Towards Hardware Ray Tracing using Fixed Point Arithmetic

Johannes Hanika* Ulm University

Alexander Keller† Ulm University

Figure 1: A car rendered using a ray tracing core completely based on fixed point arithmetic realized in integer arithmetic. For the ease of 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.

Abstract

For realistic image synthesis and many other simulation applications, ray tracing is the only choice to achieve the desired realism and accuracy. Ray tracing has become such an important algorithm that an implementation in hardware is justified and desirable. The first ray tracing hardware realizations have been using floating point and logarithmic arithmetic. While floating point arithmetic requires considerably more logic than integer arithmetic, an implementation in logarithmic arithmetic is much simpler, but still suffers from similar problems. In analogy to rasterization hardware we therefore investigate the use of fixed point arithmetic for ray tracing. Our software implementation and comparisons provide strong evidence that an implementation of ray tracing in hardware using fixed point arithmetic is an efficient and robust choice.

Index Terms: I.3.1 [Computer Graphics]: Hardware Architecture—Fixed Point Arithmetic

*e-mail: Johannes.Hanika@uni-ulm.de
†e-mail: Alexander.Keller@uni-ulm.de

IEEE/EG Symposium on Interactive Ray Tracing 2007
10 - 12 September, Ulm, Germany
978-1-4244-1629-5/07/$25.00 ©2007 IEEE

1 Introduction

Rasterization cannot efficiently render secondary effects like reflection and refraction. Even precise shadows come at a considerable cost and ray tracing (see Figure 1) as the natural solution of these problems becomes competitive.

Designing ray tracing in hardware requires selecting an arithmetic. The common choice, i.e. the IEEE floating point standard, requires considerable chip area. Simpler and faster arithmetic units can improve the situation very much.

Implementing the DDA (digital differential analyzer) algorithm used in polygon rasterization in fixed point arithmetic was a basic step towards the success of graphics hardware. In analogy to the DDA algorithm we analyze the applicability of fixed point arithmetic for hardware ray tracing. While such an implementation is much simpler to design in hardware, it requires a profound investigation of numerical ranges and quantization effects as compared to a classic floating point implementation. In the following we will describe our analysis and exemplary C99 implementation that result in the conclusion that ray tracing in fixed point arithmetic will be as useful as the DDA algorithm.

2 Arithmetic used in Hardware Ray Tracing

Integer and floating point units are the two standard kinds of arithmetic available on mainstream general purpose proces-

sors. Other kinds of arithmetics [11], like for example fixed point or logarithmic arithmetic, are less widely applied, because they are less general.

For hardware ray tracing all these kinds of arithmetic have been applied already: The first ray tracing hardware [22] used logarithmic arithmetic, with the advent of general purpose computing on graphics accelerator boards, fixed point arithmetic was applied, and with upcoming affordable reconfigurable hardware even floating point units were used [15, 21, 20, 10].

2.1 Floating Point Arithmetic

The most recent approaches to ray tracing hardware [15, 21, 20] used floating point units (without considering denormalized values) realized on FPGAs (Field Programmable Gate Arrays). While this work focussed on the proof of concept, it did not consider ray tracing problems that arise from the unequal distribution of floating point numbers along the real axis (see Figure 2a).

In fact there are many well-known precision issues to address when using floating point arithmetic [7]. In ray tracing the main problems are the quantization of numbers that are far from the origin (see Figure 3) and the self-intersection problem [19].

2.2 Logarithmic Arithmetic

Instead of mantissa and exponent, logarithmic arithmetic only stores the sign and logarithm of the number to be represented. While the spacing of the numbers remains similar to the floating point representation [2], the implementation becomes simpler.

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. [22, Cols. 13 and 14]) prooved to be efficient in space and performance as it can be realized using almost only 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.

2.3 Fixed Point Arithmetic

The main advantage of fixed point arithmetic over floating point and logarithmic arithmetic is the equidistant spacing of the numbers. However, due to a lack of an exponent in the representation, the range is very limited and an application of fixed point numbers requires a profound investigation of the ranges of all computations in an application.

The first implementations of ray tracing on graphics hardware [4, 14, 13] were using 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 envisioned hardware implementation is considerably more compact and simpler than a realization using floating point or logarithmic arithmetic.

3 Intersection Computation in Fixed Point Arithmetic

Without loss of generality we focus on triangle based ray tracing. Out of the many different ways to intersect a ray and a triangle [1, 12, 8, 16, 5, 6] we select the test of Badouel [1] based on barycentric coordinates, because it allows for increasing efficiency [18, 3, 9] by precomputation, which in addition simplifies the actual intersection procedure. For dynamic scenes the other triangle tests must be investigated, too, which, however, is out of the focus of this current paper.

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 $\mathtt{r}$ 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 therefore 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 $\mathtt{r}$ is guaranteed to be equal to one.

In order to intersect a ray (O, ω) with origin O and direction ω with a triangle according to the accelerated test from [18], the distance t along the ray from its origin to the plane of the triangle is computed as

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

Then the hitpoint of the ray and the plane projected along the component $\mathtt{r}$ is computed by

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

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

3.1 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, these 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 $\mathcal{V} := \{x_i = (x_{i,0}, x_{i,1}, x_{i,2}) : 0 \leq i < N\} \subset \mathbb{R}^3$ of N triangle vertices, we can define their axis-aligned bounding box components as

$$b_j := \min\{x_{i,j} : x_i \in \mathcal{V}\} \qquad \text{and} \qquad B_j := \max\{x_{i,j} : x_i \in \mathcal{V}\}.$$

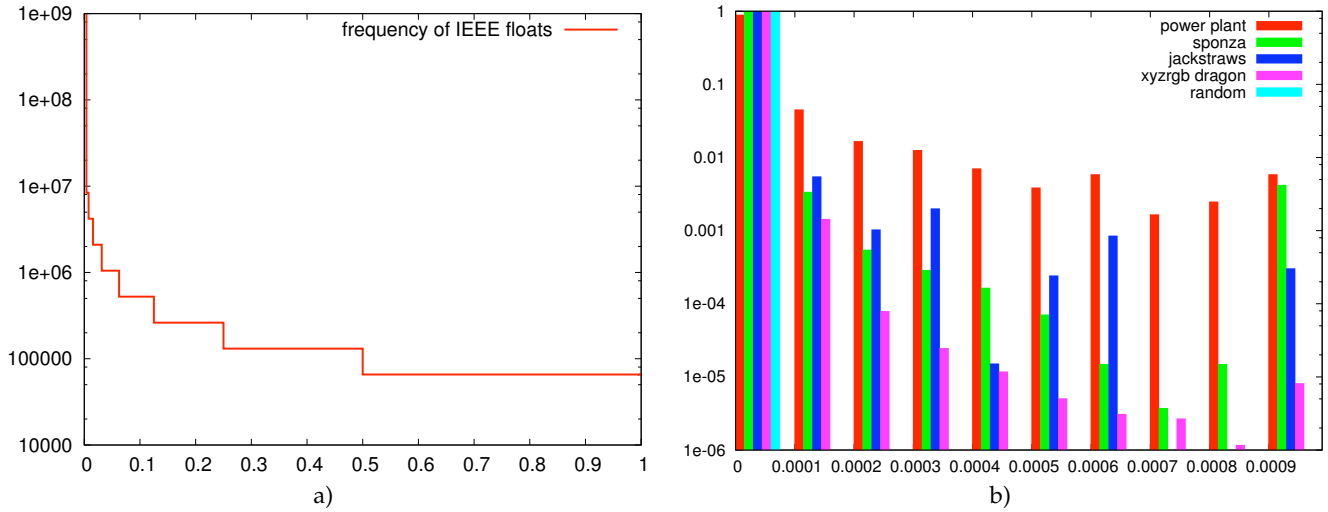


Figure 2: On a logarithmic scale: a) Frequency of the IEEE floating point numbers in $[0,1]$ and b) the relative frequency of the triangle edge components `e1p`, `e1q`, `e2p`, `e2q` for various scenes after transforming them to the integer bounding box according to Section 3.2. Most of the values are 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, however, this is not true in general as can be seen from the random triangles and the remaining scenes. Also, 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.

With the fact that for any vector $n \in \mathbb{R}^3$

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

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

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

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

$$|n_\mathtt{r}| \geq \varepsilon > 0$$

and hence

$$|\mathtt{e}_{ik}| \leq \frac{1}{\varepsilon} \max_{j\in\{0,1,2\}} (B_j - b_j). \tag{3}$$

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.

3.2 Quantization and Precision

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

Points: The scene geometry $\mathcal{V}$ is moved to the positive octant and scaled such that $b_j = 0$ for $j = 0,1,2$ and $\max_{j=0,1,2}\{B_j\} = 2^\mathtt{n} - 1$. The values `pp` and `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 $\mathtt{m}$ 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.

3.2.1 Clamping of Triangle Edge Components

In order to improve the bound on the edge data, we investigated their statistics. Figure 2b 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 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 $(-2^{-\mathtt{E}}, 2^{-\mathtt{E}})$ before mapping them to the fixed point range, where $\mathtt{E}$ is called the *edge shift*. Choosing $2^{-\mathtt{E}} \approx 0.001$, i.e. $\mathtt{E} = 10$, 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 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 is only one forth of the original area. Since the

vertices are quantized, triangles are likely to be degenerate in addition.

3.2.2 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 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;
```

3.3 Fixed Point Ray-Plane Intersection

First, the distance t has to be computed according to Equation 1, which involves a division. In finite arithmetic the mathematical equivalence

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

changes the result depending on the parameter T: The nominator *nom* 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 8, all other images have been rendered using T = 0.

3.3.1 Ray-Axis-Aligned Plane Intersection

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 ≤ 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.

3.3.2 Ray-Triangle Plane Intersection

Compared to the previous piece of code, the ray-triangle plane intersection requires to first project the ray direction onto the triangle normal. 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
}
```

3.4 Fixed Point Ray-Triangle Intersection

Whether or not the triangle is intersected by the ray is decided according to the set of Equations 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
```

3.5 Secondary Rays and the Self-Intersection Problem

A common issue with secondary rays is the so called *self-intersection* problem [19]. 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 [6].

Since in fixed point arithmetic the geometry lies on an equidistant raster with known resolution, the self-intersection can be approached in a simpler way: The distance t measured along the ray direction ω 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 ($= 2^{-\mathtt{n}}$ times the largest side of the bounding box in our implementation). The computation of the hitpoint $h = O + t \cdot \omega$ uses one more fractional bit and is rounded to the integer grid, which can cause an additional error of 0.5 in each component. 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] = hit[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 (as used in Figure 1), 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.

3.6 Acceleration

As we are heading for the simplest possible ray tracing hardware, numerically involved triangle-plane intersections as required for building kd-trees should be avoided. Hence we use a bounding interval hierarchy [17, 20] for an 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 the kd-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 [17].

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 3.3.1, 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:

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

so `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 a sufficient accuracy, ω_i^{-1} has to be stored in $\mathtt{m}+\mathtt{C}$ bits, requiring more bits for multiplication, too. The additional bits also ameliorate the fact that due to its hyperbolic nature the range of ω_i^{-1} is not well represented by equidistant quantization.

In the actual implementation, one has to take care that there will be no overflow, which in turn affects precision. The images in Figure 9 show the effect of the parameter `C` and the resulting quite visible inconsistencies for too small `C`. The computations have been executed 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];
```

4 Numerical Evidence

In Figure 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 $\mathtt{m} > 32$.

4.1 Constructing a Stress Test

Triangles with long edges and small area are numerically difficult to ray trace. 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. The triangle then was further extended until if formed a thin line at an image resolution of 640×480 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 and we therefore omitted images of this experiment.

4.2 Clamped Edge Components

Due to clamping (see Section 3.2.1) 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 [6] 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 4 are rather abstract and only included for the sake of completeness.

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 (see Figure 5). Most of the differences in Figure 6 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.

5 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.

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, pipelined implementation of the triangle test using a specialized divider unit, as well as the BIH traversal running in a simulator and will continue implementing it on a Virtex-5 FPGA.

A similar analysis can be applied to other triangle tests [12, 8] and we expect large performance benefits from applying fixed point arithmetic to the high-precision ray tracing algorithms for freeform surfaces [6].

Acknowledgements

The first author has been funded by the project *information at your fingertips - Interaktive Visualisierung für Gigapixel Displays*, Forschungsverbund im Rahmen des Förderprogramms Informationstechnik in Baden-Württemberg (BW-FIT). Our special thanks go to Sun Microsystems GmbH who kindly supported our research with computing equipment. We also want to thank the anonymous reviewers for well-meaning and helpful comments.

References

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

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

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

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

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

[6] 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, Sept. 2006.

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

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

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

[10] S.-S. Kim, S.-W. Nam, D.-H. Kim, and I.-H. Lee. Hardware-accelerated ray-triangle intersection testing for high-performance collision detection. In *Proc. WSCG*, 2007.

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

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

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

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

[15] 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.

[16] R. Segura and F. Feito. Algorithms to test ray-triangle intersection. In *Proc. WSCG*, 2001.

[17] 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.

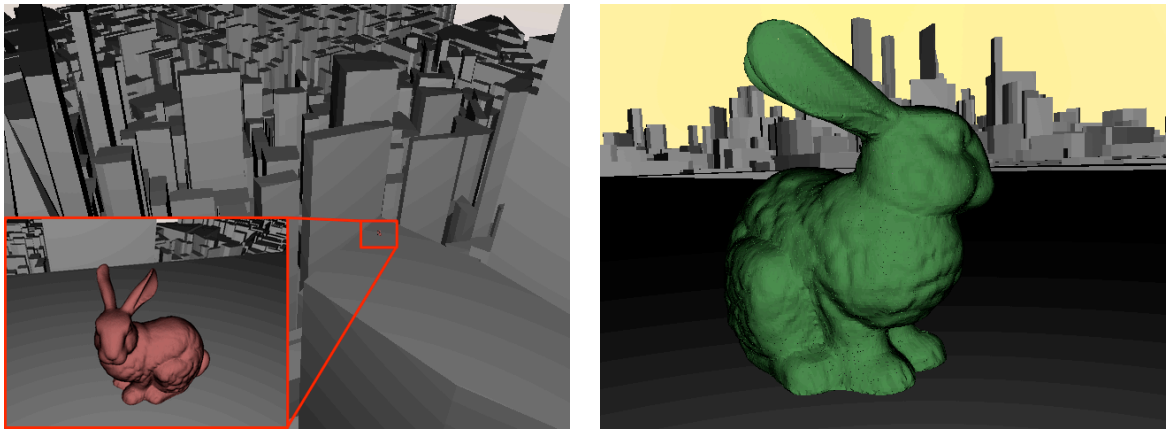
[18] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

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

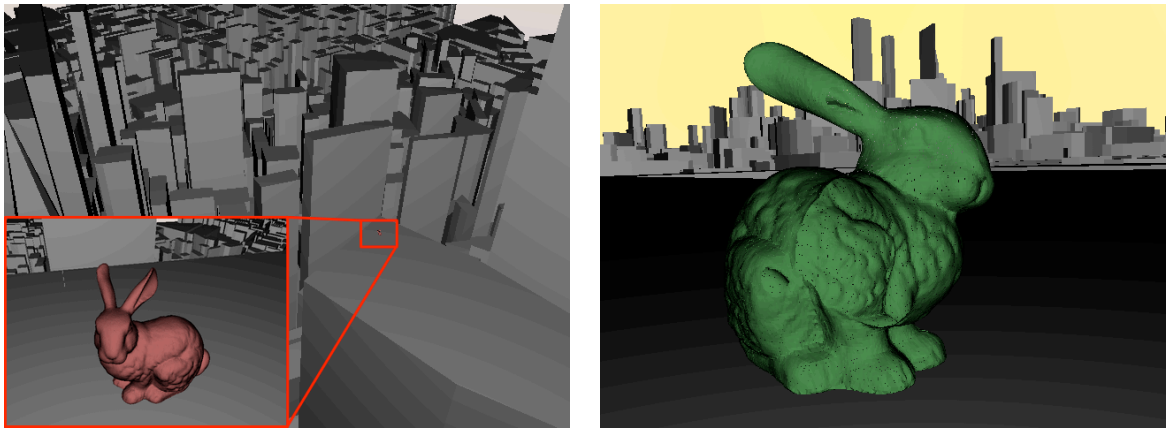
[20] S. Woop, G. Marmitt, and P. Slusallek. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware (Proc. Eurographics/SIGGRAPH 2006)*, pages 67–77, 2006.

[21] 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.

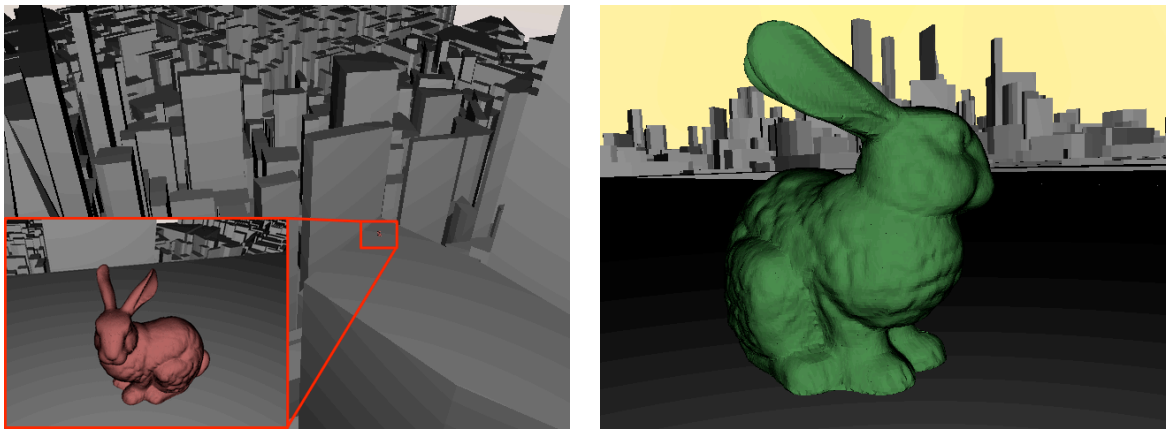
[22] A. Wrigley. Method of and apparatus for constructing an image of a notional scene by a process of ray tracing, May 1997. United States Patent US 5,933,146, 28.05.1997.



Rendered using IEEE 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 $[-10^5, 10^5] \times [-3 \cdot 10^4, 2 \cdot 10^3] \times [-10^5, 10^5]$.



Before rendering the scene has been scaled to fit inside $[-1.0, 1.0]^3$, where the floating point frequency 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.



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.

Figure 3: 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 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.

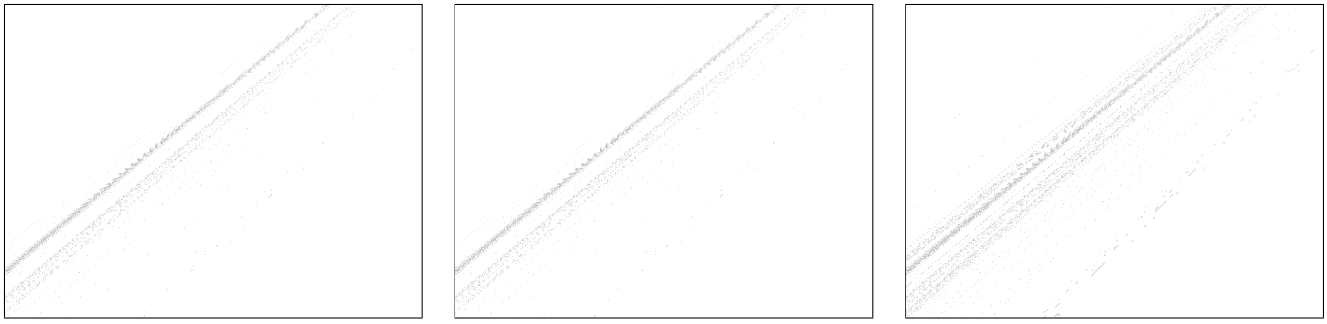


Figure 4: Triangles with components `e1p`, `e1q`, `e2p` or `e2q` outside the the representable fixed point range $(-2^{-E}, 2^{-E})$ extracted from the power plant model. To leave the range, a triangle needs to be very long and thin, and hence has almost zero area. Both kinds of investigated arithmetics (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 [6] on the left) shows mostly aliasing.

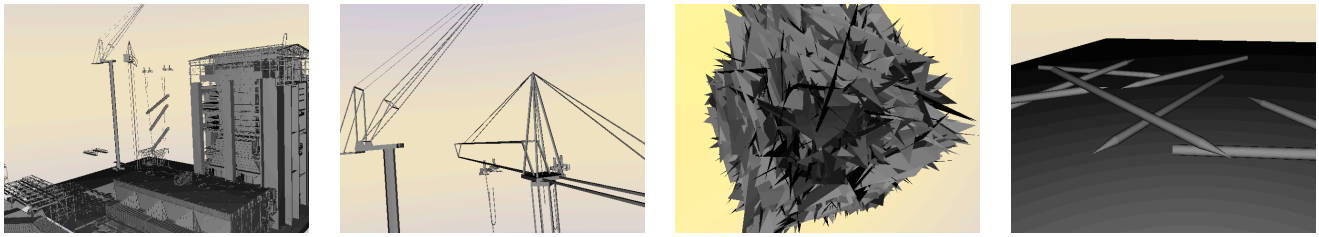


Figure 5: Some test cases where fixed point and floating point arithmetic almost cannot be distinguished. Master images were computed using a robust floating point arithmetic reference implementation [6].

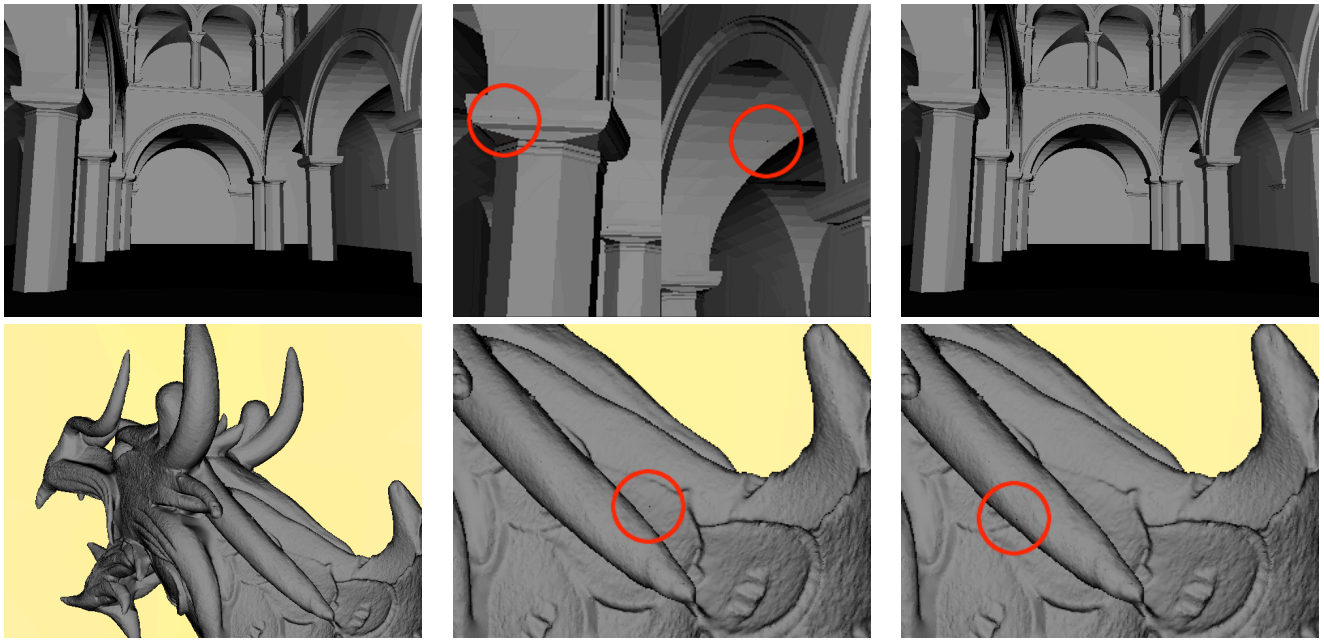


Figure 6: Comparison of screenshots made using the robust floating point arithmetic intersection [6] (left), floating point arithmetic (middle), and fixed point arithmetic (right). The red circles in the zoomed images mark the most apparent intersection errors. Concerning precision, floating point and fixed point arithmetic perform equivalently.

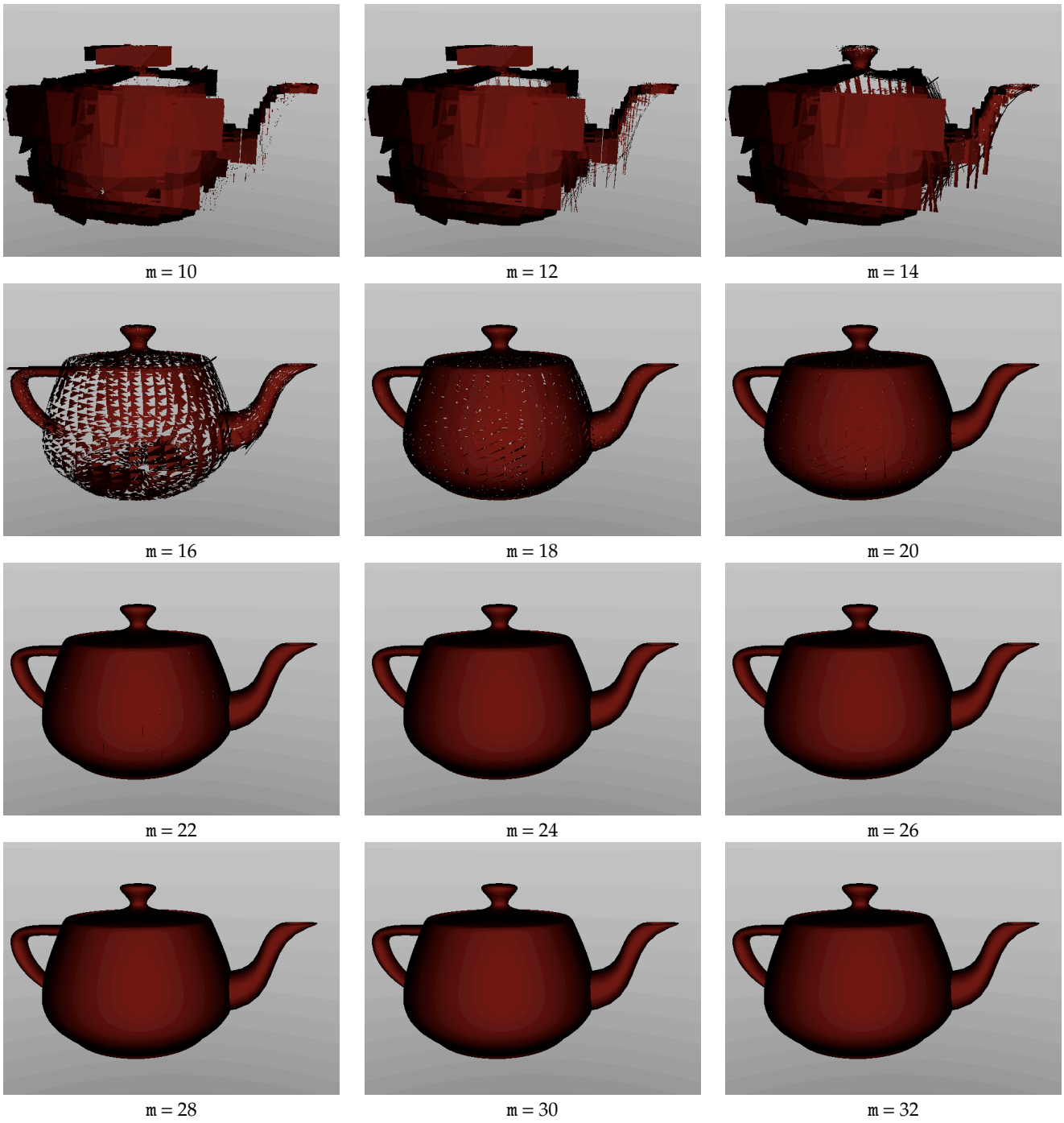
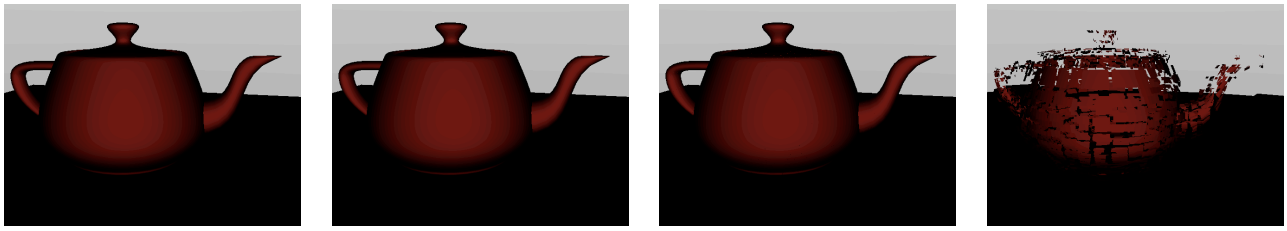
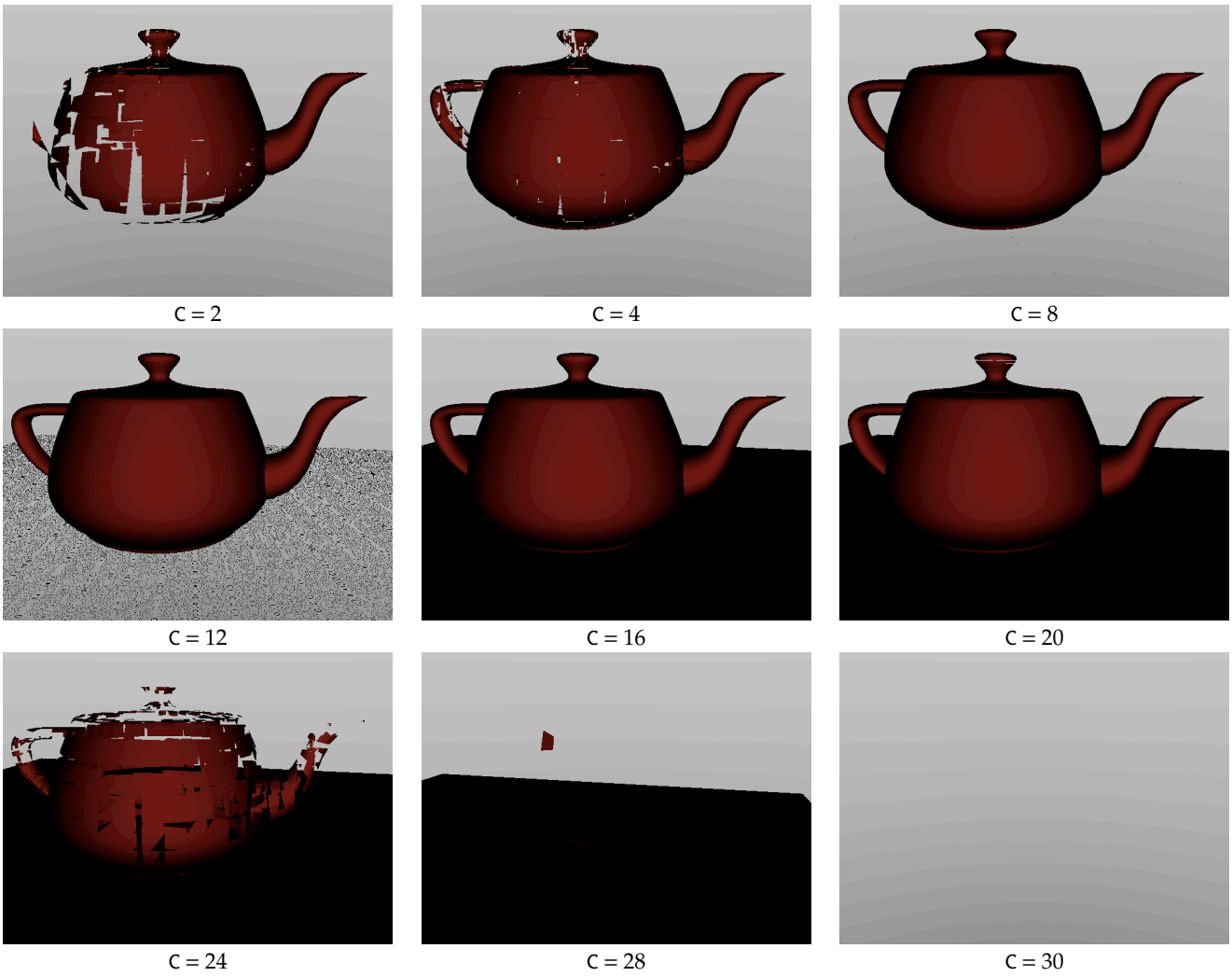


Figure 7: The Utah Teapot tesselated into 6320 triangles and rendered using different bit widths $\mathtt{m}$ for the fixed point numbers. This affects the precision of the components of the triangle edges, normals, and directions. A moderate precision already allows for artifact free renderings.



T = 0 T = 8 T = 16 T = 24

Figure 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 nominator 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.



C = 2 C = 4 C = 8

C = 12 C = 16 C = 20

C = 24 C = 28 C = 30

Figure 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 $C = 12$.