Improving Ray Tracing Precision by Object Space Intersection Computation

H. Dammertz*
Abt. Medieninformatik,
University of Ulm, 89069
Ulm, Germany

A. Keller†
Abt. Medieninformatik,
University of Ulm, 89069
Ulm, Germany

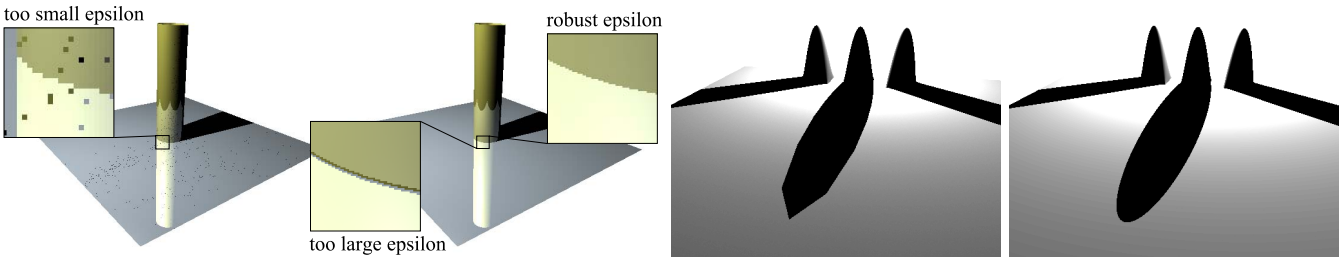

Figure 1: Common problems of ray tracing: On the left the problem of choosing a good epsilon environment in order to avoid self-intersections is shown. A too small epsilon results in self-intersections of secondary rays, while a too large epsilon results in overlaps from the extended geometry. On the right a light source close to a silhouette casts a long shadow. While the geometry approximation is not visible at the silhouette, the approximation becomes unavoidably obvious in the projection, which is difficult to predict in the tessellation process.

ABSTRACT

Instead of computing intersections along a ray, an algorithm is proposed that determines a point of intersection in object space. The method is based on the classical refinement of a hierarchy of axis-aligned bounding boxes that is computed on the fly. Typical rendering artifacts are avoided by taking into consideration the precision of floating point arithmetic. In addition the method lends itself to a simple solution of the self-intersection problem. Considering the obtained precision the algorithm is efficient, simple to use, and to implement.

1 INTRODUCTION

Finding the first intersection of a ray and a geometric surface primitive is the core operation of any ray tracing system. Due to the limitations of floating point arithmetic many of the commonly used intersection algorithms have intrinsic numerical problems that can result in visible artifacts [1, 28]. Especially for ray tracing free form surfaces these problems lead to complex algorithms with many special cases or the need for user defined parameters in order to adjust quality. In the usual case of approximation the amount of tessellation that is required for visually correct results is hard to determine and heuristics easily can fail for e.g. shadows (see figure 1), transparencies, or reflections (see figure 9). In addition already simple triangle meshes can report false ray intersections along the triangle boundaries (see figure 14). While it is infeasible to choose globally valid parameters to avoid these problems, relying on anti-aliasing to hide false intersections is not a desirable solution, either.

The general algorithm described in this paper avoids many of these problems, is very simple, and can be used with many different types of geometric surface primitives. It efficiently computes a reliable point of intersection almost up to floating point precision without the need of adjusting parameters. Especially when used with free form surfaces it results in highly accurate images with correct secondary effects even after multiple reflections or at extreme close ups. The self-intersection problem can be avoided.

*e-mail: holger.dammertz@uni-ulm.de
†e-mail: alexander.keller@uni-ulm.de

2 OBJECT SPACE INTERSECTION COMPUTATION

Usually a point of intersection is determined by computing a length from the origin along the direction of the ray. Considering the nature of floating point numbers [8], the resolution becomes coarser with increasing distance from the ray origin. In addition computing the point of intersection in object space forces the point to be quantized to the representable values along the object space coordinate axes.

Since these errors cannot be efficiently avoided in ray space, it pays off to determine the intersection in object space, offering the following two key advantages:

1. A higher numerical precision is obtained by avoiding some of the quantizations, and

2. self-intersections of secondary rays can be avoided.

Instead of computing an intersection as a distance along the ray and determining the point of intersection in object space from that distance, we directly compute a 3-dimensional interval in object space. This interval contains the true point of intersection and can also be interpreted as an axis aligned bounding box. It is found by hierarchical subdivision (see also figure 2).

The listed pseudo code describes the hierarchical traversal of bounding volumes similar to previous work [10]. Querying the nearest interval of intersection can be implemented in the report function. Depending on the application other functions like e.g. derivatives or parameterizations of the geometry can also be subdivided along with the intersection computation to yield a bounding interval.

```
Intersect(Ray, Object)
{
  bbox = axis aligned bounding box of Object
  if(Ray misses bbox)
    return
  else if(terminate)
    report(bbox as intersection interval)
    return
  else
  {
    sd = subdivide Object
    for(all objects o in sd)
      Intersect(Ray, o)
  }
}
```

Listing 1: Bounding volume hierarchy traversal.

2.1 Numerically Robust Algorithm

Bounding intervals can be computed by interval or affine arithmetic [10, 13, 4], which is the common means to address floating point precision problems. However, each arithmetic operation then has to be performed in interval or affine arithmetic resulting in a high computational effort. Using the key observation that for two floating point numbers $a \neq b$ it can happen that $(a+b)/2 = a$, the overhead of replacing arithmetic operations by operations on intervals can be avoided: Intervals are subdivided until the subdivision no longer changes in the floating point representation. For the ray intersection with an object this involves:

Numerically robust subdivision: Cracks usually result from different levels of subdivision or numerical problems. If the neighboring bounding boxes resulting from subdivision overlap or at least touch in floating point representation, no cracks, even between different levels of subdivision, can occur. In the applications section we show how to guarantee this.

Numerically robust termination: Subdivision is terminated if the size of the resulting bounding boxes no longer is changing in floating point arithmetic. The resulting bounding box then is reported as intersection interval.

It is important to select a measure that is robust with respect to the floating point arithmetic. Computing the Euclidian length of the diagonal of a bounding box involves squares. As the bounding boxes tend to become very small by subdivision, cancelation errors [8] would cause too early termination. We therefore use the L_1-norm of the bounding box diagonal, i.e. the sum of the absolute values. In contrast to the Euclidean length it only requires subtractions and additions to compute and thus remains more precise.

Fulfilling the above conditions in an implementation of the bounding volume hierarchy traversal (see the pseudo code) results in a crack and hole free intersection computation. The decision of whether or not a bounding box is intersected by a ray is decided using the algorithm of Williams et al. [27]. Seemingly contradictory to our goal, although not used for the intersection interval, this method uses distances along the ray. Note that we do not have any other choice as long as the rays are given by origin and direction. However, the computations remain consistent, because for floating point arithmetic the above conditions assure that if a box is intersected at a higher level of the hierarchy there exists at least one of them, which is intersected in the lower levels. This includes the situation where the floating point resolution along the ray is not sufficient to precisely capture the intersection interval in object space. Such a situation arises for a ray far from the origin targeting at an object close to the origin.

2.2 Secondary Rays

Realistic image synthesis requires shooting secondary rays in order to compute the illumination. The problem of self-intersections becomes apparent (see figure 1), when shooting off rays from surfaces: Due to numerical inaccuracies the same surface may be hit again, resulting in false points of intersection. A common solution to this problem is adding a small distance along the ray or normal direction. The choice of this epsilon is largely scene dependent and often left to the user of the ray tracing system. Using object identifications also does not help for non-planar or touching objects.

With the intersection interval computed by our algorithm we can avoid the self-intersection problem. The starting point of the secondary ray is selected as the corner point of the intersection interval farthest in the direction of the normal (see the illustration in 3). For transparency rays we just use the normal with inverted signs. As a valid ray interval excludes the origin of the ray, it is impossible to hit the interval of origin again.

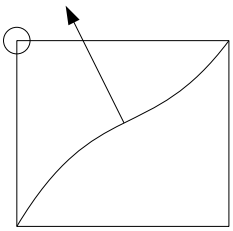

Figure 3: Avoiding the self-intersection problem by selecting the corner of the intersection interval that is farthest in the direction of the surface normal.

This method works perfectly in convex geometries. Nevertheless, an epsilon environment is required, to avoid early intersections of close to tangential rays in concave geometry. But this can be done globally without user interaction. The intersection interval is enlarged by considering its binary representation of the floating point number as integer [11]. In our implementation we subtract / add the value of 7 (3 bits). This way the epsilon value is implicitly adapting to the exponent of the floating point number. It has been found by comparing single and double precisions computation.

3 Applications

We demonstrate the general algorithm for three applications, namely polynomial free form surfaces, trimming, and improved triangle intersection testing. The speed of the algorithms is interactive, however, as we focus on precision, might be inferior to more approximative schemes.

3.1 Polynomial Free Form Surfaces

The approaches to ray tracing free form surfaces can be roughly classified into three different strategies: First the surface can be approximated by simpler surfaces. Often planar polygons or bilinear patches are used for the approximation. However, this eventually results in the problems illustrated in figures 1, 9, and 14.

Secondly, there are subdivision based algorithms that use a simple prune and search method. Early methods are described by Whitted [26] and Rubin [22]. Woodward [29] transforms the problem to two dimensions. Nishita et al. [19] developed the well known technique of Bézier clipping, which has been improved in [3]. Another subdivision based method called Chebyshev boxing is described in [6]. Recent work on subdivision based methods can be found in [18, 2].

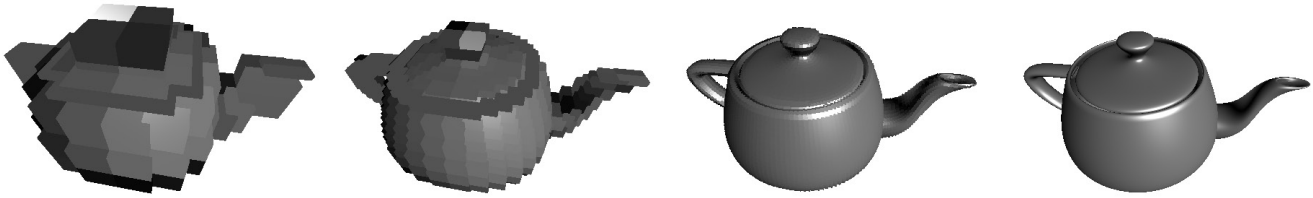

Figure 2: Illustration of the hierarchical refinement of the bounding boxes as used in the algorithm. Instead of ray parameters bounding boxes that contains the actual points of intersection are determined directly.

The third category comprises numerical solutions, where a form of Newton iteration is used in most cases to find the point of intersection. All methods based on simple Newton iteration [7, 16] have the problem of finding a good starting point and the user has to select parameters to obtain a correct image (see figure 4). Kajiya [14] uses algebraic geometry to create a numerical procedure for finding the intersection between a ray and a bivariate cubic parametric patch. A more accurate approach is described by Toth [23] and improved in [15]. It uses multivariate Newton iteration to solve the convergence problem of Newton iteration but this method is rather slow.

All the above methods suffer from disadvantages like coarse approximations, numerical problems, extensive computation, parameters that are difficult to determine for an overall scene, or restrictions on subdivision depth and surface degree.

Our method falls into the second category. We now specify the algorithm from the previous section for polynomial free form surfaces of arbitrary degree.

Figure 4: False surface intersection reported by Newton iteration, when the starting point is not sufficiently close to the fix point.

3.1.1 Polynomial Tensor Product Surfaces

Using the Bézier representation is one of the simplest ways to describe a polynomial surface. Even though it is not well suited for modeling purposes any polynomial surface can be converted into the Bézier basis (NUBS surfaces for example by knot insertion [20] - it is not NURBS, since it is not rational). For the remainder of this discussion we assume familiarity with Bézier surface patches [5].

A Bézier surface patch

$$p(u,v) := \sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(u) B_j^n(v) p_{ij} \qquad u,v \in [0 \dots 1]. \tag{1}$$

is defined by a two dimensional grid of control points p_{ij} and the Bernstein basis functions $B_i^k(t)$ of degree k. The axis aligned bounding box of a Bézier patch is computed by taking the minimum and maximum of all $(n+1)(m+1)$ control points p_{ij} in each component.

The subdivision of a Bézier surface patch is done via the de Casteljau algorithm using the recursion property of the Bernstein basis. With respect to floating point computations the best parameter value for subdivision is 0.5. This guarantees that the boundaries of the intersection intervals always touch and no cracks can appear as required in section 2.1. In addition the division by 2 is just a decrement of the exponent and is always exact unless a floating point underflow occurs [8].

The selection of the subdivision is guided by a simple heuristic. It estimates the longest direction by computing the length of the two vectors $(p_{n,0} - p_{0,0})$ and $(p_{0,m} - p_{0,0})$ and subdivides along the longer direction. The norm used to determine the length only affects the efficiency of the scheme. Therefore, we use the maximum norm, which can be evaluated fastest. The heuristic works even for very distorted self-intersecting patches of high degree as shown in figure 6, because the resulting patches become very regular after only a few subdivisions.

3.1.2 Accuracy

To analyze the accuracy obtained by our subdivision algorithm, we rounded the results of an implementation using double precision arithmetic to the nearest single precision number. These were compared to the actual single precision version and the results of a uniform triangulation of the patch. The triangle intersection algorithm used is the one described in [17]. Its intersection point was compared to the midpoint of the intersection interval of our method. Figure 8 shows the two patches that were used. The results can be found in tables 1 and 2. It becomes obvious that tessellation approaches like e.g. [2] result in larger approximation errors. Even for high tessellations our subdivision intersection algorithm performs better by two orders of magnitude.

The errors obtained by interpolating normals of triangle corners and direct computation are very similar. This can be clearly seen in the reflections in figure 9.

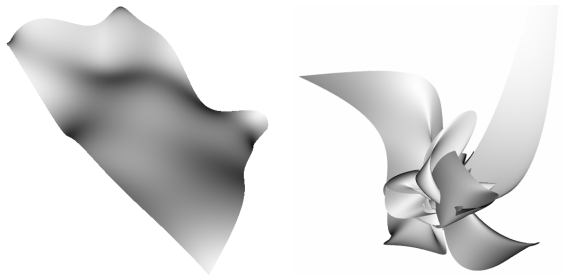

Figure 6: Two ray traced Bézier patches (left: degree 10 in u and degree 7 in v direction, right: random control points and degree 42 in both u and v direction) demonstrating robustness of the algorithm for high degrees. Finding stable starting points for Newton iteration becomes almost impossible in this setting.

Figure 5: A NURBS scene converted into 261692 Bézier patches and ray traced with our algorithm.

3.1.3 Speed of Convergence

The subdivision of a Bézier patch can be implemented efficiently using the vector instructions of modern computer architecture (see [2] for example). A simple and straightforward implementation was chosen for the analysis. The subdivision and bounding box computation was implemented using SSE instructions but only 3 of the 4 slots were used. The reason for the good performance of our algorithm is that it completely runs in the level 1 cache of the processor. As such it is compute-bound.

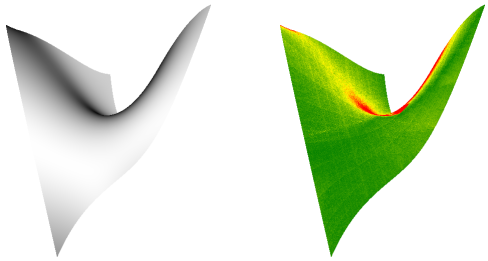

Figure 7: A Bézier patch with the color encoded number of subdivisions needed to determine the intersection interval. Green is about 50 subdivisions, yellow up to 200 and red more than 200 subdivisions.

The first interesting thing about the algorithm is the number of subdivisions needed until an intersection interval is found. As can be seen in figure 7, the effort needed to determine the intersection interval grows the more the ray direction tangential to the surface. Note that the same effect would be observed for tessellations. The reason for this is an increased number of bounding boxes that have to be intersected when the ray passes almost parallel to the surface. Furthermore the number of iterations needed to find an intersection interval depends on the size of the patch and its position in 3d space. Due floating point arithmetic larger patches and patches that are close to the origin take longer to ray trace [8]. For the analysis all patches were contained in $[1,2]^3$. We provide this interval for reference.

The control polygon of a Bézier surface converges quadratically to the surface when the de Casteljau subdivision is used [21]. Consequently we have the same rate of convergence for the bounding box of the control polygon, too.

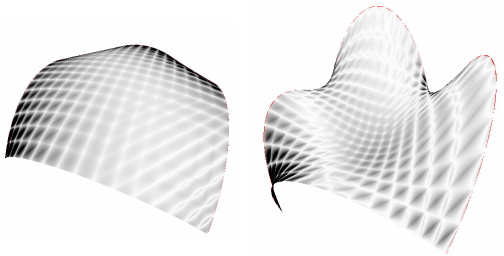

Figure 8: The two patches used for the accuracy analysis. The image shows the difference between the patch tesselated to 512 triangles and the single precision subdivision algorithm.

# Tris	Max. Error	Min. Error	Mean Error
512	$3.0642e-1$	$4.4703e-8$	$4.9815e-3$
4608	$8.5224e-2$	$8.4750e-8$	$5.5706e-4$
131072	$2.7960e-3$	$0.0000e+0$	$1.9592e-5$
our Alg.	$8.5681e-5$	$0.0000e+0$	$2.3038e-7$

Table 1: Point of intersection error for a simple patch (figure 8 on the left) compared to the double precision results.

3.1.4 Performance

The algorithm for ray tracing Bézier patches was integrated into a simple ray tracing system that also allows ray tracing of triangle models and has support for hierarchical materials and different light sources. The triangle ray tracer uses kD-trees for each object as acceleration structure. A top level axis aligned bounding volume hierarchy (BVH) is used to support instances and multiple objects. The Bézier ray tracer uses the BVH for top level and for individual objects. The test scenes are completely composed of NURBS patches and were ray traced at a resolution of 512x512 pixels with

# Tris	Max. Error	Min. Error	Mean Error
512	$1.8080e+0$	$5.9604e-8$	$9.5757e-3$
4608	$4.0562e-1$	$0.0000e+0$	$1.1005e-3$
131072	$1.1100e-2$	$0.0000e+0$	$3.8533e-5$
our Alg.	$9.3240e-5$	$0.0000e+0$	$2.2959e-7$

Table 2: Point of intersection error for a more distorted patch. The patch can be seen in figure 8 on the right.

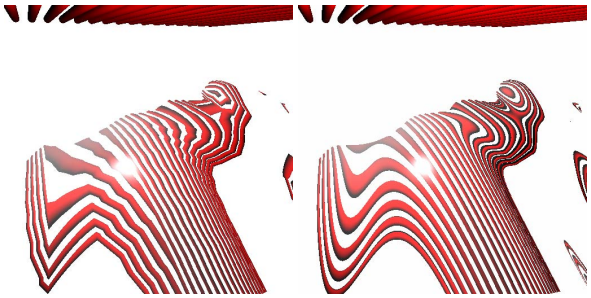

Figure 9: The scene on the left was triangulated into 10752 triangles. The discontinuities of the first derivative are clearly visible in the reflections of the bars. The image on the right was ray traced with the subdivision algorithm. The resolution of both images is 512x512.

one primary ray per pixel. The ray tracing was done on a Pentium 4 at 2.8 GHz with 2GB of RAM and the ray tracers used only a single thread. The conversion of the NURBS patches into integral Bézier patches needed no approximation in this case because of the modeling program Alias Maya 5.0, which generates rational surfaces only in rare cases. The adaptive conversion algorithm integrated into Alias Maya 5.0 was used to create the tessellation. Two test scenes were used for comparison. The dog scene has a simple Lambertian material and only primary rays were traced. The car scene features highly reflective materials. The recursion depth was set to 3 for reflections and 6 for refractions. The scene contains 29 different materials and 9 point light sources. The results are shown in figure 10. The memory consumption is the scene geometry plus the acceleration data structure size. The parse time is the time for reading and interpreting the scene description supported by our rendering system. The render time is the time for acceleration data structure construction and ray tracing a single image. Given the superior precision of our algorithm, it is very competitive.

3.1.5 Optimizations

There are various possibilities to increase the performance of the basic algorithm. Since the intersection computation dominates the rendering time, optimization of the intersection calculation is most important. Obvious improvements are iterative implementation of subdivision using a simple stack instead of a recursive implementation and organizing the data in processor friendly structures and alignments. A huge improvement in subdivision performance can be gained by subdividing the data of a patch in place, overwriting the original patch on the stack. This reduces memory access and allows for better register usage. Another improvement is traversing the subdivided bounding box that is closer to the ray first. This increases the likelihood of an early termination during the subdivision.

A common principle to accelerate the tracing of primary rays is to use coherent packets of rays as described for example by Wald in [25]. However, since bounding boxes are subdivided almost up to the resolution of floating point number, ray coherency is lost and bundle tracing as well as shaft culling [9] does not help much (see table 3).

Using coherent rays for the higher levels of the hierarchy is helpful. This corresponds to truncating accuracy. Instead of using the termination criterion as described in section 2.1, we allow early termination. For example the subdivision can be stopped when the size of the bounding box projected to the screen is smaller than half a pixel. When no secondary rays are traced the resulting images are equivalent to the images computed with the floating point termination criterion. For secondary rays, ray differentials [12] could be used, but this has not been examined yet. Table 3 shows some performance data for a single patch.

#Rays in Bundle	Float Acc.	Early Term.
1x1	0.567 fps	1.266 fps
2x2	0.657 fps	1.996 fps
4x2	0.676 fps	2.236 fps
2x4	0.680 fps	2.280 fps
4x4	0.640 fps	2.395 fps
8x8	0.297 fps	0.530 fps

Table 3: Performance when using ray bundles of different size. The performance data was collected at a resolution of 512x512 with the patch shown in figure 7

3.2 Trimming Curves

Trimming can be used to overcome the topological restriction of tensor product surfaces. For this, a hierarchy of polynomial curves uniquely defines inside and outside points in the parameter domain of the patch. These polynomial curves can be converted into Bézier representation and inside and outside points can be identified by ray tracing, using Jordan's curve theorem . This is done by shooting a ray into an arbitrary direction and counting all intersections with the curves. Odd and even numbers of intersections distinguish inside from outside.

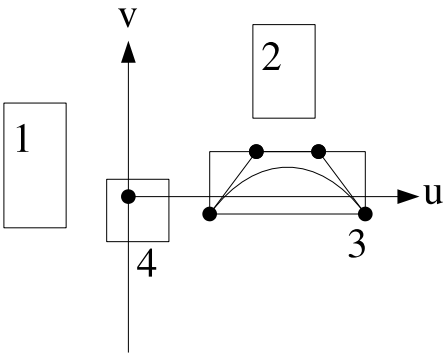

Figure 11: The four different cases for a single curve segment that can occur during the trimming test.

Since the test is arbitrary, we select the positive u-axis, which simplifies the implementation. Besides the reduction of the previous techniques to two dimensions, we only want to know whether the number of all intersections is even or odd. Transferring the principles of the previous section allows for an efficient trimming algorithm that achieves almost floating point precision.

We briefly sketch the algorithm that works for arbitrary degree along the lines of [19]. Figure 11 shows the four different cases the algorithm has to consider:

Cases 1 and 2: If all control points of the curve segment, i.e. the

Scene	# of Primitives	Parse Time	Memory	Render Time
Dog Triangles	422122	11.5	16.89MB	1.318s
Dog Bézier	24878	0.7s	6.47MB	3.283s
Car Triangles	1505118	252.4s	303MB	2573.4s
Car Bézier	261692	12.2s	29MB	812.6s

Figure 10: Performance of the Bézier subdivision algorithm compared to a triangle ray tracer. Given the superior precision of our algorithm, it is very competitive.

bounding box, lie left of the v-axis or above/below the u-axis, the ray does not intersect.

Case 3: The curve is on the right side of the v-axis but neither above nor below the u-axis. If the two endpoints of the curve are on the same side (above or below the u-axis), the ray intersects the curve an even number of times. Otherwise the number is odd. This is due to the continuity of polynomial curves.

Case 4: Now we have to recursively subdivide using the L_1 termination criterion. Upon termination we assume an intersection.

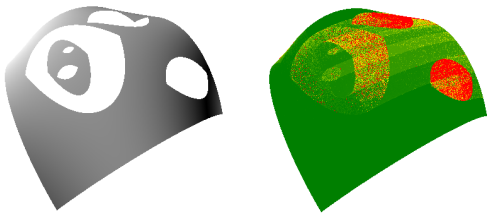

Figure 12: A trimmed Bézier patch. On the right the color encoded effort for trimming is shown. On average trimming is about 50% of the total intersection cost.

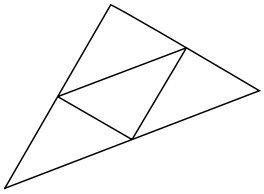

Figure 13: Quadrisection of a triangle using stable subdivision by edge midpoints.

3.3 Triangles

The principles developed so far can be used to avoid self-intersection problems (see section 2.2) and holes in triangular scenes. It is important to note that changing to double precision or different triangle tests does not avoid holes. Even the Plücker test, which avoids holes along edges, can fail around the vertices of the geometry.

Subdividing each triangle edge in the middle (see figure 13 and [24]) already fulfills the numerical requirements of section 2.1. In fact a triangle can be considered as a triangular planar Bézier patch and thus the results of section 3.1 apply.

Figure 14 shows the happy Buddha mesh. The image was ray traced at a resolution of 512x512 pixels using the triangle test of

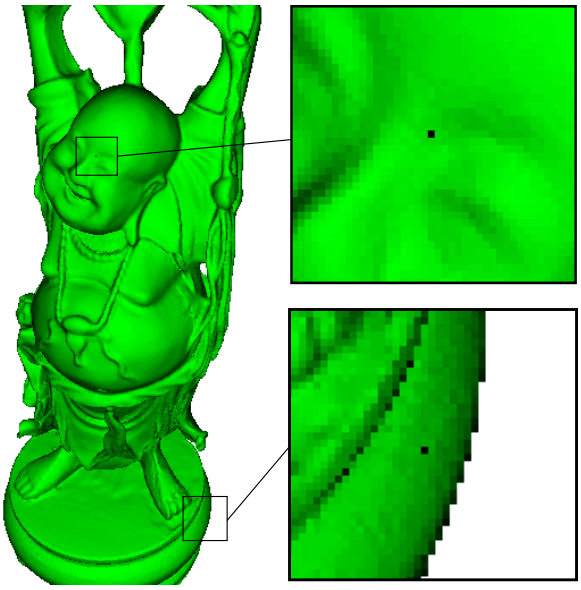

Figure 14: The happy buddha mesh with 1087716 triangles. When ray traced with standard triangle intersection algorithms false intersections can be reported. This problem does not occur with our subdivision algorithm.

Möller and Trumbore [17]. While 3.64 frames per second are obtained, some false intersections are reported along triangle edges. Using the stable algorithm avoids these errors and results in 0.73 frames per second.

If the intersection interval is not required to avoid self-intersections and only a hole free rendering is desirable, we can speed up computations by using our algorithm whenever a cheaper triangle test ([17] in our case) reports no intersection. Then the performance increases from 0.73 fps to 2.24 fps for the Buddha mesh.

4 Conclusion and Future Work

We have presented an algorithm that generally increases the precision of ray tracing without ad-hoc parameter adjustments. The results are especially interesting for precise simulation and production rendering.

Future work comprises finding a similar stable termination criterion that could be used for rational surfaces. First experiments have shown that ignoring the last few bits of the L_1-norm in the termination criterion produces good results. Until now there are only empirical results for the number of bits to ignore and they vary depending on the surface degree. Our algorithm can be used for many different kinds of subdivision surfaces. First implementations with Loop and Catmull-Clark surfaces have produced good results. Similar to the problems of rational surfaces, we still explore problems

with the subdivision accuracy at irregular vertices. Another problem is the performance when subdividing subdivision surfaces on the fly because of the many special cases that have to be considered such as boundary and irregular vertices.

ACKNOWLEDGEMENTS

We would like to thank Ingo Wald for the fruitful scientific discussion.

REFERENCES

[1] J. Amanatides and D. Mitchell. Some regularization problems in ray tracing. In *Proc. Graphics Interface '90*, pages 221–228, 1990.

[2] C. Benthin, I. Wald, and P. Slusallek. Interactive ray tracing of free-form surfaces. In *Proceedings of Afrigraph 2004*, pages 99–106, November 2004.

[3] S. Campagna and P. Slusallek. Improving Bézier Clipping and Chebyshev Boxing for Ray Tracing Parametric Surfaces, 1996.

[4] L. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications, Numerical Algorithms 37 1-4. 2003.

[5] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice, 2nd ed.* Addison Wesley, 1996.

[6] A. Fournier and J. Buchanan. Chebyshev polynomials for boxing and intersections of parametric curves and surfaces. *Computer Graphics Forum*, 13(3):127–142, 1994.

[7] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic Bézier surfaces without triangulation. In *WSCG 2005 Conference Proceedings*, Plzen, Czech Republic, 2005.

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

[9] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Praha, Czech Republic, April 2001.

[10] W. Heidrich and H. Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface*, pages 8–16, 1998.

[11] M. Herf. Robust epsilons in floating point, http://www.stereopsis.com/robusteps.html, 2000.

[12] H. Igehy. Tracing ray differentials. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 179–186, 1999.

[13] A. Junior, L. de Figueiredo, and M. Gattas. Interval methods for raycasting implicit surfaces with affine arithmetic, XII SIBGRAPHI, pages 1-7. 1999.

[14] J. Kajiya. Ray tracing parametric patches. *ACM SIGGRAPH Computer Graphics*, 16(3):245–254, July 1982.

[15] D. Lischinski and J. Gohczarowski. Improved techniques for ray tracing parametric surfaces. *The Visual Computer*, 6(3):134–152, 1990.

[16] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools*, 5(1):27–52, 2000.

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

[18] K. Müller, T. Techmann, and D. Fellner. Adaptive ray tracing of subdivision surfaces. *Computer Graphics Forum*, 22(3):553–562, 2003.

[19] T. Nishita, T. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics, ACM*, 4(24):337–345, 1990.

[20] L. Piegl and W. Tiller. *The NURBS Book*. Springer, 1997.

[21] H. Prautzsch, W. Boehm, and M. Paluszny. *Bézier and B-Spline Techniques*. Springer, 2002.

[22] S. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *ACM SIGGRAPH Computer Graphics*, 14(3):110–116, July 1980.

[23] D. Toth. On ray tracing parametric surfaces. *ACM SIGGRAPH Computer Graphics*, 19(3):171–179, 1985.

[24] D. Voorhies and D. Kirk. Ray - triangle intersection using binary recursive subdivision. In James Arvo, editor, *Graphics Gems II*, pages 257–263. Academic Press, 1991.

[25] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[26] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[27] A. Williams, S. Barrus, R. Morley, and P. Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*, 10(1):49–54, 2005.

[28] 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.

[29] C. Woodward. Ray tracing parametric surfaces by subdivision in viewing plane. *Theory and Practice of Geometric Modeling, Springer-Verlag New York, Inc.*, pages 273–287, 1989.