An Efficient FPGA-based Axis-Aligned Box Tool for Embedded Computer Graphics

Georgios Chatzianastasiou and George A. Constantinides
Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
{georgios.chatzianastasiou16, g.constantinides}@imperial.ac.uk

***Abstract*—One of the most heavily used kernels of many ray tracing algorithms is the intersection test for a ray with an Axis-Aligned Bounding Box (AABB). Floating point imprecision leads to incorrect ray/AABB intersection test results, which can lead not only to a substantial error in the photorealism of the image during rendering, by producing visually objectionable holes (*false misses*), but also to significant penalties to the ray tracer's performance and the power consumed, since the traversal is unnecessary (*false hits*). This work suggests a novel architecture that uses carefully-designed *directed* rounding and *intervals* for eliminating false misses and for investigating the trade-offs between false hit error rate, area and throughput when downscaling from high precision to low precision. The flexibility of FPGAs in terms of computational structure, pipelining and parallelism in conjunction with the massively parallel floating point operations in ray/AABB tests, makes them a very efficient choice for custom precision hardware computation. A fully-pipelined high-throughput architecture designed in RTL is demonstrated, featuring the *provable* elimination of false misses while quantifying false hits.**

***Index Terms*—Ray tracing, ray/AABB intersection, computer graphics, massively parallel computation, custom precision, reconfigurable hardware.**

I. Introduction

This work introduces a novel method, based on one of the most popular ray/AABB intersection algorithms (the Slabs Method), which achieves the *provable* elimination of false misses for *any* precision used in internal floating point computation. Carefully-selected directed IEEE-754 rounding modes have been used to help us achieve runtime interval arithmetic computation and apply tight bounds in the floating point calculations, which lead to false miss elimination even for the lowest representation of 1 bit mantissa. Furthermore, after demonstrating the new algorithm, the paper describes the manual adjustment of the FloPoCo VHDL operators for performing directed rounding to $\pm\infty$ that was explicitly required for avoiding false misses. Finally, we highlight the flexibility of FPGAs in terms of parallelism and present the hardware architecture for the proposed method. The multiple arbitrary precision floating point designs explore the trade-offs that occur between false hit error rate against area and throughput as the precision is downscaled. The corresponding results are presented in Section VI.

II. Background

A. Ray Tracing

Ray tracing is a powerful technique for rendering high quality images [1]. Tracing the paths of light rays through 3D space results in photorealistic image generation. The process of ray tracing involves an imaginary eye or a virtual camera inside the scene, from which various light segments (rays) are generated to create the image. A plane, perpendicular to the viewing direction, representing what the camera sees, is called the image plane. The image plane is broken up into a grid of pixels that will make up the image. The ray tracer computes the closest intersection point between the scene objects and each ray, and then determines the colour of the intersected object and paints the corresponding pixel with that colour. The whole process is done for all the pixels in the image.

A ray is nothing but a stream of photons pointed in a direction. Since the objects that get hit by the rays have material properties, any of the following optical effects could happen with this light segment: reflection, refraction, absorption and fluorescence. Ray tracing is very powerful because it can simulate all those optical effects. More specifically, if a ray intersects with any kind of 3D object, then shadow, transmitted and reflected rays will be traced. If any of this new set of rays intersects with another object in the scene, then additional new rays will be traced. Therefore, this leads to a recursive process which ends with the rendering of a high quality image.

The ray tracing algorithm in its uncomplicated configuration will inefficiently check for intersection against each object in the scene. This involves millions of intersection tests, especially when the scene includes a significant number of objects. Thus, a more effective way to reduce this overhead has to be applied. Rubin and Whitted proposed the idea of enclosing scene subsets within bounding volumes to decrease the number of ray-object intersection tests [2]. More specifically, if a scene subset is enclosed by a bounding volume, the generated rays will be tested against the bounding volume. Only in the case of intersection will each of the enclosed objects be tested by the ray tracer. The intersection test can be recursively applied in a tree structure until a leaf node, containing a number of objects to be rendered, is encountered. The tree forms a Bounding Volume Hierarchy (BVH). This technique heavily reduces the ray-object tests since a very small percentage of the generated rays hits the bounding volume (approximately 4-5%).

Ray-BVH traversal is heavily encountered in the ray tracing's process because the ray-bounding volume tests constitute the most reused part of ray tracing algorithms. As a result, the need for an efficient selection of a bounding volume has arisen. Since bounding volumes have to be simple in order to be efficiently tested for intersection, a very effective choice for a bounding volume is the *Axis Aligned Bounding Box* or

AABB. Axis Aligned Bounding Boxes are selected from the majority of the ray tracers since they are not only easy to be described in 3D planes by using only six coordinate values, but also very quick to traverse [3].

In particular, AABBs are chosen because they can be defined and tested for intersection just by storing their maximum and minimum extents. Therefore, only two sets of coordinates must be stored, resulting in the following six coordinate values: minimum point: (*x1*, *y1*, *z1*), maximum point: (*x2*, *y2*, *z2*). Figure 1 illustrates the AABB structure and specifies its minimum and maximum set of coordinates.

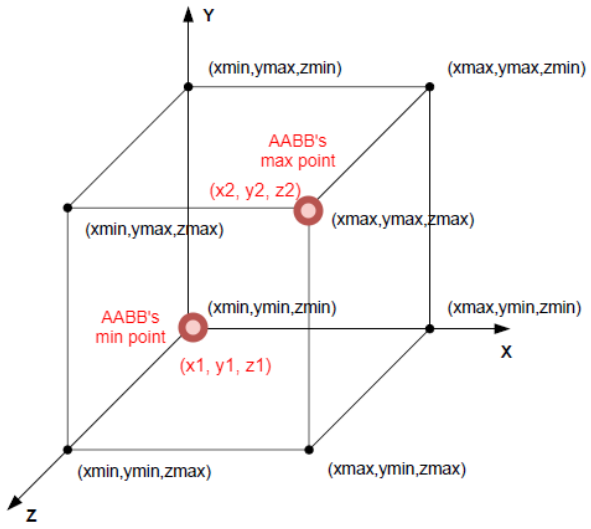

Fig. 1: The Axis-Aligned Bounding Box (AABB) [4], with red: the minimum and the maximum point of the box.

It is also necessary to define a data structure for the ray. A ray is simply defined by its origin coordinates, (*x0*, *y0*, *z0*), and a direction vector (*xD*, *yD*, *zD*).

B. Ray/AABB Intersection

The most popular ray/AABB intersection test algorithm is called the Slabs Method and was developed by Kay and Kajiya in 1986 [5]. The bounding box is defined by the intersection of a set of 'slabs', meaning the space between two parallel planes. One set of slabs exists for each dimension. The algorithm calculates three sets of intersection distances (*i.e.* an interval) along the ray. If there is an intersection for each pair of slabs by the ray and if all the intervals overlap, then the ray hits the box, otherwise it does not. The pseudo-code for the algorithm is given below. An example of a corresponding intersection hit and an intersection miss is illustrated in Figure 2.

Slabs Method pseudo-code. Only x-dimension described for simplicity.

```
if x_d = 0 then the ray is parallel to the
plane:
    if (x0 < x1)  ||  (x0 > x2) then the
    origin is not between the slabs:
      return false;
else: (ray not parallel to the plane)
    t_xmin = (x1 − x0) ÷ x_d
    t_xmax = (x2 − x0) ÷ x_d
    if  (t_xmin > t_xmax) swap them
    if  (t_xmin > t_min)   t_min = t_xmin
    if  (t_xmax < t_max)   t_max = t_xmax
    if  (t_max ⩾ t_min) && t_max > 0 do the same
    for the remaining y, z dimensions
if the ray survived all tests, then it
hits the AABB, return true;
otherwise, return false;
```

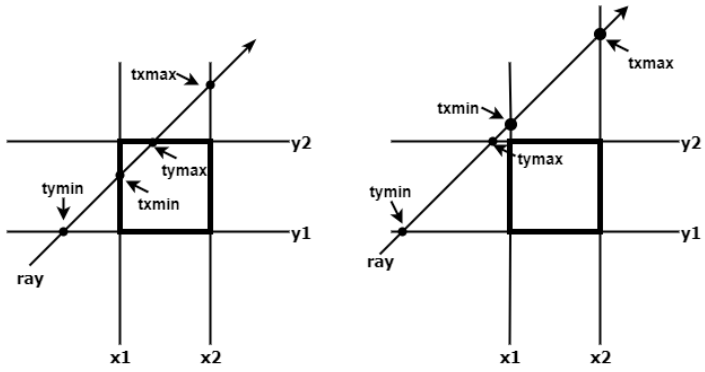

Fig. 2: Left: the ray hits the box, $[t_{xmin}, t_{xmax}]$ and $[t_{ymin}, t_{ymax}]$ overlap. Right: the ray misses the box, $[t_{xmin}, t_{xmax}]$ and $[t_{ymin}, t_{ymax}]$ do not overlap.

Instead of performing the division, which significantly decreases the performance we can multiply with the reciprocals of the ray direction vector elements. However, we have to be careful since it is possible for the division to generate $\pm\infty$ if the ray's reciprocal direction component is one of the floating point values ± 0. Since the IEEE-754 Floating Point Standard can assist in proper handling of infinities, then id_x, id_y, id_z could take infinity values and the Slabs Method correct operation will be maintained [6]. Additional care must be taken for ensuring that the algorithm efficiently handles positive and negative floating point zeros [7]. Finally, another important issue that can result in the algorithm's failure has to be confronted. More specifically, if the subtraction between the box's minimum or maximum point and the ray's origin results in a zero and concurrently the reciprocal direction component is $\pm\infty$ then the multiplication generates a NaN (Not a Number [8]). In other words, in that case the ray lies *exactly* on the edges of the Axis-Aligned Bounding Box. Also, a NaN could happen if instead of multiplying by the reciprocal

ray components, we divide by the direction component, then a $0 \div 0$ occurs. The solution is to make a case split when the ray is parallel to the corresponding plane or one or more of the direction components (x_d, y_d, z_d) is/are equal to 0. In the pseudo-code given, the first thing mentioned is this case split. A C-code implementation of the algorithm is given in Figure 3, it is worth mentioning that the case split for checking if the ray's direction components are parallel to the axes is not included. However in our implementation we took it into consideration. Each different operation is highlighted with a different color to signify the algorithm's separate phases.

```
//subtractions
//multiplications
//comparisons
bool intersection(box b, ray r) {
    double t1 = (b.min[0] - r.origin[0])*r.dir_inv[0];
    double t2 = (b.max[0] - r.origin[0])*r.dir_inv[0];

    double tmin = min(t1, t2);
    double tmax = max(t1, t2);

    for (int i = 1; i < 3; ++i) {
        t1 = (b.min[i] - r.origin[i])*r.dir_inv[i];
        t2 = (b.max[i] - r.origin[i])*r.dir_inv[i];

        tmin = max(tmin, min(min(t1, t2), tmax));
        tmax = min(tmax, max(max(t1, t2), tmin));
    }

    return tmax > max(tmin, 0.0);
}
```

Fig. 3: Simple C code implementation of the Slabs Method.

C. FPGA-based Arbitrary Precision Floating Point Numerical Computation

Reduced precision computation using reconfigurable devices has been studied for some time [9]. What we add is a novel dimension to the problem: how to reduce precision but maintain *provably correct* results for Boolean computations through avoiding either false negatives or false positives (but not both.)

Ray tracing and therefore ray/AABB tests that reside in the graphics pipeline involve numerical computation based on linear algebra. A large portion relies on massively-parallel floating point computation, now possible in FPGAs [10]. Ray tracing is an important application for low power devices, and yet is a computationally heavy task, therefore processors used for this task typically have specialized hardware. The flexibility of FPGAs in terms of computational structure, pipelining and parallelism exploitation makes them a very efficient choice for numerical hardware computation as well as for prototyping hardened accelerators.

In most cases, algorithms that use floating point arithmetic are complex in nature and are designed in single or even double precision. However, they are often worth investigation to identify whether they can maintain correct functionality while utilizing less precision. Reduced precision can not only lead to a cheaper implementation (less silicon area used), but also to a reduction of the processing time. The key point lies in the fact that in designing using FPGAs, the hardware designer can adjust the data representation to the needs of the algorithm or the application. Consequently, the trade-off between *precision* that leads to (in)accuracy and *performance* must be taken into account when designing an FPGA application-specific circuit [11]. As a result, the need of custom/arbitrary precision floating point libraries that provide fast operators has emerged.

FloPoCo [12] is a command line framework written in C++ that has been designed with this aim and proved to be a very efficient library that generates VHDL floating point cores compatible with the IEEE-754 Floating Point Standard principles. In FPGA-based designs, FloPoCo contributes to implementing circuits with just the right precision by scaling the mantissa bits and by highlighting the flexibility of the platform.

III. Problem Formulation and Algorithm Modification

The round-off error that is unavoidably introduced by floating point computation could lead to inaccuracy that impacts on the correct operation of many algorithms. One of these algorithms is ray tracing and the two types of error that arise are: *false misses* and *false hits*. An efficient way to ensure the correct functionality of the ray/AABB intersection test while concurrently downscaling precision is to compute *runtime* bounds on the numerical computation. These bounds represent each value as a range of possibilities, so using this idea to perform floating point operations yields a *set of values* summarised by an *interval*, and guarantees handling of rounding errors directly during computation.

Figure 4 illustrates the problem of the quantization error in 2D, the original ray (black color) hits the box near the edge, but the one computed with floating point operations (blue color) misses the box and produces a false miss. As can be observed for the original ray/AABB test, the computed intervals $[t_{xmin}, t_{xmax}]$ and $[t_{ymin}, t_{ymax}]$ are valid since they are non-empty and overlap the ray. On the other hand, the computed intervals for the blue ray are valid $[t'_{xmin}, t'_{xmax}]$, $[t'_{ymin}, t'_{ymax}]$ but they do not overlap the ray since $t'_{xmin} > t'_{ymax}$, leading to a false miss. In reality the differences in *x1*, *x1'*, *y1*, *y1'*, *x2*, *x2'*, *z2*, *z2'* and to the *t* distances values are very small in high precision, but as the precision is scaled down these differences become larger. The differences in these values occur due to: a) less precision used to represent the ray's origin and reciprocal direction coordinates/vector and the slabs coordinates that form the box, b) less accurate floating point subtractions and multiplications that the algorithm computes.

To solve the problem of false misses, we have introduced the following methodology:

1) Describe the AABB's minimum and maximum point, the ray's origin point and the ray's reciprocal direction vector with lower and upper values using intervals.
2) Split the computation of the intervals for t_{min} and t_{max} depending on the ray's origin in correspondence with the AABB's minimum and maximum point and for each floating point operation; perform appropriate rounding by using the correct *directed rounding* modes.

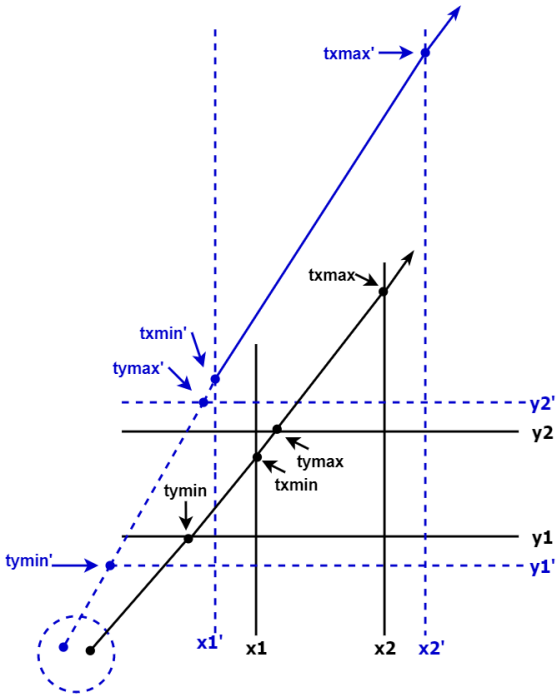

Fig. 4: In black: The original ray hits the box (infinite precision), $[t_{xmin}, t_{xmax}]$ and $[t_{ymin}, t_{ymax}]$ overlap the ray. In blue: the ray misses the box due to round-off error, $[t'_{xmin}, t'_{xmax}]$ and $[t'_{ymin}, t'_{ymax}]$ do not overlap the ray.

3) Bound $t_{xmin}, t_{ymin}, t_{zmin}$ and therefore the resulting final t_{min} with its *lower* value.
4) Bound $t_{xmax}, t_{ymax}, t_{zmax}$ and therefore the resulting final t_{max} with its *upper* value.

By calculating the lower and upper values for t_{xmin}, t_{ymin}, t_{zmin}, t_{xmax}, t_{ymax}, t_{zmax}, we guarantee that their order in terms of value, using infinite precision remains the same when downscaling the precision. Thus, the order of t_{min} and t_{max} remains the same even with representing floating point numbers with the smallest precision, resulting always in a correct result when the original/real ray hits the AABB, therefore succeeding in the elimination of any false miss error case. Consequently, by using directed rounding modes to achieve interval arithmetic, we can propose an arbitrary precision implementation for the Slabs Method that not only guarantees false miss elimination, but it also explores the false hit error rate while downscaling the precision from double down to the minimum of one bit.

In detail, there are three different possible combinations regarding the ray's origin position and the Axis-Aligned Bounding Box minimum/maximum points for each one of the three planes. For brevity the derivations will focus on the x-axis formulas. Assuming that the maximum point (*x2*, *y2*, *z2*) given is always component-wise greater than the minimum point (*x1*, *y1*, *z1*):

1) The ray's origin lies between the slabs: x1 $\leqslant$ x0 $\leqslant$ x2, or
2) The ray's origin is left to the minimum point of the AABB: x1 $>$ x0 $<$ x2, or
3) The ray's origin is right to the maximum point of the AABB x1 $<$ x0 $>$ x2

The same principles exist for the other two remaining dimensions y and z. By checking the signs of the reciprocal direction vector $\overrightarrow{\boldsymbol{id}_{\boldsymbol{x,y,z}}}$ which signify the direction of the ray in the corresponding planes and the AABB's minimum and maximum extent (x1,x2) we can avoid wasteful computation for the intersection test when:

1) The ray's direction on x axis $\overrightarrow{\boldsymbol{id}_{\boldsymbol{x}}}$ is negative and the ray's origin lies left to the minimum point of the AABB, x1 $>$ x0 $<$ x2
2) The ray's direction on x axis $\overrightarrow{\boldsymbol{id}_{\boldsymbol{x}}}$ is positive and the ray's origin lies right to the maximum point of the AABB, x1 $<$ x0 $>$ x2

This fact makes the implementation more robust since it guarantees that in the above cases there is zero possibility of an intersection hit between the ray and the AABB because the ray's origin lies either behind the box with negative direction or in front of the box with positive direction, thus efficiently reducing the amount of area that floating point operations would cost in hardware because there is no need to design the corresponding logic. A case split for the ray's direction must be done before performing the floating point subtraction and multiplication (multiply by the reciprocal direction vector instead of dividing) that the algorithm specifies. Remember that each value is represented with an interval such as: $x0 \equiv [x0_{LOW}, x0_{UP}]$, $x1 \equiv [x1_{LOW}, x1_{UP}]$, $x2 \equiv [x2_{LOW}, x2_{UP}]$, $id_x \equiv [id_{x_{LOW}}, id_{x_{UP}}]$. If the direction is positive ($id_{x_{LOW}} > 0$), we must check where the ray's origin lies.

1) If the upper bound of ray's origin $x0_{UP}$ lies to the left of the AABB's minimum point ($x1_{LOW} > x0_{UP}$, see Figure 5 left) then the interval calculations for t_{xmin} and t_{xmax} will be split to the following, where $\downarrow$ denotes rounding to $-\infty$ and $\uparrow$ denotes rounding to $+\infty$:

$$sub1 \equiv \min\{x1_{LOW} - x0_{LOW}, x1_{LOW} - x0_{UP}, x1_{UP} - x0_{LOW}, x1_{UP} - x0_{UP}\} = x1_{LOW} -_{\downarrow} x0_{UP} \quad (1)$$

$$t_{xmin} \equiv \min\{sub1 \times id_{x_{LOW}}, sub1 \times id_{x_{UP}}\} = = sub1 \times_{\downarrow} id_{x_{LOW}} \quad (2)$$

$$sub2 \equiv \max\{x2_{LOW} - x0_{LOW}, x2_{LOW} - x0_{UP}, x2_{UP} - x0_{LOW}, x2_{UP} - x0_{UP}\} = x2_{UP} -_{\uparrow} x0_{LOW} \quad (3)$$

$$t_{xmax} \equiv \max\{sub2 \times id_{x_{LOW}}, sub2 \times id_{x_{UP}}\} = = sub2 \times_{\uparrow} id_{x_{UP}} \quad (4)$$

2) If the lower bound for the ray's origin $x0_{LOW}$ lies to the right of the AABB's maximum point ($x0_{LOW} > x2_{UP}$), there is no operation required since the ray will be always in front of the box.
3) Finally, if the ray's origin lies between the slabs ($x1_{LOW} \leqslant x0_{LOW}$ && $x2_{UP} \geqslant x0_{UP}$) then the only change required for the interval calculations in comparison with

case 1), pertains to the computation of t_{xmin} :

$$t_{xmin} \equiv \min\{sub1 \times id_{x_{LOW}}, sub1 \times id_{x_{UP}}\} = \\ = sub1 \times_{\downarrow} \boldsymbol{id_{x_{UP}}} \quad (5)$$

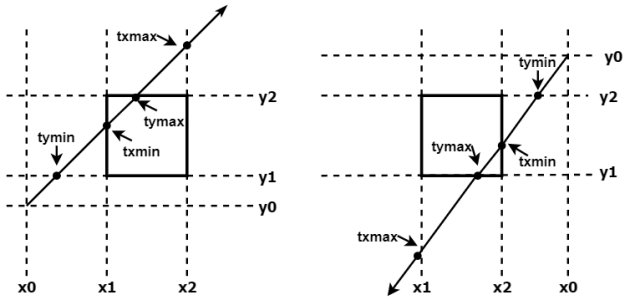

Fig. 5: Cases for Ray/AABB hit. Left: Ray's origin left to AABB's minimum point and positive direction. Right: Ray's origin right to AABB's maximum point and negative direction.

As can be identified *sub1*, *sub2* and t_{xmax} always compute the same operation of the interval, thus for the case split where the ray's direction is positive they are assigned to execute the above FP operations with *no dependence* on the ray's origin. Therefore, some parts of the code can be optimized by computing only a single value of the interval, leading to fewer floating point calculations than required for a complete interval arithmetic implementation. Consequently, for the case where the ray's direction is positive, the algorithm will only execute the following floating point operations and perform the following comparisons:

$$sub1 = x1_{LOW} -_{\downarrow} x0_{UP} \quad (6)$$

$$sub2 = x2_{UP} -_{\uparrow} x0_{LOW} \quad (7)$$

$$t_{xmax} = sub2 \times_{\uparrow} id_{x_{UP}} \quad (8)$$

If the upper bound of the ray's origin $x0_{UP}$ lies to the left of the AABB's minimum point then the interval analysis ends up only with the following case for t_{xmin} calculation:

$$t_{xmin} = sub1 \times_{\downarrow} id_{x_{LOW}} \quad (9)$$

For any other different case:

$$t_{xmin} = sub1 \times_{\downarrow} id_{x_{UP}} \quad (10)$$

The additional cases for the x-axis where the lower bound of the ray's origin $x0_{LOW}$ lies to the right of AABB's maximum point (negative direction needed for the ray to be tested for intersection) and when the ray is parallel to the plane are highlighted in the pseudocode below. The use of bold symbols attempts to distinct the slight differences in the conditions to be checked and in the floating point operators used for the multiplications when the ray's direction is changed from positive to negative or vice-versa. The algorithm follows the same pattern for y and z dimensions and by comparing the t_{max} and t_{min} values decides if a miss or hit occurs. GNU-MPFR [13] supports IEEE-754 directed rounding modes and was used to evaluate false misses elimination.

Proposed algorithm, C-code example for x-dimension using GNU-MPFR.

```
if (x) { //ray is parallel to the x-plane

   if (mpfr_less_p (x0UP, x1LOW) || (mpfr_greater_p (x0LOW, x2UP))
        return (0);  //ray misses the box
}
else if (mpfr_greater_p (idxUP, zero)) {
 //positive direction

    mpfr_sub  (sub1, x1LOW, x0UP, MPFR_RNDD);
    mpfr_sub  (sub2, x2UP, x0LOW, MPFR_RNDU);
    mpfr_mul  (txmax, sub2, idxUP, MPFR_RNDU);

    if  (mpfr_greater_p(x1LOW, x0UP))
         mpfr_mul(txmin, sub1, idxLOW, MPFR_RNDD);
    else  mpfr_mul  (txmin, sub1, idxUP, MPFR_RNDD);
}
else { //negative direction

    mpfr_sub  (sub1, x1LOW, x0UP, MPFR_RNDD);
    mpfr_sub  (sub2, x2UP, x0LOW, MPFR_RNDU);
    mpfr_mul  (txmax, sub1, idxLOW, MPFR_RNDU);

    if  (mpfr_greater_p(x0LOW, x2UP))
         mpfr_mul  (txmin, sub1, idxUP, MPFR_RNDD);
    else  mpfr_mul  (txmin, sub2, idxLOW, MPFR_RNDD);
 }
```

IV. Correct Rounding Adjustment for FloPoCo

To design a Slabs Method algorithm that eliminates false misses in RTL, we need to perform directed rounding for the floating point subtractions and multiplications. Because there is not any closed or open-source library that rounds to $\pm\infty$ it was needed to manually adjust FloPoCo VHDL operators to generate them. FloPoCo takes the default approach and rounds to nearest even for each operation. Table I gathers the information given in [14] and illustrates the conditions that must be satisfied for performing rounding in floating point addition/subtraction and multiplication. Notations to (G), (R) and (S) accordingly refer to *guard*, *round* and *sticky* bits.

TABLE I: Conditions to be satisfied for performing rounding addition in a) Floating Point Addition/Subtraction and b) Floating Point Multiplication.

Floating Point Operation	Rounding Mode	Condition for Rounding Addition
FP Add/Sub	Round to Nearest/Even	$G \cdot (LSB + R + S)$
	Round to Plus Infinity	$sign' \cdot (G + R + S)$
	Round to Minus Infinity	$sign \cdot (G + R + S)$
FP Mult	Round to Nearest/Even	$G \cdot (LSB + S)$
	Round to Plus Infinity	$sign' \cdot (G + S)$
	Round to Minus Infinity	$sign \cdot (G + S)$

A. Rounding Adjustment for FP Multiplication

To adjust a FloPoCo's floating point multiplication operator to perform negative/positive infinity rounding is quite simple. We need only the guard (G) and sticky (S) bits. FloPoCo identifies these bits, therefore we only change the condition for

rounding and instead of using the round to nearest/even take the appropriate for rounding to plus/minus infinity. Referring to Table I it is important to use the sign bit that is computed early in the correct clock cycle, so if the multiplication lasts two cycles, the rounding addition decision is taken when entering the second cycle, thus a delayed version of sign has to be used. The round bit is given as carry to the rounding adder and an addition between the normalized result and zero is performed. If the condition is true, then a unit in the last place (*ulp*) is added to the normalized result, else the result remains the same and no rounding is needed:

```
round <= '1' when (sign_d2='0' and
(sticky_d1='1' or guard='1')) else '0';
```

The only alteration to produce the negative infinity rounded value is to invert the sign bit:

```
round <= '1' when (sign_d2='1' and
(sticky_d1='1' or guard='1')) else '0';
```

B. Rounding Adjustment for FP Addition/Subtraction

The adjustment of FloPoCo's addition/subtraction is slightly trickier. If the operation requires an effective subtraction [14] the result might have leading zeros and might require a left shift of the significand by l positions corresponding to the number of leading zeros. This is the worst case in which it has been shown that three extra bits are enough for correct rounding (G,R,S) [14]. FloPoCo holds the three additional bits after the LSB with the following order: G—R—S. The shifted fraction is concatenated with the G bit (after the LSB's place), the condition of round to nearest/even is examined (see Table I) and if it is true, a *ulp* is added to the *guard* bit. Consequently only when G = 1, LSB = 1 and the condition is true will the LSB be incremented. In the final shifted fraction value the guard bit (G) gets discarded. To adjust this and correctly perform directed rounding to $\pm\infty$, the idea is to concatenate the shifted fraction with the corresponding rounding condition bit value, then insert it as carry into the rounding adder and add the shifted fraction with zero. For instance, if we want to round a positive number to plus infinity, and if LSB = 0, G = 0 , R = 0, S = 1, the condition to add a ulp is true and the *addToRoundBit* will be concatenated to the expanded fraction after the LSB such as it ends with ....01. By adding zero to the expanded fraction and setting the carry-in bit as the *addToRoundBit* we achieve to correctly round to plus infinity without adding any latency or extra area to the floating point adder/subtracter. Figure 6 illustrates these modifications by showing the default FloPoCo's rounding method compared with the adjusted rounding to $+\infty$ for a 32-bit subtractor. Again, if rounding to $-\infty$ is wanted then the only modification required is to invert the result's sign (*signR_d5*).

V. Proposed Hardware Architecture

The main design decision lies in the trade-off that interval arithmetic introduces. In particular, the use of interval arithmetic increases the number of floating point operations, leading to more hardware complexity (area) compared to the original floating point Slabs Method hardware design. As a

```
expFrac<= updatedExp & shiftedFrac_d1(26 downto 3);
addToRoundBit<= '0' when (lsb='0' and grd='1' and rnd='0' and stk='0')  else '1';
roundingAdder: IntAdder_37_f400_uid20  -- pipelineDepth=0 maxInDelay=0
   port map ( clk  => clk,
              rst  => rst,
              Cin => addToRoundBit,
              R => RoundedExpFrac,
              X => expFrac,
              Y => "0000000000000000000000000000000000000");
```

```
addToRoundBit<= '1' when (signR_d5='0' and (grd='1' or rnd='1' or stk='1'))  else '0';
expFrac1<= updatedExp & shiftedFrac_d1(26 downto 4) & addToRoundBit;
roundingAdder: IntAdder_34_f400_uid20  -- pipelineDepth=0 maxInDelay=0
   port map ( clk  => clk,
              rst  => rst,
              Cin => addToRoundBit,
              R => RoundedExpFrac,
              X => expFrac1,
              Y => "0000000000000000000000000000000000");
```

Fig. 6: Top: FloPoCo's default rounding method. Bottom: Modified VHDL code to round to positive infinity.

result, this affects the performance of the ray tracer, because the increased number of floating point operations increases the latency of the design. The fact that the ray/AABB intersection tests get executed in a graphics pipeline, where throughput is essential, drove us to aim for and achieve an initiation interval of 1. This means that the ray tracer could generate an output decision for the ray/AABB intersection test in each clock cycle after the first output. This is possible because each set of the algorithm's inputs (ray segments and AABB coordinates) is completely independent to the other inputs. To always keep all the floating point operators busy, a multiple-stage fully pipelined architecture was designed.

The inputs to the hardware module are: the ray's origin (*x0,y0,z0*) and reciprocal direction vector (id_x, id_y, id_z), the Axis-Aligned Bounding Box minimum (*x1,y1,z1*) and maximum (*x2,y2,z2*) extent and 3 Booleans (*x,y,z*) that clarify whether the ray is parallel to the corresponding plane. Figure 7 illustrates a higher level view of the proposed architecture. The hardware design of the algorithm is mainly separated in six different phases:

1) In the first phase, 18 FP Adders generate the required values for the proposed algorithm by performing an addition with zero and round accordingly to $\pm\infty$. 12 of them generate lower and upper values for the ray's origin and the reciprocal direction vector while the other 6 generate the box's upper values for its maximum point ($x2_{UP}, y2_{UP}, z2_{UP}$) and its lower values for its minimum point ($x1_{LOW}, y1_{LOW}, z1_{LOW}$) .
2) The second phase involves six FP Subtractions as described in Section III. In parallel, it is checked whether a miss occurs in case the ray is parallel to one of the planes.
3) In the third phase all the possible values for $t_{xmin,ymin,zmin}, t_{xmax,ymax,zmax}$ are generated with the use of 18 FP Multipliers (as explained in Section III).
4) In the fourth phase, the ray's direction and origin for each of the three planes is checked. Depending on them, the correctly rounded values are assigned to $t_{xmin,ymin,zmin}, t_{xmax,ymax,zmax}$. It is worth mentioning that there are no additional floating point operations required during this phase, since the signs of previously-calculated values could be propagated and used during

this phase. For example, in phase 2 we have subtract $x2_{UP}$ from $x0_{LOW}$, thus we know that if the result of the subtraction is negative then the ray's origin will lie to the right of the AABB.

5) In the fifth phase, the comparisons for the possible t intersection distances take place and the final values are assigned to t_{min} and t_{max}.
6) Finally, in the sixth phase, t_{min} and t_{max} are compared and if $t_{max} \geqslant t_{min}$ and $t_{max} > 0$, the ray hits the AABB. It is worth mentioning that the Boolean that signifies a miss that may have occurred if the ray was parallel to one of the planes is propagated and taken into consideration to produce the final Boolean result.

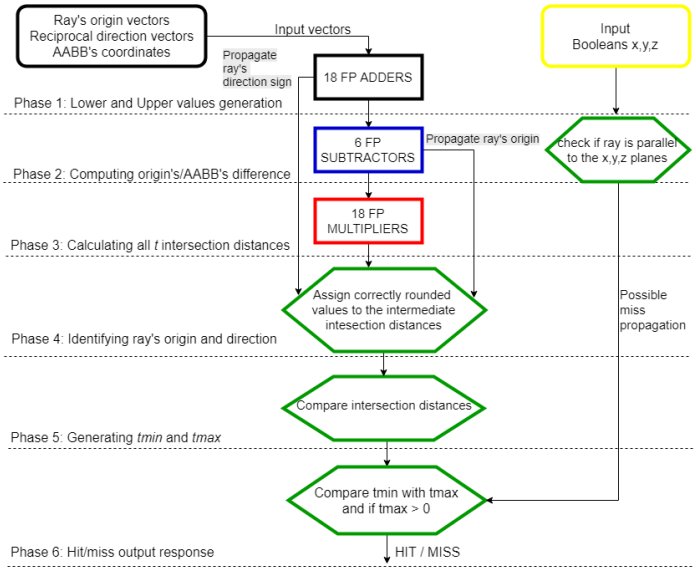

Fig. 7: Top level flow of the hardware architecture for the proposed new method.

Extra care has been taken for the case when the compared values are equal. FloPoCo uses two additional bits in front of the sign bit for encoding floating point bitwidths. This feature can be exploited when comparing values using a floating point subtraction. So, the only need is to check those two bits on the resulting floating point number and if both are zero it signifies that the compared numbers were equal. It is not necessary to use the significand of the resulting floating point number, reducing circuit complexity. The hardware implementation is fully parameterised in terms of mantissa width. However, since FloPoCo's pipeline depths scale in accordance with the precision, it may need to adjust clock frequency for correct operation of the algorithm.

VI. Results

The FPGA that was targeted is Xilinx's XC7Z020-3CLG484 consisting of 53,200 LUTs, 106,400 Slice Registers/FF, 140 36KB BRAM blocks and 220 DSP48E1s [15]. As the precision decreases in the multiple hardware designs, we expect the area to be scaled down, since each floating point number is represented with fewer bits and all FloPoCo's operators scale accordingly. Indeed, when downscaling the precision from double to 1 bit, the FPGA's most reduced resource were LUTs and secondly the Slice Registers. The amount of area (LUTs and Registers) that must be used to achieve the corresponding false hit error rate is illustrated in Figure 8 and Figure 9. As area increases, the false hit error rate reduces exponentially until the 9th bit of precision, after this point FloPoCo's operators use DSPs for the multiplications and it is shown that even with 9 or 11 bits of precision the amount of LUTs used is the same. Again, after that point the trend for the error rate continues the same (drops exponentially).

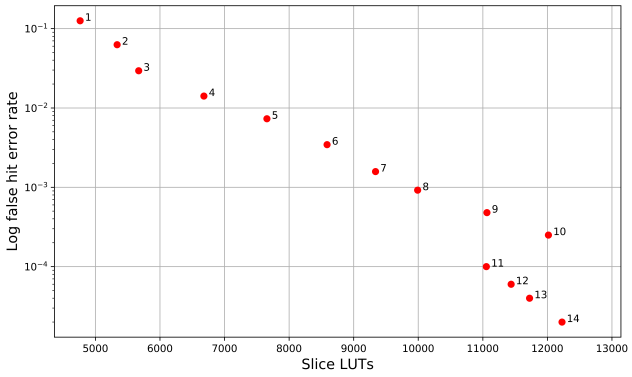

Fig. 8: False hit error rate against Slice LUTs. Precision is annotated.

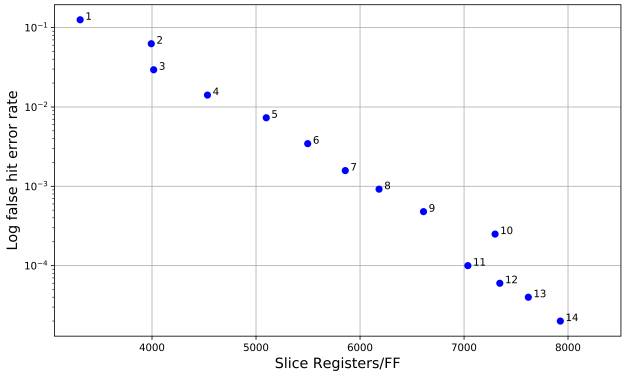

Fig. 9: False hit error rate against Slice Registers/FF. Precision is annotated.

Since we are in a graphics pipeline, parallelism must be exploited. Therefore we designed a high frequency/throughput implementation. As the precision scales up, FloPoCo increases the number of pipeline stages in order to maintain clock frequency relatively high by trading latency. As a result, we expect frequency or clock period to be maintained asymptotically constant for the different designs. In Figure 10, the minimum clock period that can be achieved (or the highest frequency) against false hit error rate is presented. A drop in throughput as precision increases can be observed, however this is not always true since FloPoCo operators can target specific FPGAs

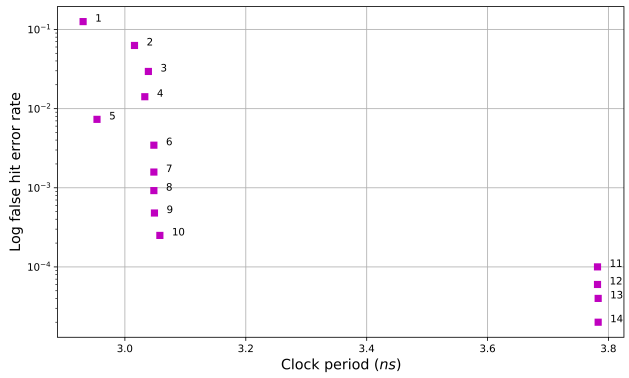

Fig. 10: False hit error rate against clock period. Precision is annotated.

to help designers identify the maximum frequency that can be achieved. Again the cost in area should be taken into consideration.

Although we demonstrated the trade-offs between false hit error rate, area and throughput when precision is downscaled, measuring false hit error rate for high precision is hard through simulation, as it is very close to zero.

VII. Conclusion

This work, based on one of the most popular ray/AABB intersection test algorithms, the Slabs Method, proposes a method using *runtime interval* calculation efficiently *avoids false misses*, using *any* precision to represent its internal floating point computation. This enhances the quality of the ray tracer in terms of generating images with high photorealism, eliminating the possibility of rendering an image with visually-objectionable holes. The trade-off of using interval arithmetic to eliminate false misses results in increased hardware complexity (silicon area) comparing our proposed circuit with the original Slabs Method. Since ray tracers require high performance, a high-throughput fully pipelined architecture that keeps all the FP operators busy in each clock cycle is designed (initiation interval $= 1$). Exploring alternative algorithms that check for ray/AABB intersection and systematically generating hardware for realizing the related trade-offs discussed is left for future work.

The paper also contributes to the floating point arithmetic by providing an implementation of FP multipliers, adders/subtracters that round to the IEEE-754 $\pm\infty$ standard. The open-source code of this work can be found online at https://github.com/constantinides/RAABB.

Acknowledgement

The authors wish to acknowledge the support of EPSRC (EP/P010040/1, EP/K034448/1, EP/I020357/1), the Royal Academy of Engineering, and Imagination Technologies.

References

1 Whitted, T., "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980.
2 Rubin, S. M. and Whitted, T., "A 3-dimensional representation for fast rendering of complex scenes," *SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 110–116, Jul. 1980.
3 Glassner, A. S., Ed., *An Introduction to Ray Tracing*. London, UK: Academic Press Ltd., 1989.
4 Mahovsky, J. A., "Ray tracing with reduced-precision bounding volume hierarchies," Ph.D. dissertation, Calgary, Alta., Canada, Canada, 2005.
5 Kay, T. L. and Kajiya, J. T., "Ray tracing complex scenes," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 269–278, Aug. 1986.
6 Smits, B., "Efficiency issues for ray tracing," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005.
7 Williams, A., Barrus, S., Morley, R. K., and Shirley, P., "An efficient and robust ray-box intersection algorithm," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005.
8 Goldberg, D., "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.
9 Constantinides, G. A., Cheung, P. Y. K., and Luk, W., "The multiple wordlength paradigm," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, March 2001, pp. 51–60.
10 Constantinides, G. A., "Massively parallel numerical computation on FPGAs," in *FPGA Developers' Forum, 2007 Institution of Engineering and Technology*, Oct 2007, pp. 1–14.
11 Constantinides, G., Kinsman, A., and Nicolici, N., "Numerical data representations for FPGA-based scientific computing," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 8–17, July 2011.
12 de Dinechin, F. and Pasca, B., "Designing custom arithmetic data paths with FloPoCo," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 18–27, July 2011.
13 Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P., "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007.
14 Ercegovac, M. D. and Lang, T., *Digital Arithmetic*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.
15 Xilinx, "Zynq-7000 all programmable soc data sheet," vol. DS190 (v1.11), pp. p.2–4, June 7, 2017.