Diplomarbeit

# Fixed Point Hardware Ray Tracing

Universität Ulm
Fakultät für Informatik
Abteilung Medieninformatik
Johannes Hanika, `hanatos@gmail.com`, 2007

| | |
|---|---|
| Betreuer/Gutachter: | Prof. Dr. Alexander Keller, Universität Ulm |
| Zweitgutachter: | Prof. Dr.-Ing. Frank Slomka |

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Johannes Hanika (Matrikelnummer 493132), 23. August 2007

# Abstract

For realistic image synthesis and many other simulation applications, ray tracing is the only choice to achieve the desired realism and accuracy. To gain performance and to concentrate on more abstract algorithms e.g. global illumination, it is desirable to encapsulate the ray tracing core (i.e. the process of finding the first intersection of a ray with the scene's boundary) in hardware. Current software implementations of ray tracing cores rely heavily on floating point arithmetic, which requires a considerable amount of logic when implemented as specialized hardware. We examine the use of fixed point arithmetic for the special case of a triangle based ray tracing core and give example implementations in C99 and VHDL along with an analysis of the resulting precision.

# Acknowledgments

First of all, I want to thank everyone on the unique *rtspam* mailing list who are, even now, keeping me from writing my thesis by making me read qmc-trash. Peace easily!

I must apologize that I'm not writing the obligatory private apologies to my friends and family for not spending all the time with them. Hope none of you were too keen on being mentioned here.

Without Alexander Keller this work would never have existed in its current form. He was the one believing in me and hardware.

999 Credits to Carsten for infiltrating my office with his win32 tablet PC, to be able to program the FPGA (one credit off for `wuauclt.exe`).

Thanks to those who read this work and helped to find misspellings and uncomprehensive passages: Alexander Keller, Sehera Nawaz, Matthias Raab, Daniel Seibert.

# Contents

# List of Figures

# 1 Introduction

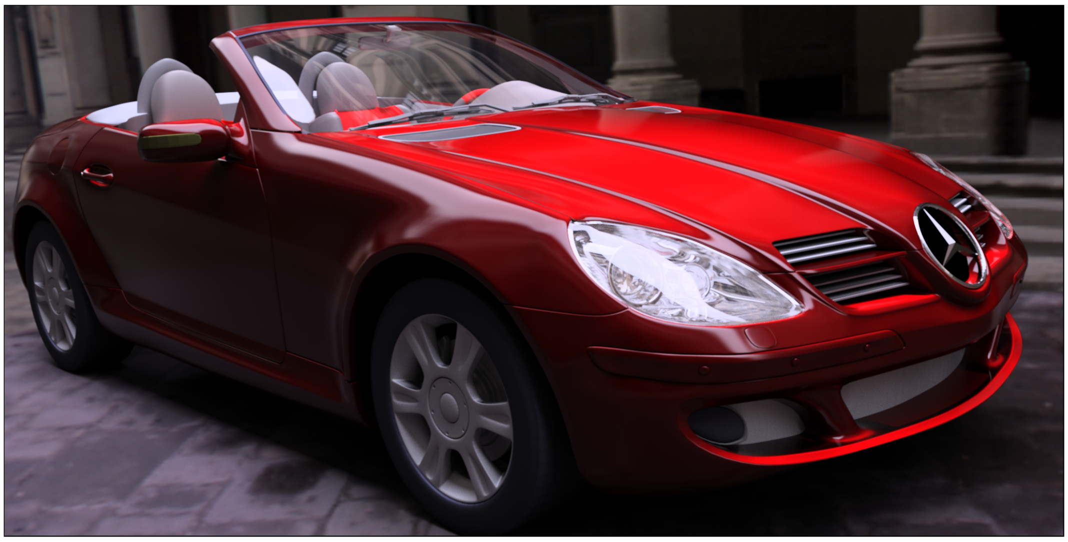


Figure 1.1: A car rendered using a ray tracing core completely based on fixed point arithmetic realized in integer arithmetic. To simplify custom shader writing, the color computations were performed using conventional floating point arithmetic. The quality of the fixed point computations is indistinguishable from computations in floating point arithmetic. Due to the equidistant spacing of fixed point numbers, the self-intersection problem can be tackled without a scene-dependent epsilon, which makes computations in fixed point arithmetic preferable.

For a wide range of applications in scientific simulation and graphics, it is necessary to determine the mutual visibility of two points or the longest free distance from one point along some direction given a mathematical description of the scene's boundary. This problem can be solved by casting a ray from the first point and determining the first intersection with the boundary. As algorithms are well-understood and become simpler, the main challenge is making them faster and more precise. We address these challenges by investigating different arithmetics with respect to their suitability for hardware implementations.

Integer and floating point units are the two standard kinds of arithmetic available on mainstream general purpose processors. Other kinds of arithmetics like for example fixed point or logarithmic arithmetic, are less widely used. For an overview of these arithmetics, the not so convenient *residue number system* and related algorithms see [Kor02].

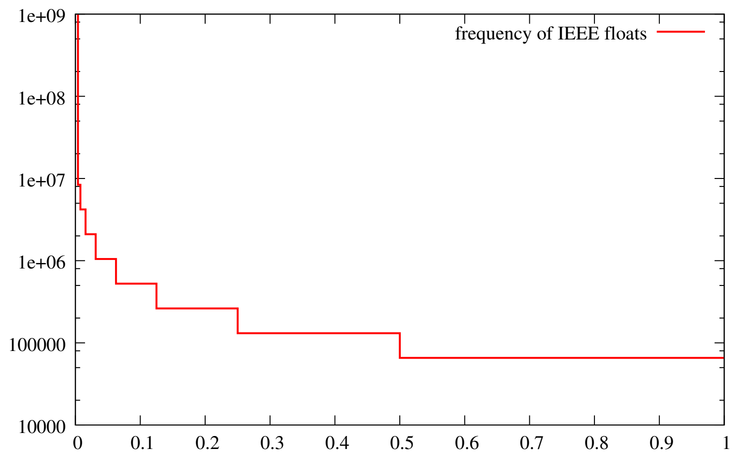


Figure 1.2: Frequency of the IEEE 754 floating point numbers in $[0,1]$ on a logarithmic scale.

For hardware ray tracing these three kinds of arithmetic have been applied already: The first ray tracing hardware [Wri97] used logarithmic arithmetic. Another implementation [Fen02] on a Field Programmable Gate Array (FPGA) is using a custom number format somewhat in between float and fixed point, employing *pseudo floating point scaling factors*. With the advent of general purpose computing on graphics accelerator boards, fixed point arithmetic was applied [CHH02, PBMH05], and with upcoming affordable reconfigurable hardware even floating point units were used [Pur04, SWW$^+$04, WSS05, WMS06, Woo06, KNKL07].

This work is organized in four chapters. The remainder of this chapter will give a short summary of available arithmetics and intersection tests. Chapter 2 and 3 are organized in a parallel way. Chapter 2 is giving a numeric analysis and details about the reference implementation in software, using C99. Chapter 3 is then mapping these results to hardware, where Sections 2.2–2.5 and Sections 3.2–3.5 correspond to each other. Chapter 4 is then giving a conclusion. The source code for the hardware part can be found in the Appendix.

## 1.1 Floating Point Arithmetic

Floating point numbers consist of a tuple $(s, m, e)$, $s \in \{0,1\}$ being the sign, $m \in \mathbb{N}$ the bits of the mantissa and $e \in \mathbb{N}$ the exponent [Kor02, Gol91]. The value of the represented number is then

$$f = (-1)^s \cdot m \cdot b^e$$

for some basis $b$.

The *IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (ANSI/IEEE Std 754-1985)* defines the single-precision normalized values

$$f = (-1)^s \cdot 1.m_{22}m_{21}\cdots m_0 \cdot 2^{e-127},$$

where the 32 bits are interpreted as $se_7e_6\cdots e_0m_{22}m_{21}\cdots m_0$, that is one sign bit, 8 bits exponent biased by 127 and a 23 bit mantissa. This means the most significiant bit, being the signbit, is stored to the left [Kor02, Gol91]. So if a 32 bit unsigned integer is counted from zero up to `0x7F7FFFFF` and reinterpreted as a single-precision floating point value, all positive floating point values are enumerated in order (including denormalized values, where the semantics of the mantissa bits change, indicated by a zero exponent). All larger values would indicate negative values, infinity or not a number (NaN). The basic four operations can be performed as follows:

**Multiplication** The sign is xor-ed, the exponents are added and the mantissae multiplied. To ensure the new mantissa can be expressed as $1.m_{22}m_{21}\cdots m_0$, i.e. $1/b \leq m < b$, a *post-normalization step* may be required [Kor02].

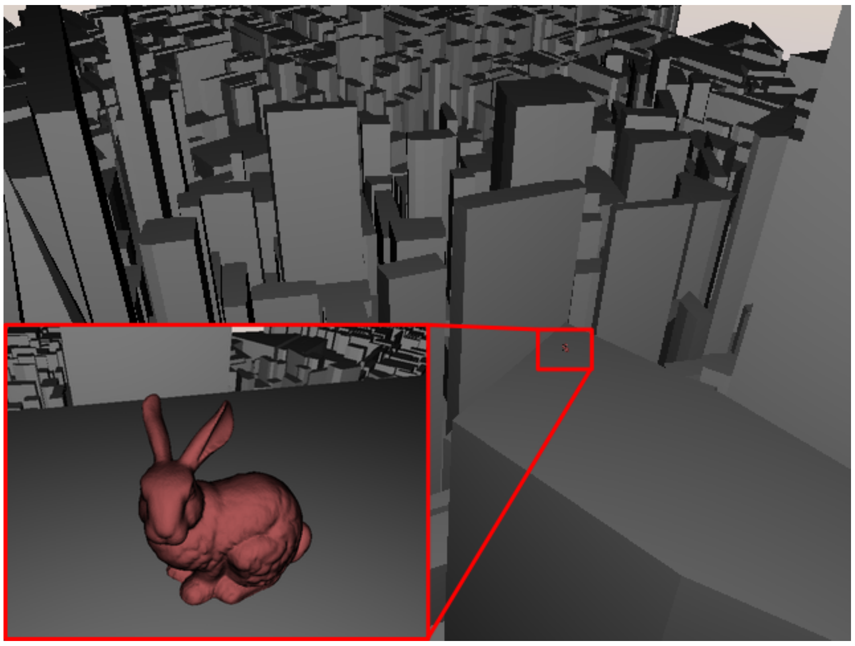
**Division** Much the same as multiplication, except the exponents are subtracted and the mantissae divided.

**Addition and subtraction** The exponents have to be equal before operating on the mantissae. So the mantissa of the smaller operand has to be adjusted. This makes post-normalization necessary which, in turn, might result in an exponent underflow.

All these operations become more complex when taking all special cases as NaN, infinity and denormalized numbers into account. Furthermore, great care has to be taken when rounding intermediary results to conform to the IEEE standard.

The most recent approaches to ray tracing hardware [SWW⁺04, WSS05, WMS06, KNKL07] used floating point units realized on FPGAs. To fit the desired design on the board, the full IEEE standard could not be implemented. So denormalized numbers have been omitted and the bit width has been reduced to 24 [WSS05]. While these approaches focused on the proof of concept, they did not consider ray tracing problems that arise from the unequal distribution of floating point numbers along the real axis (see Figure 1.2).

In fact there are many well-known precision issues to address when using floating point arithmetic [Gol91]. In ray tracing the main problems are the self-intersection problem [WPO96] and the quantization of numbers that are far from the origin. For an illustration of the latter, refer to Figures 1.3–1.5, which show the *bunnies in a city problem* (similar to the teapot in a stadium problem): Combining small and large scale data sets. The red bunny is located at the origin, while the green bunny is located close to the boundary of the city. Both bunnies are of the same size. The scene has intentionally been constructed to show the limitations of any kind of arithmetic using only 32 bits, so there are intersection errors in all images. The city model is courtesy of Leonhard Grünschloß.



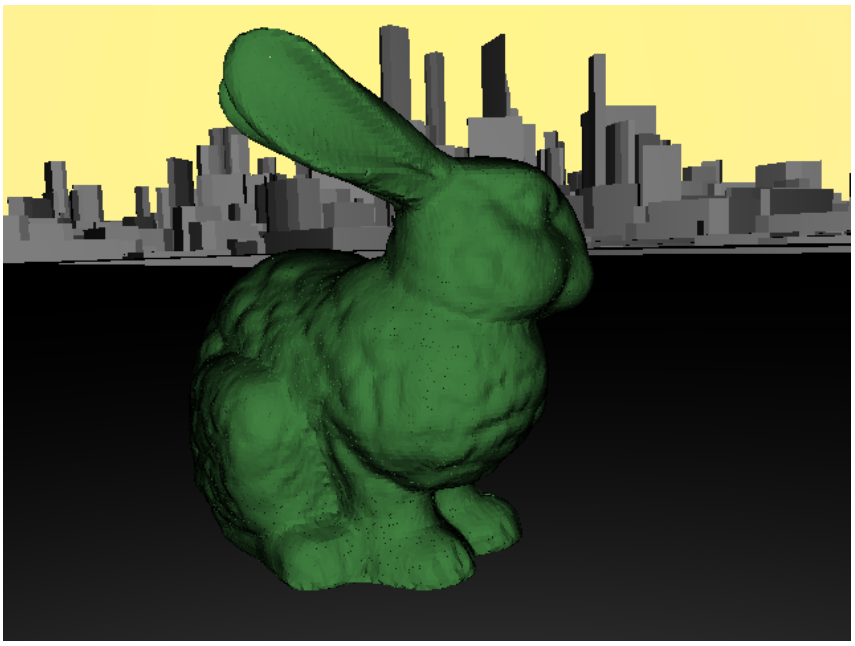


Figure 1.3: Rendered using IEEE 754 single-precision floating point arithmetic. While the red bunny is perfectly accurate, the green bunny suffers from numeric instabilities (see the dark spots). The scene bounding box was about $[-10^5, 10^5] \times [-3 \cdot 10^4, 2 \cdot 10^3] \times [-10^5, 10^5]$.

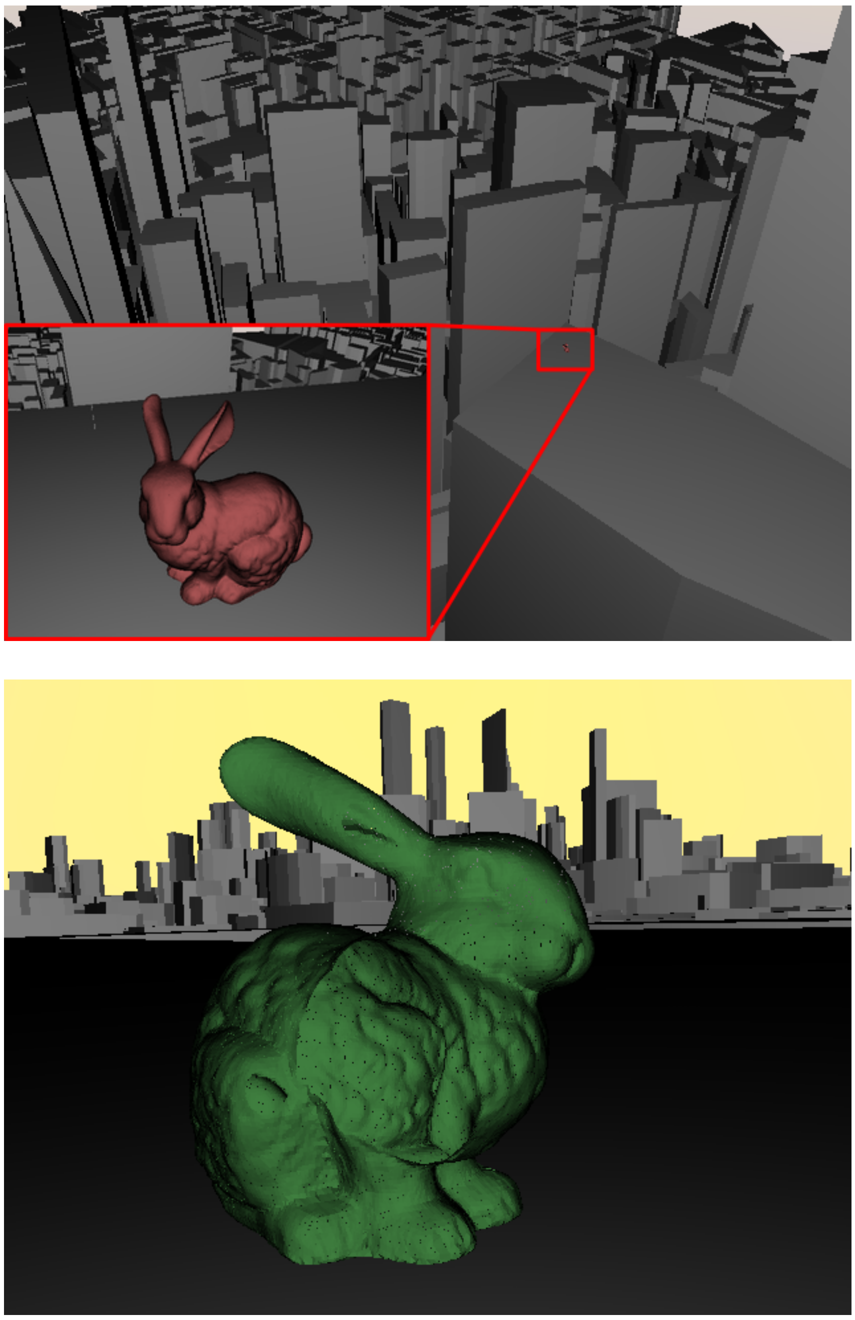


Figure 1.4: Before rendering the scene has been scaled to fit inside $[-1.0, 1.0]^3$, where the floating point resolution is particularly high. The overall accuracy is reduced as there is a very high difference in the frequency of the values near zero and near the borders. As a consequence even the red bunny now suffers from false intersections.

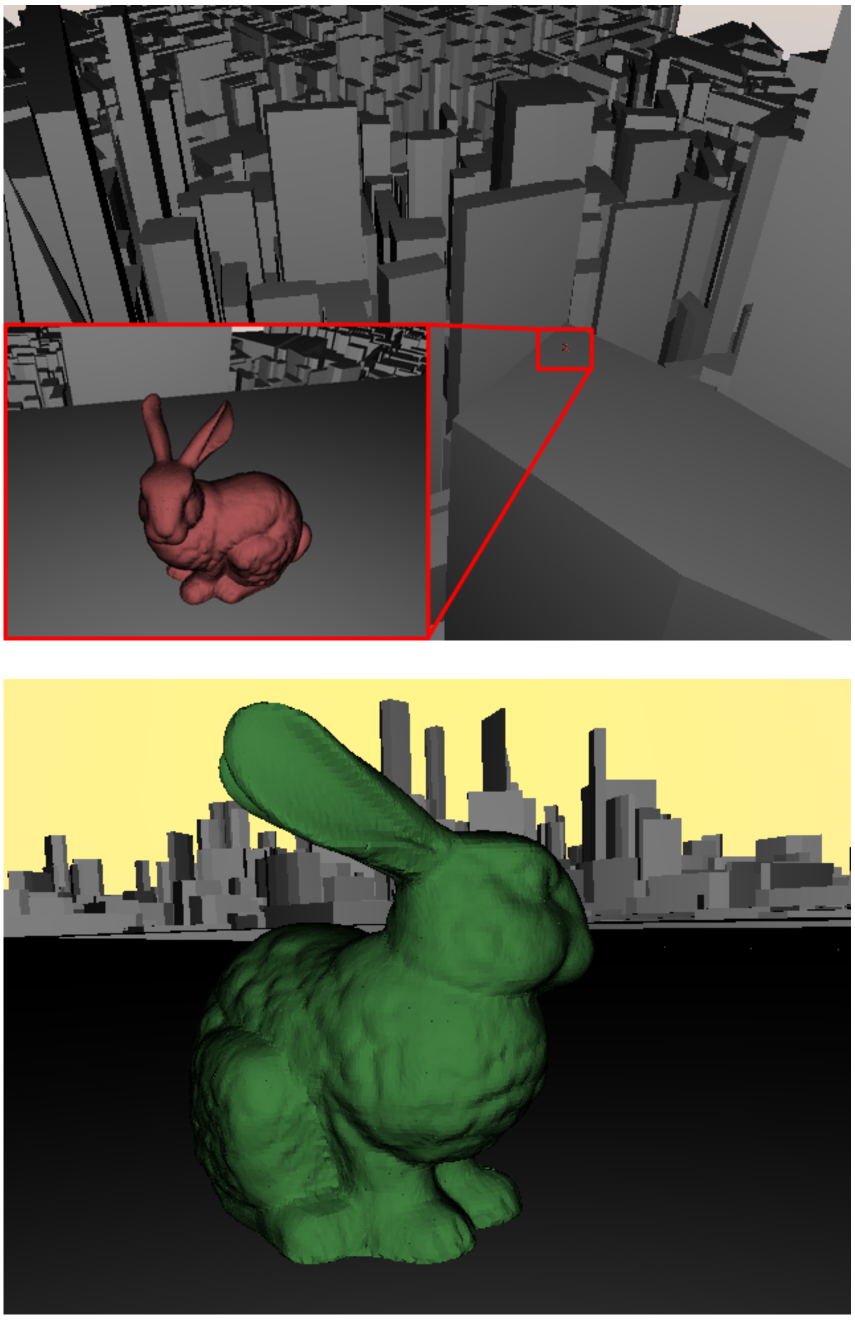


Figure 1.5: Rendered using fixed point arithmetic realized in 32 bit integer arithmetic, where the scene has been scaled to fit the bounding box to the available fixed point range. Although the rendering is not accurate, it now is invariant under translation, i.e. the same precision is reached over the whole range of representable numbers: Both bunnies show a few spurious black spots.

## 1.2 Logarithmic Arithmetic

Instead of mantissa and exponent, logarithmic arithmetic only stores the sign and logarithm of the absolute value of the number, i.e. a tuple $(s, l)$ with $s \in \{0,1\}$ and $l$ being a fixed-point value. For the special case of basis 2, the represented number $f$ is then

$$f = (-1)^s \cdot 2^l.$$

While the spacing of the numbers remains similar to the floating point representation [BB85], the implementations of multiplication, division, square root and power become simpler [Kor02]. The basic operations:

**Multiplication and division** can be reduced to addition/subtraction of the logarithm $l$.

**Addition and subtraction** are unfortunately very complicated. A complete look-up table would need $2^{2n} \times n$ bits and is therefore not realistic. So approximations with smaller look-up tables have to be used. There are several attempts to remove this disadvantage, for example [MST98].

Another source of inaccuracies is converting from and to the logarithmic system.

Due to physical constraints at that time, the first ray tracing hardware was forced to use a compact arithmetic. For this purpose logarithmic arithmetic (see e.g. [Wri97, Cols. 13 and 14]) proved to be efficient in space and performance as it can be realized using almost exclusively integer arithmetic units. The challenges of a lack of a zero and the computation of Gauss' logarithm for addition and subtraction were solved by careful algorithm design and a lookup table. However, the main disadvantage of the non-equidistant spacing of the representable numbers persists.

## 1.3 Fixed Point Arithmetic

The main advantage of fixed point arithmetic over floating point and logarithmic arithmetic is the equidistant spacing of the numbers and the direct applicability of integer arithmetic:

**Addition, subtraction, multiplication** Can be performed easily even on the FPGA, as dedicated units to perform integer operations are readily available. Our Virtex-5 FPGA for example provides 64 versatile DSP48E slices.

**Division** Unfortunately, there is no such built-in functionality for division. There are several different approaches to divide two integers which can be optimized for the special application. Our approach is discussed in Section 3.2.

Due to the lack of an exponent in the representation, the range is more limited and an application of fixed point numbers requires a profound investigation of the ranges of all computations in an application.

In fact the first implementations of ray tracing on graphics hardware [CHH02, PBMH05] were using 16-bit fixed point arithmetic and report rendering artifacts due to the limited range.

We will analyze these issues and the required ranges and present a realization of ray tracing in fixed point arithmetic that achieves the precision of a floating point implementation. The hardware implementation is considerably more compact and simpler than a realization using floating point or logarithmic arithmetic.

## 1.4 Ray/Triangle Intersection Tests

Determining the intersection point of a ray with a triangle can be done in various ways. A quick summary of the most popular tests follows.

Making use of the determinant of the matrix consisting of the edge vectors and the ray direction, the classic ray/triangle intersection test is the one by Möller-Trumbore [MT97]. Naturally an advantage for animated scenes, no precalculations are needed here at all. Yet efficiency might be increased by storing the edges instead of the vertices.

Orientations of lines represented in Plücker coordinates [Jon00] are the base of another test, which is best applied to large ray packets, amortizing the initial cost over many intersection calculations [Ben06]. Using ray packets greatly improves the performance of primary rays and a few special cases in shading. Restrictions like this should be avoided however, as we want to focus on ray tracing for general applications.

Especially appealing for hardware applications, there are intersection tests coming by without a division [SF01, Chi05]. Unfortunately at the cost of finding only a boolean result: hit or no hit. Generating secondary rays depends on the location of the hit point, so the division is only postponed as it is needed for the task of finding the closest intersection.

Lately, a robust floating point intersection test [DK06] has been developed. Yet computing the intersection point without a division, it is not easily mapped to hardware. The algorithm iteratively subdivides the triangles axis-aligned bounding box in four smaller ones by inserting a new vertex for each edge of the triangle. The ray is then tested against these bounding boxes and the procedure repeats. This avoids rays tunneling through triangles due to numeric inaccuracies inherent to the other triangle tests. As the iterative nature of the algorithm makes pipelining hard and requires a stack, we only used it as a reference implemented in software rather than hardware.

Finally, there is the test of Badouel [Bad90], based on barycentric coordinates. It has been implemented very efficiently [Wal04, Ben06, KS06] by employing precomputations. We selected this intersection test over the one by Möller-Trumbore, because our current focus is not on dynamic scenes and the precomputation makes the actual intersection code very short. The precomputations could be performed in a parallel, pipelined way on the FPGA, when streaming the triangle data as three vertices. As we are redundantly storing the triangles as three vertices and not as vertex list plus three indices per face, there is no additional memory required for the precomputed data. It just replaces the original vertex data. Storing faces as three vertex indices is slightly less convenient for a hardware implementation because the vertices have to be fetched from memory.

For dynamic scenes other triangle tests must be investigated, too. This, however, is out of the focus of this work.

# 2 Software Ray Tracing in Fixed Point Arithmetic

In this chapter, we analyse fixed point arithmetic applied to ray tracing in the context of a C99 software implementation. As mentioned, we select the test of Badouel [Bad90] out of the many different ways to intersect a ray and a triangle [Bad90, MT97, Jon00, SF01, Chi05, DK06]. We briefly summarize the procedure: For each triangle with the vertices $x_0, x_1, x_2 \in \mathbb{R}^3$ we compute the vector

$$n = (x_1 - x_0) \times (x_2 - x_0)$$

in normal direction and store the smallest index `r` of its longest absolute component

$$n_{\mathtt{r}} = \|n\|_\infty = \max\{|n_0|, |n_1|, |n_2|\}$$

in two bits. Triangles with $n_{\mathtt{r}} = 0$ have no area and are omitted. Using the additional indices

$$p := (r+1) \bmod 3 \qquad \text{and} \qquad q := (r+2) \bmod 3$$

we further store the components

$$\begin{aligned}
\mathtt{pp} &= x_{0,p}, \quad \mathtt{pq} = x_{0,q}, \\
\mathtt{np} &= \frac{n_p}{n_r}, \quad \mathtt{nq} = \frac{n_q}{n_r}, \\
\mathtt{d} &= \frac{1}{n_r}\langle n, x_0\rangle = x_{0,r} + \mathtt{pp}\cdot\mathtt{np} + \mathtt{pq}\cdot\mathtt{nq}, \text{ and} \\
\mathtt{e}_{ik} &= \frac{1}{n_r}(x_{i,k} - x_{0,k}), \ i \in \{1,2\}, k \in \{p,q\}.
\end{aligned}$$

The normalization by the maximum norm $\|n\|_\infty$ simplifies the scalar product, as the vector component `r` is guaranteed to be equal to one.

In order to intersect a ray $(O, \omega)$ with origin $O$ and direction $\omega$ with a triangle according to the accelerated test from [Wal04], the distance $t$ along the ray from its origin to the plane of the triangle is computed as

$$t = \frac{\mathtt{d} - \frac{1}{n_r}\langle O, n\rangle}{\frac{1}{n_r}\langle \omega, n\rangle} = \frac{\mathtt{d} - O_r - O_p\,\mathtt{np} - O_q\,\mathtt{nq}}{\omega_r + \omega_p\,\mathtt{np} + \omega_q\,\mathtt{nq}}. \tag{2.1}$$

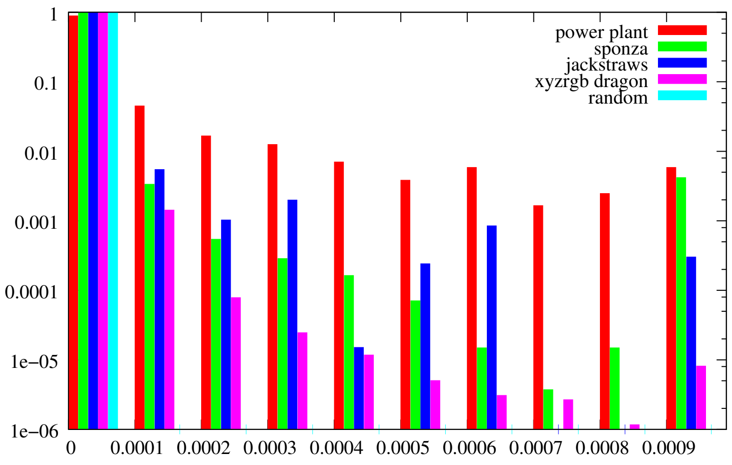


Figure 2.1: On a logarithmic scale: the relative frequency of the triangle edge components `e1p`, `e1q`, `e2p`, `e2q` for various scenes after transforming the triangles to the integer bounding box according to Section 2.1. Most of the values are smaller than 0.0001. Note that the rightmost bin contains all remaining components greater than 0.0009. For some scenes (sponza and xyzrgb_dragon) the distribution then resembles the distribution of floating point numbers (see Figure 1.2). However, this is not true in general as can be seen from the remaining scenes. Moreover, it is a misinterpretation to conclude that the edge information should be represented in floating point or logarithmic numbers, as this does not at all improve on the bigger quantization error of bigger components. Using equidistantly spaced fixed point numbers reduces these errors, however, at the cost of a reduced range.

Then the hit point of the ray and the plane projected along the component `r` is computed by

$$\begin{aligned} k_p &= O_p + t \cdot \omega_p - \mathtt{pp}, \\ k_q &= O_q + t \cdot \omega_q - \mathtt{pq}, \\ u &= \mathtt{e}_{1,p} \cdot k_q - \mathtt{e}_{1,q} \cdot k_p, \\ v &= \mathtt{e}_{2,q} \cdot k_p - \mathtt{e}_{2,p} \cdot k_q. \end{aligned} \tag{2.2}$$

An intersection is reported if the barycentric coordinates $u$ and $v$ fulfill $u \geq 0$, $v \geq 0$, and $u + v \leq 1$.

## 2.1 Quantization and Precision

### 2.1.1 Numeric Ranges

To store the accelerated representation of a triangle in finite precision without losing vital information, it is necessary to examine the ranges of the precomputed components.

As $\mathtt{pp}$ and $\mathtt{pq}$ are copies of the original data, both can be stored in the given precision. The components of the normal $\mathtt{np}, \mathtt{nq}$ are normalized using the maximum norm and consequently $\mathtt{np}, \mathtt{nq} \in [-1, 1]$.

For the remaining components we have to consider that each finite subset of the real numbers has a minimum and a maximum. Hence, given a set $\mathbb{V} := \{\mathrm{v}_i = (\mathrm{v}_{i,0}, \mathrm{v}_{i,1}, \mathrm{v}_{i,2}) : 0 \le i < N\} \subset \mathbb{R}^3$ of $N$ triangle vertices, we can define their axis-aligned bounding box components as

$$b_j := \min\{\mathrm{v}_{i,j} : \mathrm{v}_i \in \mathbb{V}\} \qquad \text{and} \qquad B_j := \max\{\mathrm{v}_{i,j} : \mathrm{v}_i \in \mathbb{V}\}.$$

For each triangle with vertices $x_0, x_1, x_2 \in \mathbb{V}$ and its normal $n$, employing the fact that for any vector $n \in \mathbb{R}^3$

$$\left\| \frac{n}{\|n\|_\infty} \right\|_2 \le \sqrt{3},$$

we can bound the absolute value of the distance $\mathtt{d}$ by

$$\begin{aligned} |\mathtt{d}| &= \left| \frac{1}{n_r} \langle n, x_0 \rangle \right| \le \max_{0 \le i < N} \left| \frac{1}{n_r} \langle n, \mathrm{v}_i \rangle \right| \\ &\le \left\| \frac{n}{n_r} \right\|_2 \cdot \max_{0 \le i < N} \|\mathrm{v}_i\|_2 \le \sqrt{3} \max_{0 \le i < N} \|\mathrm{v}_i\|_2 \le 3 \max_{j \in \{0,1,2\}} (B_j - b_j). \end{aligned}$$

In order to bound the edge components we need $\varepsilon$, which is the smallest, non-zero absolute value of the numbers representable in the arithmetic. Since triangles where computing $n_\mathrm{r}$ yields zero are omitted because they are degenerate, we have

$$|n_\mathrm{r}| = \left| (x_{1,p} - x_{0,p})(x_{2,q} - x_{0,q}) - (x_{1,q} - x_{0,q})(x_{2,p} - x_{0,p}) \right| \ge \varepsilon > 0$$

and hence

$$|\mathtt{e}_{ik}| = \left| \frac{1}{n_r} (x_{i,k} - x_{0,k}) \right| \le \frac{1}{\varepsilon} \max_{j \in \{0,1,2\}} (B_j - b_j). \tag{2.3}$$

Note that the bound using the longest side of the scene bounding box is very loose. Using the longest side of all triangle bounding boxes often yields a much tighter bound, especially when all triangles are about equally small.

### 2.1.2 Quantization

The quantization now can be parameterized by two bit widths $\mathtt{n}$ and $\mathtt{m}$:

**Points:** The scene geometry $\mathbb{V}$ is moved to the positive octant and scaled such that $b_j = 0$ for $j \in \{0, 1, 2\}$ and $\max_{j \in \{0,1,2\}} \{B_j\} = 2^\mathtt{n} - 1$. The values $\mathtt{pp}$ and $\mathtt{pq}$ then are represented using $\mathtt{n}$ bit unsigned integers resulting in $\varepsilon = 1$.

**Distances:** The representation of the distance $\mathtt{d}$ needs $\mathtt{n}$ + 3 bits, because the distance is signed (1 bit) and the maximum prolongation by a factor of 3 (see the above bound) requires two further bits.

**Vectors:** Components of normals (np and nq), ray directions, and triangle edges (e1p, e1q, e2p, and e2q) are quantized using m bits signed fixed point numbers to represent the range $[-1, 1]$.

Note that rays starting outside the bounding box of the geometry, like e.g. primary rays, must be clipped to the bounding box prior to quantization.

### 2.1.3 Clamping of Triangle Edge Components

In order to improve the bound on the edge data, we investigated their statistics. Figure 2.1 shows the relative frequency of the edge components e1p, e1q, e2p, and e2q for several typical test scenes after the transformation explained in the previous section. The distribution suggests that the majority of the edge data is smaller than 0.001 and most of the time even smaller than 0.0001.

Linearly mapping the edge data to the range of the fixed point arithmetic would result in considerable quantization errors, as the maxima can be relatively large. Instead, we clamp the values to the range $(-2^{-\mathtt{E}}, 2^{-\mathtt{E}})$ where E is called the *edge shift*. They are then quantized in this range using m bits fixed point. Choosing $2^{-\mathtt{E}} \approx 0.001$, i.e. E = 10, as indicated by the statistics, is already sufficient to render the test scenes without artifacts.

If the modeler can guarantee to keep the bound $|\mathtt{e}_{ik}| < 2^{-\mathtt{E}}$, which is much more handy than the bound in Equation 2.3 from the previous section, errors due to clamping are completely avoided. Note that this can be difficult to achieve for long triangles with small area: if the triangle is simply subdivided into four smaller ones by inserting three new vertices in the middle of the edges, the edge data would actually increase by a factor of two. This is because the edges are halved but the resulting area (and thus $n_r$) is only one quarter of the original value. Since the vertices are quantized, triangles are likely to be degenerate in addition.

### 2.1.4 Triangle Data Structure and Choice of Parameters

For the prototype implementation in C99 it was convenient to match the standard 32 bits double word width. Because of the precision requirements of d, we then have n = 32 - 3 = 29 bits. All fixed point numbers use m = 32 bits signed integers. As mentioned before, the edge shift is E = 10. To simulate smaller bit widths, the least significiant bits are simply set to zero. Note that we did not choose to store fractional bits for d. This does not introduce any additional quantization error, as with careful rounding it is possible to reconstruct the

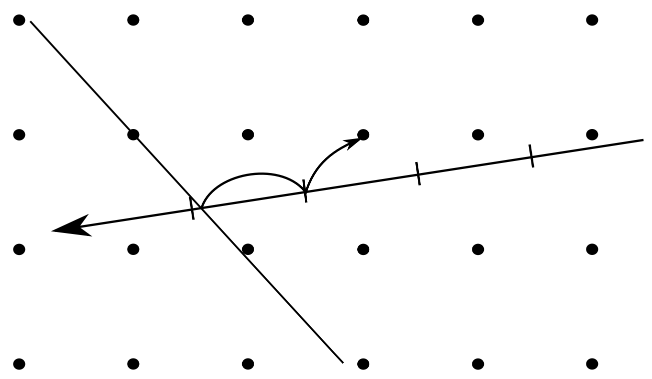


Figure 2.2: The self-intersection problem for fixed point ray tracing. The small arrow indicates the errors which can occur on the integer grid. First, the distance *t* along the ray is truncated, then the hit point is rounded to the grid. Consequently, for transmissive rays to start on the right side of the surface, the hit point has to be moved by two units.

original quantized triangle vertices using this precision. The triangles are stored as:

```
typedef struct
{
  int d;             // regular signed int
  unsigned r : 2;
  unsigned pp : 29;
  unsigned int pq;   // unsigned 29 bits
  int np, nq;        // signed fixed point in [-1,1]
  int e1q, e2q;      // signed fixed point
  int e1p, e2p;      //   in (-2^-E, 2^-E)
}
accels_t;

typedef union
{
  float x[3][3];   // original float data
  unsigned int xi[3][3];  // transformed vertices
  accels_t a;      // accelerated representation
}
triangle_t;
```

### 2.1.5 Secondary Rays and the Self-Intersection Problem

A common issue with secondary rays is the so called *self-intersection* problem [WPO96]. It refers to the fact that secondary rays sometimes end up hitting the same object they originate from because of deficiencies of the arithmetic. The usual solution is to shift the origin of the secondary ray by adding a small fraction of the normal and/or the new ray direction. For floating point algorithms the most advanced approach to this problem is the robust triangle test [DK06].

Since in fixed point arithmetic the geometry lies on an equidistant raster with known resolution, self-intersection can be approached in a simpler way: The distance $t$ measured along the ray direction $\omega$ uses the same resolution as the vertex data (only a larger range is allowed) and, therefore, can be only wrong by 1 unit of the integer grid ($= 2^{-n}$ times the largest side of the bounding box in our implementation). The computation of the hit point $h = O + t \cdot \omega$ uses one more fractional bit and is rounded to the integer grid, which can cause an additional error of 0.5 in each component. Figure 2.2 shows this setting. The worst case error along each axis thus is $1 + 0.5 < 2$ and consequently the ray origin just needs to be shifted by two units:

```
int dt = dotproduct(n, omega);
O[k] = h[k] + ((dt > 0) ^ (n[k] < 0) ? 2 : -2);
```

The variable `dt` determines transmissive rays, where the point has to be shifted into the reverse direction.

Note that in the presence of shading normals it still has to be detected whether the ray is accidentally sampled under the surface due to the perturbed normal. Thus, one has to decide for either precision ray tracing or the use of shading normals.

## 2.2 Ray/Plane Intersection

First, the distance $t$ has to be computed according to Equation 2.1, which involves a division. In finite arithmetic the mathematical equivalence

$$t = \frac{\mathrm{num}}{\mathrm{den}} = \frac{\mathrm{num} \cdot 2^{-\mathrm{T}}}{\mathrm{den} \cdot 2^{-\mathrm{T}}}, \text{ where } T \in \mathbb{N}_0,$$

changes the result depending on the parameter $T$: The numerator *num* and the denominator *den* are shifted to the right, which causes the numbers to lose their T least significiant bits, resulting in a loss of precision. However, the bit width of the division is reduced, which allows for a faster hardware implementation. The impact of the parameter T is illustrated in Figure 2.8, all other images have been rendered using $\mathrm{T} = 0$.

For the basic case of axis-aligned planes, as used in the traversal of acceleration data structures, We need to compute $t = (P - O_i)/\omega_i$, where the plane under consideration is $\{x \in \mathbb{R}^3 : x_i = P\}$.

For m $\leq$ 32, 64 bits (`long long int`) are sufficient for the temporary values in order to avoid overflows. The distance *t* can then be calculated as follows:

```
const long long int mask = 0xFFFFFFFF80000000LL;
const int den = omega[i]>>T;
if(den == 0) return no_intersection;
int t = (((P - O[i])<<(m-1-T)) & mask)/den;
```

where `omega[i]` is an m-bit signed integer representing a fixed point value in $[-1,1]$. Note that *t* is stored as a 32-bit integer, because the same precision as used for `d` is sufficient here. The variable `mask` assures that the precision is really truncated as it would be in a hardware implementation.

## 2.3 Ray/Triangle Intersection

The ray/triangle plane intersection requires to project the ray direction onto the triangle normal, as seen in Equation 2.1. To make the source listing more readable, `O` and `omega` are assumed to be arrays of `long long int`:

```
long long int den = ((omega[r] +
      (omega[p]*np >> (m-1)) +
      (omega[q]*nq >> (m-1)))) >> T;
long long int mask = 0xFFFFFFFF80000000LL;
if(den == 0) return no intersection;
int t =
  (((((d - O[r]) << (m-1)) -
   O[p]*np - O[q]*nq) >> T) & mask)/den;

if((t <= hit->t) && (t > 0))
{
  test barycentric coordinates
}
```

Whether or not the triangle is intersected by the ray is decided according to the set of Equations 2.2 using the distance *t*. Similar to the previous section, fixed point numbers in $[-1,1]$ are multiplied by $2^{m-1}$, computations are done using integer arithmetic, and finally the result is shifted back to the desired range. The implementation is:

```
int kp = O[p] + ((t*omega[p]) >> (m-1)) - pp;
int kq = O[q] + ((t*omega[q]) >> (m-1)) - pq;
long long int u = (long long int)e1p*kq -
  (long long int)e1q*kp;
long long int v = (long long int)e2q*kp -
  (long long int)e2p*kq;

if(u < 0 || v < 0 ||
  ((u + v) >> E) > (1UL << (m-1)))
    return no_intersection;
else report intersection
```

## 2.4 Acceleration

As we are heading for the simplest possible ray tracing hardware, numerically involved triangle-plane intersections as required for building *k*d-trees should be avoided. Hence we use a bounding interval hierarchy [WK06, WMS06] as acceleration data structure. For its construction only divisions by 2 and comparisons are required, which are trivial to transfer to integer arithmetic and both operations are unconditionally robust.

Compared to a *k*d-tree, the construction of a bounding interval hierarchy (BIH) is much faster and simpler, while ray tracing speed is comparable as long as the overlap of the object bounding boxes is small [WK06].

The traversal of the data structure requires intersecting a ray with a pair of axis-aligned planes. This intersection can be computed as derived in Section 2.2, however at the cost of a division which is relatively slow.

In order to accelerate the traversal, the reciprocal of the ray direction can be stored for $\omega_i \neq 0$, replacing the division by a multiplication.

$$|\omega_i| \in \left[2^{-\mathtt{m}+1}, 1\right) \Leftrightarrow \left|\omega_i^{-1}\right| \in \left(1, 2^{\mathtt{m}-1}\right],$$

thus `omega_inv[i] = (1<<(m-1))/omega[i]`.

However, computing the distance to the clip planes in a different way as the ray-triangle intersection introduces inconsistencies, especially for axis-aligned triangles. In order to obtain sufficient accuracy, $\omega_i^{-1}$ has to be stored in $\mathtt{m}+\mathtt{C}$ bits, requiring more bits for the multiplication as well. The additional bits also ameliorate the fact that due to its hyperbolic nature the range of $\omega_i^{-1}$ is not well represented by equidistant quantization.

In a software implementation, one has to take care that there will be no overflow, which in turn affects precision. The images in Figure 2.9 show the effect of the parameter `C` and the resulting quite visible inconsistencies for too small C. The computations have been performed using the following code fragment:

```
// precompute
const long long int nom = 1ULL << (m + C);
omega_inv[i] = nom/omega[i];

// plane test
int t = ((P - O[i]) >> C) * omega_inv[i];
```

## 2.5 Results

In Figure 2.7 the effect of varying the fixed point precision m is shown. Already a rather low precision allows for correct renderings. In an actual hardware implementation m will certainly be chosen in a safe way, i.e. even $m > 32$.

### 2.5.1 Constructing a Stress Test

Triangles with long edges and small area are numerically difficult to intersect. The worst-case triangle would be ranging diagonally through the complete bounding box with the third vertex exactly in the center of the bounding box. Since this triangle is degenerated, i.e. has zero area and cannot be visible, we moved one integer vertex so that the area became non-zero. A comparison to the floating point setting was impossible, as the floating point computation still classified this triangle as degenerate. Then the triangle was further extended until if formed a thin line at an image resolution of $640 \times 480$ pixels. In fact the edge data for this triangle is in $(-0.0001, 0.0001)$ and thus well represented in integer quantization. No difference between the floating point and fixed point computations could be spotted, the images are included in Figure 2.6 for the sake of completeness.

### 2.5.2 Numerical Evidence

Due to clamping of the edge data (see Section 2.1.3) intersection errors may occur. Therefore, we extracted the triangles with $|\mathtt{e}_{ik}| > 2^{-\mathtt{E}}$ from the power plant scene. As a reference and for comparison, the robust single-precision floating point triangle test [DK06] has been used.

This test revealed errors in both floating point and fixed point arithmetic, however, the fixed point version reported notably more intersections. A possible explanation for these false positives is that clamping $e_{ik}$ implicitly puts a lower bound to $n_r$ and thus to the triangle area. The visual results in Figure 2.3 are rather abstract and only included for the sake of completeness.

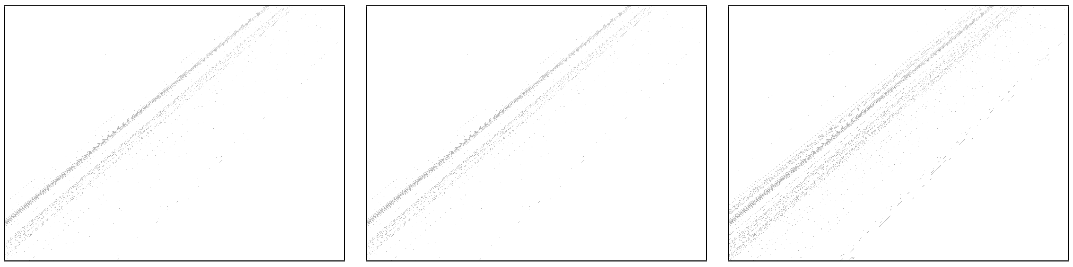


Figure 2.3: Triangles with components `e1p`, `e1q`, `e2p` or `e2q` outside the representable fixed point range $(-2^{-E}, 2^{-E})$ extracted from the power plant model. To leave the range, a triangle needs to be very long and thin, and hence has almost zero area. Both kinds of investigated arithmetics (floating point in the middle and fixed point arithmetic on the right) create intersection errors, the fixed point test even shows false positives. The triangles are so narrow that even the reference image (robust floating point intersection test [DK06] on the left) shows mostly aliasing.

Additional tests were performed for the powerplant model, the Sponza atrium, a jackstraws scene, the xyzrgb_dragon and random triangles. The floating point images have been generated using the original data set, i.e. without scaling the vertices to the integer bounding box. Often no differences can be observed (as for the test scenes in Figure 2.4). Most of the differences in Figure 2.5 originate from slightly differently quantized primary rays. Sometimes both versions cast rays through triangles spuriously.

It can be seen that the intersections are reported correctly even for triangles not so well behaved in terms of the ratio of edge length to triangle area. In the jackstraws scene one stick contains 2048 very long and narrow triangles forming a cylinder. Also the capillary crane in the power plant closeup does not cause errors.

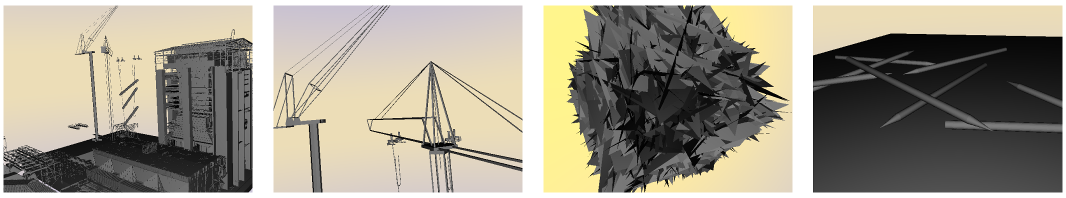


Figure 2.4: Some test cases where fixed point and floating point arithmetic almost cannot be distinguished. These master images were computed using a robust floating point arithmetic reference implementation [DK06].

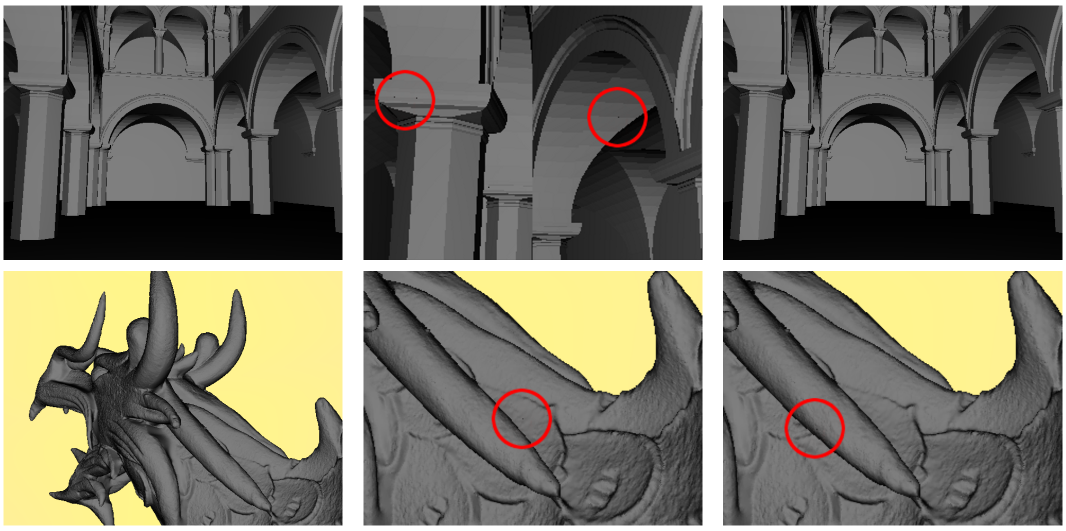


Figure 2.5: Comparison of screenshots made using the robust floating point arithmetic intersection [DK06] (left), floating point arithmetic (middle), and fixed point arithmetic (right). The red circles in the zoomed images mark the most apparent intersection errors. Concerning precision, floating point and fixed point arithmetic perform equivalently.

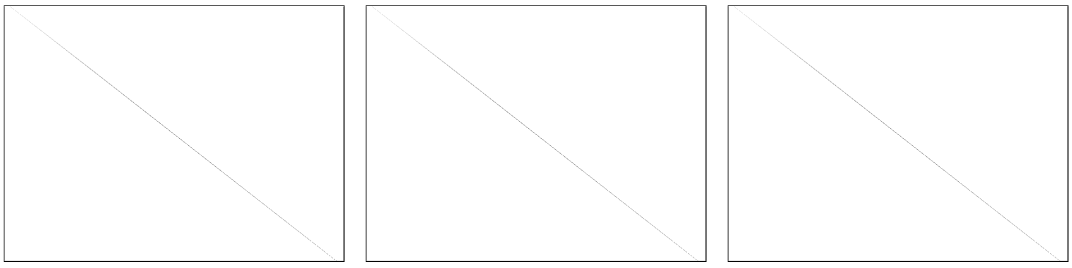


Figure 2.6: A large and thin triangle near worst-case. The theoretical worst case would not have been visible. Robust test [DK06] (left), floating point (middle) and integer arithmetic (right).

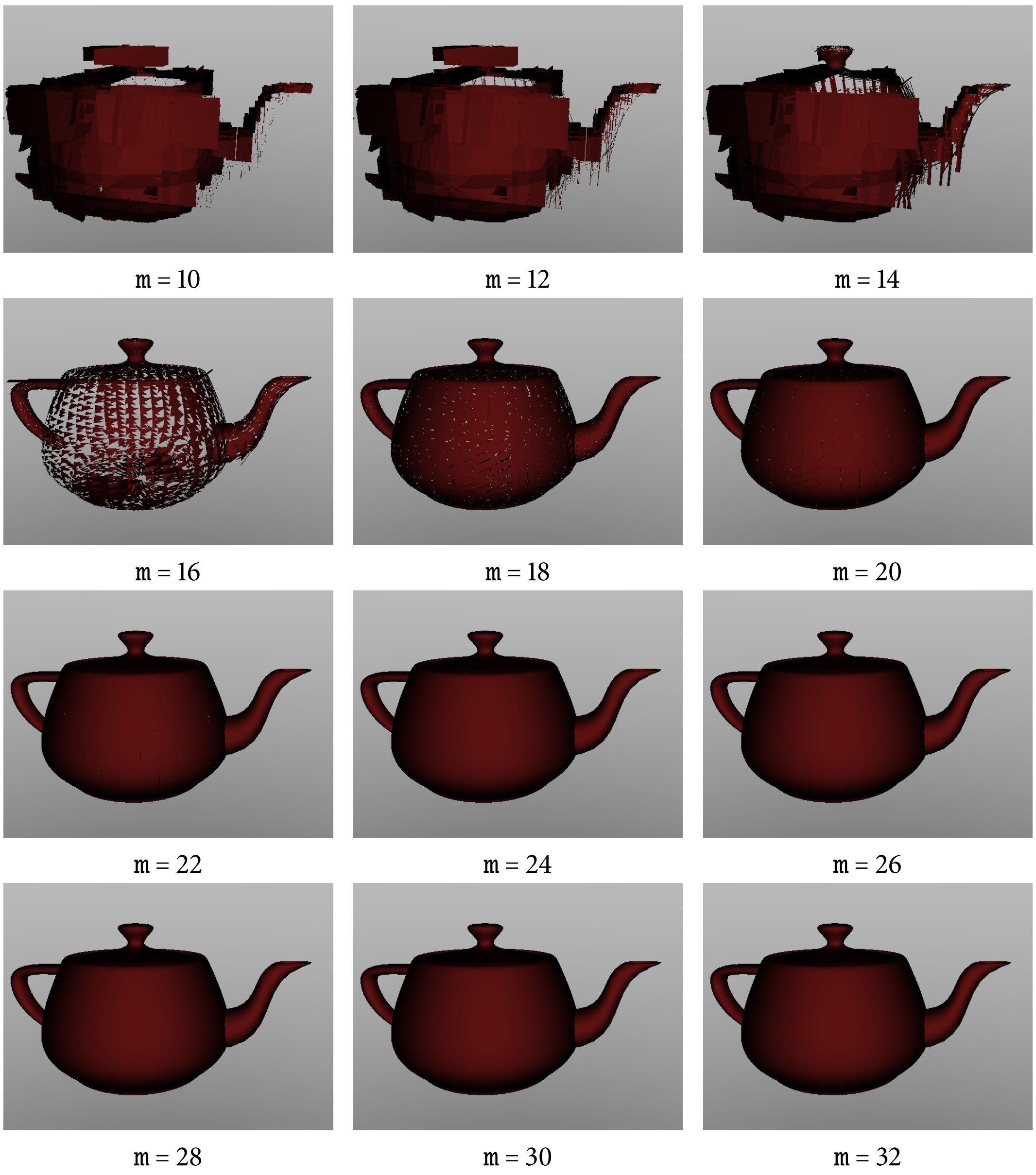


Figure 2.7: The Utah Teapot tesselated into 6320 triangles and rendered using different bit widths m for the fixed point numbers. This affects the precision of the components of the triangle edges, normals, and directions. A moderate precision already allows for artifact free renderings.

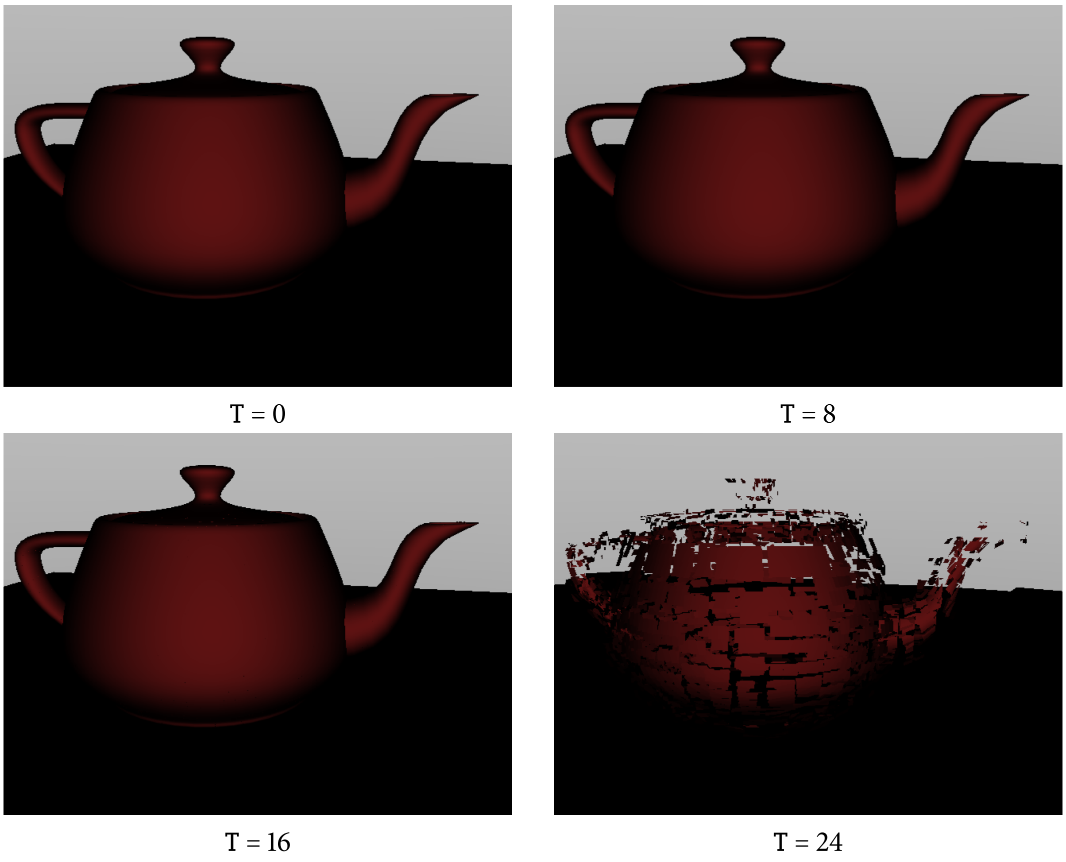


Figure 2.8: The Utah Teapot at different levels of precision for the division used in the distance computation, needed while traversing the BIH and during ray triangle intersection. The `T` least significiant bits of numerator and denominator (`m` = 32) are omitted to reduce the number of clock cycles needed for the division. Only 32 – `T` bits are used for the calculations, so for `T` = 24 only a 16/8 bit division is performed instead of 64/32 bits, resulting in unacceptable artifacts. For smaller values of `T`, the loss of precision might be negligible and acceptable compared to the gain of clock cycles.

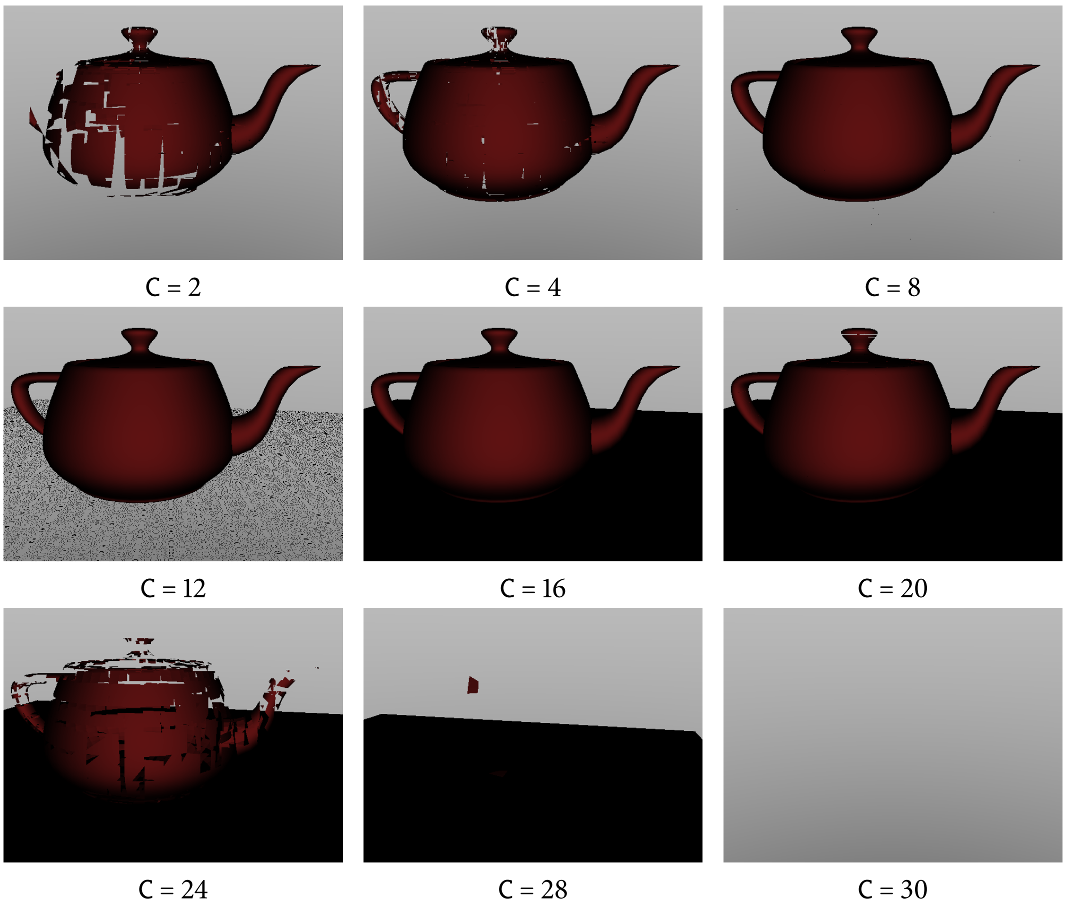


Figure 2.9: The teapot with different values of the precision parameter C. This parameter determines how many additional bits the precomputed inverse ray direction receives. This inverse value is used to avoid a division during BIH traversal. As it is not well represented using equidistant fixed point values, an additional C fractional bits are used. Artifacts for large C are due to a truncation needed to avoid overflows in 64 bit integer arithmetic in software and can be avoided in hardware by performing computations using greater bit widths. Inconsistencies for axis-aligned triangles become especially visible for C = 12.

# 3 Mapping to Hardware

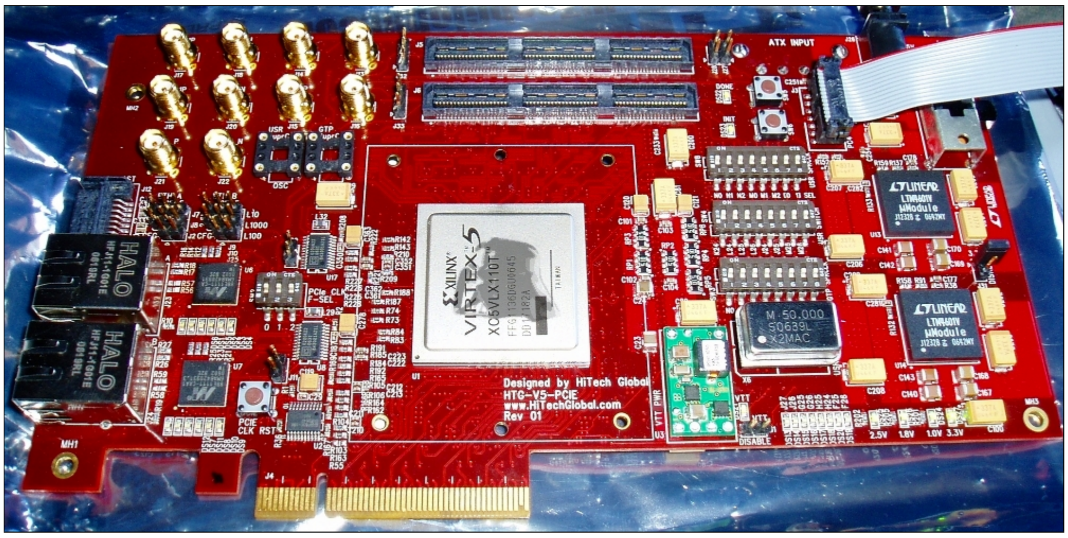


Figure 3.1: The Xilinx Virtex-5 `XC5VLX110T-1136`.

Amongst the variety of hardware description languages including Verilog, Handel-C, SystemC and JHDL, we decided to implement the algorithm in VHDL (Very High Speed Integrated Circuit Hardware Description Language), since it is an established, mature and sufficiently low-level programming language with lots of references and samples available. It is also supported by the Xilinx toolchain. For quick simulation purposes GHDL[1], a free compiler built on top of GCC, and GTKWave[2] have been used.

To help keeping the code complexity under control, a two-process design method [Gai] has been used. It involves separating combinatorial and registered logic into two processes and grouping input and output signals as well as the registers in record data types and the components in packages.

Our hardware platform is a Xilinx Virtex-5 `XC5VLX110T-1136` equipped with two gigabit Ethernet controllers, a PCI-Express interface and a DDR2 memory controller (see Figure 3.1).

[1]http://ghdl.free.fr/
[2]http://home.nc.rr.com/gtkwave/

## 3.1 Toplevel Design

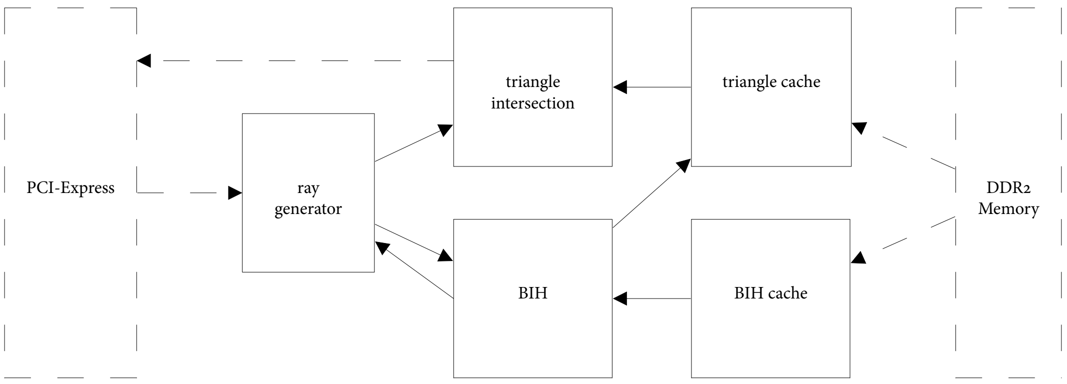


Figure 3.2: Toplevel design of the hardware implementation. Dashed parts indicate unimplemented features.

To maintain full algorithmic freedom, our design is meant to be used as a pure ray tracing co-processor. That is, no shading is done on the FPGA and there is no display attached directly to the board. The memory bandwidth problems resulting from moving rays to the board and receiving the intersection data can be tackled by the use of the two gigabit Ethernet interfaces and the PCI-Express interface. As Intel recently opened up the front side bus (FSB) for FPGA co-processors, this will probably not be an issue in the future. With regard to this prospect, we did not spend much time on interface driver development but rather on the ray tracing core itself.

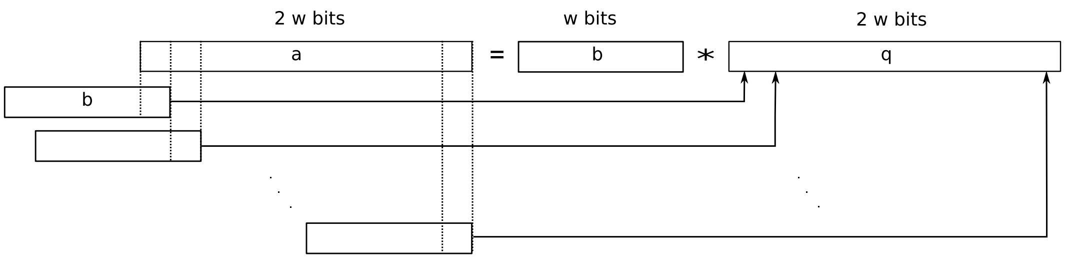
This is why in our example implementation the block-RAM representing caches for the BIH and triangle data are implemented as ROM rather than feeding them with data obtained from the DDR2 memory, and the rays are generated using a small on-board unit instead of reading them from PCI-Express. Currently, the completely built BIH and the accelerated representation of the triangles are hardcoded and moved to the ROM during FPGA configuration.

The toplevel design can be seen in Figure 3.2.

## 3.2 Ray/Plane Intersection

As the division needed in ray/triangle intersection is the most computationally complex part of tracing a ray, it should receive some extra attention. Recall that we need to calculate (see Equation 2.1):

$$t = \frac{\mathrm{d} - \frac{1}{n_r}\langle O, n\rangle}{\frac{1}{n_r}\langle \omega, n\rangle} = \frac{\mathrm{d} - O_r - O_p\,\mathrm{np} - O_q\,\mathrm{nq}}{\omega_r + \omega_p\,\mathrm{np} + \omega_q\,\mathrm{nq}}.$$



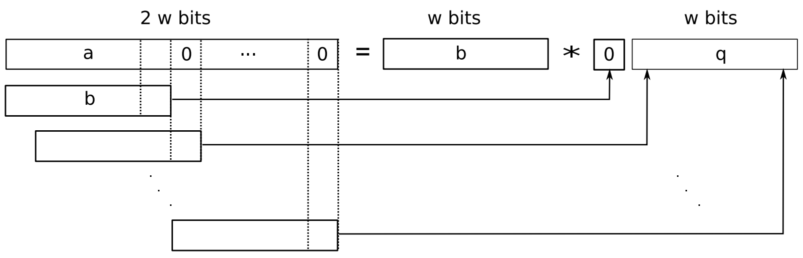


Figure 3.3: Schematic layout of our divider. On the top is a rough sketch of what a complete divider for a $2w/w$-bit divison would look like. The result has to be of width $2w$ again, as $b$ could be equal to one, so $q = a$. We also could not just cut off the least significiant bits, as these are exactly the ones needed for the location inside the bounding box. The bottom image shows our implementation of a divider, requiring $a < b$, so the $w$-th bit of $q$ will be zero.

The numerator is quantized as a signed fixed point value in $[-2^{\mathtt{n}+2}+1, +2^{\mathtt{n}+2}-1]$ using $\mathtt{n}+3$ bits, the denominator is in $[-1,1]$ using $\mathtt{m}$ bits. So for non-negative numbers a general division $n/d$ with $n \in [0, 2^{\mathtt{n}+2}-1]$ and $d \in [2^{-\mathtt{m}+1}, 1]$ would result in a quotient $q \in [0, 2^{\mathtt{n}+\mathtt{m}+1} - 2^{\mathtt{m}-1}]$, which requires an $(\mathtt{n}+\mathtt{m}+1)$-bit integer number.

For simplicity, we decided to implement a sequential, non-restoring divider [Koro2] and additionally exploit the fact that for our application, the interesting values of the quotient $t$ are only inside the bounding box: $t \in [1, 2^{\mathtt{n}+1}-1]$. All other values of $t$ indicate that the triangle has been missed, since all the triangles are contained in the bounding box. This also assumes, as noted in Section 2.1, that the ray origin is clipped to the bounding box before tracing. A quantization error of 1 is acceptable when we want the considerations of Sections 2.1.5 to hold. So it is sufficient to calculate $\mathtt{n}+1$ bits and set an error bit if $t < 1$ or $t > 2^{\mathtt{n}+1}$. The divider takes a w-bit numerator $a$ and a w-bit denominator $b$ as inputs and returns a w-bit quotient $q$:

$$q = \frac{a \cdot 2^w}{b},$$

which already takes care of the ranges of the fixed point numbers we want to use as input. To assure that there is no overflow in the quotient, the unit asserts

$$q < 2^w \Leftrightarrow \frac{a \cdot 2^w}{b} < 2^w \Leftrightarrow a < b.$$

See Figure 3.3 for a rough idea of the component. We thus only need $\mathtt{n}+1$ steps for the division as compared to $\mathtt{n}+\mathtt{m}+1$ steps necessary for the full division. The complete delay of the component `t_div` is $\mathtt{n}+3$ clock cycles, to make the output registered and to take care of the sign.

Note that we did not yet explore the full potential of fast dividers (e.g. by using a greater radix, through multiplication or finding the reciprocal using Newton-Raphson iteration [Koro2, Chapter 8]). But as these methods require significantly more complex units on the smallest scale (multiplications instead of controlled add/subtract) and sometimes even an initial lookup table implemented in ROM, it is not as clear as for floating point arithmetic that these methods would result in a performance gain.

The full VHDL source code for this divider can be found in Appendix A.1.

## 3.3 Ray/Triangle Intersection

The ray/triangle intersection test is straightforward to translate to VHDL after the division has been taken care of. Our version does not involve a state machine and thus does not have the possibility of an early out but is very suitable for pipelining instead. The pipeline can be seen in Figure 3.4, for $\mathtt{n} = 29$ and $\mathtt{m} = 32$ it has a delay of 41 clock cycles and is able to compute one ray/triangle intersection per cycle.

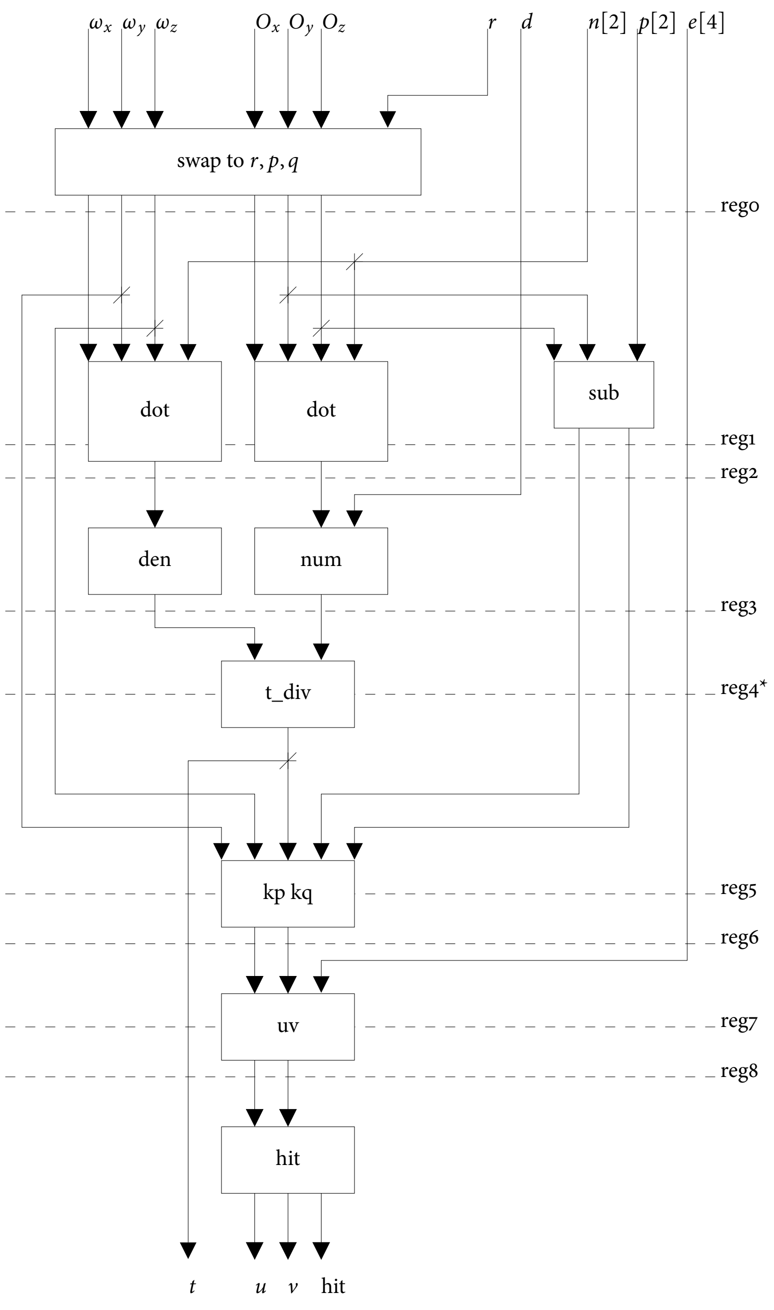


Figure 3.4: The pipeline of the Ray/Triangle intersection unit. Each register reflects one clock cycle, except *reg4**, which depends on the divider ($n + 2$ registers in our implementation, the divider is fully pipelined).

The delay can be reduced by making the combinatorial logic deeper in between the registers, but this will affect the maximum clock rate in turn.

Please refer to Appendix A.2 for the source code of this module.

## 3.4 Acceleration

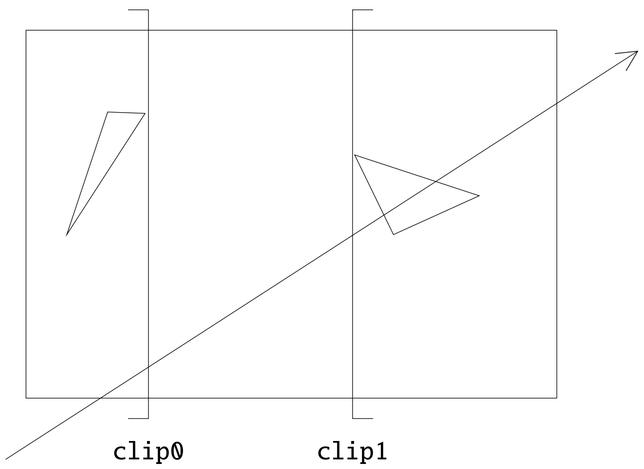


Figure 3.5: One step during BIH traversal: decide which child node has to be traversed. The left one is bounded to the right by the plane `clip0`, the right is bounded to the left by `clip1`.

As mentioned, we use a bounding interval hierarchy (BIH) [WK06] as an acceleration structure. This is a data structure similar to *k*d-trees and the B-KD tree [WMS06]. While a *k*d-tree is a space partitioning structure, the BIH and the B-KD tree partition the object list. The scene data is recursively divided in two sub-volumes, to form a tree. The BIH is a special case of a bounding volume hierarchy which uses only two split planes to partition the objects. On one level of the tree, a split plane candidate is chosen according to some heuristic. Then the objects are classified to be completely to the left or completely to the right of the split plane. It is advantageous to treat the overlapping objects as one object and make one decision for the whole block [WK07]. The child node bounding boxes are then adjusted to contain all their objects. That is, the left child is bounded to the right by `clip0` and the right child to the left by `clip1`, as illustrated in Figure 3.5. This is repeated until only fewer than a certain number of primitives remain in a node. In this case a leaf node is formed.

Triangles which have to be tested for intersection with a given ray can now be found efficiently by traversing the BIH. This is done by starting at the root node and iteratively deciding which of the two child nodes have to be traversed, depending on which volume the ray overlaps. If both nodes have to be examined, one is pushed onto a stack and the closer one is traversed first. If both of the child nodes can be pruned or the traversal has arrived at a leaf, the stack is popped.

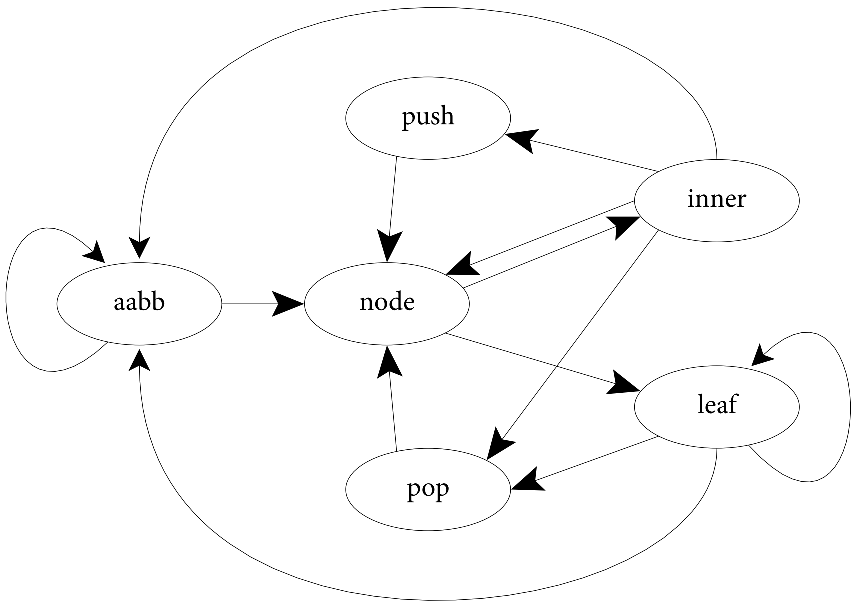


Figure 3.6: The finite state machine of the bounding interval hierarchy.

Only the traversal part of the BIH has been implemented so far. That is, the acceleration structure is not built on the FPGA but this can easily be done, even in limited memory [WK07]. The algorithm has been transformed into a finite state machine, as shown in the diagram in Figure 3.6. State transitions are described in Table 3.1, the VHDL source is in Appendix A.3.

Four 36Kbit block-RAM units are used as cache and traversal stack at a time. This limits stack size to a fixed maximum. However, if the tree is constructed on-chip, this limit can always be met. The cache is not currently connected to the memory controller, due to time constraints which prohibited the implementation of a DDR2 memory core. Such an implementation should do careful prefetching to hide memory latency [WSS05, WMS06].

There are some issues connecting the BIH output to the ray/triangle intersection unit. Firstly, the BIH is a state machine and outputs triangle numbers, as long as it is in state *leaf*. That is, when the BIH is busy traversing the inner tree, the pipeline runs empty. This could be overcome by using more than one BIH traversal unit and connect them all to a buffer which in turn is connected to one single ray/triangle intersection unit. This would also address a problem specific to our implementation, that the BIH unit only runs at lower clock rates due to deep combinatorial logic. This could be avoided by another implementation by increasing the number of states and thus the number of clock cycles spent on inner node traversal without output.

Another unimplemented thing is the early-out capability of the BIH traversal algorithm. The child nodes do not have to be traversed, if the entry point is further away than a previously found intersection with a triangle. As the delay of the intersection unit is quite long, the information sent back from the intersection result would reach the BIH much too late.

| state | next state | output | comment |
|---|---|---|---|
| aabb | aabb | done | if axis-aligned bounding box is missed |
| | node | | root node loaded |
| node | inner | | if axis $\in \{00, 01, 10\}$ |
| | leaf | | if axis = 11 |
| inner | node | | traverse one child |
| | push | | traverse two children |
| | pop | | miss both children |
| | aabb | done | miss both and stack empty |
| push | node | | store stack, then restore second node |
| pop | node | | restore stack |
| leaf | leaf | tri_num, tri_en | output triangle for intersection |
| | pop | | done dumping all tri nums |
| | aabb | done | done and stack empty |

Table 3.1: State transitions in the finite state machine for the Bounding Interval Hierarchy.

## 3.5 Results

A fully synthesizable version of the ray generator, the BIH traversal, the ray/triangle intersection unit and connecting logic have been implemented in VHDL. This only uses up about 11% of the slices of the Virtex-5. For detailed device utilization statistics see Table 3.2.

The maximum net delay is 4.231 nanoseconds, so the design could run at 233MHz, generating a ray/triangle intersection each clock cycle. The actual throughput of ray/boundary intersections depends on the traversal of the acceleration structure, which typically takes $O(\log(N))$ clock cycles, $N$ being the number of triangles in the scene.

Total power consumption is currently 866.14 mW, resulting in an estimated junction temperature of 34° C, but this number is likely to increase when data in- and output interfaces are added.

Due to the lack of interface cores, the only images generated by our implementation originate from the simulator. The resulting image of a hardcoded teapot without any shading applied can be seen in Figure 3.7. Note that our implementation does output all intersection information necessary for shading as the barycentric coordinates, distance along the ray to the intersection and triangle index.

| Slice Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| Number of Slice Registers | 3,274 | 69,120 | 4% |
| Number used as Flip Flops | 3,117 | | |
| Number used as Latches | 157 | | |
| Number of Slice LUTs | 5,278 | 69,120 | 7% |
| Number used as logic | 4,785 | 69,120 | 6% |
| Number using O6 output only | 4,344 | | |
| Number using O5 output only | 75 | | |
| Number using O5 and O6 | 366 | | |
| Number used as Memory | 488 | 17,920 | 2% |
| Number used as Shift Register | 488 | | |
| Number using O6 output only | 488 | | |
| Number used as exclusive route-thru | 5 | | |
| Number of route-thrus | 602 | 138,240 | 1% |
| Number using O6 output only | 80 | | |
| Number using O5 output only | 522 | | |
| Slice Logic Distribution | | | |
| Number of occupied Slices | 1,963 | 17,280 | 11% |
| Number of LUT Flip Flop pairs used | 5,958 | | |
| Number with an unused Flip Flop | 2,684 | 5,958 | 45% |
| Number with an unused LUT | 680 | 5,958 | 11% |
| Number of fully used LUT-FF pairs | 2,594 | 5,958 | 43% |
| Number of unique control sets | 68 | | |
| IO Utilization | | | |
| Number of bonded IOBs | 5 | 640 | 1% |
| IOB Flip Flops | 1 | | |
| Specific Feature Utilization | | | |
| Number of BlockRAM/FIFO | 11 | 148 | 7% |
| Number using BlockRAM only | 11 | | |
| Total primitives used | | | |
| Number of 18k BlockRAM used | 22 | | |
| Total Memory used (KB) | 396 | 5,328 | 7% |
| Number of BUFG/BUFGCTRLs | 3 | 32 | 9% |
| Number used as BUFGs | 3 | | |
| Number of DSP48Es | 52 | 64 | 81% |
| Total equivalent gate count for design | 1,581,587 | | |
| Additional JTAG gate count for IOBs | 240 | | |

Table 3.2: Device Utilization Summary as generated by Xilinx ISE.

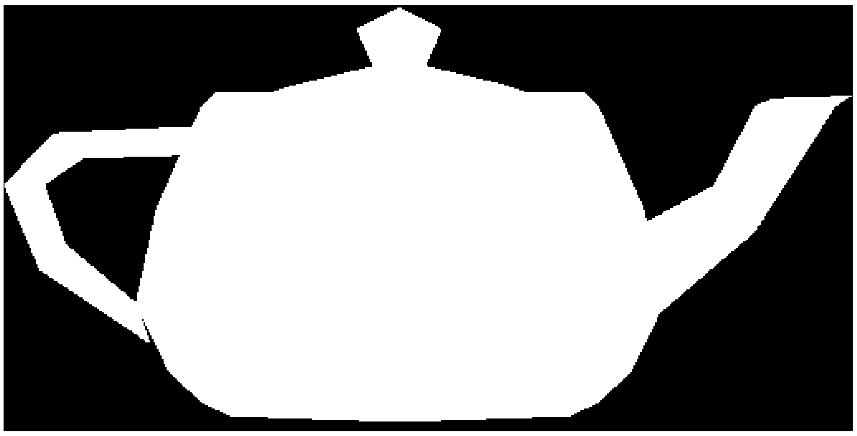


Figure 3.7: Output of the GHDL simulation: binary teapot image showing only boolean intersection results. The gap between the handle and the body of the teapot is due to the tesselation of the Bézier patches.

# 4 Conclusion and Future Work

## 4.1 Conclusion

We showed that ray tracing in fixed point arithmetic and thus a hardware implementation can achieve equally convincing results as ray tracing in floating point arithmetic. Due to the equidistant spacing of the numbers, quantization artifacts are more controllable and typical problems with floating point special cases can be avoided more easily.

A similar analysis can be applied to other triangle tests [MT97, Jon00] and we expect large performance benefits from applying fixed point arithmetic to the high-precision ray tracing algorithms for free-form surfaces [DK06].

Since integer functionality is included in hardware description languages such as VHDL, a hardware description is very simple and on FPGAs even built-in integer functionality can be exploited.

We currently have a basic, fully pipelined implementation of the triangle test using a specialized divider unit as well as the BIH traversal implemented on a Virtex-5 FPGA. Only very few resources have been utilized, as indicated in Section 3.5. This leaves a lot of room for future extensions.

## 4.2 Future Work

Although our system is capable of tracing a ray, there still remain many things to do, the most obvious being the interface to the computer. It should be very interesting to see the PCI-Express bus connected to the core, continuously streaming rays generated by some simulation application onto the board.

This directly leads to the next open task, namely initially moving the geometry into the on-board DDR2 memory, building the BIH and initializing the triangles' accelerated representations on the way.

Only then can a real attempt be made to estimate the overall performance because it will then be known how much area is still left on the FPGA to be used as parallel ray pipelines [WSS05].

The possibility to directly attach the Virtex-5 to the front side bus and use the system memory is very appealing and should be tried as soon as this is practicable.

The implementation is not optimized very intensively. There are lots of things to be tuned for efficiency:

- The BIH comes without clipnodes (BVH2 case [HHS06])
- There are a lot of parameters to be tuned: the number of triangles per leaf, BIH tree depth, how many BIH traversal units, the combinatorial depth of the pipeline, ...
- Currently, the ray/triangle intersection pipeline has a very poor fill rate of about 40%. This should be possible to overcome by detaching the iteration through the triangle list in the leaf state from the BIH state machine.
- The parameter for the precision of the division has been left at `T = 0` for the hardware implementation. Clock cycles could be saved by adjusting it according to the precision needs of the application.

Section 2.1.5 gave an analysis of the ray tracing precision, concluding that a scene and location independent epsilon value of 2 is sufficient to avoid self-intersections. It is desirable to attain results which are completely accurate rounded to the integer grid. To achieve this, the error involved in quantization of the rays and triangles before tracing the ray has to be considered as well. This is a useful extension but not related to self-intersection, as this problem only arises after quantizing the triangles and rays. One drawback of the employed Badouel intersection test in that context is that converting the triangles to the precomputed representation might result in leaks in the quantized mesh. Other triangle intersection tests should be analyzed with respect to fixed point as well.

An interesting way to incorporate early out strategies could be using a *k*d-tree as acceleration structure and a fast divisionless triangle intersection. If a triangle intersection is reported in the near node, the far node does not have to be traversed, as it would be further away from the ray's origin. However building a *k*d-tree is significantly more involved than building the BIH, so this part is even more challenging in a hardware implementation.

# Bibliography

[Bad90] D. Badouel. An efficient ray-polygon intersection. In A. S. Glassner, editor, *Graphics Gems*, pages 390–393. Academic Press Professional, 1990.

[BB85] J. Barlow and E. Bareiss. On roundoff error distributions in floating point and logarithmic arithmetic. *Computing*, 34:325–347, 1985.

[Ben06] C. Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.

[CHH02] N. Carr, J. Hall, and J. Hart. The ray engine. In *Graphics Hardware (Proc. Eurographics/SIGGRAPH 2002)*, pages 37–46, 2002.

[Chi05] N. Chirkov. Fast 3d line segment-triangle intersection test. *Journal of graphics tools*, 10(3):13–18, 2005.

[DK06] H. Dammertz and A. Keller. Improving ray tracing precision by world space intersection computation. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pages 25–32, September 2006.

[Fen02] J. Fender. The design and implementation of a hardware accelerated raytracer using the TM3a FPGA prototyping system. Bachelor thesis, University of Toronto, March 2002.

[Gai] J. Gaisler. A structured VHDL design method. `http://www.gaisler.com/doc/vhdl2proc.pdf`.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[HHS06] V. Havran, R. Herzog, and H.-P. Seidel. On the fast construction of spatial data structures for ray tracing. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pages 71–80, September 2006.

[Jon00] R. Jones. Intersecting a ray and a triangle with Plücker coordinates. *Ray Tracing News*, 13(1), July 2000.

[KNKL07] Sung-Soo Kim, Seung-Woo Nam, Do-Hyung Kim, and In-Ho Lee. Hardware-accelerated ray-triangle intersection testing for high-performance collision detection. In *Proc. Winter School of Computer Graphics*, 2007.

[Kor02] I. Koren. *Computer Arithmetic Algorithms*. A. K. Peters Ltd., 2nd edition, 2002.

[KS06] A. Kensler and P. Shirley. Optimizing ray-triangle intersection via automated search. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, pages 33–38, September 2006.

[MST98] J.-M. Muller, A. Scherbyna, and A. Tisserand. Semi-logarithmic number systems. *IEEE Trans. Comput.*, 47(2):145–151, 1998.

[MT97] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[PBMH05] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH 2005 Course Notes*, page 268, 2005.

[Pur04] T. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, March 2004.

[SF01] R. Segura and F. Feito. Algorithms to test ray-triangle intersection. In *Proc. Winter School of Computer Graphics*, 2001.

[SWW$^+$04] J. Schmittler, S. Woop, D. Wagner, W. Paul, and P. Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Graphics Hardware (Proc. SIGGRAPH/Eurographics 2004)*, pages 95–106, 2004.

[Wal04] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[WK06] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 (Proc. 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006.

[WK07] C. Wächter and A. Keller. Terminating spatial partition hierarchies by a priori bounding memory. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing*, page to appear, 2007.

[WMS06] S. Woop, G. Marmitt, and P. Slusallek. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware (Proc. Eurographics/SIGGRAPH 2006)*, pages 67–77, 2006.

[Woo06] S. Woop. *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes*. PhD thesis, Saarland University, 2006.

[WPO96] A. Woo, A. Pearce, and M. Ouellette. It's really not a rendering bug, you see... *IEEE Computer Graphics & Applications*, 16(5):21–25, September 1996.

[Wri97] A. Wrigley. Method of and apparatus for constructing an image of a notional scene by a process of ray tracing, May 1997. United States Patent US 5,933,146, 28.05.1997.

[WSS05] S. Woop, J. Schmittler, and P. Slusallek. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)*, 24(3):434–444, 2005.

# A VHDL Code

## A.1 Specialized Divider

Listing A.1: Specialized divider

```
-- this is part of intrt , a fixed point hardware raytracer
-- Copyright (C) 2007 Johannes Hanika
--
-- This program is free software: you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation, either version 3 of the License, or
-- (at your option) any later version.
--
-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
-- GNU General Public License for more details.
--
-- You should have received a copy of the GNU General Public License
-- along with this program. If not, see <http://www.gnu.org/licenses/>.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- controlled add/subtract unit with overflow, ((signed w) & (w-1 x 0))/(signed w) = signed w
-- specialized for triangle test: if q is < 0, it could have been an intersection error (hit outside the aabb).
-- sign is used as error bit as well (since t > 0)
entity t_div is
  generic(w : integer := 16);
  port
  (
    clk : in std_logic;
    n   : in  std_logic_vector(w-1 downto 0); -- signed nominator
    d   : in  std_logic_vector(w-1 downto 0); -- signed denominator
    q   : out std_logic_vector(w-1 downto 0) -- signed quotient (all 2s complement)
  );
end entity;

architecture rtl of t_div is
  -- constant ZERO : std_logic_vector (w - 2 downto 0) := (others => '0');
  type reg_t is record
    qn    : std_logic_vector (2 w-4 downto 0); -- w-1 + w-1 - 1 bits
    d     : std_logic_vector (w-2 downto 0);
    sign  : std_logic;
  end record;

  type pipe_t is array (0 to w-1) of reg_t;

  signal r   : pipe_t;
  signal rin : pipe_t;
begin

comb : process(r, n, d)
variable v : pipe_t;
begin
  v := r;

  -- read signed input, convert 2s complement to sign

  v(w-1).sign := n(w-1) xor d(w-1);
  -- qn is nom & 0 and becomes q & remainder.
  -- this causes overflows when d << n, but in this case, err is set and
  -- the triangle is not intersected, because hit would be outside the aabb.
  -- we use sign as err bit, because if t < 0 => no intersection .
  v(w−1).qn(w−3 downto 0) := (others => '0');
  if (n(w−1) = '1') then
    v(w−1).qn(2w−4 downto w−2) := not(n(w−2 downto 0)) + 1;
  else
    v(w−1).qn(2w−4 downto w−2) := n(w−2 downto 0);
  end if;

  if (d(w−1) = '1') then
    v(w−1).d := (not(d(w−2 downto 0)) + 1);
  else
    v(w−1).d := d(w−2 downto 0);
  end if;
  −−
  v(w−2).d      := r(w−1).d;
  −− detect overflow and set sign bit.
  if (r(w−1).qn(2w−4 downto w−2) >= r(w−1).d) then
    v(w−2).sign := '1';
  else
    v(w−2).sign := r(w−1).sign;
  end if;
  v(w−2).qn(w−3 downto 0)     := r(w−1).qn(w−3 downto 0);
  v(w−2).qn(2w−4 downto w−2) := r(w−1).qn(2w−4 downto w−2) − r(w−1).d;
  for i in w−3 downto 0 loop
    v(i).d      := r(i+1).d;
    v(i).sign  := r(i+1).sign;
    v(i).qn(2w−4 downto i + w−1) := r(i+1).qn(2w−4 downto i + w−1);
    v(i).qn(i − 1 downto 0)          := r(i+1).qn(i − 1 downto 0);
    if (r(i+1).qn(i + w − 1) = '0') then
      v(i).qn(i + w − 2 downto i) := r(i+1).qn(i + w − 2 downto i) − r(i+1).d;
    else
      v(i).qn(i + w − 2 downto i) := r(i+1).qn(i + w − 2 downto i) + r(i+1).d;
    end if;
  end loop;
  −− generate output from last register (0)
  −− correct for signed values:
  −− if (r(0).sign = '1') then
  −−   q <= '1' & (r(0).qn(2w−4 downto w−2) + 1);
  −− else
  −−   q <= '0' & (not(r(0).qn(2w−4 downto w−2)));
  −− end if;
  −− but neg values are errors anyways:
  if (r(0).d = ZERO) then
    q <= '1' & (not(r(0).qn(2w−4 downto w−2)));
  else
    q <= r(0).sign & (not(r(0).qn(2w−4 downto w−2)));
  end if;
  rin <= v;
end process;

clocked : process(clk)
begin
  if rising_edge(clk) then
    r <= rin;
  end if;
end process;

end architecture;
```

## A.2 Ray/Triangle Intersection

Listing A.2: Ray/triangle intersection

```
−− this is part of intrt, a fixed point hardware raytracer
−− Copyright (C) 2007 Johannes Hanika
−−
−− This program is free software: you can redistribute it and/or modify
−− it under the terms of the GNU General Public License as published by
−− the Free Software Foundation, either version 3 of the License, or
−− (at your option) any later version.
−−
−− This program is distributed in the hope that it will be useful,
−− but WITHOUT ANY WARRANTY; without even the implied warranty of
−− MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
−− GNU General Public License for more details.
−−
−− You should have received a copy of the GNU General Public License
−− along with this program. If not, see <http://www.gnu.org/licenses/>.
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.intrt_p.all;

package tri_int_p is
  type tri_int_in_t is record
    -- ray pos O
    pos_x : std_logic_vector(n downto 0);
    pos_y : std_logic_vector(n downto 0);
    pos_z : std_logic_vector(n downto 0);
    -- ray dir omega
    dir_x : std_logic_vector(m-1 downto 0);
    dir_y : std_logic_vector(m-1 downto 0);
    dir_z : std_logic_vector(m-1 downto 0);

    -- triangle
    pp,pq             : std_logic_vector(n-1 downto 0);
    d                 : std_logic_vector(n+2 downto 0);
    e1p,e1q,e2p,e2q   : std_logic_vector(m-1 downto 0);
    np,nq             : std_logic_vector(m-1 downto 0);
    r                 : std_logic_vector(1 downto 0);

    -- enable
    en    : std_logic;
  end record;

  type tri_int_out_t is record
    hit  : std_logic;
    u    : std_logic_vector(m-1 downto 0);
    v    : std_logic_vector(m-1 downto 0);
    t    : std_logic_vector(n+1 downto 0);
    rdy  : std_logic; -- passed through en
  end record;

  component tri_int
  port
  (
    clk  : in  std_logic;
    d    : in  tri_int_in_t;
    q    : out tri_int_out_t
  );
  end component;
end package;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use work.tri_int_p.all;
use work.intrt_p.all;

entity tri_int is
  port
  (
    clk  : in  std_logic;
    d    : in  tri_int_in_t;
    q    : out tri_int_out_t
  );
end entity;

architecture rtl of tri_int is
  constant ZERO  : std_logic_vector(n+2-m-1 downto 0) := (others => '0');
  constant LOSER : std_logic_vector(m-2 downto 0) := (others => '0');
  -- registers for pipeline
  type reg0_t is record
    pos_p,pos_q,pos_r : std_logic_vector(n   downto 0);
    dir_p,dir_q,dir_r : std_logic_vector(m-1 downto 0);
    d                 : std_logic_vector(n+2 downto 0);
    pp,pq             : std_logic_vector(n-1 downto 0);
    np,nq             : std_logic_vector(m-1 downto 0);
    e1p,e1q,e2p,e2q   : std_logic_vector(m-1 downto 0);
    en                : std_logic;
  end record;
  type reg1_t is record
    dir_p,dir_q,dir_r : std_logic_vector(m-1 downto 0);
    pos_r             : std_logic_vector(n   downto 0);
    d                 : std_logic_vector(n+2 downto 0);
    pmpp,pmpq         : std_logic_vector(n   downto 0);
    e1p,e1q,e2p,e2q   : std_logic_vector(m-1 downto 0);
    t1,t2             : std_logic_vector(n+1 downto 0);
    t3,t4             : std_logic_vector(m-1 downto 0);
    en                : std_logic;
  end record;
  type reg2_t is record
    dir_p, dir_q       : std_logic_vector (m−1 downto 0);
    d                  : std_logic_vector (n+2 downto 0);
    pre_nom            : std_logic_vector (n+1 downto 0);
    pre_den            : std_logic_vector (m−1 downto 0);
    pmpp,pmpq          : std_logic_vector (n   downto 0);
    e1p,e1q,e2p,e2q    : std_logic_vector (m−1 downto 0);
    en                 : std_logic ;
  end record;
  type reg3_t is record
    dir_p, dir_q       : std_logic_vector (m−1 downto 0);
    nom,den            : std_logic_vector (n+2 downto 0);
    pmpp,pmpq          : std_logic_vector (n   downto 0);
    e1p,e1q,e2p,e2q    : std_logic_vector (m−1 downto 0);
    en                 : std_logic ;
  end record;
  type reg4_t is record
    dir_p, dir_q       : std_logic_vector (m−1 downto 0);
    pmpp,pmpq          : std_logic_vector (n   downto 0);
    e1p,e1q,e2p,e2q    : std_logic_vector (m−1 downto 0);
    en                 : std_logic ;
  end record;
  −− TODO: adjust comb depth in t_div ?
  type regdiv_array is array (0 to n+2) of reg4_t;

  type reg5_t is record
    t                  : std_logic_vector (n+2 downto 0);
    e1p,e1q,e2p,e2q    : std_logic_vector (m−1 downto 0);
    pmpp,pmpq          : std_logic_vector (n   downto 0);
    t1 , t2            : std_logic_vector (m−1 downto 0);
    en                 : std_logic ;
  end record;
  type reg6_t is record
    kp,kq              : std_logic_vector (n+1 downto 0);
    t                  : std_logic_vector (n+2 downto 0);
    e1p,e1q,e2p,e2q    : std_logic_vector (m−1 downto 0);
    en                 : std_logic ;
  end record;
  type reg7_t is record
    t                : std_logic_vector (n+2 downto 0);
    t1 , t2 , t3 , t4  : std_logic_vector (n+m+1 downto 0);
    en               : std_logic ;
  end record;
  type reg8_t is record
    t   : std_logic_vector (n+2 downto 0);
    u   : std_logic_vector (n+m+1 downto 0);
    v   : std_logic_vector (n+m+1 downto 0);
    en  : std_logic ;
  end record;

  type reg_t is record
    reg0 : reg0_t;
    reg1 : reg1_t;
    reg2 : reg2_t;
    reg3 : reg3_t;
    reg4 : regdiv_array;
    reg5 : reg5_t;
    reg6 : reg6_t;
    reg7 : reg7_t;
    reg8 : reg8_t;
  end record;

  component t_div is
    generic(w : integer := 16);
    port
    (
      clk  : in  std_logic ;
      n    : in  std_logic_vector (w−1 downto 0);
      d    : in  std_logic_vector (w−1 downto 0);
      q    : out std_logic_vector (w−1 downto 0)
    );
  end component;

  signal r, rin : reg_t;
  signal t_div_output : std_logic_vector (n+2 downto 0);
begin

−− TODO save some logic here (parameter T )!
−− finishes after w = n+3 clock cycles .
div      : t_div      generic map(n+3) port map(clk, r.reg3.nom, r.reg3.den, t_div_output );

comb : process(r, d, t_div_output)
variable v : reg_t;
variable t1, t2 : std_logic_vector (2 n+2 downto 0);
variable t3, t4 : std_logic_vector (2 m−1 downto 0);
variable sum : std_logic_vector (m+n+1 downto 0);
begin
  v := r;
  −− propagate registers .
  −− reg0
  case d.r is
  when "00" =>
    v.reg0.pos_r(n downto 0) := d.pos_x(n downto 0);
    v.reg0.pos_p(n downto 0) := d.pos_y(n downto 0);
    v.reg0.pos_q(n downto 0) := d.pos_z(n downto 0);
    v.reg0.dir_r(m−1 downto 0) := d.dir_x(m−1 downto 0);
    v.reg0.dir_p(m−1 downto 0) := d.dir_y(m−1 downto 0);
    v.reg0.dir_q(m−1 downto 0) := d.dir_z(m−1 downto 0);
  when "01" =>
    v.reg0.pos_q(n downto 0) := d.pos_x(n downto 0);
    v.reg0.pos_r(n downto 0) := d.pos_y(n downto 0);
    v.reg0.pos_p(n downto 0) := d.pos_z(n downto 0);
    v.reg0.dir_q(m−1 downto 0) := d.dir_x(m−1 downto 0);
    v.reg0.dir_r(m−1 downto 0) := d.dir_y(m−1 downto 0);
    v.reg0.dir_p(m−1 downto 0) := d.dir_z(m−1 downto 0);
  when others =>
    v.reg0.pos_p(n downto 0) := d.pos_x(n downto 0);
    v.reg0.pos_q(n downto 0) := d.pos_y(n downto 0);
    v.reg0.pos_r(n downto 0) := d.pos_z(n downto 0);
    v.reg0.dir_p(m−1 downto 0) := d.dir_x(m−1 downto 0);
    v.reg0.dir_q(m−1 downto 0) := d.dir_y(m−1 downto 0);
    v.reg0.dir_r(m−1 downto 0) := d.dir_z(m−1 downto 0);
  end case;
  v.reg0.d    := d.d;
  v.reg0.pp   := d.pp;
  v.reg0.pq   := d.pq;
  v.reg0.e1p := d.e1p;
  v.reg0.e1q := d.e1q;
  v.reg0.e2p := d.e2p;
  v.reg0.e2q := d.e2q;
  v.reg0.np   := d.np;
  v.reg0.nq   := d.nq;
  v.reg0.en   := d.en;

  −− reg1
  −− first step in dotproduct 1:
  t1 := r.reg0.pos_p r.reg0.np;
  t2 := r.reg0.pos_q r.reg0.nq;
  v.reg1.t1 := t1 (2 n + 2 downto m−1);
  v.reg1.t2 := t2 (2 n + 2 downto m−1);
  −− dotproduct 2:
  t3 := r.reg0.dir_p r.reg0.np;
  t4 := r.reg0.dir_q r.reg0.nq;
  v.reg1.t3 := t3 (2 m−2 downto m−1);
  v.reg1.t4 := t4 (2 m−2 downto m−1);
  −− both unsigned
  v.reg1.pmpp := r.reg0.pos_p − ('0'& r.reg0.pp);
  v.reg1.pmpq := r.reg0.pos_q − ('0'& r.reg0.pq);
  v.reg1.pos_r := r.reg0.pos_r;
  v.reg1.dir_p := r.reg0.dir_p;
  v.reg1.dir_q := r.reg0.dir_q;
  v.reg1.dir_r := r.reg0.dir_r;
  v.reg1.d := r.reg0.d;
  v.reg1.e1p := r.reg0.e1p;
  v.reg1.e1q := r.reg0.e1q;
  v.reg1.e2p := r.reg0.e2p;
  v.reg1.e2q := r.reg0.e2q;
  v.reg1.en  := r.reg0.en;

  −− reg2
  v.reg2.dir_p := r.reg1.dir_p;
  v.reg2.dir_q := r.reg1.dir_q;
  v.reg2.d      := r.reg1.d;
  v.reg2.e1p    := r.reg1.e1p;
  v.reg2.e1q    := r.reg1.e1q;
  v.reg2.e2p    := r.reg1.e2p;
  v.reg2.e2q    := r.reg1.e2q;
  v.reg2.pmpp := r.reg1.pmpp;
  v.reg2.pmpq := r.reg1.pmpq;
  −− rest of dot1: signed (n+2) + signed (n+2) + unsigned (n+1)
  v.reg2.pre_nom := r.reg1.t1 + r.reg1.t2 + ('0'& r.reg1.pos_r);
  −− rest of dot2
  v.reg2.pre_den := r.reg1.t3 + r.reg1.t4 + r.reg1.dir_r;
  v.reg2.en     := r.reg1.en;

  −− reg3
  v.reg3.dir_p := r.reg2.dir_p;
```

```
v.reg3.dir_q  :=  r.reg2.dir_q;
v.reg3.e1p    :=  r.reg2.e1p;
v.reg3.e1q    :=  r.reg2.e1q;
v.reg3.e2p    :=  r.reg2.e2p;
v.reg3.e2q    :=  r.reg2.e2q;
v.reg3.pmpp :=  r.reg2.pmpp;
v.reg3.pmpq :=  r.reg2.pmpq;
v.reg3.nom   :=  r.reg2.d − r.reg2.pre_nom;
v.reg3.den :=  r.reg2.pre_den(m−1) & r.reg2.pre_den;
v.reg3.en   :=  r.reg2.en;

−− reg4
v.reg4 (0). pmpp :=  r.reg3.pmpp;
v.reg4 (0). pmpq :=  r.reg3.pmpq;
v.reg4 (0). dir_p  :=  r.reg3.dir_p;
v.reg4 (0). dir_q  :=  r.reg3.dir_q;
v.reg4 (0). e1p    :=  r.reg3.e1p;
v.reg4 (0). e1q    :=  r.reg3.e1q;
v.reg4 (0). e2p    :=  r.reg3.e2p;
v.reg4 (0). e2q    :=  r.reg3.e2q;
v.reg4 (0). en      :=  r.reg3.en;
−− have to  wait   until   div   finishes .  in  the  meantime, forward  rest  of  the  pipe.
for  i  in  1  to  n+2 loop
   v.reg4( i ). pmpp :=  r.reg4( i −1).pmpp;
   v.reg4( i ). pmpq :=  r.reg4( i −1).pmpq;
   v.reg4( i ). dir_p  :=  r.reg4( i −1).dir_p;
   v.reg4( i ). dir_q  :=  r.reg4( i −1).dir_q;
   v.reg4( i ). e1p    :=  r.reg4( i −1).e1p;
   v.reg4( i ). e1q    :=  r.reg4( i −1).e1q;
   v.reg4( i ). e2p    :=  r.reg4( i −1).e2p;
   v.reg4( i ). e2q    :=  r.reg4( i −1).e2q;
   v.reg4( i ). en      :=  r.reg4( i −1).en;
end loop;

−− reg5
t3  :=  t_div_output(m downto 1)r.reg4(n+2).dir_p;
t4  :=  t_div_output(m downto 1)r.reg4(n+2).dir_q;
v.reg5.t1  :=  t3 (2 m−3 downto m−2);
v.reg5.t2  :=  t4 (2 m−3 downto m−2);
v.reg5.t     :=  t_div_output;
v.reg5.pmpp:= r.reg4(n+2).pmpp;
v.reg5.pmpq:= r.reg4(n+2).pmpq;
v.reg5.e1p :=  r.reg4(n+2).e1p;
v.reg5.e1q :=  r.reg4(n+2).e1q;
v.reg5.e2p :=  r.reg4(n+2).e2p;
v.reg5.e2q :=  r.reg4(n+2).e2q;
v.reg5.en   :=  r.reg4(n+2).en;

−− reg6
v.reg6.kp   :=  r.reg5.pmpp + r.reg5.t1;
v.reg6.kq   :=  r.reg5.pmpq + r.reg5.t2;
v.reg6.t     :=  r.reg5.t;
v.reg6.e1p :=  r.reg5.e1p;
v.reg6.e1q :=  r.reg5.e1q;
v.reg6.e2p :=  r.reg5.e2p;
v.reg6.e2q :=  r.reg5.e2q;
v.reg6.en   :=  r.reg5.en;

−− reg7
v.reg7.t   :=  r.reg6.t;
v.reg7.t1  :=  r.reg6.e1p    r.reg6.kq;
v.reg7.t2  :=  r.reg6.e1q    r.reg6.kp;
v.reg7.t3  :=  r.reg6.e2p    r.reg6.kq;
v.reg7.t4  :=  r.reg6.e2q    r.reg6.kp;
v.reg7.en :=  r.reg6.en;

−− reg 8
v.reg8.t   :=  r.reg7.t;
v.reg8.u   :=  r.reg7.t1  − r.reg7.t2;
v.reg8.v   :=  r.reg7.t4  − r.reg7.t3;
v.reg8.en :=  r.reg7.en;

−− output
q.rdy  <= r.reg8.en;
q.t  <= r.reg8.t(n+2 downto 1);
q.u  <= r.reg8.u(m−1 downto 0);
q.v  <= r.reg8.v(m−1 downto 0);
sum := r.reg8.u + r.reg8.v;
if  sum(m+n+1 downto edge_shift+m−1) /= LOSER or
   r.reg8.u(m+n+1) = '1'  or  r.reg8.v(m+n+1) = '1'  or
   r.reg8.t(n+2) =  '1'   then
   q.hit  <=  '0';
else
   q.hit  <=  '1';
```

```
  end if;

  rin <= v;
end process;

clocked : process(clk)
begin
  if rising_edge(clk) then
    r <= rin;
  end if;
end process;

end architecture;
```

## A.3 Bounding Interval Hierarchy

Listing A.3: Bounding Interval Hierarchy

```
-- this is part of intrt, a fixed point hardware raytracer
-- Copyright (C) 2007 Johannes Hanika
--
-- This program is free software: you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation, either version 3 of the License, or
-- (at your option) any later version.
--
-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
-- GNU General Public License for more details.
--
-- You should have received a copy of the GNU General Public License
-- along with this program. If not, see <http://www.gnu.org/licenses/>.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.intrt_p.all;

package bih_p is
  type bih_in_t is record
    -- ray pos + precomputed inverse omega
    pos_x : std_logic_vector(n-1 downto 0);
    pos_y : std_logic_vector(n-1 downto 0);
    pos_z : std_logic_vector(n-1 downto 0);
    -- rcp ray dir omega
    idir_x  : std_logic_vector(m-1+C downto 0);
    idir_y  : std_logic_vector(m-1+C downto 0);
    idir_z  : std_logic_vector(m-1+C downto 0);
    -- sync reset
    rst     : std_logic;
  end record;

  type bih_out_t is record
    -- done (next ray please)
    done     : std_logic;
    -- triangle number
    tri_num  : std_logic_vector(28 downto 0);
    -- enable (tri number)
    tri_en   : std_logic;
  end record;

  component bih
  port
  (
    clk  : in  std_logic;
    d    : in  bih_in_t;
    q    : out bih_out_t
  );
  end component;
end package;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use work. tri_int_p . all ;
use work.bih_p.all ;
use work. intrt_p . all ;
use work.bih_ram_p.all;

entity bih is
  port
  (
    clk  : in   std_logic ;
    d    : in   bih_in_t ;
    q    : out bih_out_t
  );
end entity;

architecture rtl of bih is
  -- fill in scene's aabb here (lower bounds are 0).
  constant aabb_x : std_logic_vector (n−1 downto 0) := (others => '1');
  constant aabb_y : std_logic_vector (n−1 downto 0) := "01111101011100101101111001001 1";
  constant aabb_z : std_logic_vector (n−1 downto 0) := "10011111010011001100011111111 1";
  -- always works, but inefficient :
  -- constant aabb_y : std_logic_vector (n−1 downto 0) := (others => '1');
  -- constant aabb_z : std_logic_vector (n−1 downto 0) := (others => '1');

  type state_t is (aabb_s, push_s, pop_s, node_s, inner_s , leaf_s );
  -- attribute ENUM_ENCODING: STRING;
  -- attribute ENUM_ENCODING of STATE_TYPE:type is "001 010 011 100 101 110 111";
  type reg_t is record
    state          : state_t ;
    -- hit distance so far :
    hit_dist       : std_logic_vector (n downto 0);
    -- tri num state for leaf_s
    tri_num        : std_logic_vector (31 downto 0);
    -- stack pointer ( < 32)
    stack_pos      : std_logic_vector (4 downto 0);
    tri_en         : std_logic ;
    -- registers for tmin/tmax, as ram addr is changed.
    tmin           : std_logic_vector (62 downto 0);
    tmax           : std_logic_vector (62 downto 0);
    dist0          : std_logic_vector (62 downto 0);
    dist1          : std_logic_vector (62 downto 0);
    saved_node : std_logic_vector (9 downto 0);
  end record;

  signal r , rin : reg_t := (aabb_s, (others => '0'), (others => '0'), (others => '0'), '0', (others => '0'),
       (others => '0'), (others => '0'), (others => '0'), (others => '0'));

  -- current node
  signal node_data  : std_logic_vector (31 downto 0);
  signal node_clip0 : std_logic_vector (31 downto 0);
  signal node_clip1 : std_logic_vector (31 downto 0);

  signal data0     : std_logic_vector (35 downto 0);
  signal data1     : std_logic_vector (35 downto 0);
  signal data2     : std_logic_vector (35 downto 0);
  signal data3     : std_logic_vector (35 downto 0);
  signal data0_in  : std_logic_vector (35 downto 0);
  signal data1_in  : std_logic_vector (35 downto 0);
  signal data2_in  : std_logic_vector (35 downto 0);
  signal data3_in  : std_logic_vector (35 downto 0);
  signal addr3x    : std_logic_vector (9 downto 0) := (others => '0');
  signal addr1x    : std_logic_vector (9 downto 0) := (others => '0');
  signal we        : std_logic := '0';

begin
  -- implemented with correct memory dump, stack: addr = '11111'&stack_pos,
  -- 4x1Kx36 bits block ram, two addresses : 3x node, 1x tri #, together : 4x stack
  -- async read , address is clocked .
  ram0 : raminfr_bih0 port map(clk, we, addr3x, data0_in , data0 );
  ram1 : raminfr_bih1 port map(clk, we, addr3x, data1_in , data1 );
  ram2 : raminfr_bih2 port map(clk, we, addr3x, data2_in , data2 );
  ram3 : raminfr_bih3 port map(clk, we, addr1x, data3_in , data3 );

  -- directly connect node_data, clip0 , clip1 , q.tri_num to ram output pins as tmp signal .
  node_data  <= data0(31 downto 0);
  node_clip0 <= data1(31 downto 0);
  node_clip1 <= data2(31 downto 0);
  q.tri_num  <= data3(28 downto 0);
  q. tri_en     <= r . tri_en ;

comb : process(r , d, data0, data1 , data2, data3 , node_data, node_clip0, node_clip1, addr1x)
variable v : reg_t ;
variable aabb_dist_x : std_logic_vector (n downto 0);
variable aabb_dist_y : std_logic_vector (n downto 0);
variable aabb_dist_z : std_logic_vector (n downto 0);
```

```
-- tmp values for aabb idir : (n +1)( m+C) = 31+32+16 bits = 79
variable tx : std_logic_vector (78 downto 0);
variable ty : std_logic_vector (78 downto 0);
variable tz : std_logic_vector (78 downto 0);
-- variable c : std_logic_vector (1 downto 0);
begin
  v := r;
  q.done <= '0';
  -- step state machine.
  case r.state is
  when aabb_s =>
    we <= '0';
    -- TODO: add more states for faster clock?
    -- calculate tmin, tmax
    v.tmin := (others => '0');
    -- calculate t0, t1, t2 in parallel and let tmax be min(t0, t1, t2)
    if d.idir_x (47) = '1' then
      aabb_dist_x := '1'&( not(d.pos_x)+1);
    else
      aabb_dist_x := '0'&( aabb_x - d.pos_x);
    end if;
    if d.idir_y (47) = '1' then
      aabb_dist_y := '1'&( not(d.pos_y)+1);
    else
      aabb_dist_y := '0'&( aabb_y - d.pos_y);
    end if;
    if d.idir_z (47) = '1' then
      aabb_dist_z := '1'&( not(d.pos_z)+1);
    else
      aabb_dist_z := '0'&( aabb_z - d.pos_z);
    end if;
    tx := aabb_dist_x d.idir_x;
    ty := aabb_dist_y d.idir_y;
    tz := aabb_dist_z d.idir_z;
    if tx(78 downto C) < ty(78 downto C) then
      if tx(78 downto C) < tz(78 downto C) then
        v.tmax := tx(78 downto C);
      else
        v.tmax := tz(78 downto C);
      end if;
    elsif ty(78 downto C) < tz(78 downto C) then
      v.tmax := ty(78 downto C);
    else
      v.tmax := tz(78 downto C);
    end if;

    -- tmax < 0 ? => miss aabb.
    if v.tmax(62) = '1' then
      v.state := aabb_s;
      q.done <= '1';
    else
      -- init stack as root node (0)
      v.stack_pos := (others => '0');
      v.state := node_s;
      addr3x <= (others => '0');
      addr1x <= (others => '0');
      v.hit_dist := (others => '1');
    end if;

  when pop_s =>
    we <= '0';
    -- load block ram tmin/tmax to registers
    -- and node_pos to ram address
    v.tmin := data1(26 downto 0) & data0(35 downto 0);
    v.tmax := data3(21 downto 0) & data2(35 downto 0) & data1(35 downto 31);
    addr3x <= data3(35 downto 26);
    -- if hit_dist < tmin, goto pop_s
    v.state := node_s;

  when push_s =>
    we <= '0';
    -- restore address for near node.
    addr3x <= r.saved_node;
    v.state := node_s;

  when node_s =>
    we <= '0';
    -- set state according to axis
    case node_data(1 downto 0) is
    when "11" =>
      v.state := leaf_s;
      -- load tri index,
      addr1x <= node_data(12 downto 3);
      v.saved_node := node_data(12 downto 3) + 1;
      -- set tri num for counter in reg
      v.tri_num := node_clip0;
      v.tri_en := '1';
    when "00" =>
      -- start to do some calculations ( dist0 , dist1 )
      v.state := inner_s;
      tx := (node_clip1(n downto 0) - ('0'&d.pos_x )) d.idir_x ;   -- (uns - uns ) sgn
      ty := (node_clip0(n downto 0) - ('0'&d.pos_x )) d.idir_x ;
      if d.idir_x (47) = '1' then
        v.dist0 := tx(78 downto C);
      else
        v.dist0 := ty(78 downto C);
      end if;
      if d.idir_x (47) = '1' then
        v.dist1 := ty(78 downto C);
      else
        v.dist1 := tx(78 downto C);
      end if;
    when "01" =>
      v.state := inner_s;
      tx := (node_clip1(n downto 0) - ('0'&d.pos_y )) d.idir_y ;   -- (uns - uns ) sgn
      ty := (node_clip0(n downto 0) - ('0'&d.pos_y )) d.idir_y ;
      if d.idir_y (47) = '1' then
        v.dist0 := tx(78 downto C);
      else
        v.dist0 := ty(78 downto C);
      end if;
      if d.idir_y (47) = '1' then
        v.dist1 := ty(78 downto C);
      else
        v.dist1 := tx(78 downto C);
      end if;
    when others => --"10" =>
      v.state := inner_s;
      tx := (node_clip1(n downto 0) - ('0'&d.pos_z )) d.idir_z ;   -- (uns - uns ) sgn
      ty := (node_clip0(n downto 0) - ('0'&d.pos_z )) d.idir_z ;
      if d.idir_z (47) = '1' then
        v.dist0 := tx(78 downto C);
      else
        v.dist0 := ty(78 downto C);
      end if;
      if d.idir_z (47) = '1' then
        v.dist1 := ty(78 downto C);
      else
        v.dist1 := tx(78 downto C);
      end if;
    end case;

  when inner_s =>
    -- axis is 00 01 10
    if r.dist0 < r.tmin and r.dist1 > r.tmax then
      we <= '0';
      -- pop stack
      if r.stack_pos = 0 then
        v.state := aabb_s;
        q.done <= '1';
      else
        v.stack_pos := r.stack_pos - 1;
        addr3x <= "11111" & v.stack_pos;
        addr1x <= "11111" & v.stack_pos;
        v.state := pop_s;
      end if;
    elsif r.dist0 >= r.tmin and r.dist1 > r.tmax then
      we <= '0';
      -- load node 0 (near)
      if r.tmax > r.dist0 then v.tmax := r.dist0; end if;
      if (d.idir_x (47) = '1' and node_data(1 downto 0) = "00")
                     or (d.idir_y (47) = '1' and node_data(1 downto 0) = "01")
                     or (d.idir_z (47) = '1' and node_data(1 downto 0) = "10") then
        addr3x <= node_data(12 downto 3) + 1;
      else addr3x <= node_data(12 downto 3); end if;
      v.state := node_s;
    elsif r.dist0 < r.tmin and r.dist1 <= r.tmax then
      we <= '0';
      -- load node 1 ( far )
      if r.tmin < r.dist1 then v.tmin := r.dist1 ; end if;
      if (d.idir_x (47) = '1' and node_data(1 downto 0) = "00")
                     or (d.idir_y (47) = '1' and node_data(1 downto 0) = "01")
                     or (d.idir_z (47) = '1' and node_data(1 downto 0) = "10") then
        addr3x <= node_data(12 downto 3);
      else addr3x <= node_data(12 downto 3) + 1; end if;
      v.state := node_s;
    -- if r.dist0 >= r.tmin and r.dist1 <= r.tmax then
    else
```

```
      -- push node 1,  will  be  stored  on next  rising  edge.
      we <= '1';
      -- overwrite  current  stack  position ,  stack_pos++
      addr3x <= "11111"&r.stack_pos;
      addr1x <= "11111"&r.stack_pos;
      v.stack_pos := r.stack_pos + 1;
      if (d. idir_x (47) = '1'  and node_data(1 downto 0) = "00")
                     or (d. idir_y (47) = '1'  and node_data(1 downto 0) = "01")
                     or (d. idir_z (47) = '1'  and node_data(1 downto 0) = "10") then
        data3_in(35 downto 26) <= node_data(12 downto 3);
      else data3_in(35 downto 26) <= node_data(12 downto 3) + 1; end if;

      data3_in(21 downto 0) <= r.tmax(62 downto 41);
      data2_in(35 downto 0) <= r.tmax(40 downto 5);
      data1_in(35 downto 31) <= r.tmax( 4 downto 0);
      if r.tmin > r.dist1 then
        data1_in(26 downto 0) <= r.tmin(62 downto 36);
        data0_in(35 downto 0) <= r.tmin(35 downto 0);
      else
        data1_in(26 downto 0) <= r.dist1 (62 downto 36);
        data0_in(35 downto 0) <= r.dist1 (35 downto 0);
      end if;

      -- load node 0  ( set  registers )
      -- save addr3x  for  push_s
      if (d. idir_x (47) = '1'  and node_data(1 downto 0) = "00")
                     or (d. idir_y (47) = '1'  and node_data(1 downto 0) = "01")
                     or (d. idir_z (47) = '1'  and node_data(1 downto 0) = "10") then
        v.saved_node := node_data(12 downto 3) + 1;
      else v.saved_node := node_data(12 downto 3); end if;
      if r.tmax > r.dist0  then v.tmax := r.dist0 ; end if;
      v.state := push_s;
    end if;

  when leaf_s =>
    we <= '0';
    -- TODO: detach this  counter  from  the  fsm  of  the  bih.
    -- axis  is  11,  first  triangle  index  has been loaded.
    -- count down tri  num (reg  signal )
    if v.tri_num = 1 then
      v.tri_en := '0';
      if r.stack_pos = 0 then
        -- stack  empty
        v.state := aabb_s;
        q.done <= '1';
      else
        -- pop stack
        v.stack_pos := r.stack_pos - 1;
        addr3x <= "11111" & v.stack_pos;
        addr1x <= "11111" & v.stack_pos;
        v.state := pop_s;
      end if;
    else
      -- loop  until  all  tris  dumped out
      v.tri_num := r.tri_num - 1;
      v.saved_node := r.saved_node + 1;
      addr1x <= r.saved_node;
      v.tri_en := '1';
    end if;
  end case;
  rin <= v;
end process;

clocked : process(clk, d)
begin
  if rising_edge(clk) then
    if d.rst = '1' then
      r <= (aabb_s, (others => '0'), (others => '0'), (others => '0'), '0', (others => '0'),
      (others => '0'), (others => '0'), (others => '0'), (others => '0'));
    else
      r <= rin;
    end if;
  end if;
end process;

end architecture;
```