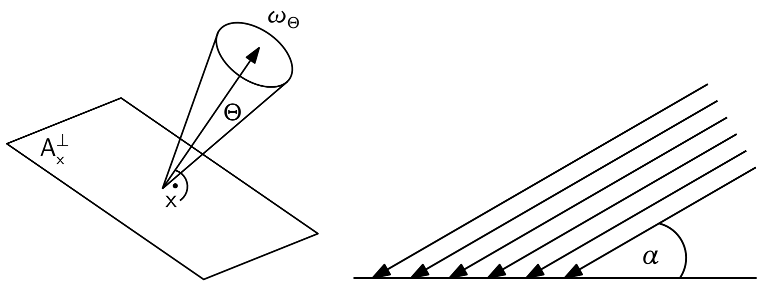
where $\cos(N_x,\Theta)$ is the cosine of the angle between Θ and the surface normal. This result makes sense when imagining a circular shaft of light shining on a surface with a non-perpendicular angle ($\cos(N_x,\Theta) < 1$). Since the surface area receiving light resembles a stretched circle, it is larger than a perpendicularly projected area receiving this light. The cosine term in (2.6) is required to cancel out this increase in area.

From now on, we will categorize radiance by its orientation to a surface point and write L_o for exitant radiance and L_i for incident radiance. In space, these functions satisfy

$$L_i(x,\Theta) = L_o(x,-\Theta). \tag{2.7}$$

Thus, there is no distinction between incident or outgoing radiance. However, at a surface points, these functions measure different quantities.

[1]The usage of this word stems from Eric Veach's Ph. D. thesis [10]



(**a**) Definition of radiance (**b**) Non-perpendicular incident light

Figure 2.1: A graphical illustration of radiance

The irradiance at a surface point x can be obtained by integrating the radiance function over the hemisphere above x. In (2.8), the cosine factor models the fact that incident light at a grazing angle will deposit less radiant energy than light at a perpendicular angle.

$$E(x) = \int_{\Omega^+} L_i(x,\Theta)\cos(N_x,\Theta)\,\mathrm{d}\omega_\Theta \tag{2.8}$$

One important property of radiance is that, in absence of participating media such as fog or flame, its value is invariant along straight paths.

2.3 Geometric term

Let us consider the situation of figure 2.3: there are two differential surfaces A_x and A_y and a normalized direction vector pointing from x to y.

$$\Theta := \frac{y-x}{\|y-x\|}.$$

Dutré et al. [2] use a transformation for expressing a solid angle ω_Θ as a differential surface:

$$\mathrm{d}\omega_\Theta = \frac{\cos(N_y,-\Theta)\,\mathrm{d}A_y}{\|x-y\|^2} \tag{2.9}$$

Suppose that we want to express the radiance traveling on a ray from x to y. (2.6) and (2.9) imply that

$$L(x,\Theta) = \frac{\mathrm{d}^2\Phi}{\cos(N_x,\Theta)\,\mathrm{d}A_x\,\mathrm{d}\omega_\Theta} \tag{2.10}$$

$$= \frac{\mathrm{d}^2\Phi}{\mathrm{d}A_x\,\mathrm{d}A_y}\cdot\frac{1}{G(x,y)} \tag{2.11}$$

where

$$G(x,y) := \frac{\cos(N_x,\Theta)\,\cos(N_y,-\Theta)}{\|x-y\|^2} \tag{2.12}$$

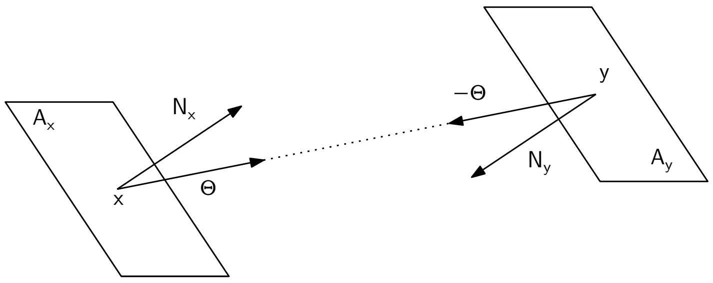


Figure 2.2: Illustration of the geometric term

$G(x, y)$ is called the *geometric term* and describes how much radiance is passed from A_x to A_y. The denominator expresses how the subtended solid angle of the differential surface A_y on the hemisphere of x will shrink quadratically with increasing distance.

2.4 Bidirectional reflectance distribution function

The *bidirectional reflectance distribution function* (BRDF) describes what fraction of the incident irradiance from a certain direction (Ψ) leaves as reflected radiance in another direction (Φ). It was introduced by Nicodemus in 1970 and models how light interacts with a material. Using (2.8), we can derive the irradiance function for a solid angle:

$$\mathrm{d}E(x,\Theta) = L_i(x,\Theta)\,\cos(\Theta, N_x)\,\mathrm{d}\omega_\Theta \tag{2.13}$$

Now we can write down the definition of $f_r : \mathbb{R}^3 \times \Omega^+ \times \Omega^+ \to \mathbb{R}$.

$$f_r(x,\Psi,\Theta) := \frac{\mathrm{d}L_o(x,\Theta)}{\mathrm{d}E(x,\Psi)} = \frac{\mathrm{d}L_o(x,\Theta)}{L_i(x,\Psi)\,\cos(\Psi, N_x)\,\mathrm{d}\omega_\Psi} \tag{2.14}$$

In order to be physically plausible, the BRDF needs to have several properties:

(i) **Range**: The value of a BRDF is always greater than or equal to zero.

(ii) **Reciprocity**: The BRDF value does not change when Θ and Ψ are exchanged. The notation $f_r(x, \Psi \longleftrightarrow \Theta)$ denotes this property.

(iii) **Energy conservation**: The law of conservation of energy demands that, given incident irradiance along an angle Ψ, the total reflected radiance is less than or equal to this amount.

$$\int_{\Omega^+} f_r(x,\Psi,\Theta)\cos(N_x,\Theta)\,\mathrm{d}\omega_\Theta \leq 1 \tag{2.15}$$

The *bidirectional scattering distribution function* (BSDF) is an extension of the BRDF to the whole sphere around a surface point and as such, it can model refraction in addition to reflection.

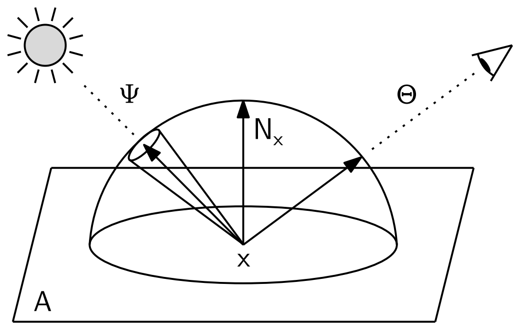


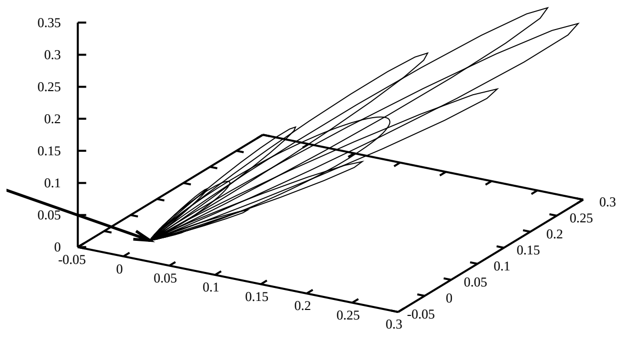
Figure 2.3: The Bidirectional Reflectance Distribution Function

2.4.1 Modified Blinn-Phong BRDF

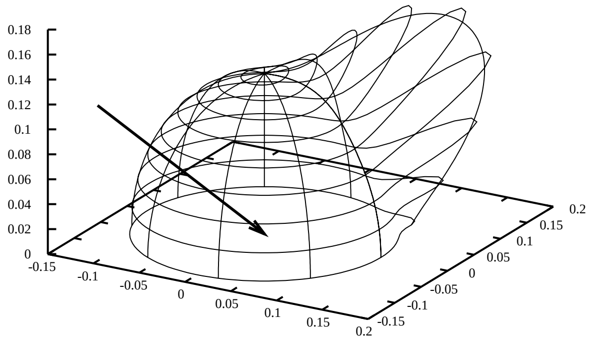
Since the commonly used Phong BRDF does not satisfy the physical plausibility properties of energy conservation and reciprocity, a modified [5] Blinn-Phong model can be used instead:

$$f_r(x, \Psi \longleftrightarrow \Theta) = k_s \frac{n+2}{2\pi} \cos^n(N_x, H) + k_d \frac{1}{\pi} \tag{2.16}$$

where H is a half-vector in the plane of Ψ and Θ and $k_d + k_s \leq 1$ (energy conservation). k_d and k_s respectively control how diffuse and specular the surface is. This model raises the cosine of the angle between H and the surface normal to a power n, which determines how sharp the specular reflection is – a low value causes glossy reflection.



(**a**) Perfect specular reflection ($k_d = 0, n = 160$)



(**b**) Diffuse and glossy specular reflection ($k_d = 0.4, n = 40$)

Figure 2.4: Polar plots of the modified Blinn-Phong BRDF ($k_s = 0.02$ in both cases). The black arrow denotes the (constant) direction of incoming irradiance. The hemisphere radius in a direction denotes the amount of reflectance.

2.5 Rendering equation

When solving a global illumination problem, we are most interested in a radiance function describing the state after an energy equilibrium has been reached. In 1986, Kajiya presented the *rendering equation* [3] as a generalization of various global illumination algorithms. It is based on similar equations known from radiative heat transfer research. We will use a slightly modified [2] version expressed using angles and a BRDF.

The exitant radiance $L_o(x,\Theta)$ can be split up into an emitted radiance term L_e (emitted by the surface itself) and a reflected radiance term L_r (created as a result of incident radiance).

$$L_o(x,\Theta) = L_e(x,\Theta) + L_r(x,\Theta) \tag{2.17}$$

Using the definition of radiance (2.5) and the definition of the BRDF (2.14), we can calculate L_o using an intuitive integral equation:

$$L_o(x,\Theta) = L_e(x,\Theta) + \int_{\Omega^+} L_i(x,\Psi)\, f_r(x,\Psi,\Theta)\, \cos(N_x,\Psi)\, \mathrm{d}\omega_\Psi \tag{2.18}$$

This is the rendering equation in its basic hemispherical form – being a Fredholm equation of the second kind, it cannot be analytically solved except for some absolutely trivial cases.

2.5.1 Area formulation

There are several equivalent formulations of this equation – the second important one does not integrate over the hemisphere of a point but does so over *all* geometry contained in the scene. For this, we need a ray casting function:

$$r(x,\Theta) := x + \Theta \cdot \min\{t \in \mathbb{R}^+ \mid x + t\cdot\Theta \in \mathscr{A}\}$$

where $\mathscr{A} \subseteq \mathbb{R}^3$ contains all surface points of the scene. Thus, r finds the closest intersection point between a surface and the ray starting at x in direction Θ. Using r, we can also define a mutual visibility function V:

$$V(x,y) := \begin{cases} 1, & \text{if } r(x,\overrightarrow{xy}) = y \\ 0, & \text{otherwise} \end{cases}$$

Using V and the geometric term (2.12), the L_i term in (2.18) can be substituted.

$$L_o(x,\Theta) = L_e(x,\Theta) + \int_{\mathscr{A}} L_o(y,\overrightarrow{yx})\, f_r(x,\overrightarrow{xy},\Theta)\, V(x,y)\, G(x,y)\, \mathrm{d}A_y \tag{2.19}$$

2.5.2 Path integral formulation

The third approach [1] introduces three new integral operators K, G and T:

$$(Kh)(x,\Theta) := \int_{\Omega^+} h(x,\Psi)\, f_r(x,\Psi,\Theta)\, \cos(N_x,\Psi)\, \mathrm{d}\omega_\Psi$$

K is called the *local scattering operator* because it matches the reflectance integral L_r of the rendering equation.

$$(Gh)(x,\Theta) := \begin{cases} h(r(x,\Theta),-\Theta), & \text{if } r(x,\Theta) \text{ is defined} \\ 0, & \text{otherwise} \end{cases}$$

G is called the *propagation operator*, since it converts exitant radiance on distant surfaces to incident radiance on the local surface such that $GL_o = L_i$. For a visual illustration of the operators, refer to figure 2.5.

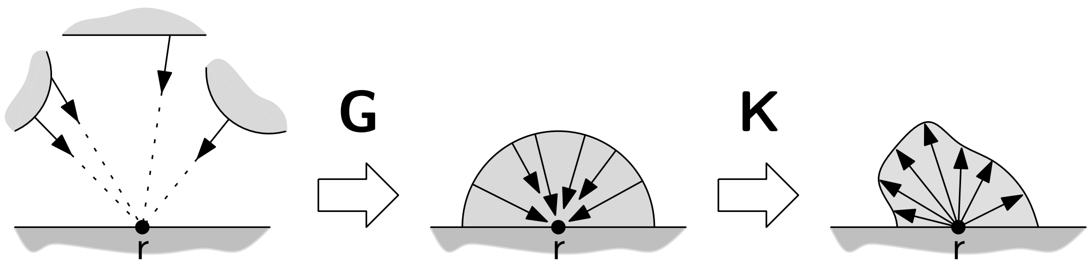


Figure 2.5: An illustration of the operators K and G [1].

$$T := K \circ G$$

The third operator T is the concatenation of the previous two operators and is called the *light transport operator*. Using T, the rendering equation simplifies to

$$\begin{aligned} & L_o = L_e + KL_i \\ \Leftrightarrow\quad & L_o = L_e + TL_o \\ \Leftrightarrow\quad & L_e = (I-T)L_o \\ \Leftrightarrow\quad & L_o = (I-T)^{-1}L_e \end{aligned} \tag{2.20}$$

assuming that $(I-T)$ has an inverse. Similarly to the geometric series, the inverse exists if $||T|| < 1$, where $||\cdot||$ is the *operator norm* given by:

$$||S|| := \sup_{||f||_\sim \leq 1} ||Sf||_\sim$$

for some norm $||\cdot||_\sim$. In this case, the domain of this norm is the set of all radiance functions, which becomes a banach space using the L_p norm defined by:

$$||f||_p = \left[\int_{R^3}\int_\Omega |f(x,\Theta)|^p \,\mathrm{d}\omega_\Theta \,\mathrm{d}x\right]^{\frac{1}{p}}$$

In 1995, Arvo [1] has shown for one-sided reflective surfaces with physically plausible BRDFs that $||G|| \leq 1$ and $||K|| \leq 1$ if $||\cdot||_\sim$ is a L_p-norm with $1 \leq p \leq \infty$. By futher assuming that no surface in the scene is perfectly reflective, he has furthermore shown that $||K|| < 1$ and thus

$$||T|| = ||KG|| \leq ||K||\,||G|| < 1 \tag{2.21}$$

Therefore, $(I - T)$ is invertible and its inverse is the *Neumann series* of T

$$(I - T)^{-1} = I + T + T^2 + \cdots = \sum_{n=0}^{\infty} T^n \tag{2.22}$$

Combining (2.20) and (2.22) yields

$$L_o = L_e + T L_e + T^2 L_e + T^3 L_e + \cdots \tag{2.23}$$

In other words, the rendering equation can also be expressed as the sum of contributions by paths of lengths 1, 2, etc. coming from an emissive scene object. This is the integral we will try to solve.

3 Monte Carlo Integration

Monte Carlo Integration tries to solve integrals which might otherwise be hard or impossible to solve using analytical methods. Consider the one-dimensional definite integral over the interval $[0, 1]$.

$$I := \int_0^1 f(x)\,\mathrm{d}x. \tag{3.1}$$

Many functions from the domain of computer graphics do not possess elementary antiderivatives. In these cases, numerical quadrature methods are usually applied to approximate I. However, these methods require the evaluation of f at many sample points. Furthermore, when integrals of higher dimension are to be solved, quadrature methods become increasingly impractical since they require the evaluation of f over a multidimensional grid with a polynomial amount of sample points. This phenomenon is known as the *curse of dimensionality*. In these cases, Monte Carlo Integration may still provide very accurate numerical solutions while not suffering from any of these problems. Monte Carlo Integration is covered in depth in [2], [4] and [10].

3.1 Basic Monte Carlo integration

One important quality of Monte Carlo methods is that their convergence rates are completely independent of the integration domain's dimensionality. This property makes them the method of choice for solving integrals like (2.23), which require the integration over an infinite amount of dimensions. For simplicity, the following estimators are restricted to one dimension. However, they could easily be extended to multiple dimensions without changing their fundamental properties.

3.1.1 Primary estimator

Given n independent and identically-distributed random variables $X_1, \ldots, X_n \sim U(0, 1)$ with U being the standard uniform distribution, we can define n primary estimators

$$\langle I_i \rangle := f(X_i)$$

In this chapter, p_{X_i} denotes the probability distribution function of X_i. The expected value of these estimators is the value of the definite integral (3.1):

$$E[\langle I_i \rangle] = \int_0^1 f(x)\,p_{X_i}(x)\,\mathrm{d}x = \int_0^1 f(x)\,\mathrm{d}x = I \tag{3.2}$$

The estimator is *unbiased* because the expected values matches the quantity which is to be estimated. The variance $\sigma^2_{\langle I_i \rangle}$ of $\langle I_i \rangle$ can also be calculated:

$$\sigma^2_{\langle I_i \rangle} = E[(f(x) - I)^2] = \int_0^1 f^2(x)\,\mathrm{d}x - I^2 \tag{3.3}$$

In practice, one would never use this estimator by itself because the variance is constant and generally too high to yield acceptable results.

3.1.2 Secondary estimator

A secondary estimator can be created by averaging the contributions of the primary estimators. An actual implementation generates a series of pseudo-random numbers and averages the function evaluations at these points.

$$\langle J \rangle := \frac{1}{n} \sum_{i=0}^{n} \langle I_i \rangle$$

The expected value of this estimator also matches the integral (3.1) and it is thus unbiased.

$$E[\langle J \rangle] = E[\frac{1}{n} \sum_{i=0}^{n} \langle I_i \rangle] = \frac{1}{n} \sum_{i=0}^{n} E[\langle I_i \rangle] = I \tag{3.4}$$

Compared to (3.3), the variance is reduced by a factor of n.

$$\sigma^2_{\langle J \rangle} \overset{\text{iid}}{=} \frac{1}{n^2} \sum_{i=0}^{n} \sigma^2_{\langle I_i \rangle} = \frac{1}{n} \sigma^2_{\langle I_0 \rangle} \tag{3.5}$$

However, the standard deviation σ is only reduced by a factor of $\sqrt{n}$. This is also the main practical problem with $\langle J \rangle$ and Monte-Carlo integration in general. In order to improve the quality of the result I by a factor of two, the number of samples has to be increased four-fold.

3.2 Variance reduction techniques

Many techniques have been developed to mitigate this property of Monte Carlo methods. Prominent examples are *importance sampling* and *stratified sampling*.

3.2.1 Importance Sampling

The underlying idea of importance sampling is that some parts on the interval $[0,1]$ are more important for the calculation of I than others, specifically the areas where $f(x)$ has a relatively large value. Ideal random variables $X_1, \ldots, X_i$ would only sample f within these areas while still resulting in an unbiased estimator.

Suppose that $X_0 \dots X_n$ are independent and distributed according to a probability density function p and $p(x) \neq 0$ $(x \in [0,1])$. We can define new estimators

$$\langle I_i' \rangle := \frac{f(X_i)}{p(X_i)} \qquad \text{and} \qquad \langle J' \rangle := \frac{1}{n} \sum_{i=0}^{n} \langle I_i \rangle$$

Again, these estimators are unbiased, but they have different variance terms:

$$E[\langle I_i' \rangle] = \int_0^1 \frac{f(x)}{p(x)} p(x) \, \mathrm{d}x = I \tag{3.6}$$

$$\sigma^2_{\langle I_i' \rangle} = \int_0^1 \left(\frac{f(x)}{p(x)} \right)^2 p(x) \, \mathrm{d}x - I^2 = \int_0^1 \frac{f^2(x)}{p(x)} \, \mathrm{d}x - I^2 \tag{3.7}$$

$$\sigma^2_{\langle J' \rangle} \overset{\text{iid}}{=} \frac{1}{n^2} \sum_{i=0}^{n} \sigma^2_{\langle I_i' \rangle} = \frac{1}{n^2} \sum_{i=0}^{n} \left(\int_0^1 \frac{f^2(x)}{p(x)} \, \mathrm{d}x \right) - \frac{1}{n} I^2 \tag{3.8}$$

As can be seen in (3.8), the variance $\sigma^2_{\langle J' \rangle}$ is zero if $p(x) = f(x)/I$. Unfortunately, sampling according to this probability density function involves both normalizing p and calculating the inverse of its cumulative distribution function. Both of these steps require the integration of f over $[0,1]$, which we were trying to avoid in the first place. In practice, any probability density function f with a "similar shape" will cause a major variance reduction.

3.2.2 Stratified sampling

Sampling uniformly distributed random variables often causes the accumulation of many sample points in one location ("clumping"). This effect slows down convergence of Monte Carlo methods and can be avoided by distributing samples more uniformly. The fundamental idea of stratified sampling is to subdivide the integration domain into several disjoint intervals ("strata") D_i such that

$$\int_0^1 f(x) \, \mathrm{d}x = \int_{D_0} f(x) \, \mathrm{d}x + \cdots + \int_{D_m} f(x) \, \mathrm{d}x \tag{3.9}$$

and then equally distributing the samples amongst these intervals, which causes negligible computational overhead. According to Lafortune [4], the variance of stratified sampling is always less than or equal to the variance of the estimator $\langle J \rangle$. This technique is effective, easy to implement, and can be combined with importance sampling.

3.3 Russian roulette

A naïve algorithm numerically approximating each integral in the path formulation of the rendering equation (2.23) would never terminate. For a well-defined algorithmic implementation, a stop condition, where evaluation terminates, is needed. Care must be taken as not to introduce any bias into the estimator.

Russian-roulette defines an estimator $\langle J \rangle_{rr}$ based on $\langle J \rangle$, which evaluates to zero with a probability a (as in *absorption*).

$$\langle J \rangle_{rr} := \begin{cases} 0, & \text{with probability } a \\ (1-a)^{-1} \cdot \langle J \rangle, & \text{with probability } 1-a \end{cases}$$

This estimator is still unbiased, although the variance is obviously increased.

$$E[\langle J \rangle_{rr}] = a \cdot 0 + (1-a) \cdot \frac{1}{1-a} \cdot E[\langle J \rangle] = E[\langle J \rangle] \tag{3.10}$$

3.4 Estimators for Fredholm equations

Integral equations of the form

$$f(x) = g(x) + \int_0^1 K(x,y) f(y) \, \mathrm{d}y \tag{3.11}$$

where K is the kernel of the associated integral operator, g is a known function and f is unknown are called *inhomogeneous Fredholm equations of the second kind*. Lafortune [4] uses an unbiased estimator for f using importance sampling:

$$\begin{aligned}
\langle f(x) \rangle &= g(x) + \frac{K(x,X_1)}{p_1(X_1)} \langle f(X_1) \rangle \\
&= g(x) + \frac{K(x,X_1)}{p_1(X_1)} \left(g(X_1) + \frac{K(X_1,X_2)}{p_2(X_2)} \langle f(X_3) \rangle \right) \\
&= g(x) + \frac{K(x,X_1)}{p_1(X_1)} g(X_1) + \frac{K(x,X_1)}{p_1(X_1)} \frac{K(X_1,X_2)}{p_2(X_2)} g(X_2) + \cdots \\
&= \sum_{i=0}^{\infty} \left(\prod_{j=1}^{i} \frac{K(X_{j-1},X_j)}{p_j(X_j)} \right) g(X_i)
\end{aligned} \tag{3.12}$$

where $X_1, \ldots, X_n$ are random variables distributed according to the probability density functions $p_1, \ldots, p_n$ and $X_0 = x$. This estimator can be combined with russian roulette for an algorithmic implementation (Listing 3.1).

3.5 Sampling the hemisphere

There are several different approaches to reduce variance when sampling the hemisphere above a surface point.

(i) **Cosine lobe sampling**: The rendering equation (2.18) includes a cosine term since light at grazing angles contributes less radiant energy to the irradiance at the differential surface position. Cosine lobe sampling uses this fact to prefer angles that are closer to the surface normal.

```
MonteCarlo-Integrate (x)
s ← Uniform sample from [0, 1]
v ← 0
if (s < a)
    return // Absorption
$s$ ← Sample [0,1] using the PDF $p$
$v \leftarrow g(s)$ + (MonteCarlo-Integrate($s$) $\cdot K(x,s))/p(s)$
return v / (1 − a)
```

Listing 3.1: An unbiased primary estimator for Fredholm equations

To create random numbers with the distribution p_{cl}, we can, for example, calculate the inverse of the cumulative distribution function F_{cl}. The probability density function p_{cl} is given by

$$p_{cl}(\Theta) = \frac{\cos(N_x, \Theta)}{\pi} = \frac{\cos(\theta)}{\pi}$$

where $\Theta = (\varphi, \theta)$ and θ is the elevation on the hemisphere.

$$\begin{aligned} F_{cl}(\varphi, \theta) &= \int_0^{\varphi} \int_0^{\theta} \frac{\cos(\theta')}{\pi} \sin(\theta')\, d\theta'\, d\varphi' \\ &= \int_0^{\varphi} \frac{1}{\pi} \left[\frac{\cos^2(\theta')}{2} \right]_0^{\theta} d\varphi' \\ &= \frac{\varphi}{2\pi} \cdot (1 - \cos^2(\theta)) =: F_{\varphi}(\varphi) \cdot F_{\theta}(\theta) \end{aligned}$$

The cumulative distribution function F_{cl} is separable and the inverse of its parts can easily be calculated by setting φ to 2π or θ to $\frac{\pi}{2}$.

$$F_{\varphi}^{-1}(\xi_1) = 2\pi\xi_1 \tag{3.13}$$

$$F_{\theta}^{-1}(\xi_2) = \cos^{-1}(\sqrt{1 - \xi_2}) \tag{3.14}$$

Since ξ_i will be uniform random numbers on the interval $[0, 1]$, the term $1 - \xi_2$ in (3.14) can be replaced by ξ_2. Applying these transformations to ξ_1 and ξ_2 will result in directions that are distributed according to p_{cl}.

(ii) **BRDF sampling**: When a surface reflection occurs, BRDF sampling prefers directions which transport much radiant energy. Unfortunately, creating samples with such a distribution is not always possible using analytic methods. While BRDF sampling is superior to cosine lobe sampling, it generally has a higher computational cost. This project uses the modified Blinn-Phong BRDF because proper samples can be generated using a simple closed form.

4 Bidirectional path tracing

Bidirectional path tracing was independently developed by Veach and Guibas [10] and Lafortune and Willems [4]. It unites and generalizes the path tracing and light tracing algorithms while substantially reducing their variance.

4.1 Path tracing and light tracing

The *path tracing* technique proposed by Kajiya [3] iteratively solves the rendering equation by recursively tracing rays through the scene. Whenever a ray hits a non-emissive surface, it is randomly either absorbed, reflected or refracted. When a ray hits a light source, the contribution of the hitting path is accumulated on the camera's film. The construction of these so-called *random walks* constitutes an estimator of the rendering equation's solution. The reciprocity laws make this approach physically correct even though photons actually travel in the opposite direction. A major improvement of the technique samples the light sources at every intersection to rectify cases where the probability of this event is almost zero. A significant drawback of the path tracing technique is its low convergence rate. Even with the mentioned modifications, more than 1024 samples per pixel are needed for the most basic of scenes.

The *light tracing* technique is the dual version of path tracing – photons are traced from the light source to the viewpoint. Light tracing can handle some scenes well where path tracing takes a very long time to converge. However, the number of of photons required for an image without noise is staggeringly high. Instead of tracing radiance, the light tracing technique propagates a quantity called *importance*. Importance describes how much influence an infinitesimal light source positioned at some point has on the scene. Importance can be handled like radiance and all presented equations can also be applied to it. Tracing radiance and importance are essentially symmetric operations and it can be proven that the linear operators expressing them are mutually adjoint.

4.2 A bidirectional estimator

Bidirectional path tracing works by performing two random walks at the same time – one from the light source and another one from the viewpoint. Both radiance and importance are traced and later combined. Importance sampling, stratified sampling and russian roulette can all be used to create samples. The path's vertices are denoted by x_i $(i = 0, \ldots, S)$ on the eye path and y_i $(i = 0, \ldots, T)$ on the light path.

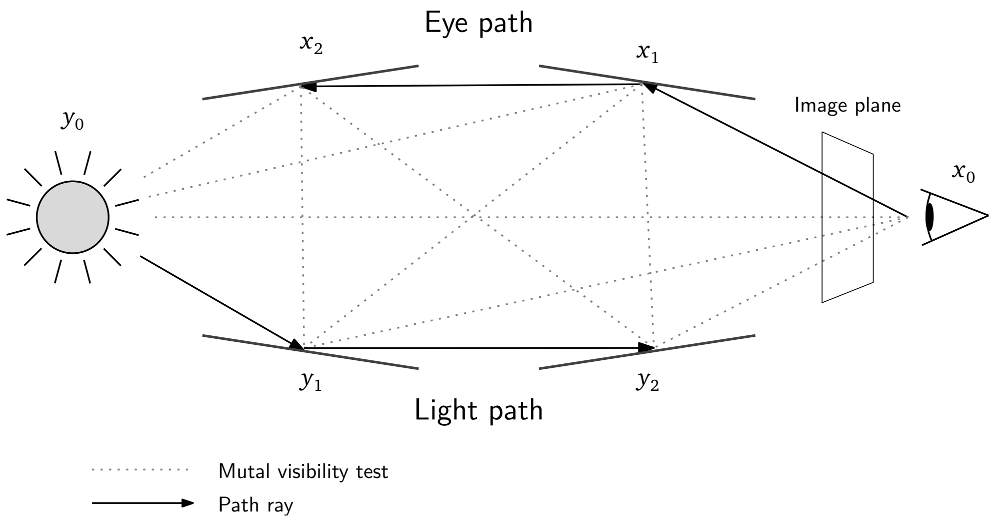


Figure 4.1: Overview of the bidirectional path tracing algorithm

In a second step, the mutual visibility of hit points on the two paths is tested and stored (Figure 4.1). Finally, the contribution of every possible composite path is calculated. Let $C_{t,s}$ be the path created by concatenating the vertices $y_0, \ldots, y_t$ and $x_s, \ldots, x_0$. It consists of t steps on the light path, s steps on the eye path and one additional deterministic step connecting the two sub-paths. We would like to create a partial estimator $\langle C_{t,s} \rangle$ to calculate the flux over this composite path. Lafortune [4] differentiates several cases:

(i) $s = 0, t = 0$: The light source is directly visible from the eye and the affected pixel coordinates have to be determined a posteriori. G is the geometric term known from (2.12).

$$\langle C_{0,0} \rangle = L_e(y_0, \overrightarrow{y_0 x_0})\, G(x_0, y_0)$$

(ii) $s > 0, t = 0$: Paths with this configuration correspond to classic path tracing – the light has to be explicitly sampled for this path configuration. Let the normal vectors at the surface interaction points be named n_{x_i} and n_{y_i}:

$$\begin{aligned}\langle C_{0,s} \rangle = & L_e(y_0, \overrightarrow{y_0 x_s})\, f_r(x_s, \overrightarrow{x_s y_0}, \overrightarrow{x_s x_{S-1}})\, \cos(n_{x_s}, \overrightarrow{x_s y_0}) \\ & \prod_{i=1}^{S-1} f_r(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}})\, \cos(n_{x_i}, \overrightarrow{x_i x_{i+1}})\end{aligned}$$

This complicated-looking estimator is simply the product of the BRDF and exitant cosine values between surface interactions on the eye path and between the light source and the last eye path interaction.

(iii) $s = 0, t > 0$: Paths with this configuration correspond to classic light tracing. The affected pixel coordinates have to be determined a posteriori.

$$\langle C_{t,0}\rangle = L_e(y_0, \overrightarrow{y_0 y_1}) \left(\prod_{i=1}^{T-1} f_r(y_i, \overrightarrow{y_i y_{i+1}}, \overrightarrow{y_i y_{-1}}) \cos(n_{y_i}, \overrightarrow{y_i y_{i+1}}) \right)$$
$$f_r(y_t, \overrightarrow{y_t x_0}, \overrightarrow{y_t y_{T-1}}) \cos(n_{y_s}, \overrightarrow{y_s x_0})$$

(iv) $s > 0, t > 0$: The composite path consists of t reflections on the light path and s reflections on the eye path.

$$\langle C_{t,s}\rangle = L_e(y_0, \overrightarrow{y_0 y_1}) \left(\prod_{i=1}^{T-1} f_r(y_i, \overrightarrow{y_i y_{i+1}}, \overrightarrow{y_i y_{-1}}) \cos(n_{y_i}, \overrightarrow{y_i y_{i+1}}) \right)$$
$$f_r(y_t, \overrightarrow{y_t x_s}, \overrightarrow{y_t y_{T-1}})\, G(y_t, x_t)\, f_r(x_s, \overrightarrow{x_s y_t}, \overrightarrow{x_s x_{S-1}})$$
$$\left(\prod_{i=1}^{S-1} f_r(x_i, \overrightarrow{x_i x_{i+1}}, \overrightarrow{x_i x_{i-1}}) \cos(n_{x_i}, \overrightarrow{x_i x_{i+1}}) \right)$$

When russian roulette or importance sampling are used, additional factors have to be taken into account. The flux arriving at the eye location x_0 can be expressed as a weighted sum of the partial estimators multiplied by the visibility function V:

$$\Phi = \sum_{t=0}^{T} \sum_{s=0}^{S} \alpha_{t,s} \cdot V(y_t, x_s) \cdot \langle C_{t,s}\rangle$$

These weights cannot be chosen arbitrarily – some constraints need to be satisfied or the resulting estimator might be biased:

$$\sum_{t=0}^{N} \alpha_{t,N-t} = 1 \quad (N = 0, 1, \ldots)$$

The rationale is that the groups of composite paths with lengths 1, 2, ... each receive weights that add up to one. For this project, simple coefficients of the form

$$\alpha_{t,s} = \frac{1}{t+s+1}$$

were chosen. It can be seen that they satisfy the above condition.

4.3 Pure software implementation

The following code consists of simplified C++ excerpts of a pure software implementation of the bidirectional path tracing algorithm.

The main rendering loop (Listing 4.2) uses a loadable sampling technique to generate an image plane sample and transforms it into an eye ray. Afterwards, a light ray is sampled

```
struct Vertex {
    Point p;              // Surface point in world coordinates
    Normal n;             // Surface normal (world coordinates)
    Vector dgdu, dgdv;    // Tangent and binormal vectors
    Vector wi, wo;        // Incident and exitant directions (local)
    const BSDF *bsdf;     // Pointer to the surface's BSDF
    Spectrum cumulative; // Cumulative attenuation factor
};
```

Listing 4.1: Surface interaction data structure

```
Sample eyeSample;
Ray eyeRay;

while (m_sampler->getNextSample(eyeSample)) {
    m_camera->generateRay(eyeSample, eyeRay);

    Spectrum Le =  sampleLight(lightRay, lightPdf);

    if (lightPdf != 0.0f) {
        Le /= lightPdf;
        renderPaths(eyeSample, eyeRay, lightRay, Le);
    }
}
```

Listing 4.2: Main rendering loop

and the method `BiDir::renderPaths` is invoked on these values. These two rays form the start of the eye and light path random walks. Before the `renderPaths` function is discussed, we will first introduce two utility functions:

The method `BiDir::randomWalk` in listing 4.3 performs a random walk either on an eye or a light path. This is done by following one initial ray and then sampling the BSDF at each intersection point. A cumulative attenuation factor is stored in the `cumulative` variable. The evaluation is only stopped once an absorption event occurs.

The method `BiDir::evalPath` in listing 4.4 takes two arrays containing the surface interactions of previously created eye and light paths. The paths are then evaluated according to case (iv) of the algorithm.

The `BiDir::renderPaths` function uses these methods to evaluate cases (ii) to (iv). Case (i) is omitted here for simplicity. It can be seen how the source code implementation corresponds exactly to the previously introduced estimator types.

```
void BiDir::randomWalk(Ray ray,        // Initial ray
        std::vector<Vertex> &vertices, // Target array
        bool eye) const {              // Eye or light path?
    Intersection its;
    Spectrum cumulative(1.0f); // Cumulative attenuation factor

    while (true) { // Repeat until an absorption event occurs
        if (!rayIntersect(ray, its)) // Intersect the ray with
            break;                   // the scene

        const BSDF *bsdf = its.mesh->getBSDF();
        float bsdfPdf;
        Vertex v;           // Store the local surface geometry:
        v.p = its.p;        //  - intersection point
        v.n = its.n;        //  - normal vector
        v.dgdu = its.dgdu; //  - tangent vector
        v.dgdv = its.dgdv; //  - binormal vector
        v.bsdf = bsdf;     //  - local BSDF
        // Ensure wi/wo are stored with regard to the requested type
        if (eye) {
            /* Convert incoming dir. into local coordinate system */
            v.wo = BSDF::toLocal(-ray.d, its);
            cumulative *= bsdf->sample(v.wo, v.wi, bsdfPdf);
            ray.d = BSDF::toWorld(v.wi, its); // New ray direction
            /* Multiply by the absolute value of the outgoing cosine
                and divide by the PDF value (importance sampling) */
            cumulative *= std::abs(BSDF::cosTheta(v.wi)) / bsdfPdf;
        } else {
            /* Convert incoming dir. into local coordinate system */
            v.wi = BSDF::toLocal(-ray.d, its);
            cumulative *= bsdf->sample(v.wi, v.wo, bsdfPdf);
            ray.d = BSDF::toWorld(v.wo, its); // New ray direction
            /* Multiply by the absolute value of the outgoing cosine
                and divide by the PDF value (importance sampling) */
            cumulative *= std::abs(BSDF::cosTheta(v.wo)) / bsdfPdf;
        }
        v.cumulative = cumulative;
        if (cumulative.isBlack()) // Absorption
            break;
        /* Store the surface interaction */
        vertices.push_back(v);
        ray.o = its.p; // New ray origin: current intersection point
    }
}
```

Listing 4.3: Random walk function

```
Spectrum BiDir::evalPath(
       const std::vector<Vertex> &eyePath,   // Eye and light path
       const std::vector<Vertex> &lightPath, // surface interactions
       int nEye, int nLight) const {          // Number of steps
                                              // on each path
    const Vertex &ev = eyePath[nEye-1];
    const Vertex &lv = lightPath[nLight-1];

    Spectrum L(1.0f); /* Start with an importance of 1.0 */

    /* Account for reflections on the eye and light paths */
    if (nEye > 1)
        L *= eyePath[nEye-2].cumulative;
    if (nLight > 1)
        L *= lightPath[nLight-2].cumulative;

    /* Calculate the geometric term */
    Vector etl = lv.p - ev.p;
    float lengthSquared = etl.lengthSquared();
    etl /= std::sqrt(lengthSquared);

    float geoTerm = absDot(etl, ev.n) * absDot(-etl, lv.n)
        / lengthSquared;

    /* Extremely close points cause numerical problems */
    if (lengthSquared < 0.05)
        return Spectrum(0.0f);

    /* Evaluate the BSDFs at the last light and eye path
       surface interaction points */
    L *= ev.bsdf->f(
        ev.wi, BSDF::toLocal(etl, ev.dgdu, ev.dgdv, ev.n)
    );

    L *= lv.bsdf->f(
        BSDF::toLocal(-etl, lv.dgdu, lv.dgdv, lv.n), lv.wo
    );

    L *= geoTerm; // Attenuate by the geometric term

    return L;
}
```

Listing 4.4: Path evaluation function

```
void BiDir::renderPaths(const Sample &eyeSample,
        const Ray &eyeRay, const Ray &lightRay,
        const Spectrum &Le) {
    Vector wi;
    Point onLight;

    /* Perform two random walks */
    randomWalk(eyeRay, m_eyePath, true);
    randomWalk(lightRay, m_lightPath, false);

    /* Implementation of cases (ii) and (iv) */
    for (unsigned int i=1; i<m_eyePath.size()+1; i++) {
        const Vertex &ev = m_eyePath[i-1];
        // Sample the light source from the current surface location
        Spectrum localLe = m_lights[0]->sample(ev.p, onLight, wi);
        // Convert the incident light dir. to the local coord. sys.
        Vector localWi = BSDF::toLocal(wi, ev.dgdu, ev.dgdv, ev.n);
        float cosTheta = std::abs(BSDF::cosTheta(localWi));
        if (V(ev.p, onLight) && cosTheta > 0) { // Visibility test
            /* Account for multiple reflections */
            if (i > 1)
                localLe *= m_eyePath[i-2].cumulative;
            /* Store the contribution on the film */
            m_film->addSample(eyeSample,
                localLe * ev.bsdf->f(ev.wo, localWi) * cosTheta *
                pathWeight(i, 0)
            );
        }
        /* Case (iv) */
        for (unsigned int j=1; j<m_lightPath.size()+1; j++) {
            const Vertex &lv = m_lightPath[j-1];
            if (V(ev.p, lv.p)) { // Mutual visibility test
                /* Store the contribution on the film */
                m_film->addSample(eyeSample, Le *
                    evalPath(m_eyePath, i, lightRay, m_lightPath, j)
                    * pathWeight(i, j)
                );
            }
        }
    }

    // Continued on the next page ..
```

Listing 4.5: Path rendering function (part 1)

```
    // BiDir::renderPaths continued ..

    /* Implementation of case (iii) */
    for (unsigned int j=1; j<m_lightPath.size()+1; j++) {
        const Vertex &lv = m_lightPath[j-1];

        if (V(lv.p, m_camera->getPosition())) { // Visibility test
            Spectrum localLe = Le;
            /* Calculate the exitant direction */
            Vector wo = m_camera->getPosition() - lv.p;
            float lengthSquared = wo.lengthSquared();
            wo /= std::sqrt(lengthSquared);

            /* Extremely close points cause numerical problems */
            if (lengthSquared < .05)
                continue;

            /* Convert exitant vector to the local coord. sys. */
            Vector localWo = BSDF::toLocal(
                wo, lv.dgdu, lv.dgdv, lv.n
            );

            /* Account for multiple reflections */
            if (j > 1)
                localLe *= m_lightPath[j-2].cumulative;

            localLe *= lv.bsdf->f(lv.wi, localWo)
                * std::abs(BSDF::cosTheta(localWo))
                / lengthSquared;

            /* Determine the position on the image plane
               and store the contribution if the position
               is valid */
            Sample s;
            if (m_camera->positionToSample(lv.p, s))
                m_film->addSample(s, localLe * pathWeight(0, j));
        }
    }
}
```

Listing 4.6: Path rendering function (part 2)

4.4 Results

Example renderings using the pure software implementation of the bidirectional path tracing algorithm showed very good results at 128 to 256 samples per pixel. The noise was considerably lower compared to path tracing renderings created with an equal amount of samples. The rendering time, however, was quite long – about 30 minutes were required to render a Cornell box scene at 512x512 without visible noise.

Caustics caused by specular transmission and refraction are usually notoriously hard to render – as expected, the algorithm was able to reproduce a caustic without problems. The result of the Cornell box scene and the image representations of the partial estimators can be seen in figures 4.2 and 4.3. Note the color bleeding on the walls and the light reflections on the ceiling and floor caused by the mirror object.

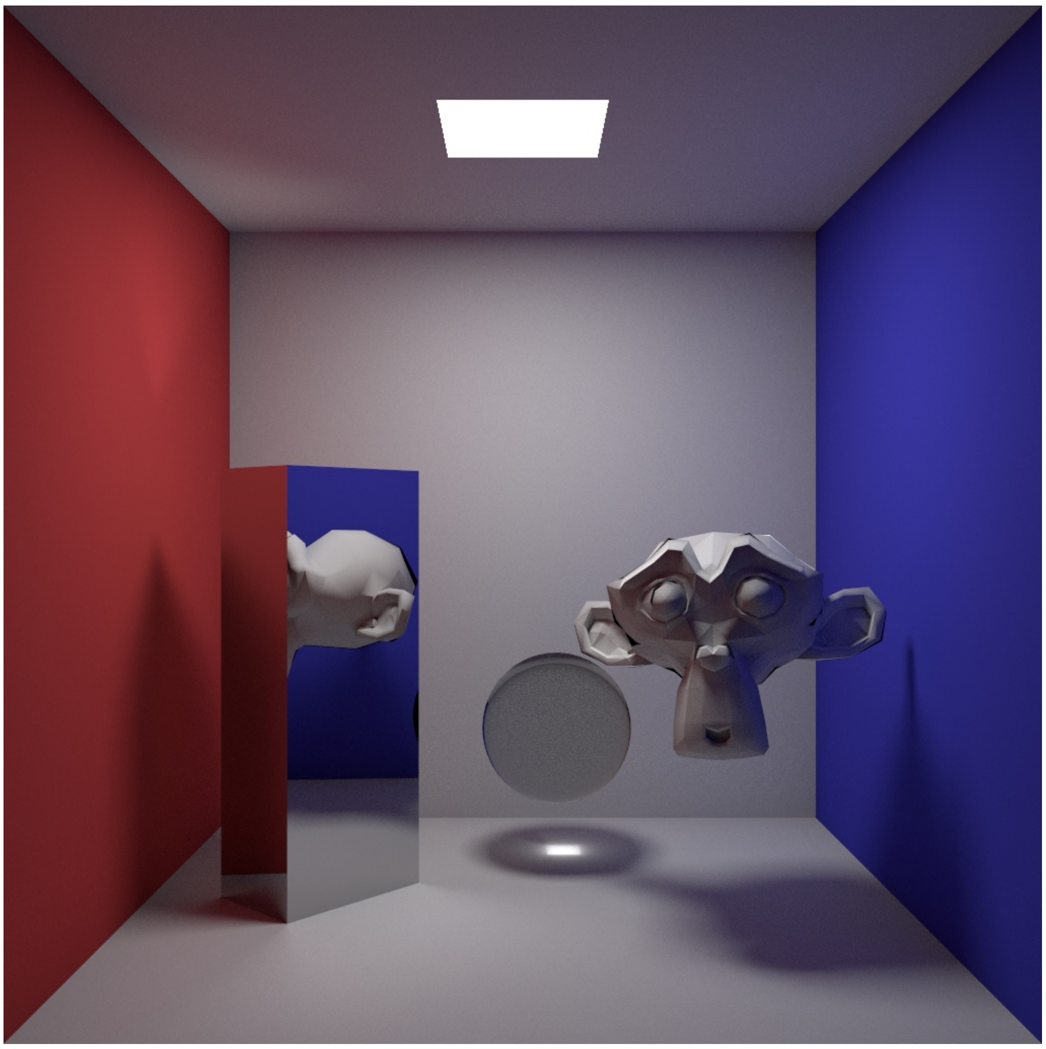
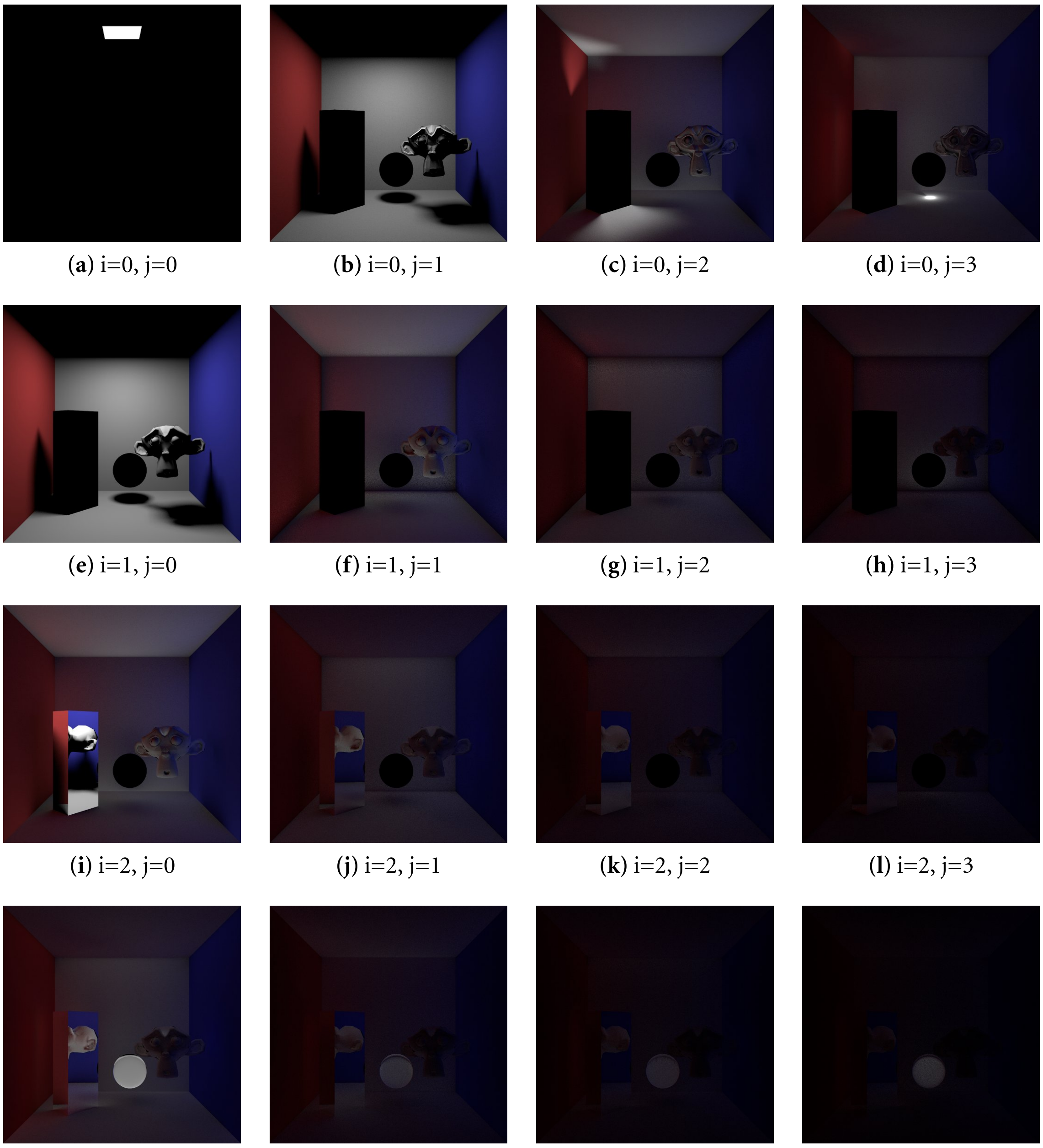


Figure 4.2: Cornell box rendering (256 samples, ray depth: 3). Monkey courtesy of the Blender project.



(a) i=0, j=0 **(b)** i=0, j=1 **(c)** i=0, j=2 **(d)** i=0, j=3

(e) i=1, j=0 **(f)** i=1, j=1 **(g)** i=1, j=2 **(h)** i=1, j=3

(i) i=2, j=0 **(j)** i=2, j=1 **(k)** i=2, j=2 **(l)** i=2, j=3

(m) i=3, j=0 **(n)** i=3, j=1 **(o)** i=3, j=2 **(p)** i=3, j=3

Figure 4.3: Cornell box rendering grouped by path configuration. Each image contains the paths generated by one of the partial estimators – "i" and "j" denote the eye and light path steps, respectively.

4.5 The bipartite mutual visibility query

Since $(n+1)^2$ mutual visibility checks have to be performed for a pair of paths of length n, these costs will eventually dominate the total computation time with increasing values of n. Therefore, this operation was chosen for acceleration using dedicated hardware. The implemented operation works as follows:

```
Bipartite-Mutual-Visibility-Query(X, Y, S, T)
for i ← 0 to S do
    for j ← 0 to T do
        check whether $x_i$ and $y_j$ are mutually visible
        store the result in bit vector
return bit vector
```

This operation has the highly beneficial property of transporting a computationally complex problem with minimal communication overhead – especially, when using long light and eye random walks. Only $6n$ floating-point values need to be transferred to the intersection co-processor which, in turn, sends a small result bit vector. On the other hand, such a request causes $(n+1)^2$ intersection tests on the co-processor. This makes it possible to befefit from such acceleration using only low-bandwith links.

5 Target architecture & design flow

A *Field-Programmable Gate Array* (FPGA) is a semiconductor device consisting of reconfigurable logic resources embedded within a reconfigurable interconnect. The high degree of programmability enables the implementation of almost any logic circuit, including recent developments such as fully asynchronous circuits. The size and speed improvements of modern FPGAs make them viable for very complex designs, especially in the domain of high performance computing, where the parallelism of their logic resources pays off compared to current general-purpose CPUs. This chapter discusses the FPGA architecture and design flow of Xilinx [13] parts, however, parts by other manufacturers are usually very similar in structure and some of the information provided here will also apply to them.

5.1 Logic resources

The *Configurable Logic Block* (CLB), is the main resource for the implementation of combinatorial and sequential circuits. It can be further subdivided into four *slices* and a *switch matrix*, as shown in figure 5.2. The switch matrix provides very fast local connectivity amongst the four slices in a CLB and somewhat slower connectivity to adjacent CLBs. Each slice (Figure 5.3) contains two registers and two 4-input *look-up tables* (LUTs), some of which can also be configured to operate as 16-bit RAM/ROM or as a shift register. Additionally, each slice contains dedicated logic for the creation of extremely fast ripple-carry [1] adders.

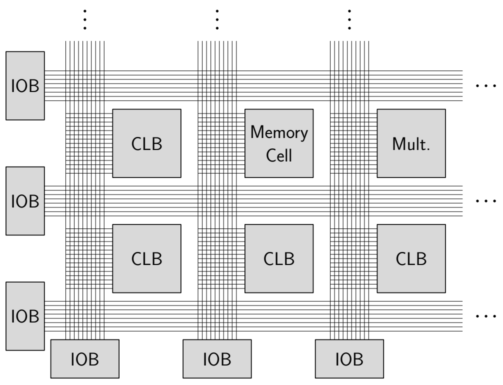


Figure 5.1: A generic FPGA array architecture – including I/O blocks, on-chip memory slices and multipliers embedded in a reconfigurable interconnect

[1]These adders consist of a cascade of 1-bit full adders, where the carry bits "ripple" from the last significant digit to the most significant digit.

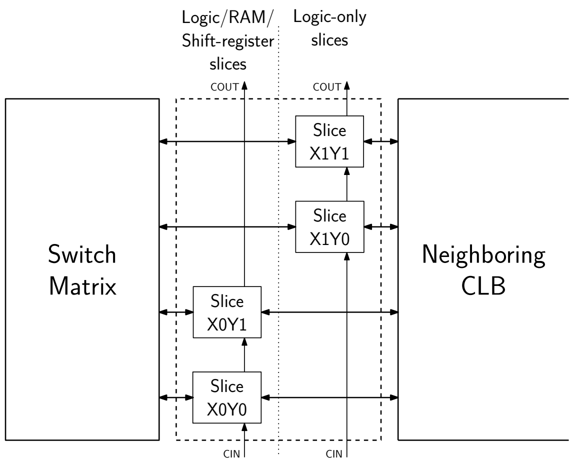


Figure 5.2: Layout of a Configurable Logic Block

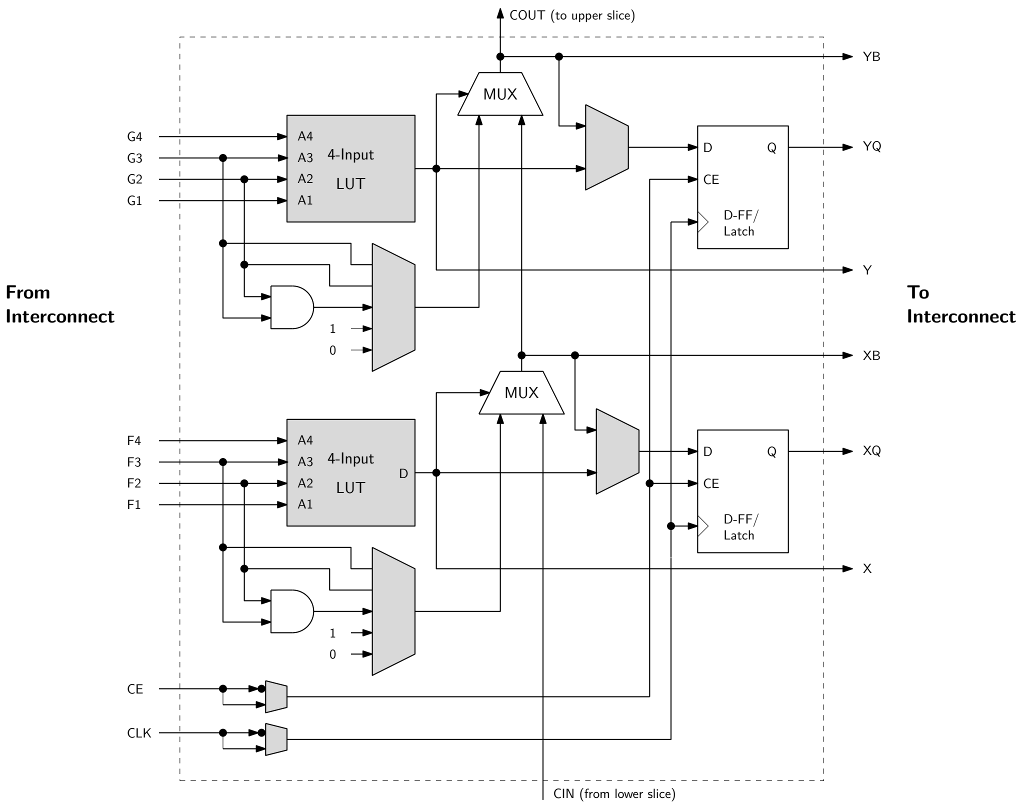


Figure 5.3: Simplified diagram of a Virtex-4 slice (wide function circuitry not drawn). The highlighted elements can be configured by a bitstream.

5.1.1 Look-up tables

A look-up table can implement an arbitrary boolean function with four inputs. The time required for the function generator to react to a change in inputs is very short ($T_{ILO} = 0.2ns$) and independent of the implemented function. By combining look-up tables with dedicated wide function multiplexers available in each slice (not drawn in figure 5.3), fast boolean functions with up to 8 inputs are possible.

5.1.2 Interconnect

Routing is one of the most complex tasks of FPGA design and this reflects in the abundance of interconnect types available to the developer: There are CLB-internal lines, direct lines, double lines, hex lines, long lines (Figure 5.4), each possessing individual electrical capacitance and skew characteristics. Fortunately, there is software which will perform automatic placement and routing of logic on the FPGA. However, it is necessary to be aware of skew, which might be introduced by inappropriate usage of the interconnect resources. *Skew* is the difference in path delays when a signal simultaneously travels to several different destinations. This is a particular problem when occurring on nets distributing clock pulses to thousands of flip-flops because it removes the synchronization that is so essential to any synchronous circuit. Thus, it has to be reduced at all costs.

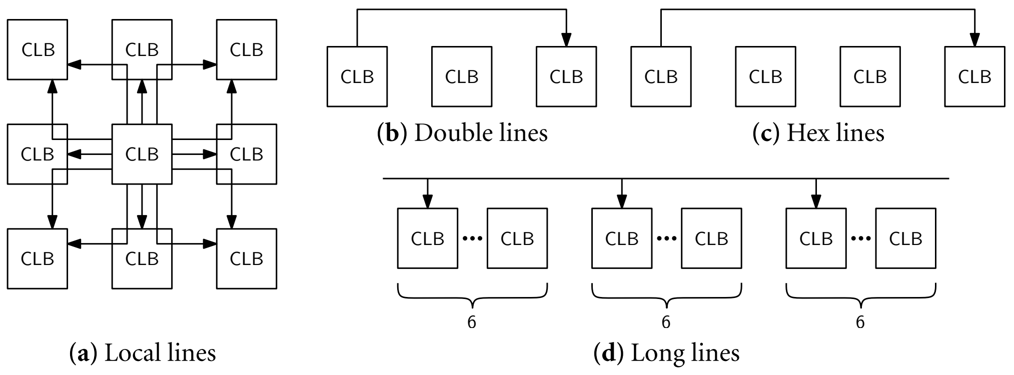


Figure 5.4: Hierarchical interconnect resources

5.1.3 Clocking

Since the most of today's logic circuits are synchronous designs, they have to be supported by an external clock source. Virtex-4 devices include an extensive number of systems to facilitate this task. A total of 32 dedicated low-skew clock distribution networks exist on the chip, a subset of which leads to the clock input of every logic resource on the FPGA. These networks can only be driven by special clock buffers, which ensures that a clock impulse can be delivered within hundreds of picoseconds and almost no skew.

Another important clocking component is the *Digital Clock Manager* (DCM), which is just a Xilinx-specific term for *delay-locked loop* (DLL). High-speed systems running at frequencies in excess of 100 Mhz frequently encounter synchronization anomalies as a result of this speed. Suppose two components C_1 and C_2 located on the same circuit board exchange data through a communications link. Additionally, C_1 forwards a clock signal to C_2, which C_2 uses to internally clock its synchronous logic resources. However, the copper traces connecting C_1 and C_2 have a capacitive value, which causes a propagation delay in the order of several nanoseconds when forwarding the clock signal. As a result, C_1 and C_2 receive the rising edge of the clock signal at different times and thus, are not properly synchronized from a system perspective.

This is where the DCM comes in. In order to remove this anomaly, the circuit board needs to provide a *feedback loop* trace leading from component C_1 to component C_2 and back again. The DCM will send a clock signal through the feedback loop and measure the delay caused by I/O drivers and capacitive values. Through observation of the clock signal, it is able to "anticipate" its future behavior and, using this information, constructs a predicted version of the clock signal. This signal is then forwarded to C_2 and again delayed by the trace capacitance. However, this time, the offset into the future and path delays cancel each other out, causing C_1 and C_2 to be in perfect synchronization.

5.1.4 DSP48 tiles

Newer Xilinx FPGAs include "DSP48" tiles, which basically consist of a 18×18 two's complement multiplier followed by an 48-bit sign-extended adder/subtracter/accumulator [14]. On the chip surface, these are arranged in vertical columns and use a dedicated 48-bit internal bus so that a tile can "cascade" data upwards to the next tile. The pipelining of input operands, intermediate products and the adder/subtracter-outputs is completely programmable. The arithmetic mode of operation of a tile can be changed dynamically using an opcode input. An optional 17-bit right shift makes it possible to perform any of these operations with an arbitrary precision.

5.1.5 Block RAM tiles

On the target FPGA, there is also an ample amount of 18Kb block RAM tiles, which can be configured to use "aspect ratios" from 512×36, to 16K×1. Each tile has two totally independent ports supporting full read and write access. Some very interesting applications are possible thanks to the dual-port capability — including fast direct–mapped caches and shared memory communication. Furthermore, the memory block can be "pre-configured" using the configuration bitstream and can thus be used as ROM or as a microsequencer.

5.2 Design flow

This section documents the creation of a tiny piece of hardware to illustrate the typical design flow. The circuit is a moore state machine, which reads a stream of bits and determines

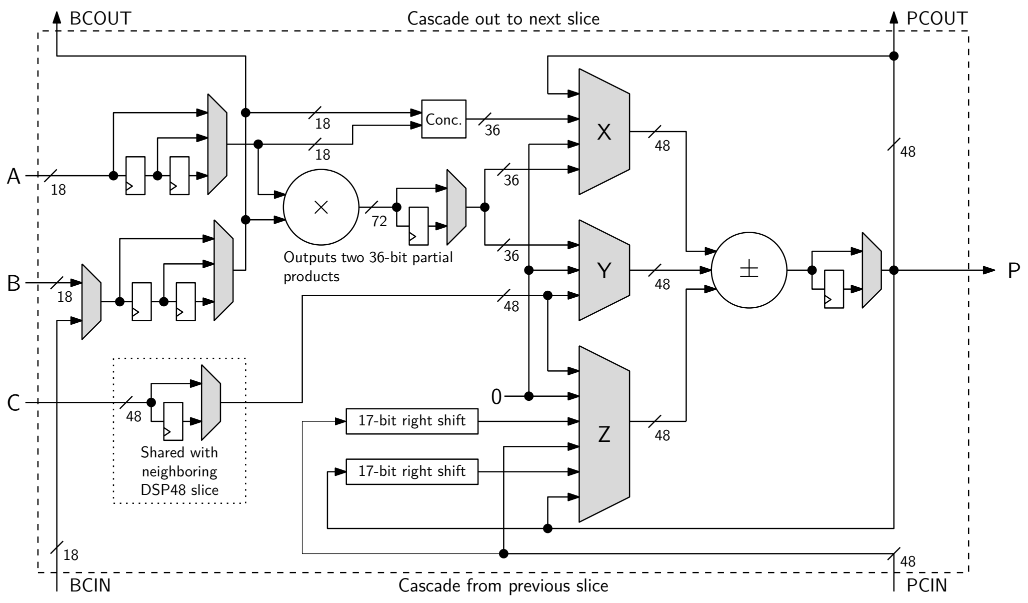

BCOUT
Cascade out to next slice
PCOUT
A
18
B
18
C
48
Conc.
18
18
36
36
36
72
48
Outputs two 36-bit partial products
X
Y
Z
48
48
48
48
±
P
0
17-bit right shift
17-bit right shift
Shared with neighboring DSP48 slice
18
BCIN
Cascade from previous slice
48
PCIN

Figure 5.5: DSP48 slice architecture as described in [14].

Device	CLB matrix	Slices	Total LUTs	Slice RAM (Kb)	Block RAM (Kb)	DSP48s
XC4VLX15	64×24	6,144	12,288	96	864	32
XC4VLX25	96×28	10,752	21,504	168	1,296	48
XC4VLX60 (∗)	128×52	26,624	53,248	416	2,880	64
XC4VLX100	192×64	49,125	98,394	768	4,320	96
XC4VLX200	192×116	89,088	178,176	1,392	6,048	96

Table 5.1: Logic resources available in current Virtex-4 devices [12].

whether the amount read so far contained an even or odd number of ones (parity).

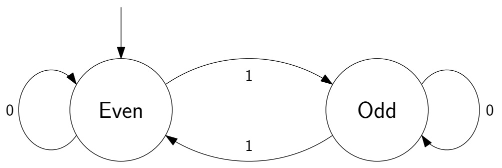


Figure 5.6: Diagram of the state machine

5.2.1 Logic design

Logic design is done using the *Very High Speed Integrated Hardware Description Language* (VHDL), a language commonly used for the simulation and synthesis of digital circuits. VHDL supports modularity, re-use and encourages the developer to partition his design into many interconnected *entities*. An entity possibly possesses multiple *architectures*. This can be compared to Java's interfaces and implementation classes. Exemplary code for the state machine example is provided in listings 5.1 and 5.2.

```
library ieee;                         -- Import std_logic data type
use ieee.std_logic_1164.all;          --  (corresponds to a single bit)

entity my_fsm is                      -- Declare the entity 'my_fsm'
    port (                            -- Interface to the outside world
        clk    : in std_logic;        -- Clock input
        rst    : in std_logic;        -- Synchronous reset input
        input  : in std_logic;        -- FSM input
        output : out std_logic        -- FSM output
    );
end my_fsm;
```

Listing 5.1: Entity declaration of the finite state machine

5.2.2 Synthesis and mapping

The synthesis step collects a number of VHDL-files and transforms this *behavioral* circuit description into an equivalent *logic* description — while the result has very little resemblance with the original source code, it is completely faithful to the specified functionality.

This device-independent synthesized logic description is then translated and mapped onto the available device resources (e.g. LUTs, flip flops). Figure 5.7 is the result of invoking the Xilinx tools on the provided state machine code.

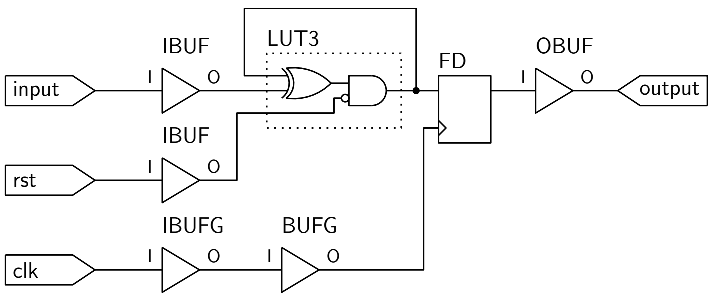


Figure 5.7: Synthesized parity calculation circuit

5.2.3 Placement and routing

The placement and routing steps continue with the results from synthesis and mapping. Furthermore, they need a so–called *constraints* file, which forces them to adhere to additional functional specifications such as operating frequency in MHz or more fine-grained timing requirements. Unfortunately, placement and routing of a complex circuit can take

```
architecture rtl of my_fsm is    -- Declare an architecture 'rtl'
    type state_t is              -- 'RTL'=Register Transfer Level
        (STATE_EVEN, STATE_ODD); -- New state data type
    signal state_r : state_t;    -- Declare the state register
begin
    output <= '0' when state_r = STATE_EVEN else '1';

    fsm_update: process (clk)    -- Sensitive to clock signal
    begin
        if rising_edge(clk) then -- When a rising clock pulse occurs
            if rst = '1' then    -- Synchronous reset detection
                state_r <= STATE_EVEN;
            else                 -- otherwise invoke the state
                case state_r is  -- transition function
                    when STATE_EVEN =>
                        if input = '1' then
                            state_r <= STATE_ODD;
                        end if;

                    when STATE_ODD =>
                        if input = '1' then
                            state_r <= STATE_EVEN;
                        end if;
                end case;
            end if;
        end if;
    end process fsm_update;
end rtl;
```

Listing 5.2: Architecture declaration of the finite state machine

from hours to days — dependent on how "strict" these constraints are. The reason is that the underlying problems are NP-hard and have to be solved heuristically. If the whole design is supposed to run at 100 Mhz, signals need to travel from flip-flop through combinatorial logic to flip-flop within 10 nanoseconds! Otherwise, the previous result is stored and erroneous circuit operation will ensue. As a consequence, these steps need to ensure that all components are placed appropriately (e.g. close enough on the chip surface) to satisfy the constraints.

While there are many third-party synthesis and mapping solutions, the final placement and routing has to be performed with proprietary tools made by the device manufacturer. As last part of the design flow, these tools produce a configuration *bitstream*, which can then be uploaded onto the target FPGA.

6 Building blocks

When developing a core of this complexity, it becomes essential to make use of resources already embedded in the FPGA fabric. This project was designed to fit into a Xilinx Virtex-4 LX60 FPGA and extensively uses its internal dual-ported memory blocks and DSP48 pipelined multipliers. For reasons of cost and simplicity, a USB 2.0 interface was designed to connect the intersection processor to a CPU. While a PCI or PCI Express interface would likely involve much less CPU overhead and latency, the USB 2.0 variant still allows fast data transfers and serves its purpose of demonstrating the performance of such a hardware acceleration device.

6.1 Larger multipliers

The 18-bit operand width limitation of the DSP48 multiply-accumulate blocks would usually severely limit their use. However, by cascading four of these multiplier units, it is possible to build a larger (35×35-bit) signed integer multiplier. The exact multiplication scheme can be derived using divide-and-conquer: Let a_i and b_i be the i-th digit of the two's complement representation of a and b such that:

$$a = -2^{34} \cdot a_{34} + \sum_{i=0}^{33} 2^i \cdot a_i \quad \text{and} \quad b = -2^{34} \cdot b_{34} + \sum_{i=0}^{33} 2^i \cdot b_i$$

By rearranging the summation order, the sum can be split up into the sum of a shifted, signed 18-bit value and an unsigned 17-bit value.

$$a = \underbrace{(-2^{17} \cdot a_{34} + \sum_{i=0}^{16} 2^i a_{i+17})}_{=:a_h} \cdot 2^{17} + \underbrace{\sum_{i=0}^{16} 2^i a_i}_{=:a_l} .$$

$$b = \underbrace{(-2^{17} \cdot b_{34} + \sum_{i=0}^{16} 2^i b_{i+17})}_{=:b_h} \cdot 2^{17} + \underbrace{\sum_{i=0}^{16} 2^i b_i}_{=:b_l} .$$

The multiplication can then be rewritten as:

$$\begin{aligned} a \cdot b &= (a_h \cdot 2^{17} + a_l)(b_h \cdot 2^{17} + b_l) \\ &= a_h b_h \cdot 2^{34} + 2^{17} \cdot a_h b_l + 2^{17} \cdot a_l b_h + a_l b_l \end{aligned}$$

The summands are now composed of 17-bit unsigned multiplications and 18-bit signed multiplications, which can be realized using the built-in DSP48-blocks.

6.2 Non-restoring integer divider

Of the four basic arithmetic operations, division is the slowest and most complex in terms of hardware. Even on modern processors such as the Pentium-4, division takes in excess of 40 clock cycles. Signed integer division is defined by the equation $z = d \cdot q + s$ and the constraint $\mathrm{sgn}(z) = \mathrm{sgn}(s)$, where z is the dividend, d is divisor, q is the quotient and s is the remainder.

A fully pipelined, non-restoring divider is used to realize the division operation occurring inside the kd-tree traversal and triangle intersection units. This core is characterized by a short critical path and moderate size. To clarify its operation, we will first examine the simpler, restoring divider: Both are digit recurrence schemes, generating one quotient bit during each cycle. Restoring division is very intuitive in that it is similar to how a human would perform division of two large numbers. If the trial difference of the shifted divisor and the partial remainder from the $n-1^{\text{th}}$ iteration is positive, the result is stored for the next iteration. Otherwise, the previous value is restored and a 0 quotient bit is generated.

```
RESTORING-DIVIDE(z, d)
r ← z
for i ← n−1 downto 0 do
  if r ≤ $2^i \cdot d$ then
    r ← r − $2^i \cdot d$
    q ← $2 \cdot q + 1$
  else
    q ← $2 \cdot q + 0$
  end
end
```

Listing 6.1: Restoring division algorithm

However, restoring division has two disadvantages: Because the decision whether or not to store the subtraction result depends on its sign, the timing-critical paths of the resulting logic are very long. Furthermore, additional sign conversion steps are required if the division core needs to handle two's-complement numbers.

The non-restoring division algorithm avoids these drawbacks using a *binary signed-digit* (BSD) number representation. Depending on the partial reminder and dividend signs, either an addition or a subtraction is performed during each cycle. The resulting quotient bits are, unlike restoring division, not in $\{0, 1\}$, but $\{-1, 1\}$ – as a result, this intermediate quotient is always an odd number. This method can be compared to adding weights to both sides of a scale until balance is reached. After the intermediate quotient bits in BSD form have been calculated, two additional correction cycles are required for two's-complement conversion and remainder calculation. A thorough review of various hardware division algorithms can be found in [7].

```
Nonrestoring-Divide(z, d)
r ← z
for i ← n−1 downto 0 do
  if sgn(r) ≠ sgn(d) then
    r ← r − $2^i \cdot$ d
    $q_i$ ← 1
  else
    r ← r + $2^i \cdot$ d
    $q_i$ ← −1
  end
end
```

Listing 6.2: Non-restoring division algorithm

6.3 Floating-point unit

During simulation runs, an implementation of the triangle intersector using 32-bit fixed-point arithmetic showed signs of serious numerical problems. When a distant ray is transformed into the unit triangle space of a very small triangle, overflows occur. The inability of fixed-point arithmetic to represent very large and very small numbers makes it unsuitable for this design.

Instead, a simple 24-bit floating point format was chosen. It is neither compatible with the IEEE 754 standard, nor does it support denormalized numbers or rounding modes. However, the simplicity reflects in a simple and efficient hardware implementation, which still achieves an amazing precision during triangle intersection. Since this design's floating point calculation results are only stored temporarily, no rounding error accumulation will occur.

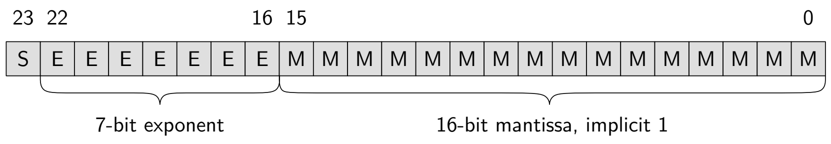

23 22
16 15
0
S E E E E E E E M M M M M M M M M M M M M M M M
7-bit exponent
16-bit mantissa, implicit 1

Figure 6.1: 24-bit floating-point format

6.3.1 Floating-point adder/subtracter

When comparing the integer versus floating-point complexity of the four basic arithmetic operations, the adder/subtracter incurs the greatest hardware overhead. The reason for this complexity is that the operands may need to be swapped and costly alignment pre- and post-shifts have to be performed. Given two floating point values $a = (-1)^{s_a} \cdot 2^{e_a} \cdot m_a$ and $b = (-1)^{s_b} \cdot 2^{e_b} \cdot m_b$, the core performs the following operations in sequence:

(i) Calculate the difference of the operand's exponents: $\delta \leftarrow |e_a - e_b|$.

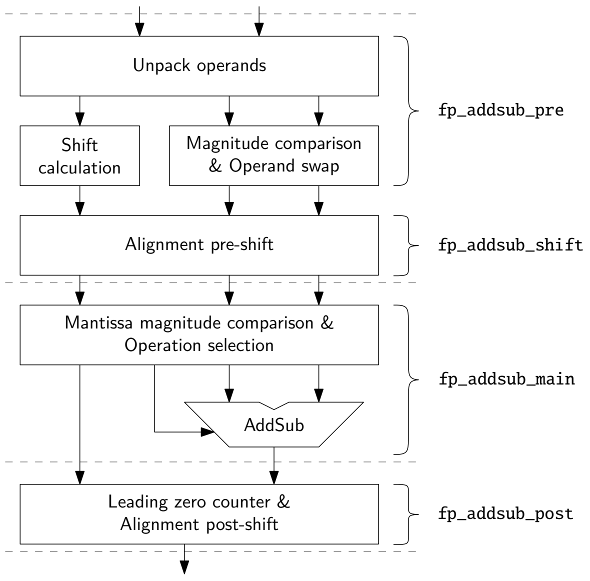

Unpack operands
Shift calculation
Magnitude comparison & Operand swap
fp_addsub_pre
Alignment pre-shift
fp_addsub_shift
Mantissa magnitude comparison & Operation selection
AddSub
fp_addsub_main
Leading zero counter & Alignment post-shift
fp_addsub_post

Figure 6.2: Block diagram of the floating-point adder/subtracter. Dashed lines denote pipeline stages.

(ii) Possibly swap the operands so that $a = \langle\text{Operand with large exponent}\rangle$ and $b = \langle\text{Operand with small Exponent}\rangle$.

(iii) Limit the alignment pre-shift: $\delta \leftarrow \max\{\delta, 18\}$. If the exponent difference is greater than 18, b can be considered zero.

(iv) Perform an alignment pre-shift of the operand with the smaller exponent: $m_b \leftarrow m_b \cdot 2^{-\delta}$

(v) Operation selection depending on the operand signs and the requested operation

(vi) Either $c \leftarrow a \pm b$ or $c \leftarrow b \pm a$, so that the mantissa c_m is non-negative.

(vii) Perform a normalization post-shift and correct the exponent

On Virtex4, steps 1-4 can be performed within 10 ns and form the first pipeline stage. Steps 5-6 and step 7 are implemented in pipeline stages 2 and 3, respectively.

6.3.2 Floating-point multiplier/divider units

The floating point multiplier is much simpler. Given a and b, it performs the following operations in sequence:

1. Calculate the result sign: $s_c \leftarrow s_a \oplus s_b$
2. Calculate an intermediate exponent: $e_c \leftarrow e_a + e_b$
3. Calculate an intermediate mantissa: $m_c \leftarrow m_a \cdot m_b$

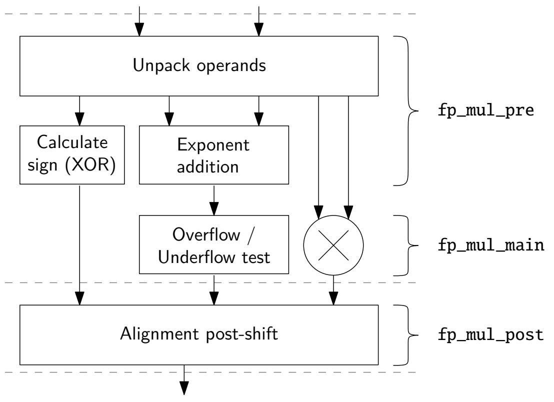


Figure 6.3: Block diagram of the floating-point multiplier

4. In case of an exponent underflow: $m_c \leftarrow 0$. In case of an overflow, flag the error.
5. The multiplication result of the two mantissas is in $[1, 4)$. If $m_c > 2$, an alignment shift has to be performed and the exponent needs to be incremented.

The floating point divider works analogous to the multiplier: Instead of adding the exponents, they are now subtracted and the alignment post-shift performs a shift in the other direction. The pipelined non-restoring divider previously mentioned is used by the divider core.

Core	4-Input LUTs	Flip-flops	SRL16[1]	DSP48
FP add/sub	376	139	8	0
FP multiplier	69	92	1	1
FP divider	533	509	29	0
FP comparator	66	49	0	0
Available	53248	53248	26624	64

[1] 16-bit linear shift register

Table 6.1: Implementation costs of the floating-point cores

6.4 Memory hierarchy

6.4.1 DDR SDRAM Controller

Intersect uses a custom, pipelined DDR SDRAM controller capable of achieving 3,2Gb/s (381 MiB/s) at 100 Mhz. It takes care of refresh cycles, command timeouts, data strobes, open rows and delay/skew issues. It supports read and write bursts of length 2 (16 bit width), thus requiring one command every clock cycle to sustain full read/write speed.

6.4.2 Caching

The project uses a 1024-line direct-mapped cache to speed up read accesses. Each cache line stores sixteen 24-bit words amounting to a total of 48 KB of addressable memory. Internally, the cache uses a series of dual ported RAM blocks to provide a single write port and a read port for each of the three intersection units.

After a cache miss occurs, a read request is sent to the cache miss handler state machine. The cache miss handler then instructs the DDR controller to bring in 16 consecutive words using a burst transfer and updates the cache memory, which takes about 25 clock cycles altogether. Since the cache uses no FIFO to store cache misses occurring while the state machine is busy, special care must be taken to avoid cases where several threads fight to load data into the same cache line, which gets overwritten just before each thread is ready to receive its requested data.

6.5 USB Interface

The USB interface makes use of a Cypress EZ-USB FX2 controller chip on the circuit board. This chip has been programmed with a firmware putting the device into FIFO mode. A simple state machine on the FPGA moves arriving/leaving data between the FX2's "slave" FIFO and internal FPGA FIFOs, which enables a theoretical peak performance of 480MBit/s. The firmware of the USB controller is stored on an external EEPROM and automatically loaded on power-up. On the FPGA side, there is also some endianness detection/conversion logic. As a consequence, the different possible hosts (e.g. Linux/i386, Linux/PowerPC) can use the intersection processor without spending any CPU cycles on endianness conversion.

6.5.1 Device driver

As for now, there is only a Linux device driver. It uses DMA and asynchronous I/O for high-speed data transfers. A special USB scheduler manages a list of ongoing requests to keep the USB bus saturated at all times. Up to sixteen simultaneous requests can be scheduled; this only works very well with the `ehci-hcd` host controller and current Linux versions (e.g. $\geq$ 2.6.17). The driver stores received packets in a 128-deep temporary packet buffer and combines them to reduce the number of system calls (and context switches) required to read data. The optimum transfer speed was achieved when scheduling up to eight bulk output USB requests and eight inbound interrupt transfers at the same time. To evaluate different modes of operation and chip configurations, a simple benchmark was designed. It measures the raw read/write performance without any additional processing on the FPGA's side. The device driver achieved 27 MiB/s sustained write bursts and 31 MiB/s sustained read bursts on a Pentium-4 PC running Linux 2.6.17 at 3 Ghz.

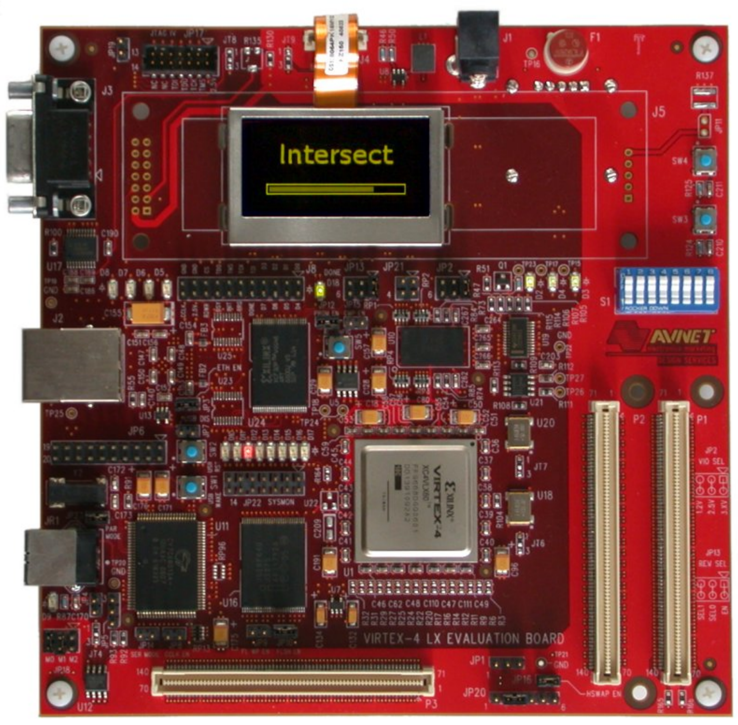


Figure 6.4: Implementation hardware

7 Intersection unit

Table 7.1 shows an excerpt of a profiler report generated while running the pure software implementation of the bidirectional path tracing algorithm. Since most time is spent doing ray-triangle intersection and kd-tree traversal, an acceleration device could drastically reduce the run-time of the algorithm.

% CPU time	Self seconds	Calls	Name
67.69	90.71	402295180	Triangle::rayIntersect
11.68	15.65	12519060	KDTree::rayIntersect
4.90	6.57	12540538	AABB::rayIntersect
4.80	6.43	12540538	KDTree::rayIntersect
2.00	2.68	1048576	BiDir::renderPaths
1.60	2.15	8347638	BiDir::V
1.35	1.81	3312701	BiDir::evalPath
1.13	1.52	9723139	coordinateSystem
1.05	1.41	2097152	BiDir::randomWalk
0.45	0.60	4192484	BSDF::toWorld
0.36	0.48	14799230	BSDF::toLocal
0.23	0.31	1	KDTree::buildTree

Table 7.1: A profiler report excerpt after rendering a scene using the pure-software implementation. Note that almost all CPU time is spent tracing rays!

7.1 Triangle intersection unit

Various intersection schemes were compared in the context of hardware implementation feasibility. Established algorithms such as the fast intersection test by Möller and Trumbore [6] are unfortunately too complex for this purpose. Instead, a modified of Arenberg's algorithm [9] was chosen, which is also used on the SaarCOR chip by the University of Saarland's graphics group.

7.1.1 Intersection algorithm

For high performance operation, the ray-triangle intersection unit uses a redundant triangle storage format, which first has to be generated from the raw triangle data. This requires two steps for every triangle : First, an affine transformation is created to transform the unit triangle $\{(0,0,0),\ (1,0,0),\ (0,1,0)\}$ onto the specified triangle. During the second step, the affine transform's inverse is calculated and converted to the processor's 24-bit floating-point format. This pre-computation speeds up the actual intersection calculations and reduces hardware complexity by a great deal.

Let n be the normal vector of the triangle defined by $A, B, C \in \mathbb{R}^3$.

$$n := \frac{(B - A) \times (C - A)}{\|(B - A) \times (C - A)\|}.$$

The affinity $T : \mathbb{R}^3 \to \mathbb{R}^3$ maps the unit triangle onto $\triangle ABC$ and maps $(0, 0, 1)$ onto the orthogonal vector n.

$$T := \begin{pmatrix} B_x - A_x & C_x - A_x & n_x \\ B_y - A_y & C_y - A_y & n_y \\ B_z - A_z & C_z - A_z & n_z \end{pmatrix} + A$$

If the triangle is well-formed (i.e. the points are not collinear), the normal vector n completes $\{B - A,\ C - A\}$ to a base of $\mathbb{R}^3$. Thus, T is an isomorphism and its inverse can be calculated – for example using the Gauss-Jordan algorithm.

All that is left to be done by the hardware in order to calculate an intersection point is to translate the ray into unit triangle space with the help of the inverse affine transformation and then perform the intersection there, which is now a much easier problem.

```
INTERSECT-TRIANGLE($T^{-1}, O, D$)
$O' \leftarrow T^{-1} \cdot O$    // Affine transform
$D' \leftarrow T^{-1} \cdot D$    // Linear transform
$t \leftarrow -O'_z / D'_z$
$u \leftarrow O'_x + D'_x \cdot t$
$v \leftarrow O'_y + D'_y \cdot t$

if $t \geq 0$ and $u \geq 0$ and $v \geq 0$ and $u + v \leq 1$ then
   return $(t, u, v)$
else
   return nil
```

Listing 7.1: Ray-triangle intersection pseudocode

7.1.2 Hardware implementation

A diagram of the ray-triangle intersection data-path can be found in figure 7.1. This design utilizes a massive amount of parallelism by unwrapping and pipelining the whole calculation. This results in the use of 20 fp24-multipliers, 18 fp24-adders, one fully pipelined fp24-divider, and many register stages. A consequence of the pipelined design is that a triangle intersection has a latency of thirty clock cycles. However, since up to thirty triangle intersection requests can be simultaneously contained inside the pipeline, one request is completed during each clock cycle, giving the unit a throughput of 100M Triangles per second at 100Mhz.

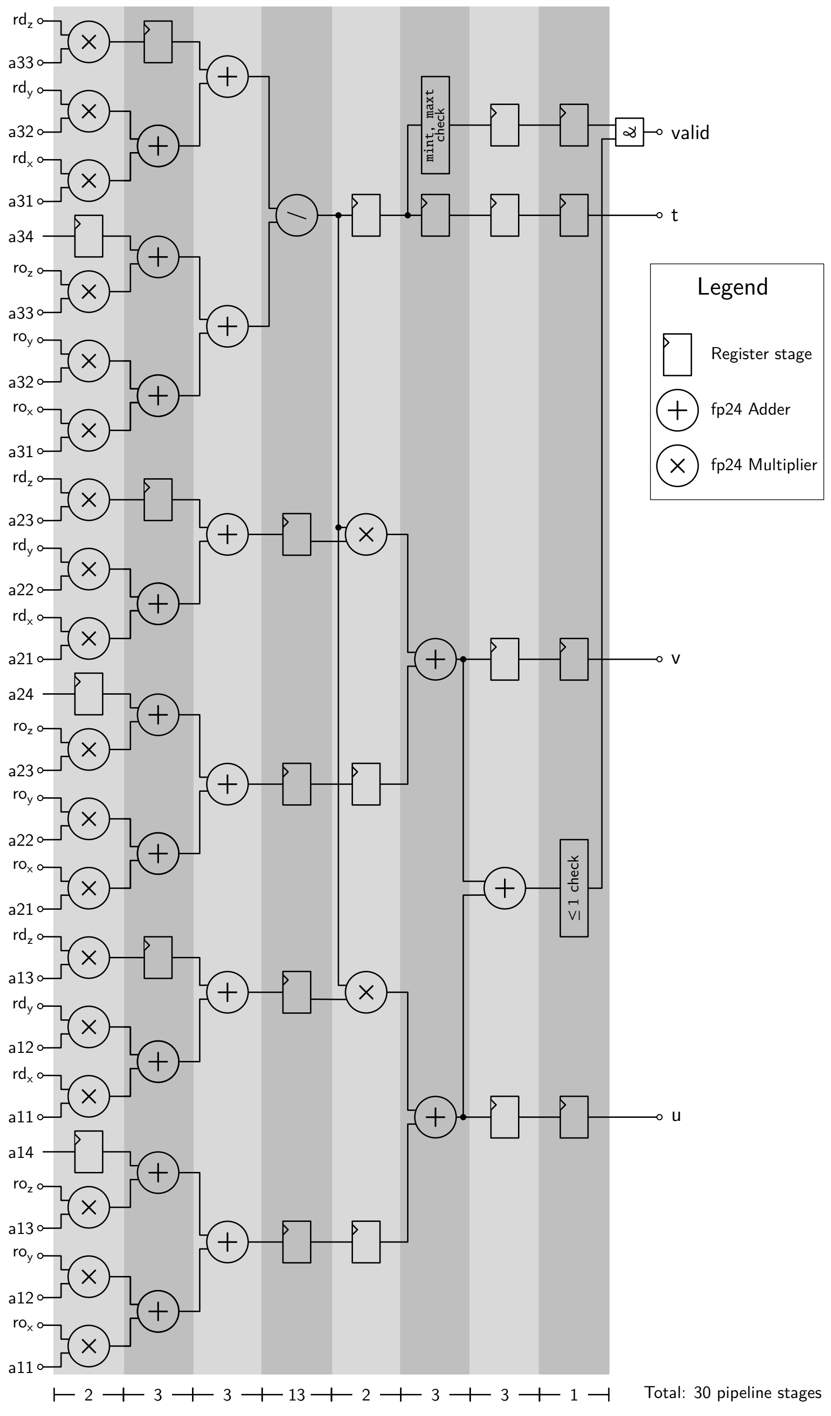


Figure 7.1: Ray-triangle intersection data path

7.2 kd-tree intersection unit

A *kd-tree* is the geometric equivalent of a binary search tree. It is a data structure frequently used to organize points or surfaces in a k-dimensional space. In comparison to the well-known quadtree or octree data structures, kd-trees benefit from their simplicity and good cache behavior, giving them better overall performance. A kd-tree is created by recursively partitioning the scene geometry using a generated hyperplane. In contrast to the BSP-tree, the hyperplanes are always axis-aligned, therefore the only information needed to uniquely characterize a hyperplane is an axis index and a single coordinate. kd-trees that are used to organize geometry usually store all primitives inside the tree's leaf nodes.

7.2.1 Intersection algorithm

kd-trees are very efficient data structures for elementary operations such as nearest-neighbor queries or ray intersections. Listing 7.2 contains some pseudo-code to illustrate how the partitioning information makes the intersection operation very efficient. Only triangles in

```
TRAVERSE-TREE(node, ray, mint, maxt)
if node.isLeaf() then
   return the closest intersection in node.triangles or $-1$
if ray.d[node.axis] $= 0$ then
   noSplitIntersection = true // Corner case: The ray is parallel to the intersection plane
else // Calculate the distance along the ray to the ray−plane intersection
   splitIntersection = (node.splitPos − ray.o[node.axis]) / ray.d[node.axis]
if ray.o[node.axis] $\leq$ node.splitPos then // Determine which cell contains the ray's origin
   near = node.left
   far = node.right
else
   near = node.right
   far = node.left
if noSplitIntersection or splitIntersection < 0 or splitIntersection >= maxt then
   return TRAVERSE-TREE(near, ray, mint, maxt) // Whole interval on near cell
else if splitIntersection $\leq$ mint then
   return TRAVERSE-TREE(far, ray, mint, maxt) // Whole interval on far cell
else // Traverse both cells in the proper order
   t $\leftarrow$ TRAVERSE-TREE(near, ray, mint, splitIntersection)
   if t >= 0 then   // An intersection has been found in the near cell, the far
      return t      // cell does not need to be checked.
   return TRAVERSE-TREE(far, ray, splitIntersection, maxt)
```

Listing 7.2: kd-tree traversal pseudocode

leaf nodes, which are actually encountered by the ray, are checked for intersections. Also, once the closest intersection in a node has been found, the evaluation can be stopped since any intersection in other nodes will be farther away. The `mint` and `maxt` parameters specify the considered search interval along the ray.

7.2.2 Generating kd-trees

The kd-tree construction algorithm is based on the *Surface Area Heuristic*, which is explained in detail in [11]. At each recursion level, a split plane is chosen using greedy selection. It can be proven that such a "perfect" greedy split plane passes through a triangle vertex, restricting the set of possible candidates to a finite subset.

Given the set of candidate planes, the algorithm tries to calculate the overall cost of intersecting arbitrary rays against each of the possible trees resulting from the partitioning operation. It then selects the split plane with the lowest cost function value. The cost function is defined as

$$c(A,B) := c_{\mathrm{kd}} + c_{\mathrm{tri}} \cdot (p_A \cdot n_A + p_B \cdot n_B)$$

where c_{tri} and c_{kd} are the costs of intersecting a single triangle and performing one traversal step, respectively. n_A and n_B are the number of triangles contained on each side of the partition (A,B) – overlapping triangles are counted twice. Finally, the conditional probabilities p_A and p_B that, given an intersection of the parent node, A or B are also intersected, are calculated using the following approximation:

$$p_A = \frac{S_A}{S} \quad \text{and} \quad p_B = \frac{S_B}{S}$$

where S is the surface area of the parent node and S_A, S_B are the surface areas of A and B, respectively. This heuristic generally produces very good results. It can be proven that the algorithmic average case complexity of intersecting a ray against a tree containing n primitives is in $O(\log n)$. Separate trees are built for the host CPU and intersection processor, since the relation between intersection and traversal costs is so fundamentally different amongst the two.

7.2.3 Hardware implementation

Designing a kd-tree unit for hardware implementation is a challenge because of the complex control flow and stack requirements. This project used a bottom-up approach: first, flat data-paths were developed for the triangle and kd-tree intersection units. Later, these were embedded into a control unit handling stack accesses and control flow.

Data path

The kd-tree data path (Figure 7.2) is a straight-forward implementation of the function body in listing 7.2. Again, there is one expensive division step, which requires 13 pipeline stages and makes up a big part of this unit's hardware consumption. Since the length of 19 stages is much smaller than the triangle unit's 30 stages, 11 empty register stages are appended so that the results of both units are available at the same time. This may be seen as a waste of available FPGA resources – however, such register stages can be implemented very cheaply using the built-in `SRL16` linear shift registers.

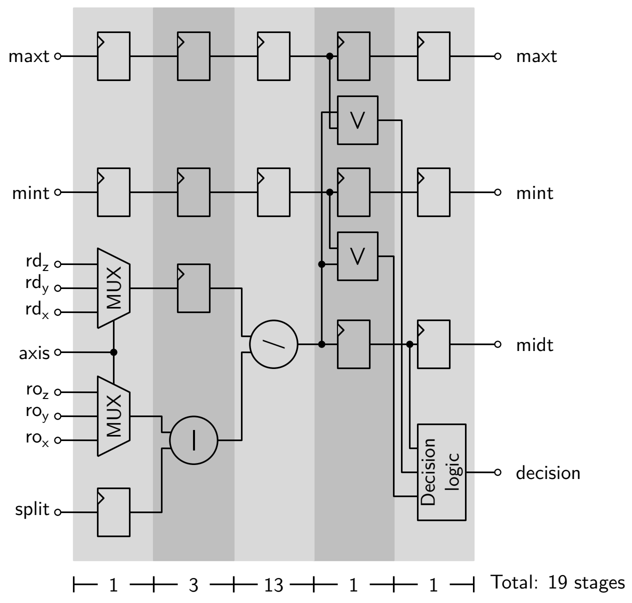


Figure 7.2: kd-tree traversal data path

Control unit

The final core has a pipeline length of 32 stages and embeds the triangle and kd-tree intersection data paths. Each pipeline stage has an associated thread ID between 0 and 31, which is used to ensure that each intersection thread only sees its own, private stack memory area. The stack was implemented in a way such that the top item is always kept inside registers and the remaining data is stored inside memory blocks.

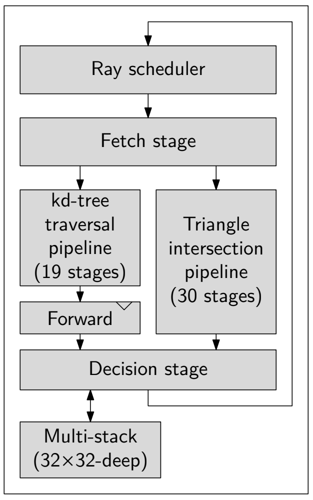

Ray scheduler
Fetch stage
kd-tree traversal pipeline (19 stages)
Triangle intersection pipeline (30 stages)
Forward
Decision stage
Multi-stack (32×32-deep)

Figure 7.3: kd-tree control unit

When the *decision* stage pushes a value onto the stack, the registered top item is replaced with new data and the old value is written out into the private stack memory area. When the decision stage pops an item from the stack, the top value is overwritten and will be available during the next iteration. This all works because only one push or pop operation can occur per thread per iteration. Here, the term iteration specifies the event of a single thread passing through pipeline stages 1 to 30. The VHDL stack data type declaration is shown in listing 7.3. The data type in listing 7.4 describes the complete state of an intersection thread and exists as register storage inside each pipeline stage. This creates a massive amount of registers but, again, these registers can be cheaply implemented using linear shift registers.

```vhdl
type stack_t is record
    addr    : memptr_t; -- Node address in memory
    mint    : fpval;    -- Start of the checked interval
    maxt    : fpval;    -- End of the checked interval
end record;
```

Listing 7.3: Control unit – stack data type

```vhdl
type intersection_t is record
    id       : thread_id; -- Thread id
    reqid    : reqid_t;   -- Request id
    top      : stack_t;   -- Top of the stack
    valid    : boolean;   -- Is this stage's data valid or a bubble?
    shadow   : boolean;   -- Does this stage contain a shadow ray?
    fetch_ok : boolean;   -- Was the data fetch sucessful?
    kd       : boolean;   -- Does the addr. point to a kd-tree node?
    sp       : thraddr_t; -- Stack pointer
    ro       : fpval3;    -- Ray origin
    rd       : fpval3;    -- Ray direction
    u        : fpval;     -- U coord. of the closest intersection
    v        : fpval;     -- V coord. of the closest intersection
    hit      : boolean;   -- Was there an intersection so far?
    tri_idx  : triidx_t;  -- Index of the current triangle
    best_idx : triidx_t;  -- Index of the best triangle
end record;
```

Listing 7.4: Control unit – pipeline stage data type

The kd-tree control unit performs the following operations in each stage:

(i) **Schedule:**[1]

- If there is an active thread in the last pipeline stage, schedule it. This operation basically copies all registers from the last pipeline stage.

[1]This is technically not a pipeline stage because the schedule & fetch operations are performed within the same clock cycle – however, it is listed separately here for clarity.

- If the last stage contains a bubble and the input queue contains a ray to be intersected, schedule it. This operation initializes the thread's stack pointer to zero and pushes the ray and the memory address `0x000000`[2] on top of the stack. No stack memory access is actually performed since the top was previously unused.

(ii) **Fetch**:

- Fetch 16×24 bits = 48 bytes from the memory address specified by the top of the stack using the direct-mapped cache. 48 bytes can contain one triangle or up to four kd-tree nodes.
- If there is a cache miss, notify the cache and ignore this thread for the current iteration by setting the decision stage to read-only when the thread arrives there. This behavior repeats until the stack has finally brought in the requested cache line.

(iii) **Intersect or traverse** (30 stages):

- If the fetched memory block contains a triangle, check for an intersection with the thread's ray along the current search interval.
- If the top item references a kd-tree node, determine the order in which the child nodes should be traversed, and check for corner cases.

(iv) **Decision**:

- If the previously performed memory fetch of this thread was unsuccessful, do nothing.
- If the top item of the stack referenced a kd-tree node, push the far node onto the stack and overwrite the top of the stack with the near node. The search interval is also reduced for both nodes. If a corner case occurred and one of the nodes does not need to be checked at all, only overwrite the top of the stack.
- If the top item of the stack referenced the *last* triangle of a leaf kd-tree node, there are several possibilities:
  - If the remaining stack is empty, the search is done and the core announces whether an intersection has been found so far.
  - If the remaining stack is not empty, the search is only terminated if an intersection has been found so far. If this is not the case, a node is popped from the stack memory and the search continues with the next iteration.
- If the top item of the stack referenced a triangle of a leaf kd-tree node – but not the last one, there are also several possibilities:
  - If no intersection was found, the top of the stack is changed to point to the next triangle of the leaf node.

[2]The root of the kd-tree is located at this position in memory

- If an intersection was found and the thread does not contain a shadow ray, the core checks whether this intersection is closer than all previous intersection. The t, u and v coordinates are stored as well as the triangle index, if this is found to be true.
- If an intersection was found and the thread contains a shadow ray, the search is aborted because an obstruction has been found.

Table 7.2 lists the resource requirements of the presented VHDL cores in terms of Virtex-4 resources. As expected, the triangle unit uses a relatively large amount of logic resources, while the kd-tree control unit uses mostly storage elements. Because of the core's size, three intersection units are included on the final FPGA version, which allows 96 rays to be processed in parallel with a peak performance of 300M intersection operations per second at 100Mhz.

Core	4-Input LUTs	Flip-flops	SRL16[1]	DSP48[2]	RAMB16[3]
Triangle unit	8676	3221	606	20	0
kd-tree data path	1392	949	206	0	0
kd-tree control	3625	8465	2788	0	4
Combined	13693 (26%)	12635 (24%)	3600 (14%)	20 (31%)	4 (3%)
Available	53248	53248	26624	64	160

[1] 16-bit linear shift register
[2] 18-bit pipelined multiplier
[3] 16kbit dual-port memory block

Table 7.2: Implementation costs of the intersection units

7.3 Bidirectional path-tracing frontend

The bidirectional path-tracing frontend is located between the USB interface and the intersection cores. It reads commands sent from the host CPU and executes them. This core automatically performs fp32 to fp24 conversion and vice versa to free the host CPU from this burden. Commands carry a request identification number, which is included when sending the response back to the CPU. The rationale is that responses may not be received in the order, in which the request were previously sent. When using a software pipeline on the host CPU, this information is needed to associate responses with the correct data.

This core also implements the *bipartite mutual visibility query* operation. When the CPU sends such an operation to the FPGA, the FPGA first expects the three-dimensional coordinates of the eye path surface interactions followed by the light-path surface interactions. The core issues shadow ray intersection requests for each possible pair (e_i, l_j) and sends visibility results back as soon as they exit the pipeline.

8 Software

The design of the *beam* global illumination renderer was strongly influenced by the book Physically Based Rendering [8] by Matt Pharr and Greg Humphreys. beam uses the same plugin-based approach, where almost anything is an external plugin: sampling techniques, light types, camera types and BRDFs are loaded on demand as the scene references them. The plugin interfaces are specified in a very genereral way so that almost any technique can be implemented without a change in the rendering core. beam uses a simple XML-based scene format – an example can be seen in listing 8.1.

One additional feature is the use of the EXR high dynamic-range format to store image data. When images turn out too bright or too dark after a long rendering time, they can be re-developed using a different exposure time.

8.1 Software pipeline

One major difference, however, is the main rendering loop. Since the rendering task now involves communication and external processing latencies, a software pipeline using 4 threads of execution was introduced. These are as follows:

(i) **Ray generator**: This thread uses a custom sampling technique, a light source, and a camera to sample the start rays of a pair of random walks. In an additional step, these random walks are continued until a maximal path length is reached or an absorption event occurs.

(ii) **Write thread**: This thread receives a packet of rays from the ray generator and sends a mutual visibility query to the FPGA.

(iii) **Read thread**: This thread receives query responses from the FPGA, writes the results to the affected packets, and then passes them to the processor thread.

(iv) **Processor thread**: The processor thread receives packets containing a number of random walks and visibility information. The only thing left to be done is to write the contribution of each possible path onto the camera's film. Afterwards, the packet is recycled and given back to the ray generator.

The write and read threads have an elevated priority to get data into and out of the FPGA at the earliest possible time. The pipeline works quite efficiently when many packets are inserted at the start. This approach keeps both the host CPU and the FPGA busy and mitigates the effects of external processing latencies.

```xml
<scene>
    <!-- Specifies a triangle mesh using the OBJ file loader -->
    <trimesh type="obj">
        <string name="filename" value="monkey.obj"/>

        <!-- Sets the surface properties of the monkey mesh -->
        <bsdf type="blinnphong">
            <float name="kd" value="0.7"/>
            <float name="ks" value="0.0"/>
        </bsdf>
    </trimesh>

    <!-- Instantiates a point light source -->
    <light type="point">
        <transform name="lightToWorld"> <!-- Affine transform. -->
            <translate x="0" y="1" z="0"/>
        </transform>
        <spectrum name="intensity" value="30,␣30,␣30"/>
    </light>

    <!-- Instantiates a perspective -->
    <camera type="perspective">
        <transform name="cameraToWorld">
            <translate x="0" y="0" z="2.8"/>
        </transform>
        <float name="fov" value="50"/>

        <!-- The camera uses the stratified sampling technique -->
        <sampler type="stratified">
            <integer name="xSamples" value="2"/>
            <integer name="ySamples" value="2"/>
            <boolean name="jitter" value="true"/>
        </sampler>

        <!-- The camera uses a high-dynamic range film -->
        <film type="exrfilm">
            <integer name="xRes" value="512"/>
            <integer name="yRes" value="512"/>
        </film>
    </camera>
</scene>
```

Listing 8.1: A scene using beam's XML-based scene description language

9 Results

During the course of this project, three components were planned and developed: a global illumination renderer, a linux kernel driver, and an intersection processor architecture. Various building blocks had to be designed in order to facilitate its development. Finally, the presented intersection architecture was validated and synthesized for the target hardware. As of this writing, the implementation of the co-processor works very reliable – although it has a series of drawbacks, which made it impossible to exploit the full advantage of the acceleration. These disadvantages are mostly related to the evaluation board's hardware and are listed below:

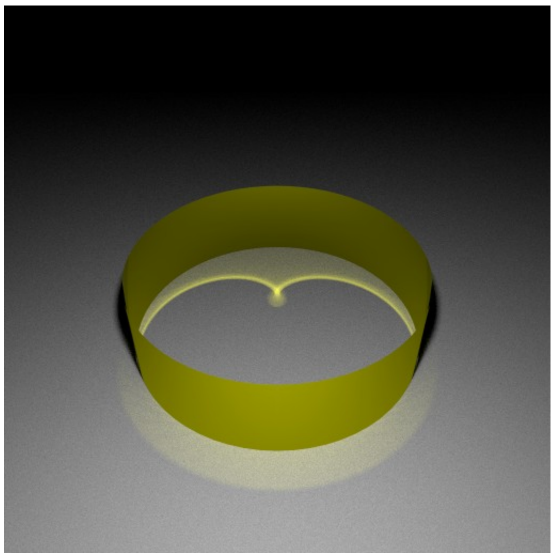
(i) **USB 2.0 Link**: The USB connection proved to be a significant bottleneck. Even though the implemented operation was very efficient in terms of required data transfer, the FPGA usage was only about 5 percent under full USB load. Gigabytes of visibility information had to be transferred even for the most simple of scenes. Despite this severe bottleneck, a speedup of about 30% was measurable when offloading the visibility computation.

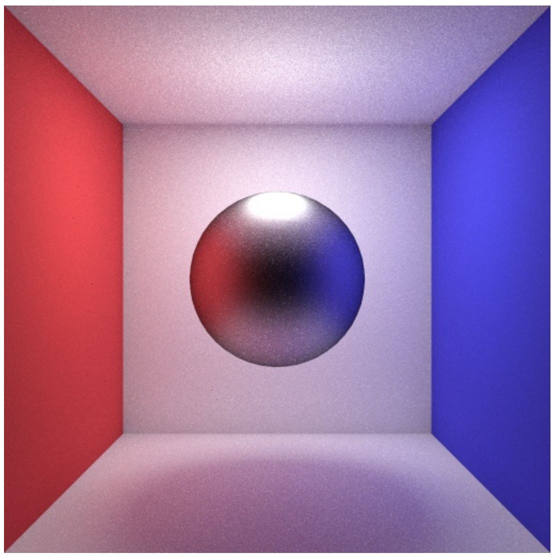
(ii) **Memory bandwidth**: The memory connection of the used FPGA board had a bandwidth of 381 MiB/s and was only 16 bits wide, which is tiny compared to the 6,4 GiB/s of bandwidth and 128 bits of connectivity available on commonly used DDR2 memory modules.

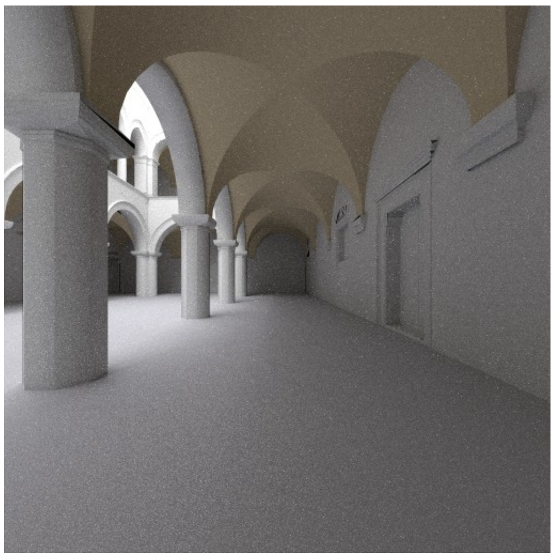
(iii) **Cache size**: The 1024-line cache proved to be too small to achieve good performance when rendering large scenes. This was especially true for the Sponza Atrium test scene, which took up almost all of the 32 MB of on-board memory. Cache hit rates below 10% made the use of a cache practically useless.

With the benefit of hindsight, a different board configuration would have eliminated these problems. A PCI Express board with an integrated DDR2 memory module socket would have eliminated the bandwidth problems. Furthermore, a FPGA with more on-chip SRAM resources or a series of on-board SRAM chips would have reduced the cache thrashing when using very large scenes.

Since the project is written in generic VHDL, it can easily be ported to different target hardware. The following section analyzes the projected performance of the co-processor when the bandwidth and cache issues are not the limiting factors. While these conditions are idealized, they are realistic in conjunction with the proposed target hardware and show what could be done with proper funding.


(a) Rendering of a caustic


(b) Glossy reflection


(c) Sponza Atrium

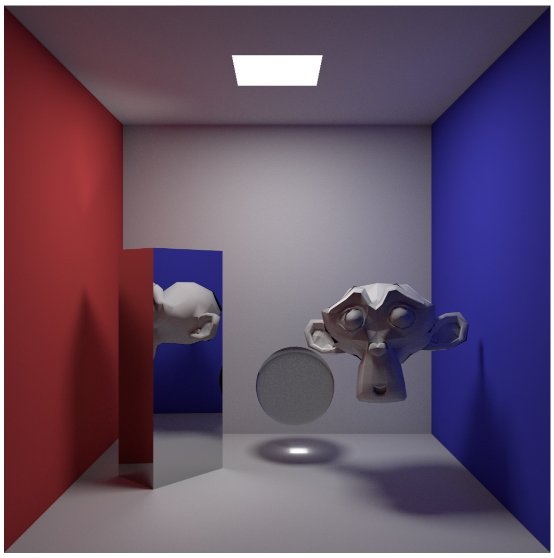

(d) Cornell Box

Figure 9.1: Renderings of the used test scenes

9.1 Projected performance

When using a 128-bit memory connection using double data rate I/O, a 384-bit fetch takes only two clock cycles. To calculate the expected memory access timeout, the worst case was assumed: Every memory access references an inactive DDR memory row, which results in a "pre-charge" and "activate" command including two no-operation cycles before actually being able to access the data. Thus, a total of six clock cycles is needed for any external memory access. Assuming a cache hit ratio of 90%, the expected memory access latency is

$$0.9 \cdot 1 + 0.1 \cdot 6 = 1.5 \,\text{cycles}$$

To calculate the projected performance, the average number of kd-tree node and triangle data accesses per intersection operation were measured (Table 9.1).

Scene name	Number of triangles	Avg. kd-tree node accesses	Avg. triangle data accesses
Caustic	130	6	4
Cornell Box	2,270	10	9
Sponza Atrium	76,136	41	29

Table 9.1: Average number of accesses per intersection operation

These values were then multiplied by the memory access latency to calculate the average number of clock cycles required for an intersection operation in these scenes. Based on these numbers, the maximum number of intersections per second at 100Mhz was calculated for different core configurations (Table 9.2).

Scene name	Clock cycles per intersection	1 Core	3 Cores	8 Cores
Caustic	16.5	6,060,606	18,181,818	48,484,848
Cornell Box	28.5	3,508,771	10,526,316	28,070,175
Sponza Atrium	105	952,380	2,857,142	7,619,047

Table 9.2: Intersections per second using different core configurations

Further improvements could be achieved by increasing the core's clock speed beyond 100 Mhz – for example by performing critical path optimizations or switching to a higher speed grade FPGA device. Implementing this core using an ASIC[1] process would also dramatically increase its speed.

9.2 Conclusion and future work

A highly space-efficient processor architecture using floating point arithmetic was presented. The recursive kd-tree intersection code was mapped into an efficient, pipelined hardware design. A multi-core version of this architecture was implemented on an FPGA

[1] *Application-specific integrated circuit* - A custom, highly efficient, non-reconfigurable integrated circuit.

and both verified using test-benches and a specially designed global illumination renderer. While the performance of the prototype did not meet the initial expectations, much improvement will be possible using more adequate evaluation hardware. A custom floating point core design made it possible to fit the exceptional number of 126 floating point cores onto a single FPGA. The designed architecture proved to be highly efficient in the number of required clock cycles per intersection operation.

The single most important future work will be to acquire more adequate evaluation hardware, and to adapt the project to it. With some effort, it might be possible to design a hardware implementation of the ray generator, which runs directly on the FPGA – thus significantly lowering the communication bandwidth requirements. Additional areas of future work include such optimizations as mailboxing[2] or studies on how variations in pipelining and cache design affect the performance of the intersection core.

[2] *Mailboxing* is a technique which avoids unnecessary intersection tests when a triangle strikes multiple kd-tree nodes.

Bibliography

[1] Arvo, J. R. *Analytic Methods for Simulated Light Transport*. PhD thesis, Yale University, CT, USA, 1995.

[2] Dutré, P., Bekaert, P., and Bala, K. *Advanced Global Illumination*. A K Peters, Natick, MA, USA, 2003.

[3] Kajiya, J. T. The rendering equation. In *SIGGRAPH* (1986), pp. 143–150.

[4] Lafortune, E. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 1996.

[5] Lafortune, E., and Willems, Y. Using the modified phong reflectance model for physically based rendering. Tech. Rep. CW 197, Department of Computing Science, Katholieke Universiteit Leuven, 1994.

[6] Möller, T., and Trumbore, B. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1 (1997).

[7] Parhami, B. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, NY, USA, 2000, pp. 211–221, 279–307.

[8] Pharr, M., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[9] Schmittler, J., Woop, S., Wagner, D., Paul, W. J., and Slusallek, P. Realtime ray tracing of dynamic scenes on an FPGA chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 95–106.

[10] Veach, E. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, CA, USA, 1997.

[11] Wald, I., and Havran, V. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 61–69.

[12] Xilinx, Inc. *Virtex-4 Family Overview*, v1.5 ed., Feb. 2006. http://www.xilinx.com.

[13] Xilinx, Inc. *Virtex-4 User Guide*, v1.5 ed., Mar. 2006. http://www.xilinx.com.

[14] Xilinx, Inc. *XtremeDSP for Virtex-4 FPGAs User Guide*, v2.2 ed., July 2006. http://www.xilinx.com.