Diplomarbeit

Floating-Point Precision Ray Tracing of Free-Form Surfaces

Universität Ulm
Fakultät für Informatik
Abteilung Medieninformatik

Holger Dammertz, 2005

Betreuer/Gutachter: Prof. Dr. Alexander Keller, Universität Ulm
Zweitgutachter: PD Dr. Alfred Strey, Universität Ulm

Erklärung

Name: Holger Dammertz Matrikel-Nr.: 439510

Ich erkläre, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Ulm, den ....

Floating-Point Precision Ray Tracing of Freeform Surfaces

Holger Dammertz

June 15, 2005

Contents

Chapter 1

Introduction

1.1 Ray Tracing

Ray casting is an image generation method, where rays are shot from the camera through a pixel into a 3d scene to determine the closest point of intersection. Ray tracing [Whi80] extends this idea to trace multiple additional rays from the first point of intersection in order to calculate the visibility of light sources (to compute shadows and indirect illumination) and to simulate reflection and refraction. The result is that each color of a pixel is computed via tracing a tree of rays through the scene. This principle is shown in figure 1.1.

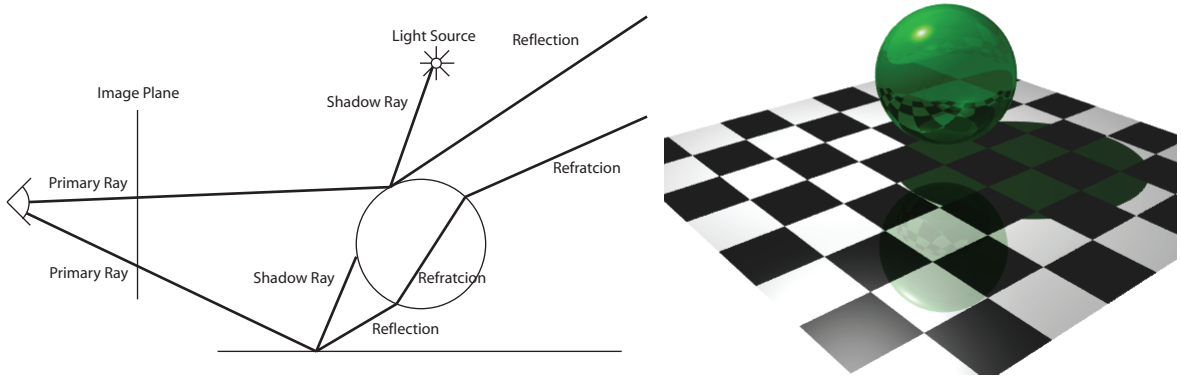


Figure 1.1: On the left two primary rays with some additional secondary rays are schematically traced. On the right an image is shown that was generated by tracing primary, reflection, refraction and shadow rays.

The method and history of ray tracing in general is well documented in Glassner's book "An Introduction to Ray tracing" [Gla89]. Tracing a ray through a 3d scene is also the basic operation needed for most realistic image synthesis algorithms as for example global illumination computations [DBB03].

1.2 Free Form Surfaces

Generally free form surfaces are higher order surfaces that are described by a set of data points and are differentiable up to a specified degree. There are many different types of free form surfaces and especially for modeling purposes a lot of representations were developed. The most common type are non uniform rational basis-spline (NURBS) surfaces (see for example [FvDFH96] for an introduction or [PT97] for a more complete algorithmic treatment). These use rational polynomials as basis functions and can guarantee continuity across patch boundaries. Figure 1.2 shows a car hull model that consists

of several NURBS surface patches. These surfaces are also called parametric surfaces because the basis polynomials lead to a natural parameterization of each single surface patch. Another name are tensor product surfaces. Most polynomial 2d surfaces in use are tensor product surfaces because they can be constructed by the tensor product of two 1d polynomial curves. In many applications (especially in the entertainment industry) the properties of rational basis functions are not needed and so often the models are just described by polynomials. The smoothness is needed to guarantee various properties of the surfaces. For a good visual appearence of reflections on a car the surface has to be at least C^2 continuous. These high quality surfaces are needed by many industrial applications. Especially in product design or the automobile industry so called class A surfaces are desired. Many applications in the CAD area are designed to handle these kind of surfaces. In the entertainment industry such surfaces are also widely used because they offer an easy way to model and animate complex objects. Another type of free

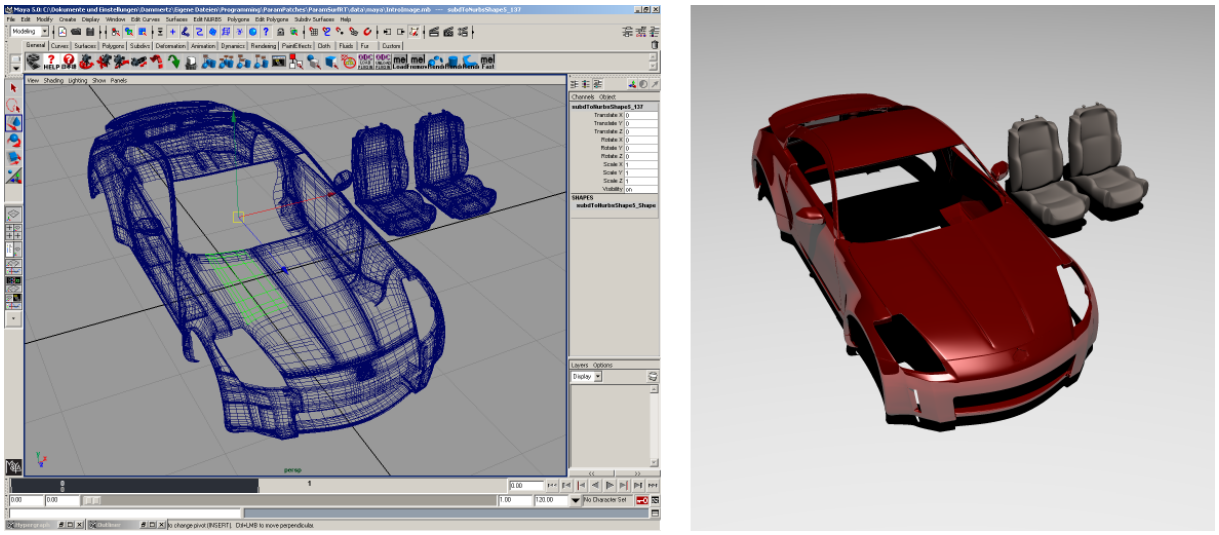


Figure 1.2: Part of a car model from Dosh design (www.doschdesign.com). On the left the NURBS model is shown in the modeling application Alias Maya. The right image shows the result of ray tracing this scene.

form surfaces that are recently used are subdivision surfaces. They describe a smooth surface by infinitly many times applying a subdivision algorithm to a control mesh. These subdivision methods produce often surfaces that are C^2 continous in the limit in almost all points. This fits the needs of the digital entertainment industry where they are often used.

1.3 Ray Tracing of Free Form Surfaces

As free form surfaces are widely used in ray tracing environments, a lot of approaches are found in the literature in order to solve the ray-surface intersection problem for different types of surfaces. They can be roughly classified into three different strategies. The first one converts (tessellates) the surface prior to ray tracing into another (simpler) primitive while discarding the original surface description completely. The primitive of choice is the triangle to linearly approximate the surface. Second there are subdivision based methods using a simple prune and search method. The convex hull property of most surfaces is exploited for this. The last approach comprises of numerical solutions, where a form of Newton iteration is used in the most cases to find the intersection point. Some algorithms can not be assigned to one of the three approaches, but in the following subsections they are described,where they fit best. As can be seen below many articles on subdivision based methods or numerical solutions concentrate on improving or combining algorithms. The main contributions are either more robust algorithms (in the sense of fewer false pixels and less parameters to adjust per scene) or faster ones by utilizing the coherence of primary rays.

1.3.1 Subdivision based Methods

The simplest subdivision based methods are described by Whitted [Whi80] and Rubin [RW80]. They use the convex hull property and perform a binary search through repeated subdivision and ray - bounding volume intersection. The search is stopped as soon as the bounding volumes reach a predefined minimum size.

Woodward [Woo89] transforms the problem to two dimensions. The patch is projected onto the plane that is orthogonal to the ray and the subdivision is done in 2d. For reasons of speed the patch is also translated and the control points are scaled to large integers.

Nishita et al. [NSK90] developed the well known technique of Bézier clipping which is an iterative geometric algorithm. The ray is described as the intersection of two orthogonal planes and the subdivision is performed in 2d by projecting the patch along the ray. The Bézier clipping algorithm now finds parameter regions that are guaranteed to not intersect the ray and prunes them. It has some nice properties like that it can be used for surfaces of arbitrary degree and for trimming curves alike. The problem with the original algorithm is that it can report false intersections. Swen Campagna and Phillipp Slusallek describe the problem in [CS] and show that it is inherent to this algorithm and not a numerical problem. As solution they add additional checks to the basic algorithm. In the same paper they also extend a method called Chebyshev Boxing [FB94] and combine it with Bézier clipping. Chebyshev Boxing is a fast method that uses Chebyshev polynomials to calculate bilinear approximations for the patch and a bounding volume hierarchy.

Müller et al. [MTF03] describe an adaptive method for ray tracing subdivision surfaces (with a convex hull property) without prior tessellation. They use the projection of the ray onto a surface patch and work in 2d. The patch is refined if it has a high curvature, is large or contributes to the silhouette of the object. So the refinement is controlled by user adjustable parameters. The final intersection is then computed using the triangles of the refined patch. Since neighboring subdivisions may be of different depth special gap triangles need to be inserted to avoid cracks. The algorithm is quite complex to implement but handles all features of subdivision surfaces.

A very straight forward and fast approach for ray tracing cubic non-rational Bézier and Loop subdivision surfaces is described in [BWS04]. There is a predefined number of refinement steps performed on the fly. At this predefined subdivision level the ray is then intersected with a triangular approximation of the surface. One problem of this approach (as with all other algorithms that have user definable parameters) is that the parameter has to be adjusted for each object in the scene separately. Another problem is that the same depth has to be used for the whole object because otherwise cracks between neighboring subdivisions will occur. This may also lead to visual artifacts in shadows and reflections as by all triangle based methods.

1.3.2 Numerical Solutions

One interesting method is described by Kajiya in [Kaj82]. He uses algebraic geometry (resultants) to create a numerical procedure for finding the intersection between a ray and a bivariate cubic parametric patch. This transforms the problem into finding the roots of univariate polynomial of degree 18. The main disadvantage besides the quite high amount of computations needed for this is that is only works for polynomial surfaces of degree three.

The direct formulation of the ray surface intersection problem (for surfaces of degree 3 or higher) leads to a nonlinear system of equations. A variety of numerical (i.e. iterative) solution methods can be used to solve them. The one used in most cases is Newton's method because it is very easy to implement and converges quickly with a good starting point. Finding this starting point (or detecting that Newton's algorithm does not converge) is the core of most hybrid algorithms that combine subdivision and Newton iteration. These algorithms use some way of subdivision to find a good starting point.

A more accurate approach is described by Toth [Tot85]. He uses multivariate Newton iteration to solve the convergence problem of Newton iteration. The method uses interval arithmetic to compute so called interval extensions for the surface. This provides a safe criterion for Newton to converge to the correct solution. If the criterion is not met, the surface is subdivided into smaller ones. The method allows to ray trace all surfaces for which the interval extension can be computed. But the implementation has some special cases that need to be checked for each iteration and the computation of the interval extensions is quite costly. Another problem is that the algorithm involves numerical critical operations

like matrix inversion which can cause problems in real applications.

Lischinski and Gonczarowski [LG90] describe various methods to improve the performance of Toth's method. A uniform space subdivision and surface tree caching, a hybrid between Toth's algorithm and subdivision, is used. The main goal is to exploit ray coherence to speed up the intersection calculation. They also use a non-scanline sampling order (in this case a Peano curve) for primary rays.

A method to directly ray trace (trimmed) NURBS surfaces is described by Martin et al. in [MCFS00]. Newton iteration is used to find an intersection. Because of the Newton iteration a starting point near the actual point of intersection is needed so that the iteration converges to the correct solution. For this a bounding volume hierarchy is created. The bounding volumes are generated directly for the NURBS surface via refinement using a heuristic to stop when the refined surface is flat enough. This hierarchy is then traversed first to get a good starting point. The main problem with this method is that the parameter for the flatness criteria needs to be adjusted by the user for each scene. Moreover the convergence may fail on flat angles or when objects are scaled.

Figure 1.3: Problem of wrong parameters for Newton iteration. Image taken from [MCFS00].

A combination of Newton iteration and Bézier clipping is described in [WSC01]. An obstruction detection technique is developed to verify the intersection point found. When the Newton method fails to converge Bézier clipping is used.

Geimer et. al. [GA05] perform a simple method similar to what is done by Martin et. al. [MCFS00]. They ray trace cubic Bézier patches by constructing a bounding volume hierarchy to find a good starting point for Newton iteration. With this method they do not have the problem of the triangulation as in the method from Benthin et. al. [BWS04] but they have scene dependant parameters that have to be adjusted by the user in order to produce the correct result.

1.3.3 Tessellation

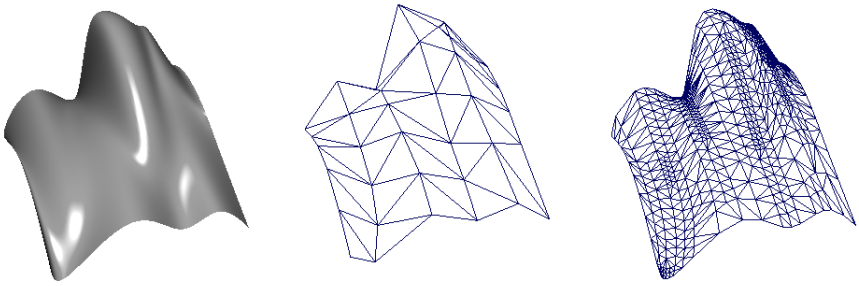


Figure 1.4: On the right two different tessellations of a surface are shown. The tessleation in the middle used a simple uniform technique and the one on the right makes use of a simple adaptive method.

In almost all cases tesselation means to linearly approximate the surface by triangles. Since modern graphics hardware can process triangles very fast and triangles are generally the primitive of choice for many ray tracers there exist a lot of different methods in the literature. Generally some kind of error metric and heuristics are used to generate more triangles in areas where they are needed and less in others.

These are for example high curvature regions or silhouettes that require a high tessellation for visually pleasing results. Some examples for the visual artifacts that can occur by wrong or insufficient tessellation are shown in the next chapter. The high tessellation required for good images is often a problem because the total amount of triangles is limited by either speed or more seriously by the memory. CAD models that have several hundreds of megabytes of NURBS data can not be tessellated into too much triangles on commodity hardware.

Chapter 2

Floating Point Precision Ray Tracing

One big problem of most algorithms described in the previous chapter is that the result depends on parameters that have to be adjusted by the user for each scene and model. The parameters are either needed to get correct results (when using Newton iteration) or to get visually pleasing results when using subdivision or tessellation. For Newton iteration there can always be constructed special cases where the result is wrong. This occurs for example if the camera gets close to silhouettes or objects are scaled. What makes the parameter settings also difficult is that at preview resolution the image looks good but at production resolution with anti aliasing some rays fail and generate wrong results. This is the main reason why these methods are not used in industrial ray tracers.

Another problem that is not directly apparent is that many of the algorithms perform transformations on the patch (for example projection in 2d). In almost all cases this leads to a loss of accuracy due to floating point arithmetic. This is generally of no concern in the entertainment industry but it may be important in CAD applications.

A-priori tessellation and on the fly subdivision methods both have similar problems. The linear approximation by triangles gets visible on silhouettes and badly in shadows and reflections. While the

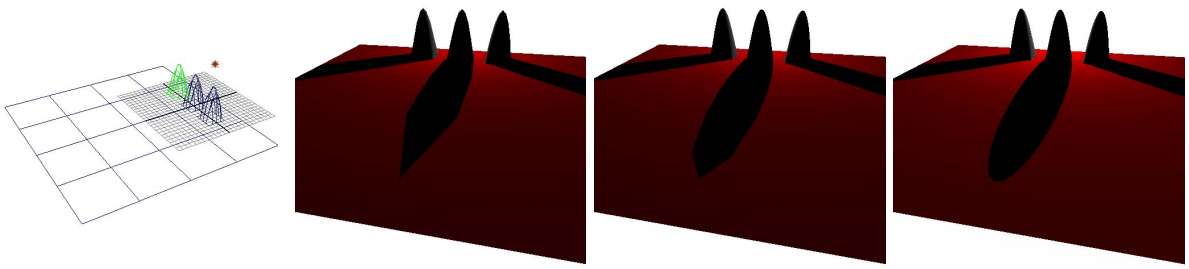


Figure 2.1: Silhouette problems with triangulations. The left image shows a test scene in Maya 5.0. The two images in the middle show the scene rendered with mental ray 3.2.2.1. The tessellation was set to *mid quality* (512 triangles) and *high quality* (2048 triangles). The right image was rendered directly without approximating the scene by triangles.

silhouette problem can be overcome by screen based tessellation this method fails mostly for shadows and reflections because it is much more difficult to detect critical regions. As told above most commercial ray tracers use a-priori tessellation because of rendering speed and due to the fact that the artist has control over arbitrary regions of the object. It is a lot of work to adjust the tessellation for each object to produce the desired results. This parameter tweaking is feasible for still images but as soon as animations are involved it is becoming even more difficult. Changing tessellation may be visible from frame to frame if the renderer does not automatically interpolate and so the tessellation parameters have to be adjusted for the worst case in the animation (like a close up on the silhouette or a long shadow). But fine tessellation of large objects leads to an incredible amount of memory needed to store all triangles.

The images in figure 2.1 and 2.2 show the problems of linear approximation in simple test scenes. The

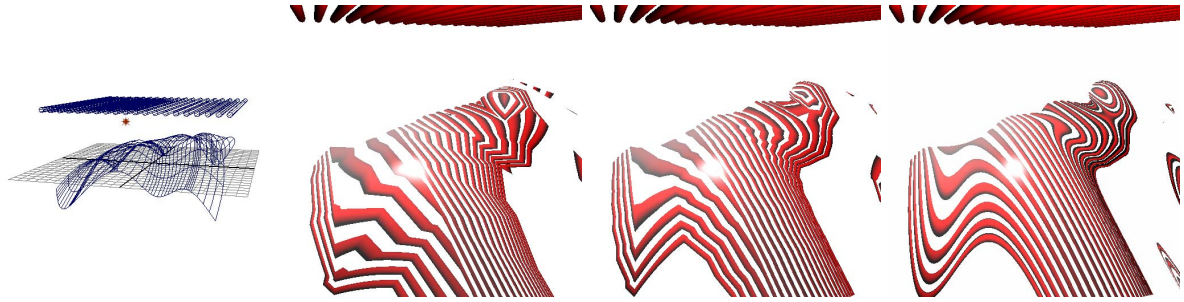


Figure 2.2: Problem of interpolating a normal across an approximated surface. The left image shows the Maya 5.0 test scene. The two images in the middle were rendered by mental ray 3.2.2.1 using the preset tessellation quality regular grid. The *mid quality* setting (center left) produced 2688 triangles and the *high quality* setting 10752 triangles for the test scene. The right image was rendered with out new scheme.

images in the middle were rendered with the global surface tessellation quality presets "Regular Grid Mid Quality" and "Regular Grid High Quality" for mental ray 3.2.2.1 in Maya 5.0. The right images were rendered using the algorithm developed below.

Using the principle algorithm developed in this section avoids some of the problems mentioned above. There are no scene-dependent parameters and the memory usage is low compared to prior tessellation. The resulting algorithm is very simple (much simpler then any of the ones described in the previous chapter), crack free and has no special cases. The intersection computations are done directly in world space with the original data. That means that no transformations are done to the ray or patch that could add inprecisions. In section 2.3 it will be shown that this algorithm can be implemented for cubic surfaces in integral Bézier representation with all those properties and that it will run quite fast on modern hardware. The next sections examine the handling of trimming curves, implementations for polynomial surfaces of arbitrary degree and rational surfaces (NURBS). In the last section of this chapter subdivision surfaces are considered.

The basic idea used for the algorithm is a brute force divide and conquer algorithm. The idea for using this for ray tracing is quite old and was used for example in [RW80] as a derivation for a scene data structure and in [Whi80]. Both methods apply a user definable parameter that defines the minimum size of the bounding volume. In theory we can go a step further. Given a surface S and a subdivision rule that allows to split the surface into multiple smaller surfaces S_i so that $S_i \subsetneq S$ and $\bigcup_i S_i = S$, we can apply this subdivision rule an infinite number of times and ultimately get an infinite number of points that define the surface. In an implementation only a finite number of subdivisions is needed due to the limited accuracy of the floating point representation of a point of intersection. But with this observation there is an independent criterion to terminate the subdivision: stop when the maximum accuracy is reached. Of course the term maximum accuracy is not fixed. It depends on the number formats and the algorithms used.

2.1 Floating Point Representation

Recent computer hardware supports at least a subset of the IEEE standard 754 for floating point numbers. Since the algorithm should compute at maximum accuracy the basic layout and numerical problems of floating point numbers are reviewed here. In most cases the following implementations use only *single precision* (32 bit) for reasons of speed. A 32 bit *single precision* floating point number consists of three

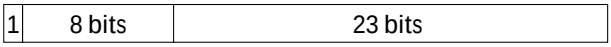


Figure 2.3: Memory layout of a 32 bit IEEE floating point number.

parts: A sign bit, an 8 bit exponent and a 23 bit mantissa. Besides the normal floating point numbers that

use a normalized mantissa (an implicit leading 1) there are some bit patterns reserved for special numbers. The reserved patterns are for zero, denormalized numbers, infinity and not a number (NaN). Goldberg [Gol91] describes the floating point standard in detail and shows the common problems of floating point arithmetic. In his article he covers error metrics and improved algorithms to avoid common floating point arithmetic errors like cancellation.

For the implementations of the ray tracing algorithm below only a few things are important. The first one is to avoid denormalized numbers because they slow down computations significantly when using SSE instructions on a Pentium 4. This is achieved easily by turning on the flush to zero mode (in fact this is generally the default mode when using the Intel compiler) and to avoid any input that contains denormalized numbers. This decreases the accuracy around the origin but since the algorithm works in world space this hardly affects the overall accuracy. Another important thing to note is that a division by 2 is exact if no underflow occurs. This division is just a decrease of the exponent by 1. Another important thing to note is that the floating point number can also be treated as bit string by loading it unchanged into an integer register. Now one can access the sign bit, the exponent and the mantissa independently by bit masking operations and interpret them as integer numbers. This is used for example in 2.7.2 to define a distance on floating point numbers that is independent of the exponent.

2.2 The General Algorithm

In order to compute the intersection between a ray and a surface patch the algorithm developed here uses two operations. The first one is the calculation of a bounding volume to enclose the surface patch as tight as possible. The second one is a numerically adequate subdivision rule to split a given surface patch into multiple smaller sub-patches of the same kind. The strategy of getting the intersection is now a classical divide and conquer algorithm. If the ray intersects the bounding volume but the termination criterion is not yet met the patch is subdivided again and the bounding volumes of the sub-patches are tested again. If the criterion is met a hit is reported and the bounding volume is returned as interval containing the point of intersection. Of course the criterion should be scene independent and in the best case is automatically determined without user interaction.

This intersection calculation can be formulated compactly in a recursive way:

```
IntersectRayPatch(Ray, Patch)
  bv = CalculateBoundingVolume for Patch
  if (Ray misses bv) return;
  else if (Termination criterion met) report bv as hit and return;
  else
    sd = subdivide Patch
    for all q in sd do IntersectRayPatch(Ray, q)
```

Algorithm 2.2.1 *Calculating the intersection of a ray and a freeform surface.*

The algorithm above reports all intersections with the surface. In many applications it is only required to search for the nearest intersection; this can be implemented for example in the report function. Later this is used to speed up the calculation. What is ignored in the outline above is how to keep track of the surface parameterization if there exists one.

The usage of only bounding volumes for the intersection calculation leads to one important and desirable property. There can not occure any cracks even if the depth of subdivision varies over the surface. This is for example a problem when the surface is approximated by triangles. Of course this is only true if the subdivision rule is numerically stable so that computed bounding volumes contact or overlap each other after subdivision.

Another characteristic of this algorithm is that no single point of intersection is returned when a hit was found. Instead the complete bounding volume is returned as the hit interval. When needing an actual point (for example as origin for secondary rays) it can be chosen on the surface of the bounding volume to guarantee that it lies outside or inside the object. For this reason the notion of a single point of intersection is dropped and the final bounding volume is returned. This is closely related to a special

type of computation called interval analysis [Sny92].

To visualize the principle of the algorithm figure 2.4 shows the Utah teapot which is ray traced using the above algorithm. Axis aligned bounding boxes are used as bounding volumes and for each image a different but fixed size is used as termination criterion instead of an automatic one.

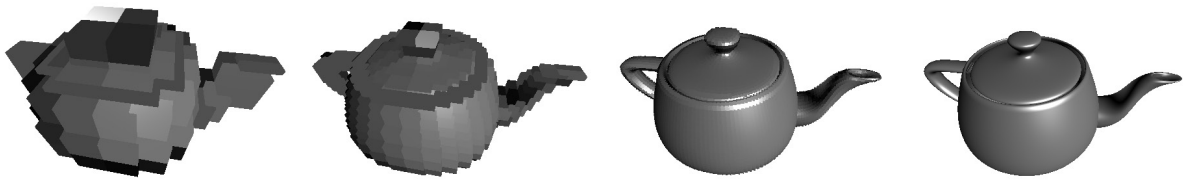


Figure 2.4: Visual result of the algorithm using different bounding box sizes as termination criterion

All images in figure 2.4 are rendered in a resolution of 512x512. The size of the bounding box used as termination criterion is decreasing from left to right.

Besides the mentioned lack of cracks due to subdivision the algorithm has some other nice properties. At first it operates directly on the surface and uses only two operations (subdivision and bounding volume calculation). These operations are available for many commonly used free form surfaces. Furthermore the original surface patch is not needed any more after subdivision and so the memory location can be overwritten. This turned out to be generally faster and saves memory during subdivision.

In the following chapters axis aligned bounding boxes are used as bounding volumes if not noted otherwise. They are often very easy to compute, need only six floats to store, can be exactly computed and the intersection calculation is quite fast.

The last thing to examine in algorithm 2.2.1 is the termination criterion. As derived in the introduction of this chapter a termination when the maximum accuracy is reached would be desirable. When using floating point arithmetic and only simple operations there is an inherent limit to accuracy. This can be used as a simple termination criterion if the subdivision algorithm is appropriate. If after one iteration the computed bounding volume is the same as before we must have reached the maximum accuracy that can be computed by this algorithm and number format. Of course there are a lot of computations performed with this method but in section 2.3.3 an optimized implementation shows that it can be fast enough for ray tracing a whole image. If the subdivision algorithm for the surface does not fulfill the numerical properties this simple termination criterion will of course fail and produce holes. A possible solution to this problem is developed in 2.7.2. For faster intersections other scene independent termination criteria (at least for primary rays) are also possible like a screen space based method (2.5.3).

2.3 Integral Bézier Surface Patches

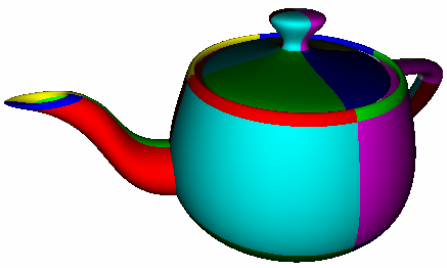


Figure 2.5: The Bézier patches of the teapot model.

Using the Bézier representation is one of the simplest ways to describe a polynomial surface. Even though it is not so well suited for modeling purposes any polynomial surface can be converted into the

Bézier basis. A model is generally composed of several polynomial surface patches that join with some continuity properties at the boundaries. Figure 2.5 shows the teapot with each Bézier patch colored.

A Bézier surface patch is defined by a two dimensional grid of control points p_{ij} where i is in the range of $0 \dots n$ and j in $0 \dots m$. There are $(n+1)(m+1)$ control points needed to define a patch

$$p(u,v) := \sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(u) B_j^n(v) p_{ij} \qquad u,v \in [0 \dots 1]. \tag{2.1}$$

$B_i^n(t)$ are the Bernstein basis functions of degree n. There are $n+1$ Bernstein basis functions

$$B_i^n(t) := \binom{n}{i} (1-t)^{n-i} t^i. \tag{2.2}$$

A Bézier surface patch with $n = 3$ and $m = 3$ is called a cubic patch (figure 2.6) and is defined by 16 control points.

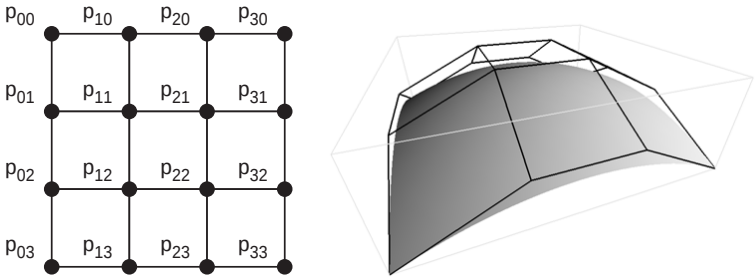


Figure 2.6: Cubic (degree 3) Bézier patch. On the left the layout of the 16 control points is shown. On the right the control grid (black), the axis aligned bounding box (gray) and the surface.

2.3.1 Determining the Convex Hull

The Bernstein basis functions (2.2) have the property of forming a partition of unity:

$$\sum_{i=0}^{n} B_i^n(t) = 1 \qquad \forall t \in \mathbb{R} \tag{2.3}$$

Equation 2.1 can be rewritten as

$$s(u,v) = \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n} B_j^n(v) p_{ij} \qquad u,v \in [0,1]. \tag{2.4}$$

A Bézier patch is a tensor product surface. The inner sum describes a Bézier curve with the parameter v. For a fixed parameter value v the outer sum describes a Bézier curve with the control points being calculated by the inner sum. Because of the Bernstein basis summing to one (2.3), each curve is a convex combination of its control polygon. So the whole surface is a convex combination of its control polygon and lies completely in its convex hull. This property makes it easy to compute an axis aligned bounding-box for a given patch:

```
aabb[0] = aabb[1] = patch.m_ControlPoints[0];
for (int i = 1; i < patch.m_NumControlPoints; i++)
{
  aabb[0].SetIfLess(patch.m_ControlPoints[i]);
  aabb[1].SetIfGreater(patch.m_ControlPoints[i]);
}
```

Code 2.3.1 *Axis aligned bounding-box calculation for Bézier patch.*

The axis aligned bounding box `aabb` is defined by two 3d points `aabb[0]` and `aabb[1]`. The first one contains the minimum of each coordinate and the second one the maximum. The two methods SetIfLess and SetIfGreater compare each coordinate separately and assign it only if the values are less than (or greater than) the value already stored. This are the equivalent to the SSE instructions `minps` and `maxps`.

2.3.2 Subdivision

The Bernstein basis functions (2.2) satisfy the recursion formula

$$B_i^{n+1}(t) = tB_{i-1}^n(t) + (1-t)B_i^n(t) \qquad \text{with } B_{-1}^n := B_{n+1}^n := 0 \text{ and } B_0^0 := 1 \tag{2.5}$$

A point on a Bézier curve

$$c(t) = \sum_{i=0}^{n} b_i^0 B_i^n(t)$$

can be calculated by the de Casteljau algorithm using the recursion property 2.5 of the Bernstein basis.

$$c(t) = \sum_{i=0}^{n} b_i^0 B_i^n(t) = \sum_{i=0}^{n-1} b_i^1 B_i^{n-1}(t) = \cdots = \sum_{i=0}^{0} b_i^n B_i^0(t) = b_0^n \tag{2.6}$$

where b_i^0 are the original control points of the patch and

$$b_i^{k+1} := (1-t)b_i^k + tb_{i+1}^k$$

are the intermediate "control points" of the de Casteljau algorithm. These points can be written in the

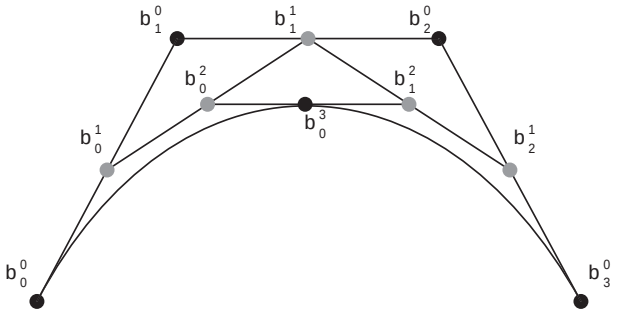


Figure 2.7: Graphical application of the de Casteljau algorithm to a cubic Bézier curve at a parameter value of 0.5. Each intermediate control point is created in the middle of the line segment. The last control point b_0^3 lies on the curve.

following way:

$$\begin{array}{ccccc} b_0^0 & & & & \\ b_1^0 & b_0^1 & & & \\ b_2^0 & b_1^1 & b_0^2 & & \\ \vdots & & & \ddots & \\ b_n^0 & b_{n-1}^1 & b_{n-2}^2 & \cdots & b_0^n \end{array}$$

where each intermediate control point is a weighted sum of two previous points. This is illustrated graphically in figure 2.7 for a parameter of $t = 0.5$.

The important property of the de Casteljau algorithm for subdivision is that the intermediate control points on the border of the above evaluation scheme describe a Bézier curve themselves. On the interval $[0, 1]$ the points $b_0^0, b_0^1, b_0^2, \ldots, b_0^n$ define a curve that is equivalent to the original curve on the interval $[0, t]$. The same holds true for the points $b_0^n, b_1^{n-1}, b_2^{n-2}, \ldots, b_n^0$ that define a curve equivalent to $c(t)$ for

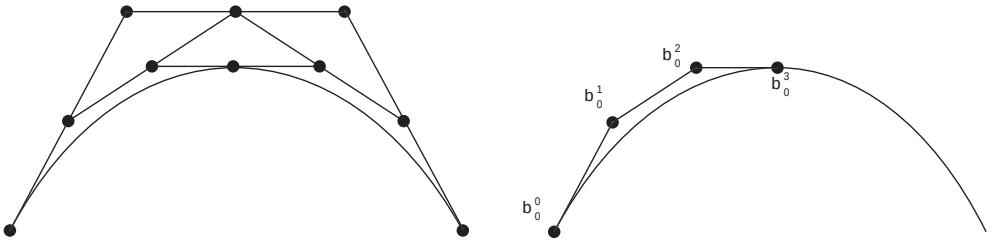


Figure 2.8: Subdivision at $t = 0.5$ using de Casteljau's algorithm. On the right side the control points for the left part of the curve are shown.

$t \in [t, 1]$. For the cubic case this is shown in figure 2.8. The de Casteljau algorithm can be used to subdivide a surface patch into two smaller ones. The subdivision can be done either in u or in v direction in parameter space. In the first case we exchange the summations in equation 2.1 to get

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

Now we can apply the subdivision described above to the inner sum for each j to split the surface in u-direction at a parameter value t to get $s_1(u, v_1)$ and $s_2(u, v_2)$. s_1 and s_2 together form the original surface. The best parameter value for subdivision in this case is 0.5. Even though the subdivision can be done at any parameter value $u, v \in (0, 1)$ it is not trivial to find a good point in parameter space and for the implementation using floating point arithmetic 0.5 turns out to be the numerically most stable.

One possible way to use the subdivision in algorithm 2.2.1 is to subdivide the patch at first in u-direction at $u = 0.5$ and the resulting two patches in v-direction at $v = 0.5$ to get four new smaller patches. But for implementation purposes generating only two patches per iteration and using a simple heuristic to decide in which direction to subdivide is a lot faster. The heuristic used in the implementation below tries to guess the parameter in which the patch is larger by comparing the length of the two vectors $(b_{n,0} - b_{0,0})$ and $(b_{0,m} - b_{0,0})$ and subdivides along the larger direction. Another way to decide in what parameter direction to split is examined in section 2.5.2.

2.3.3 Implementation

In this section a complete implementation for computing the point of intersection between a ray and an integral cubic Bézier patch is described. The code shown here is optimized for the Intel Pentium 4 processor architecture.

Data Structures

Vector3d: The `class Vector3d` is used for points and vectors alike. It should be clear from the context or the name what a variable is used for. The class has the usual operators like addition and scalar multiplication overloaded.

```
class __declspec(align(16)) Vector3d
{
  public:
    float x,y,z,w;
};
```

Code 2.3.2 *Vector class*
Because of the P4 architecture the `Vector3d` has a fourth component called `w` that is ignored during normal 3d operations. It makes sure that each element of a `Vector3d` array is 16 byte aligned and will be used later for rational Bézier patches.

Ray: A ray is stored as a origin in 3d space and a direction.

```
class Ray
{
  public:
    Vector3d m_Origin;
    Vector3d m_Direction;

    /** additional information for speedup */
    Vector3d m_InvDir;
    int m_Sign[3];
};
```

Code 2.3.3 *Data structure of a ray*

For a faster intersection with an axis aligned bounding box additional information is precomputed once a ray is generated. The member variable `m_InvDir` stores the inverse of each component of the direction. `m_Sign` stores the sign of each component of the direction. It is used later to index the correct side of the bounding box.

Bézier patch: The data structure for a cubic Bézier patch is quite simple. Only the grid of control points is stored. The indexing of the points is given by

$$\begin{array}{cccc} 12 & 13 & 14 & 15 \\ 8 & 9 & 10 & 11 \\ 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 \end{array} \qquad \begin{array}{c} v \\ \uparrow \\ \end{array} \quad \begin{array}{c} \\ \\ \rightarrow \end{array} \quad \begin{array}{c} \\ \\ u \end{array}$$

```
class __declspec(align(16)) BezierPatch4x4
{
  public:
    Vector3d m_ControlPoints[16];
}
```

Code 2.3.4 *Data structure of a cubic Bézier patch*

Axis Aligned Bounding Box Computation

Algorithm 2.3.1 is used for the computation of the bounding box. Since this operation is performed very often in the intersection calculation it is crucial to run fast. Code 2.3.5 shows one possible implementation.

```
void BezierPatch4x4::CalculateRoughAABBox(Vector3d* f_pBBox)
{
  __m128 f_pBBox0=_mm_load_ps((float*)&m_ControlPoints[0]);
  __m128 f_pBBox1=_mm_load_ps((float*)&m_ControlPoints[0]);

  for (int i = 1; i < 16; i++)
  {
    f_pBBox0 = _mm_min_ps(f_pBBox0, _mm_load_ps((float*)&m_ControlPoints[i]));
    f_pBBox1 = _mm_max_ps(f_pBBox1, _mm_load_ps((float*)&m_ControlPoints[i]));
  }

  _mm_store_ps((float*)&f_pBBox[0],f_pBBox0);
  _mm_store_ps((float*)&f_pBBox[1],f_pBBox1);
}
```

Code 2.3.5 *Axis aligned bounding box computation for a cubic Bézier patch*

The search for the minimum and maximum is performed in the xmm registers. In the last two lines the result is stored in the memory region for the bounding box. Taking a closer look it can be seen that $\frac{1}{4}$ of the floating point comparisons done are wasted because the SSE instructions operate on 4 floats. But using the additional slot would mean to sort out the results at the end which turned out to be slower. Another advantage of this formulation is that it can be used without a change for rational cubic Bézier patches that are described in section 2.7. The method is called "rough" because the result is just determined by the control points of the patch. As can be seen from the control points in figure 2.8 a much closer bounding box could be determined if the patch is subdivided once.

Subdivision in Two Smaller Patches

The subdivision described in section 2.3.2 can be implemented in a straight forward way. The only decision to make is where to store the resulting patches. Since the intersection calculation algorithm 2.2.1 does not need the old patch once it is split, the subdivision is performed in place. That means one of the new patches overwrites the memory of the original patch.

```
void BezierPatch4x4::SubdividePatchInPlace_VDir(BezierPatch4x4& f_BottomPatch)
{
  (__m128&)f_BottomPatch.m_ControlPoints[12] = (__m128&)m_ControlPoints[12];
  (__m128&)f_BottomPatch.m_ControlPoints[13] = (__m128&)m_ControlPoints[13];
  (__m128&)f_BottomPatch.m_ControlPoints[14] = (__m128&)m_ControlPoints[14];
  (__m128&)f_BottomPatch.m_ControlPoints[15] = (__m128&)m_ControlPoints[15];
  static const __m128 f05=_mm_set1_ps(0.5f);

  // using simple de Casteljau at v = 0.5 for each column of the patch
  for (int i = 0; i < 4; i++)
  {
    (__m128&)f_BottomPatch.m_ControlPoints[8+i] = _mm_mul_ps(
             _mm_add_ps((__m128&)m_ControlPoints[8+i], (__m128&)m_ControlPoints[12+i]), f05);
    register __m128 tmp = _mm_mul_ps(
             _mm_add_ps((__m128&)m_ControlPoints[4+i], (__m128&)m_ControlPoints[8+i]), f05);

    (__m128&)f_BottomPatch.m_ControlPoints[4+i] = _mm_mul_ps(
             _mm_add_ps(tmp, (__m128&)f_BottomPatch.m_ControlPoints[8+i]), f05);

    (__m128&)m_ControlPoints[4+i] = _mm_mul_ps(
             _mm_add_ps((__m128&)m_ControlPoints[0+i],(__m128&)m_ControlPoints[4+i]),f05);
    (__m128&)m_ControlPoints[8+i] = _mm_mul_ps(
             _mm_add_ps((__m128&)m_ControlPoints[4+i],tmp),f05);
    (__m128&)f_BottomPatch.m_ControlPoints[0+i] = (__m128&)m_ControlPoints[12+i] = _mm_mul_ps(
             _mm_add_ps((__m128&)m_ControlPoints[8+i],
                        (__m128&)f_BottomPatch.m_ControlPoints[4+i]),f05);
  }
}
```

Code 2.3.6 *Subdivision of a Bézier patch in v-direction*

Each column of the control points describes a Bézier curve that is split into two halves by the above algorithm. Even though the code looks quite messy due to the typecasts for the SSE intrinsics it is just an application of what is described in section 2.3.2. The code for subdividing in u-direction is analogue but with rows and columns exchanged.

Intersection of a Ray with an Axis Aligned Bounding Box

The numerical stability of the intersection test is an important part of the algorithm because the boxes tend to get very small. Williams et al. describe in [WBMS03] an efficient check that uses the properties of the IEEE floating point standard to get an efficient and stable algorithm that handles all special cases and provides all the functionality needed for the ray-patch intersection calculation.

The implementation uses the precomputed values stored in the `Ray` to directly index at the correct side of the bounding box and to avoid expensive divisions.

```
bool intersectRayBox(Vector3d p[2], Ray& ray)
{
        float tymin, tymax, tzmin, tzmax;

        bb_t0 = (p[ray.m_Sign[0]].x - ray.m_Origin.x) * ray.m_InvDir.x;
        bb_t1 = (p[1-ray.m_Sign[0]].x - ray.m_Origin.x) * ray.m_InvDir.x;
        tymin = (p[ray.m_Sign[1]].y - ray.m_Origin.y) * ray.m_InvDir.y;
        tymax = (p[1-ray.m_Sign[1]].y - ray.m_Origin.y) * ray.m_InvDir.y;
        if ( (bb_t0 > tymax) || (tymin > bb_t1) )
                return false;
        if (tymin > bb_t0)
                bb_t0 = tymin;
        if (tymax < bb_t1)
                bb_t1 = tymax;
        tzmin = (p[ray.m_Sign[2]].z - ray.m_Origin.z) * ray.m_InvDir.z;
        tzmax = (p[1-ray.m_Sign[2]].z - ray.m_Origin.z) * ray.m_InvDir.z;
        if ( (bb_t0 > tzmax) || (tzmin > bb_t1) )
                return false;
        if (tzmin > bb_t0)
                bb_t0 = tzmin;
        if (tzmax < bb_t1)
                bb_t1 = tzmax;

        return ( (bb_t0 < t1) && (bb_t1 > t0) );
}
```

Code 2.3.7 *Ray - axis aligned bounding box intersection*

The variables `bb_t0` and `bb_t1` are global variables to store the distance of the entry and exit point of the ray into the bounding box. The global variables `t0` and `t1` store the range of the ray. Even though the code has a lot of if-statements it is faster than a streamlined version in SSE without any if-statement. The reason for this is mainly that the early jumps out of the function save more cycles in the average (here) than a pipeline flush costs. But this may change with other processor architectures.

Ray - Patch Intersection

The most interesting aspects of the implementation are shown in this section. At first the complete code listing (code 2.3.8) of an implementation of algorithm 2.2.1 is shown. Then the most important parts are explained.

```
void intersectRayPatchFloatAcc()
{
  int top = 0;
  float dOld = FLT_MAX;

  paramStack[top].x = paramStack[top].z = 0.0f;
  paramStack[top].y = paramStack[top].w = 1.0f;
  dStack[top] = FLT_MAX;

  do
  {
    patchStack[top].CalculateRoughBBox(g_TempBBox);
    float d = (g_TempBBox[1] - g_TempBBox[0]).SumXYZ();

    if (d >= dStack[top])  // an interseciton was found
    {
      g_u = (paramStack[top].y + paramStack[top].x) * 0.5f;
      g_v = (paramStack[top].w + paramStack[top].z) * 0.5f;

      t1 = bb_t0;
    }
    else if (intersectrayBox(g_TempBBox)) // the patch is not small enough yet
    {
      float lu = (patchStack[top].m_Data[3] - patchStack[top].m_Data[0]).GetLength2();
      float lv = (patchStack[top].m_Data[12] - patchStack[top].m_Data[0]).GetLength2();
      if (lu > lv)
      {
        paramStack[top+1].y = paramStack[top].y;
        paramStack[top].y = paramStack[top].x + (paramStack[top].y-paramStack[top].x)*0.5f;
        patchStack[top].CreateLeftRightSDPatchInPlace(patchStack[top+1]);
        paramStack[top+1].x = paramStack[top].y; paramStack[top+1].z = paramStack[top].z;
        paramStack[top+1].w = paramStack[top].w;
      }
      else
      {
        paramStack[top+1].w = paramStack[top].w;
        paramStack[top].w = paramStack[top].z + (paramStack[top].w-paramStack[top].z)*0.5f;
        patchStack[top].CreateTopBottomSDPatchInPlace(patchStack[top+1]);
        paramStack[top+1].x = paramStack[top].x; paramStack[top+1].y = paramStack[top].y;
        paramStack[top+1].z = paramStack[top].w;
      }
      dStack[top] = dStack[top+1] = d;
      top+=2;
    }
    top--;
  } while (top >= 0);
}
```

Code 2.3.8 *Intersection of a ray with a cubic Bézier patch*

There are three stacks that are used instead of a recursive implementation. The `patchStack` is a stack that contains `BezierPatch4x4` elements and stores all the patches that need to be checked against the ray. The `paramStack` consists of `Vector3d` and is used to remember the interval where the current patch was split. The vector stores in `x` and `y` the parameter interval for the u direction and in `z` and `w` the v direction. If a final intersection is found the middle of the parameter interval is used for the point of intersection. The last stack (`dStack`) which stands for diameter stack is used for the termination criterion.

The function works in the following way; the caller puts the patch that should be intersected on top

of the stack (at position 0). The parameter stack is initialized with the full range $[0, 1]$ and the diameter stack contains the maximum float value as first element. Then the main loop starts which terminates either if the patch is not hit or a point of intersection has been found. At first the axis aligned bounding box of the patch is calculated and stored in `g_TempBox`. After this the diameter of this box is calculated using the L_1 norm. The next check with the `dStack` is only relevant after the first iteration (at the beginning it is `FLT_MAX`). If the bounding box has been hit by the ray, the patch is split into two sub-patches. The decision in which direction to split is made heuristically as explained at the end of section 2.3.2. The values `lu` and `lv` are a guess of the length of the patch in each parameter direction. After the decision in which parameter direction the split should occur the parameter values are updated and the two new patches are placed on the stack (with the first patch overwriting the old one on the stack). The diameter of the old bounding box is finally stored on the `dStack`.

The termination criterion works in the following way: If the patch was split into two smaller patches the new diameter at the beginning of the loop is smaller and so the patch is subdivided again if its bounding box was hit by the ray. But if we have reached the maximum of numerical accuracy we want to terminate. The subdivision uses only convex combinations. If two floating point numbers are close together the convex combination (with a parameter of 0.5) results in one of the initial values. So if the diameter of the bounding box after one subdivision is the same as before we can safely terminate because without much work we can not get a better approximation of the point of intersection. Using this method we have a safe and scene/scale independent criterion to terminate the intersection calculation without the requirement to prior guess the number of iterations for a predetermined accuracy.

2.3.4 Analysis of the Algorithm

Effort for Intersection Calculation

The first interesting thing about the algorithm is the number of subdivisions needed until an intersection point is found. Figure 2.9 shows a single Bézier patch on the left and a color coded image on the right that contains the number of subdivisions performed per point of intersection.

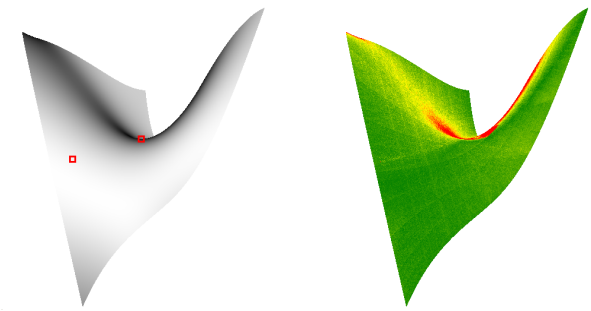


Figure 2.9: Number of subdivisions needed to calculate the point of intersection. Green is about 50 subdivisions, yellow up to 200 and Red more then 200 subdivisions to calculate the point of intersection. In the left image the position of the two rays that were used to create figure 2.10 and figure 2.11.

As can be seen in the image, the effort needed to determine the point of intersection grows the closer the ray direction approaches an angle of 90 degree with the surface normal. The reason for this is due to the increased number of bounding boxes that are hit when the ray passes parallel to the surface.

Closely related to this is the speed of convergence. The control polygon of a Bézier surface converges quadratically to the surface when the de Casteljau subdivision is used [PBP02]. So one would expect a quick convergence of the bounding box of the control polygon too. To analyze this the size of the bounding box is measured by the L_1 norm of the diagonal. This approach is the same as used above for the termination criterion. Two assumptions can be made to get a theoretical optimum for the size of the bounding boxes over the number of generated bounding boxes. The first one is that each bounding box is split exactly in the middle. This is only approximated in the algorithm when using a parameter value of 0.5. The second assumption is that the algorithm always considers the bounding box first in which

the final point of intersection actually lies. With this assumptions, the curve for the optimum can be described with

$$d(n) = s\left(\frac{1}{2}\right)^{\frac{n}{2}} \tag{2.7}$$

where n is the number of bounding boxes and s is the size of the first bounding box. The $\frac{n}{2}$ in the exponent is because in each iteration the patch is divided either in u or in v direction. So the diameter of the box halves roughly after 2 subdivisions (one in u and one in v).

In the following figures the size of the bounding boxes that were pushed onto the stack are shown over the number of generated bounding boxes. The violet curve in each image is the graph of equation 2.7. The data was created with the test scene shown above in figure 2.9

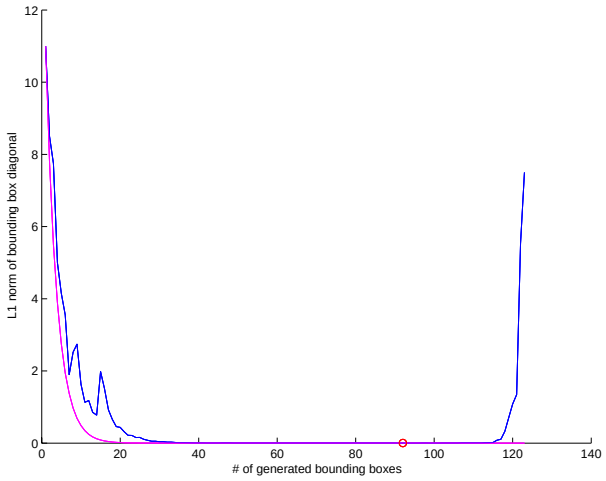


Figure 2.10: Single ray with 62 subdivisions.

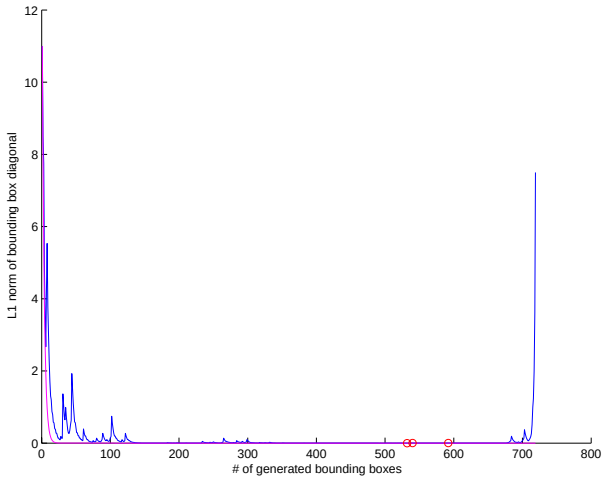


Figure 2.11: Single ray with 360 subdivisions.

Figure 2.10 shows a single ray that needed 62 subdivisions (124 bounding boxes) to determine the point of intersection. The bounding box that was finally used as interval for the point of intersection is marked with the red circle at position 92. The peaks at the beginning of the curve are the results of visiting the false bounding box first. The peak at the end is due to checking the bounding boxes that were placed at the beginning on the stack and have not yet been visited.

The graph in figure 2.11 shows a ray that has a much worse behavior than the previous one. The ray was taken out of the red region in figure 2.9. In the graph can be seen that the wrong path followed very often. There are two boxes small enough to be considered a possible point of intersection (marked as red circles) but are farther away than the actual point of intersection. Depending on the structure of the patch that has to be ray traced the behavior of the subdivision can be even worse for a few rays (for example if there are self intersecting patches). But since these cases are very rare when real scenes are ray traced (see chapter 3) and since the algorithm always produces a correct result the average case is more relevant.

Figure 2.12 shows the average size of the bounding box for the 5396 rays that hit the surface in figure 2.9. Even though there are a few rays (51) that required more than 200 iterations the graph was cut off at 200 iterations (400 bounding boxes) for display purposes. The fewest iterations needed to find a point of intersection were 49 in this scene. The most were 707 and the average was 86.22 iterations.

Accuracy of the Normal and the Point of Intersection

In order to get information about the accuracy of the result and to compare it to the accuracy of triangulation a reference image is needed. To calculate the reference image the complete algorithm above has been implemented using double precision arithmetic. The results are then rounded to the nearest 32bit float precision number which leads to the best possible floating point representation of the point of

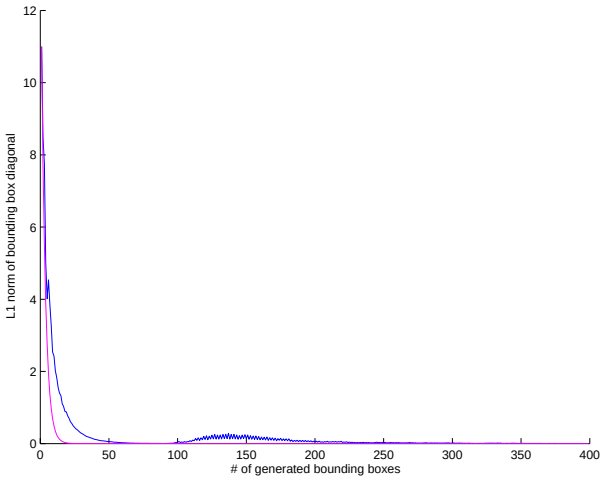


Figure 2.12: Average size of bounding box for 5396 rays.

intersection. For comparison with triangles the test patches are tessellated uniformly into three different fixed amounts of triangles. In the analysis below the calculated point of intersection and the normal are compared to the double precision result. Of course the code 2.3.8 does not produce a single point of intersection like a normal triangle ray tracer but a 3d interval (the final bounding box). The middle of the returned interval is used as point of intersection for comparison.

Rays that hit the patch in double precision but miss the patch completely using single precision or triangles (also in single precision) are marked red in the image but are ignored in the generation of the statistics. Since the point of intersection and the normal are a three dimensional quantity the comparison is done using the L_1 norm on the difference of the corresponding vectors.

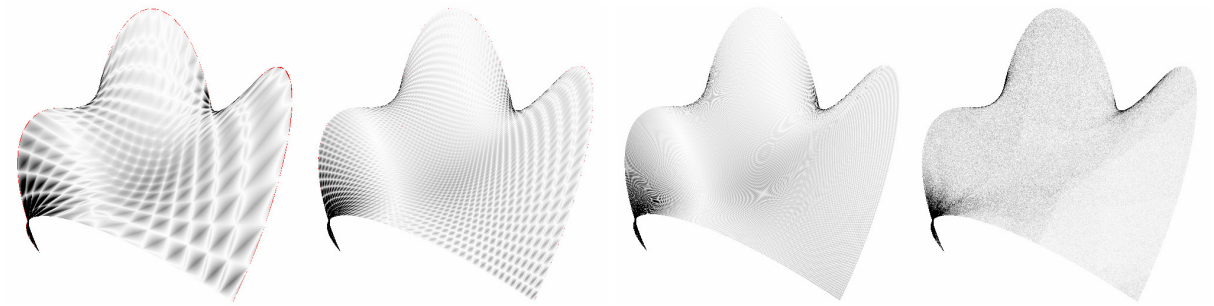


Figure 2.13: Visualization of the error (also see table 2.1) of the point of intersection compared to the double precision version for a single Bézier patch.

All images in figure 2.13 and figure 2.14 are generated at a resolution of 512x512. From left to right the patches use 512 triangles, 4608 triangles, 131072 triangles and the rightmost is calculated with the 32 bit floating point precision subdivision algorithm. The error is clamped using the 3σ rule and is scaled in each error image to use the full range from black (high error) to white (no error). That means error values larger than $\overline{x} + 3\sigma$ are set to black where $\overline{x}$ is the mean error and σ the standard deviation. The scaling is needed because the mean error changes from image to image by a factor of about 100. For absolute values of the error see the tables below. The aliasing being visible in the third image is due to the regular distribution of the error when using uniform triangle tessellation. The values in the tables below are also generated at a resolution of 512x512. The row with zero triangles is computed with the floating point precision subdivision algorithm.

# Triangles	Max. Error	Min. Error	Mean Error	Variance
512	$1.808000e+000$	$5.960464e-008$	$9.575743e-003$	$3.294521e-004$
4608	$4.056229e-001$	$0.000000e+000$	$1.100530e-003$	$7.464949e-006$
131072	$1.110046e-002$	$0.000000e+000$	$3.853264e-005$	$8.651681e-009$
0	$9.324029e-005$	$0.000000e+000$	$2.295893e-007$	$3.624202e-013$

Table 2.1: Single precision point of intersection accuracy compared to the double precision calculation.

Max error is the maximum difference encountered among all points of intersection and *min error* the minimal difference. Table 2.1 shows the values that are generated with the patch seen in figure 2.13. As can be seen in the table the average error reduces with increased tessellation but only slowly compared to the increase of triangles. Table 2.2 shows the same comparison done for the surface normals. Normals are very important for the computation of reflection, refraction and lighting properties of a surface. The normals for the triangles are computed using the standard method of interpolating the vertex normals. The vertex normals are computed during tessellation using the cross product of the derivatives of the original patch at the parameter position of the vertex. The normals for the floating point precision subdivision algorithm are computed by the cross product of the derivatives at the parameter location found for the point of intersection.

# Triangles	Max. Error	Min. Error	Mean Error	Variance
512	$1.018858e+000$	$1.757592e-005$	$2.124975e-002$	$8.998101e-004$
4608	$9.822437e-001$	$5.908310e-006$	$2.384546e-003$	$6.719863e-005$
131072	$2.336000e-002$	$3.594905e-007$	$8.145076e-005$	$1.311810e-008$
0	$2.231598e-004$	$0.000000e+000$	$7.541509e-007$	$2.601197e-012$

Table 2.2: Comparison of normal accuracy.

The error in the normals usually features the same behavior as the error of the point of intersection. An error image for the normals looks similar to figure 2.13. Of course triangles approximate a flat patch better than a highly distorted one. Table 2.3 and figure 2.14 show the errors when ray tracing a less distorted patch.

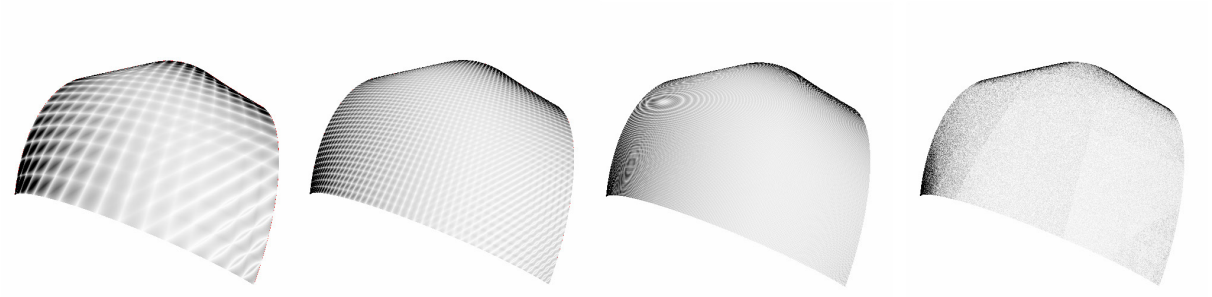


Figure 2.14: Visualization of the error (table 2.3) with a less distorted patch.

# Triangles	Max. Error	Min. Error	Mean Error	Variance
512	$3.064177e-001$	$4.470348e-008$	$4.981469e-003$	$4.928906e-005$
4608	$8.522448e-002$	$8.475035e-008$	$5.570586e-004$	$8.131825e-007$
131072	$2.795994e-003$	$0.000000e+000$	$1.959216e-005$	$1.022489e-009$
0	$8.568168e-005$	$0.000000e+000$	$2.303760e-007$	$3.908901e-013$

Table 2.3: Point of intersection error for a less distorted patch.

If the patch that is ray traced is completely flat and all control points on an edge are on one line triangles are an exact representation of it. What is left is only the error of the ray plane intersection and a small error introduced when computing the tessellation triangles. If the flat patch is orthogonal to one of the coordinate axis the subdivision algorithm generates at least in this coordinate direction the correct result because the point of intersection is computed directly on the object. One would expect that the subdivision algorithm would output worse results compared to triangles for a flat patch that is not orthogonal to one of the coordinate axis. Table 2.4 shows that this is true but the error is only slightly worse compared to the triangulation and still of the same order of magnitude than the accuracy of the previous patches.

# Triangles	Max. Error	Min. Error	Mean Error	Variance
512	$3.099442e-006$	$0.000000e+000$	$2.827907e-007$	$7.407670e-014$
4608	$4.291534e-006$	$0.000000e+000$	$4.181265e-007$	$1.344111e-013$
131072	$3.099442e-006$	$0.000000e+000$	$3.181449e-007$	$8.416109e-014$
0	$2.861023e-006$	$0.000000e+000$	$3.372990e-007$	$6.654892e-014$

Table 2.4: Point of intersection error when using a completely flat patch that is not parallel to one of the coordinate axis.

The error analysis in this section showed that the floating point precision algorithm produces results that are several orders of magnitude better than the approximation by triangles even if many triangles are used. Especially for shadows and reflections this improved accuracy is directly visible in the resulting images. This was shown for example in figure 2.1 and figure 2.2 at the beginning of this chapter. Further examples can be found in section 3.4 where larger scenes are compared.

One important thing to note is that the subdivision algorithm also allows to compute an interval for the normals by simultaneously subdividing the Bézier representation of the derivative polynomial. When a final intersection interval for the surface has been found the second interval contains the range of the normals in this region. Besides improved error bounds this can be used for example by another adaptive subdivision criterion or to adjust the number of secondary rays.

2.4 Secondary Rays

As discussed in the first chapter light sources and some surface properties can be handled by tracing secondary rays. For the calculation of reflection and refraction the ray has to start from the point of intersection on the surface. This leads to the numerical problem of self intersection [JA90]. A point of intersection on the surfaces lies below due to floating point imprecisions. The left image in figure 2.15 shows this problem for shadow rays. Some intersection points of the flat surface lie below and the shadow rays intersect this surface. A common solution is to add a fixed epsilon to the origin of the ray but this is not a very convenient solution, because the size of this epsilon is scene dependent. Using a value too small does not remove all artifacts and using a value too large leads to artifacts for example in regions where an object touches a reflecting surface. Both problems can be seen in figure 2.15. A trick used in triangle ray tracers is to exclude the point of intersection when the same triangle is hit again. This can cause only problems across borders. But this does not work for free form surfaces because they are not flat and an intersection with the same patch may be valid.

The solution for this problem used here is a combination of a meaningful selection of the starting point for the ray cast from a surface and a robust floating point epsilon that automatically adapts to the precision needed.

2.4.1 Selection of the Ray Origin

Even though the computation of the point of intersection collapses sometimes into one single point this is not always the case. So the notion of a point of intersection is dropped and the ray tracer returns an axis aligned bounding box that is called intersection box (intersection interval). This box is guaranteed to enclose the surface and the point of intersection. When a secondary ray is shot there are at least 8

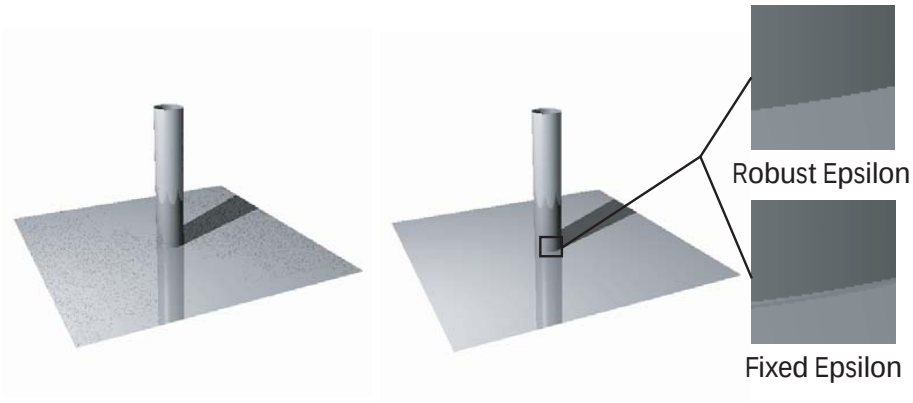


Figure 2.15: The right image shows a scene rendered with shadows and reflection. The left image shows the result without special handling of the point of intersection. The two small images on the right show a closeup of the reflection. When using a fixed epsilon a reflection may start to far away as can be seen in the bottom close up image.

well defined points (from which some may be the same) to choose from. The point that is used is selected through the normal on the surface at the parameter value for this hit box. Depending on the octant in which the normal is, the corner of the bounding box is selected. With this method it is for example possible to select a starting point that lies above the surface for a reflection ray and below the surface for a refrection ray. Figure 2.16 shows this principle at an example in 2d.

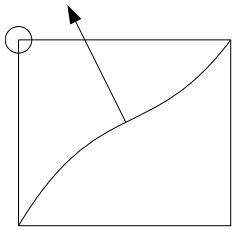


Figure 2.16: 2d example for selecting the ray origin.

2.4.2 Robust Floating Point Epsilon

When the hit box collapses in one or more dimensions the selection described above is not enough to avoid self intersections. As a solution every hit box is enlarged by an epsilon in every dimension. To get an epsilon that is not scene dependent but only on the precision of the hit box we can do the following: looking at the IEEE floating point format the number is stored as an exponent and a mantissa. If the mantissa is looked at as an integer ([Her00]) we can increase it by a small constant amount and get an epsilon whose true value depends on the exponent of the floating point number. The value used in the implementation is 7. One simple way of implementing this in C++ is the use of a union: Figure 2.17 and figure 2.18 show two scenes that were rendered with the parametric surface ray tracer described in the next chapter.

```
#define EPSILON 7

typedef union
{
  float f;
  unsigned int i;
} FltInt;

FltInt value;        // some float value
...
value.i += EPSILON;  // apply epsilon to value
```

Code 2.4.1 *Implementing robust floating-point epsilon*

Figure 2.17: Show room scene. The car model is taken from Dosch Design (www.doschdesign.com). The scene contains 9 light sourcees and was rendered at 4640x1920 with 13 rays per pixel.

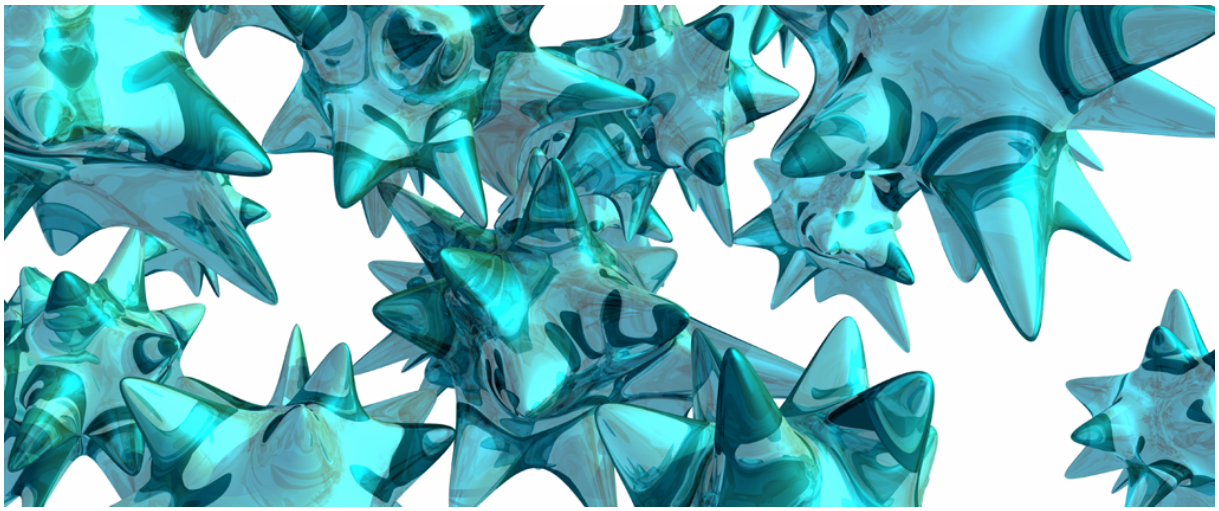


Figure 2.18: Another scene that was rendered at 4640x1920 by the parametric surface ray tracer with 13 rays per pixel.

2.5 Optimizations

As the functions for subdivision, bounding box calculation and intersection are called very often for each pixel and patch even minor changes can have a huge impact on the performance of this algorithm. Depending on the used compiler is has been for example useful to unroll the loops over the control points in the bounding box calculation and the subdivision by hand.

An optimization concerning the algorithm would be to use a heuristic to guess the side from which a patch is hit and follow this side first in the subdivision. The reason for this is that if a point of intersection is found in the near part, the ray is already shorter and the far part can be discarded sooner. But the implementation of this has to be done very careful because simple usage would mean to copy the data on the stack (for exchanging near and far sub-patch) and that turned out to be too slow. Using an indexed stack leads to non neighbored data in memory which is also discussed in 2.5.1. One solution would be to have 2 different methods for each subdivision direction that stores the result already in the right order in memory.

2.5.1 Ray Coherence

One of the first things that come to mind when looking at the subdivision routines above is that the work done is discarded after each intersection test and has to be done again for the next ray even if it hits the same patch. But retaining the data generated during subdivision has some drawbacks that hinder the performance. At first the data has to be arranged in a tree structure instead of a linear stack which means the data is not locally in the memory but may be scattered. The second more serious problem is that the calculation as formulated in section 2.3.3 is done mainly in the level 1 cache of the processor because the patch is subdivided in place. So the computation is compute bounded. When starting to save and restore older sub-patches the memory access latency becomes the bottleneck. Another thing to consider is, that there are a lot of subdivisions when going down to floating point accuracy so even two similar rays do not share many subdivision levels.

A way of avoiding to store the intermediate patches but to use the coherence of neighboring rays is to shoot a bundle of rays simultaneously ([Wal04]. The performance gain here depends on the size of the patches. If they are large so that almost all rays in the bundle hit the same patch it performs best. One way of implementing this for arbitrary ray bundles is to store all rays of the bundle in an array. During the subdivision a mask array is stored additionally on the stack that signalizes if a ray is active in the current subdivision level or not. At the beginning (stack level 0) all rays are active and they

are deactivated when they miss the bounding box of the current subdivision level. This is done with a loop over all active rays. A simple (non optimized) implementation of this is shown in code 2.5.1. When popping elements from the stack rays that were deactivated for this branch may be active on this subdivision level. So an additional branch in the subdivision is followed.

```
hit = false; // to check if at least one ray hits the bounding box
for (int i = 0; i < NUM_RAYS_IN_BUNDLE; i++)
{
  if (rayBundelMask[top][i])  // ray is active
  {
    // deactivate ray if it misses the bounding box
    if ((!intersectRayBox(bundle[i].ray, g_TempBBox)))
      rayBundelMask[top][i] = false;
    else // if any of the rays hits the bounding box we have to subdivide further
      hit = true;
  }
}

if (hit)
{
  if (termination criterion)
  {
    set current bounding box as hit interval for  all active rays
    and pop the current patch from the stack
  }
  else
  {
    subdivide in two sub-patches and push them onto the stack
  }
}
else
{
  no ray of the bundle has hit the bounding box of the current patch so
  pop it from the stack
}
```

Code 2.5.1 *Deactivation of rays in a ray bundle if they miss the bounding box of the current subdivision level*

This method is the most efficient when the scene consists only of one large patch. To get a rough idea of the maximum performance gain when using ray bundles a simple but a little distorted patch is used that can be seen in figure 2.19.

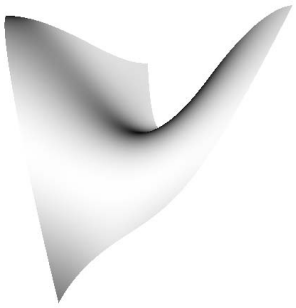


Figure 2.19: Test patch for ray bundles

#Rays in Bundle	Float Acc.	Fixed 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 2.5: Performance when using ray bundles of different size.

Of course the problem described at the beginning that even coherent rays share only few subdivision levels when subdividing to floating point precision is still present. For comparison a fixed termination criterion is also used. The fixed criterion was chosen so that the resulting image shows no visual difference to the exacter one but a significantly less depth subdivision is used. All performance measurements in table 2.5 were done at a resolution of 512x512 pixels with one ray per pixel.

As can be seen in table 2.5 the performance gain for floating point accuracy is at most 20% for 4x2 rays in the bundle. This gets worse when there are more and smaller patches in the scene. So when this approach is applied to a real scene, the overhead of the bundle management is much larger then the performance gain. If a smaller subdivision depth is used the performance gain is nearly 90% for bundles of 4x4 rays in the test scene. The reason for this is mainly the increased coherence of the rays (the ratio of shared bounding boxes is higher). So when using other termination criteria like the one described in section 2.5.3 it is more useful but when the patches get smaller and there are thounds of objects the overhead turned out to be too high.

2.5.2 Other Heuristics for the Subdivision Direction

In 2.3.2 the heuristic to decide in which parameter direction the subdivision should occur tries to shorten the longer side of the bounding box. Since subdivision is the most performance critical part, optimizations to discard rays which do not hit the patch as early as possible should have a major performance impact. One idea to get a better heuristic is illustrated in figure 2.20. If the bounding box is subdivided in the direction that is more parallel to the ray it is likely that a large part can be discarded without further subdivision. Of course the heuristic has to assure that even if the ray is orthogonal to one side of the

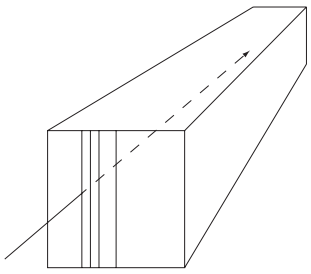


Figure 2.20: A better way to subdivide the bounding box when the ray is nearly parallel to one side.

bounding box the other direction is also subdivided because otherwise no point of intersection can be found.

Since there is no correlation between the orientation of the patch and its bounding box the direction has to be chosen with data of the patch. To get a rough guess of a direction vector for the u and v

direction the difference $du = (b_{n,0} - b_{0,0})$ and $dv = (b_{0,m} - b_{0,0})$ is used again. On the one hand we want to subdivide in v-direction if the ray is more parallel to du and in u-direction otherwise. On the otherhand we have to assure that if one direction becomes to short the other direction is subdivided. One way to compute a heuristic with this propperties is by using the length of the cross product:

$$\begin{aligned} lu &= |du \times \text{ray.m\_Direction}|^2 \\ lv &= |dv \times \text{ray.m\_Direction}|^2 \end{aligned} \tag{2.8}$$

The cross product has the property that the resulting vector is short when the angle between the two vectors is small. But the length of the result is also small when the incoming vectors are short. So if $lu > lv$ we subdivide in u-direction otherwise in v-direction.

Figure 2.21 shows the comparison between the old (lenght based) and the new heuristic (cross product). The result is that there is only a small difference in the effort and point of intersection when using the cross product heuristic.

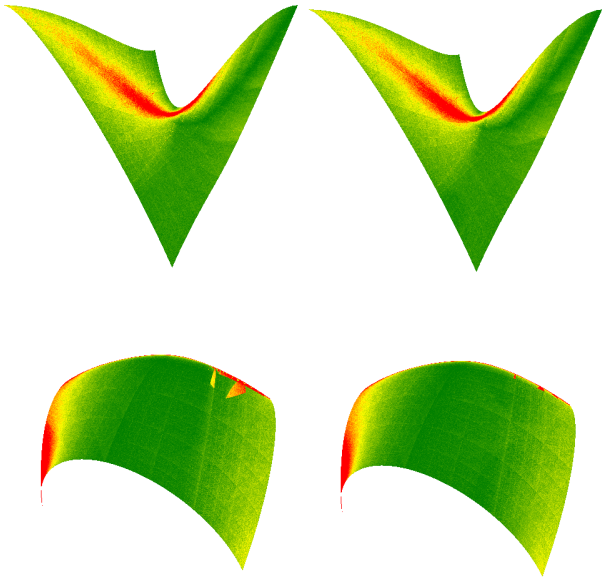


Figure 2.21: Comparing the effort of the cross product heuristic (left) with the length heuristic (right). There are only small differences. Table 2.6 list the data resulting from the comparison.

Scene	Method	min effort	max effort	avgEffort	fps
top	length	47	525	104.53	0.313
top	crossp	47	499	103.76	0.305
bottom	length	48	4028	85.94	0.460
bottom	crossp	46	3686	87.44	0.435

Table 2.6: Measurements for the comparison of the length based heuristic and the cross product heuristic. The associated images can be seen in figure The scene name top refers to the more distorted patch.2.21.

The cross product heuristic is slightly slower then the length based heuristic in most cases. The reason for this is mainly due to the computations needed by the cross product. But there are some camera positions, where it is a little bit faster. Table 2.7 shows an accuracy comparision of the two methods. As seen there is no significant difference in the accuracy.

# Method	Max. Error	Min. Error	Mean Error	Variance
length	$4.875660e-005$	$0.000000e+000$	$4.975422e-007$	$8.452438e-013$
crossp	$4.875660e-005$	$0.000000e+000$	$5.083150e-007$	$9.324388e-013$

Table 2.7: Accuracy comparison for the bottom scene of figure 2.21

2.5.3 Adaptive Subdivision

Another way of speeding up the intersection calculation for primary rays is to use the size of the bounding box as a termination criterion. Similar to the REYES architecture [CCC87] the subdivision is stopped when the size of the bounding box projected onto the image plane is smaller than half a pixel which corresponds to the Nyquist limit of an image. This simple projection does not work for secondary rays. Because calculating the exact area of the projected bounding box for each iteration would be too costly it is better to determine an upper bound for the projected size. This can be done using the theorem of intersecting lines.

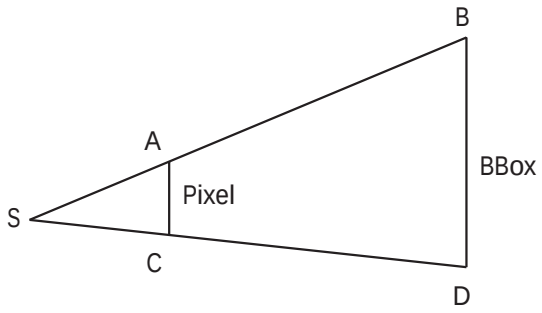


Figure 2.22: Using the intersection theorem as termination criterion

With $a = \overline{SA}, b = \overline{SB}, c = \overline{AC}, d = \overline{BD}$ the theorem says that $\frac{a}{b} = \frac{c}{d}$. To use this as a termination criterion a is the distance of the eye point to the pixel and b the length of the ray that hits the bounding box. If d is the size of the bounding box and c the half size of the pixel we have to subdivide until $\frac{ad}{b} \leq c$. Of course this adaptive subdivision is also crack-free because the bounding boxes touch each other even if neighboring boxes are subdivided to a different depth. The final point of intersection may be enclosed with a larger parameter interval. The center of this interval is used for the computation of a single normal. Figure 2.23 shows the effort of the adaptive method on the right in comparison to the float accuracy method. Table 2.9 shows an accuracy comparison as done in 2.3.4 of the adaptive

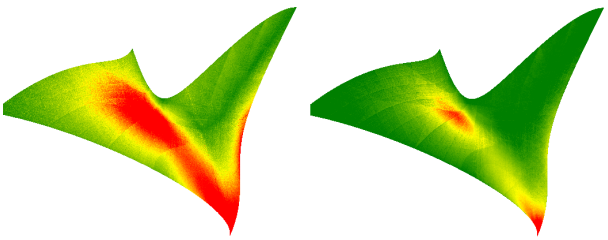


Figure 2.23: The left image shows the effort for the float accuracy termination criterion and the right the effort for the adaptive method. The numerical evidence can be found in table 2.8.

method with triangles and the float criterion. The results show that the adaptive method for one patch is slightly better than tesselating this patch to 4608 triangles but since the hit interval is much larger, there is always at least a small error. When using the adaptive termination criterion the resulting images look identical except for the boundaries where the patch may be slightly thicker.

A property of this termination criterion that may be an advantage but also a disadvantage is that

Term. Crit.	min effort	max effort	avgEffort	fps
fltAcc	48	2915	144.06	0.306
adaptive	30	1304	79.26	0.486

Table 2.8: Comparison of the effort for the adaptive termination criterion with the float accuracy criterion.2.23.

# Method	Max. Error	Min. Error	Mean Error	Variance
$512 Tris$	$3.599518e+000$	$1.117587e-008$	$1.623814e-002$	$4.583233e-003$
$4608 Tris$	$1.075897e+000$	$0.000000e+000$	$1.678691e-003$	$6.519672e-005$
$131072 Tris$	$3.934562e-003$	$0.000000e+000$	$5.665203e-005$	$5.116020e-009$
adaptive	$1.037640e+000$	$1.251698e-006$	$2.765996e-004$	$2.369071e-005$
Flt.Acc.	$4.875660e-005$	$0.000000e+000$	$5.083150e-007$	$9.324388e-013$

Table 2.9: Accuracy comparison of the adaptive method with the float accuracy method and simple triangulations.

far away patches are approximated with only a few boxes. The parameter value associated to the point of intersection is always the same if the rays hit the same box. For textured patches for example this significantly reduces the visible aliasing while moving the camera through the scene. But when using super-sampling this property is generally not desired. One solution would be to use the sub-pixel size as termination criterion. Another possibility is to use the sample position (that should be in $[0, 1)$) also in the parameter interval found for the hit box. With this modified parameter the normal is calculated.

2.6 Trimming Curves

Trimming curves are an important part of models described by tensor product surfaces (like Bézier or NURBS surfaces). They are needed to overcome the topological restrictions of a rectangular parameter region and are used to cut away an arbitrary part of a surface patch. In the following implementation closed cubic polynomial curves in Bézier representation are used to define a trimmed region in parameter space. The algorithm has to check whether an intersection point with the surface is inside or outside of the trimmed region. To calculate this the Jordan curve theorem is used which basically says:

> Any continuous simple closed curve in the plane separates the plane into two disjoint regions, the inside and the outside.

To check if a point is inside or outside of a closed curve a ray is shot from the point and the number of intersections with the curve is counted. If the number is even, the point is outside else it is inside. Since each curve consists of several Bézier segments the ray has to be checked against each of them.

For this calculation nearly the same algorithm as used for ray surface intersection can be applied. So all the nice properties are retained like exact shadows and no need for external parameters. The only change besides the reduction to 2d is to find all intersections instead of the nearest one. But because we only want to know, whether the number of intersections is odd or even, and the ray shot from the point is arbitrary, some simplifications can be made [NSK90].

Figure 2.24 shows the four different cases the algorithm has to consider. As the ray can be chosen arbitrary the positive u-axis is selected. Since we want to count the number of intersections the cases 1 and 2 are the easiest. In case 3 we can use a property of continous curves to directly calculate if the number of intersections is odd or even. In case 4 we have to subdivide until either we get case 1 to 3 or the maximum subdivision depth is reached. To get at the end the result wheter the number of intersections is even or odd there is a boolean variable `oddIntersections` that is inverted each time an odd number of intersections is encountered (case 3 and 4).

Case 1 and 2 If all control points of the curve segment (i.e. the bounding box) are left of the v-axis or above/below the u axis the ray does not intersect.

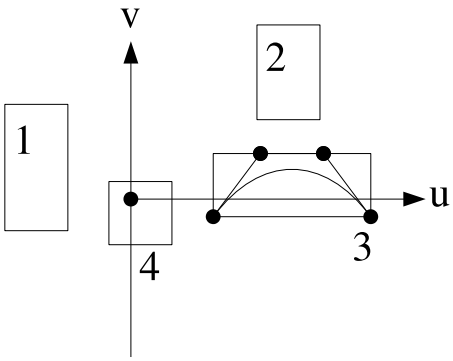


Figure 2.24: The four different cases for a single curve segment that can occure during the trimming test.

Case 3 The curve is on the right side of the v-axis but neither above nor below the u-axis. Now an intersection is possible and we are interrested in the number of intersections. But the calculation if the number is odd or even is simple. 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 (in the cubic case either zero or two intersections). If they are on different sides, we have an odd number of intersections and the variable `oddIntersections` is inverted.

Case 4 Here the curve is split via the de Casteljau algorithm into two smaller curves and they are again checked for case one to three. If the maximum subdivision depth is reached and the remaining curve is still case 4 one single intersection is assumed. The maximum subdivision depth is again automatically determined by using the L_1 norm of the bounding box diameter. If it does not change the subdivision is stopped.

Using this algorithm the parameter value of the intersection can be checked if it is inside or outside the trimmed region. If the curve should trim away an outer part of the surface the intersection counter has just to be initialized with 1 and the inside/outside relation is exchanged.

In many cases the algorithm terminates quite quickly because the parameter values checked are not too close to the curves and so case 1 to 3 are reached very fast. Other values that are near a trimming curve require a calculation to the maximum accuracy. This calculation effort can be quite high. In the implementation the ray was checked against all segments of the trimming curves. Figure 2.25 shows the distribution of the effort to decide whether the parameter of the point of intersection is in the trimmed region or not. As can be seen in table 2.10 the trimming curves cost about 50% of performance for a single patch in this two cases when computing every parameter value to the maximum accuracy.

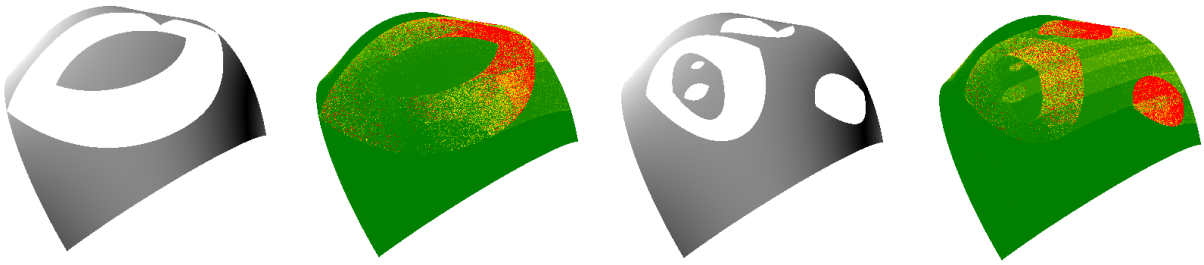


Figure 2.25: Effort for trimming a surface patch (See also table 2.10). The trim in the left image had 4 curve segments and the trim in the right image 12 curve segments.

The implementation above is very simple without any optimizations using the structure of the trimmed regions. A more common method of storing a trim-region is a tree structure [MCFS00]. This needs the additional restriction of non-selfintersecting curves which is not needed by the algorithm above. The tree

# of Segments	min effort	max effort	avgEffort	fps
4	6	142110	404.46	0.294
12	47	46080	439.74	0.293

Table 2.10: Effort for the trimming in figure 2.25. The speed of ray tracing the same patch without trimming was 0.578 fps. All measurements were done at a resolution of 512x512.

structure is organized as shown in figure 2.26 where a trim curve references all curves that lie inside as children. This also restricts the curves to not intersect. This structure should result in a speed up if

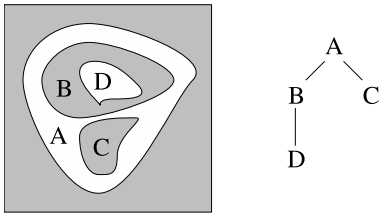


Figure 2.26: Storing the trimming curves in a tree structure

there are many complex trim hierachies in the scene. But in many cases trimmin curves are used to fit a patch to another one and so only one curve is needed.

The computations in the algorithm above are done in the interval $[0, 1]$. In this interval floating point numbers have the highest resolution. If this high accuracy is not needed a simple way of reducing the accuracy and increasing the speed is scaling and shifting the parameter region. Scaling with 2^{-n} and shifting by $1 - 2^{-n}$ leads to a computation in the interval $[1 - 2^{-n}, 1]$. In table 2.11 is an example for the increase in speed when this method is used.

n	min effort	max effort	avgEffort	fps
0	12	46080	439.74	0.293
1	12	47268	413.73	0.303
2	12	40960	400.61	0.308
3	12	39316	383.81	0.313
4	12	38540	367.36	0.319
8	12	34590	307.96	0.342

Table 2.11: Reducing the accuracy by shifting and scaling the parameter region. The measurements were done for the patch with 12 trim segments.

2.7 Rational Bézier Patches and Arbitrary Degree

As described in section 2.3 Bézier patches can be of arbitrary degree in each parameter direction. Higher degrees are needed for example in the industrial design of cars or planes. In this area rational patches are also used because they can represent conic sections exactly. A rational Bézier surface patch is defined as

$$s(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v)w_{ij}p_{ij}}{\sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v)w_{ij}} \qquad u,v \in [0,1] \tag{2.9}$$

where w_{ij} are weights assigned to each control point. Intuitively the value of the weight controls how close a curve is to the corresponding control point. Large values create curves that are close to the control points. The division in this equation leads to numerical problems in the float precision subdivision algorithm. These imprecisions lead to holes in the resulting image if no adjustments to the algorithm are made. A possible solution is discussed in section 2.7.2.

2.7.1 Handling arbitrary Degree Patches

The only things that have to be adjusted are the calculation of the bounding box and the subdivision of the patch to handle arbitrary degree patches. In the bounding box code (2.3.5) the loop has to consider all $(n+1)(m+1)$ control points instead of the 16 of cubic patches. The subdivision code (2.3.6) has to be changed to implement an arbitrary degree version of the de Casteljau algorithm described in section 2.3.2.

Figure 2.27: One Bézier patch with degree 10 in u direction and 7 in v.

Of course using a for loop that has a variable number of iterations slows down the computation significantly. If the application domain of the ray tracer allows to make a general statement about the degree of the patches it is much faster to provide extra optimized code for the most frequently used degrees. Only exceptions are then handled by the general code. Since the decision which algorithm to use can be done before the intersection calculation there is only a very small overhead compared to the amount of calculation done for the intersection.

2.7.2 Ray Tracing Algorithm for Rational Bézier Patches

A rational Bézier patch (equation 2.9) is an integral Bézier patch in homogenous 4d space [Far91]. The 4d control points are then $p_{ij}^w = (w_{ij}x_{ij}, w_{ij}y_{ij}, w_{ij}z_{ij}, w_{ij})$. Inserting these points in equation 2.1 results in equation 2.9 after homogenization. So the subdivision can be done with the same code as before for cubic patches (see code 2.3.6). Nothing has to be changed because the SSE instructions already operate on four values.

The intersection computation of the ray with the bounding box in homogenous 4d space seems to be no easy task. So the points have to be projected back (homogenized) into 3d instead. The homogenization of the control points has to be done only when the bounding box is calculated. This leads to a problem in combination with the previously used termination criterion because the division needed for the homogenization reduces the accuracy of the bounding box and this leads to holes in the surface when subdividing too far. The holes do not occur when using for example the adaptive subdivision of section 2.5.3. But this works only for primary rays. A slightly better approach is to use a "floating point adaptive" method similar to the robust epsilon of section 2.4.2 that is independent of the scale of the scene. In theory the criterion would be: stop just before the next subdivision makes the bounding box so small that the homogenization introduces so much error that cracks occur. This would fulfill the premise of computing to the maximum accuracy with a given algorithm and number format. In practice the following approach was developed.

The previous termination criterion based on the L_1 norm of the bounding box diameter is still used to guarantee termination. Additionally a distance that is based only on the mantissa of the bounding box coordinates `bbox_min` and `bbox_max` is used. In the general implementation each coordinate has to be considered separately. A basic implementation for two single floats is shown in code 2.7.1.

```
bool areTwoFloatsClose(Float low, Float high, int dist)
{
  if ((low < 0.0f) && (high > 0.0f)) return false;
  if (low.GetExponent() != high.GetExponent()) return false;
  return (abs(low.GetMantissa() - high.GetMantissa()) < dist);
}
```

Code 2.7.1 *Checking if two floats are close together*

In the first line of this method it is checked whether the two floats have the same sign. If not they are assumed to not be close together. Ignoring the numbers that are close but have different sign is not a problem as will be discussed below. For the same reason it is not a problem to ignore numbers that are close but have a different exponent. The last line now returns true if the absolute difference of the two mantissa (that is an integer) is smaller than a predefined value `dist`. To apply this method to a vector it is applied to every coordinate and returns true only if all coordinates are close. This is now used as additional termination criterion in the recursive subdivision. If the two vectors `bbox_min` and `bbox_max` are close enough together the subdivision is stopped and a hit box returned. The termination criterion now looks like

```
if (areTwoVectorsClose(g_TempBBox[0], g_TempBBox[1], dist) || (d >= dStack[top]))
```

The predefined number `dist` represents basically the accuracy of the subdivision algorithm and is of course the critical point. If it is chosen too small there may be cracks and choosing it too large reduces the accuracy of the result more then necessary. It can be considered as the number of bits that should be ignored. If `dist` is for example 512 the last 9 bits in the mantissa of the two floats are ignored in the comparison. Up to now there are only experimental results for the value of `dist` but it might be possible to derive the value from the algorithms used.

Code 2.7.2 shows the simplest implementation where the homogenization is done just with multiplication of the inverse of the weight of each control point. When using this implementation a `dist` value of 2048 was experimentally determined to produce no cracks even for rational Bézier with random control points.

```
void BezierPatch4x4::CalculateBBox_Rational_MulInverseW(Vector3d f_pBBox[2])
{
  register __m128 w =_mm_set1_ps(1.0f/m_Data[0].w);
  register __m128 tmp = _mm_mul_ps(_mm_load_ps((float*)&(m_Data[0])), w);

  __m128 f_pBBox0 = tmp;
  __m128 f_pBBox1 = tmp;

  for (int i = 1; i < 16; i++)
  {
    w =_mm_set1_ps(1.0f/m_Data[i].w);
    tmp = _mm_mul_ps(_mm_load_ps((float*)&(m_Data[i])), w);

    f_pBBox0 = _mm_min_ps(f_pBBox0, tmp);
    f_pBBox1 = _mm_max_ps(f_pBBox1, tmp);
  }

  _mm_store_ps((float*)&f_pBBox[0],f_pBBox0);
  _mm_store_ps((float*)&f_pBBox[1],f_pBBox1);
}
```

Code 2.7.2 *Simple implementation of bounding box computation for a cubic rational Bézier patch*

Table 2.12 shows the development of the error when using different values for `dist`. The patch used for this analysis is the same as used at the beginning of section 2.3.4 in figure 2.13. As one would expect the error increases with larger `dist` values. The jump in the *max error* at a `dist` value of 8192 is due to the increased size of the silhouette. Here a ray hits the patch in a different area. The mean error shows that even for a `dist` value of 2048 the accuracy is comparable to the accuracy of a highly tesselated patch.

# dist	Max. Error	Min. Error	Mean Error	Variance
0	$4.875660e-005$	$0.000000e+000$	$5.083150e-007$	$9.324388e-013$
16	$1.078248e-004$	$0.000000e+000$	$9.945716e-007$	$3.994041e-012$
32	$2.151132e-004$	$0.000000e+000$	$1.756948e-006$	$1.332506e-011$
64	$4.485250e-004$	$0.000000e+000$	$3.398831e-006$	$5.238159e-011$
128	$8.134246e-004$	$0.000000e+000$	$6.707922e-006$	$2.010109e-010$
256	$1.731277e-003$	$0.000000e+000$	$1.335928e-005$	$8.140084e-010$
512	$3.275633e-003$	$0.000000e+000$	$2.663262e-005$	$3.197630e-009$
1024	$6.326735e-003$	$0.000000e+000$	$5.333002e-005$	$1.251082e-008$
2048	$5.812660e-001$	$4.329195e-010$	$1.189041e-004$	$7.439624e-006$
4096	$5.810031e-001$	$4.329195e-010$	$2.384628e-004$	$1.164149e-005$
8192	$3.150732e+000$	$4.329195e-010$	$5.506144e-004$	$2.824981e-004$
16384	$3.176900e+000$	$4.329195e-010$	$1.024043e-003$	$3.723727e-004$
32768	$3.198011e+000$	$4.329195e-010$	$2.045889e-003$	$7.092118e-004$
65536	$3.296033e+000$	$4.180038e-009$	$4.331357e-003$	$2.094727e-003$

Table 2.12: Accuracy of using different `dist` values as additional termination criterion.

Improving the Accuracy

One way of improving the accuracy of the computations is to switch to the appropriate rounding mode before homogenization. Here appropriate means to round to the smaller value when computing the minimum and round to the greater value when computing the maximum. An implementation of this is shown in code 2.7.3. It is important to divide by `w` and not to compute the inverse as before because else the rounding mode has to be changed depending on the signs also. For this code a `dist` value of 64 seems to be enough.

```
void BezierPatch4x4::CalculateRoughBBox_Rational_DivisionByWRoundingMode(Vector3d f_pBBox[2])
{
  register __m128 tmp = _mm_div_ps(_mm_load_ps(&(m_Data[0])), _mm_set1_ps(m_Data[0].w));

  __m128 f_pBBox0 = tmp;
  __m128 f_pBBox1 = tmp;

  _MM_SET_ROUNDING_MODE(_MM_ROUND_DOWN);
  for (int i = 0; i < 16; i++)
  {
    tmp = _mm_div_ps(_mm_load_ps((float*)&(m_Data[i])), _mm_set1_ps(m_Data[i].w));
    f_pBBox0 = _mm_min_ps(f_pBBox0, tmp);
  }

  _MM_SET_ROUNDING_MODE(_MM_ROUND_UP);
  for (int i = 0; i < 16; i++)
  {
    tmp = _mm_div_ps(_mm_load_ps((float*)&(m_Data[i])), _mm_set1_ps(m_Data[i].w));
    f_pBBox1 = _mm_max_ps(f_pBBox1, tmp);
  }

  _mm_store_ps((float*)&f_pBBox[0],f_pBBox0);
  _mm_store_ps((float*)&f_pBBox[1],f_pBBox1);
  _MM_SET_ROUNDING_MODE(_MM_ROUND_NEAREST);
}
```

Code 2.7.3 *Simple implementation of bounding box computation for a cubic rational Bézier patch*

2.8 Non Uniform Rational Basis Splines (NURBS)

NURBS surfaces are a common primitive used for modeling especially in the CAD area [PT97]. They are a representation for polynomial surfaces with guaranteed continuity across patch boundaries and can be defined in homogenous coordinates as

$$s^w(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u)N_{j,q}(v)p_{ij}^w \tag{2.10}$$

where p_{ij}^w are the 4d control points of the control polygon and the $N_{i,p}$ are the pth-degree B-spline basis functions. The basis functions can be defined recursively over a non-uniform and non-periodic (also called clamped) knot vector

$$U = \{\underbrace{a,\ldots,a}_{p+1}, u_{p+1},\ldots,u_{n-p-1}, \underbrace{b,\ldots,b}_{p+1}\} \tag{2.11}$$

and can be written by means of the Cox-de-Boor recursion:

$$N_{i,0} = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$
$$N_{i,p}(u) = \frac{u-u_i}{u_{i+p}-u_i} N_{i,p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}} N_{i+1,p-1}(u). \tag{2.12}$$

One possible way for ray tracing NURBS with algorithm 2.2.1 would be to use the knot insertion algorithm [PT97] to split each surface. Since a NURBS surface always lies in the convex hull of its control points this would suffice. But this is quite slow.

NURBS surfaces can be converted exactly into Bézier representation using knot insertion. The result of this conversion is that each patch of the NURBS surface is represented in the Bézier basis. A NURBS patch is here the 2d analog of a single segment of a NURBS curve and not a complete NURBS surface as commonly called in 3d modeling applications.

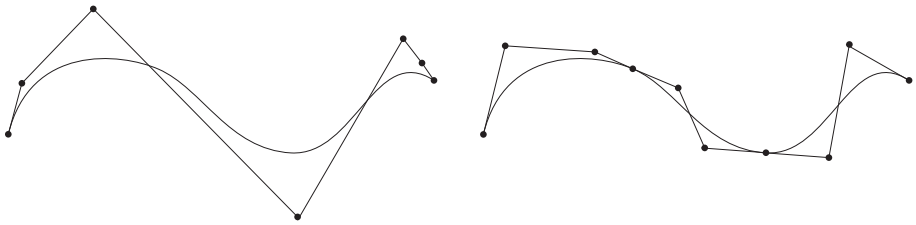


Figure 2.28: NURBS curve and its corresponding Bézier representation.

There is no loss in accuracy compared to the direct method described above because the same points are created by the algorithm. There is also no possibility for cracks between the generated Bézier patches because they share the same points at the boundary. Having the scene decomposed into Bézier patches the ray tracing can be done as described in chapter 3. The only thing that has to be taken care of is to transform the parameter values correctly so that textures are applied as intended when defined for the NURBS patches.

One disadvantage of the conversion is that the scene needs more memory because each patch of a NURBS surface is now described with $(m+1)\cdot(n+1)$ control points where m and n are the degrees in u and v direction of the surface. A NURBS surface with $k \cdot l$ control points has $(k-m)\cdot(l-n)$ surface patches. So the ratio of memory increase can be calculated as

$$\frac{(k-m)\cdot(l-n)\cdot(m+1)\cdot(n+1)}{k\cdot l}$$

which is quite large when there are a lot of huge NURBS surfaces in the scene. Table 2.13 shows the size of the raw NURBS data for two test scenes and the size of the converted Bézier patches. Images of the scenes can be found in section 3.4.

Scene	# of NURBS surfaces	Size of NURBS (kB)	Size of Bézier (kB)
dog	378	531.45	4664.625
car	3968	6791.43	61305.375

Table 2.13: Increase of memory when converting NURBS scenes into Bézier representation.

If the additional memory needed for the storage of the scene should become a problem in a practical application it is also possible to convert a NURBS surface segment on the fly in its Bézier representation. This is especially simple when using a bounding volume hierarchy as acceleration structure. The bounding volumes can be created down to each NURBS surface segment. Of course this slows down the ray tracing significantly since for each ray that hits a leaf bounding box this conversion has to be done.

2.9 Subdivision Surfaces

Subdivision surfaces are a recent approach to describe smooth surfaces. The ACM SIGGRAPH course notes [Ze99] give a good overview of the most commonly used types and methods. Subdivision surfaces are defined by a control mesh and a set of rules to recursively subdivide it. This rules are designed to produce a surface that is smooth in the limit (inifinite number of subdivision steps) at almost all points. The control mesh that is used to describe these surfaces has generally less restrictions on its topology than tensor product spline surfaces. The subdivision rules are defined for vertices of arbitrary valence and sometimes even for arbitrary polygons.

Since these surfaces are based on subdivision they should also work with the ray tracing algorithm 2.2.1. But the implementation is not as easy as for tensor product spline surfaces. The first thing to assure is the convex hull property that may not be given for some subdivision schemes (for example interpolating subdivision). Next there are generally many different rules which have to be chosen based on local topological information. This slows down repeated subdivision significantly. In the following section an unoptimized implementation for the Catmull-Clark subdivision scheme is developed.

2.9.1 Catmull-Clark Subdivision Surfaces

The Catmull-Clark subdivision scheme [CC78] is a generalization of the tensor product bicubic b-spline. The subdivision rules are defined for arbitrary polygonal meshes but after one subdivision there are only quadrilateral faces left. The subdivision rules for quadrilateral faces are illustrated in figure 2.29.

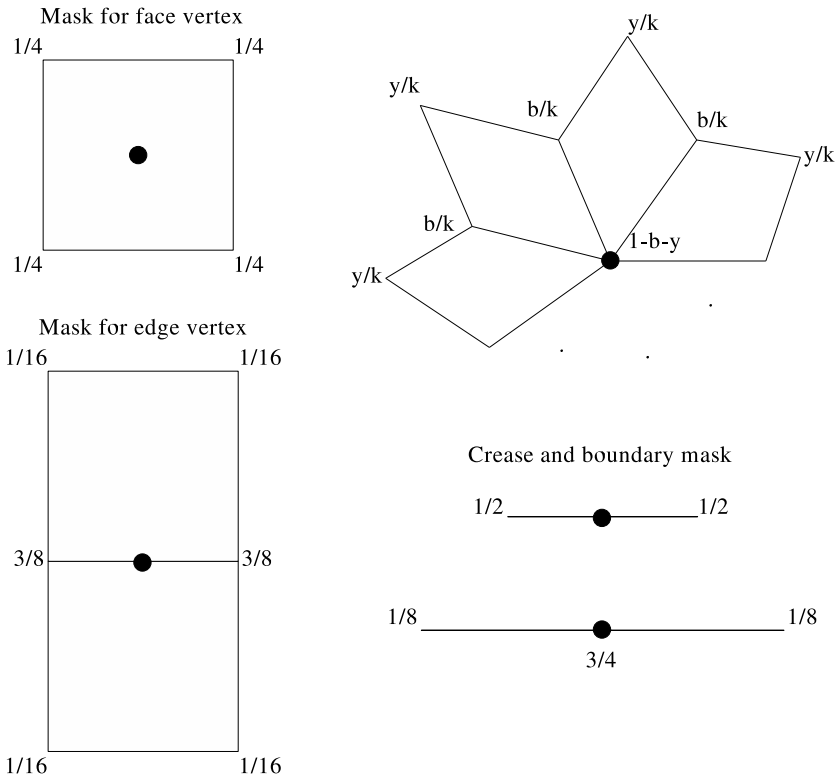


Figure 2.29: Catmull-Clark subdivision masks for quadrilateral meshes. The parameters can be chosen as $b = \frac{3}{2k}$ and $y = \frac{1}{4k}$ where $\frac{k}{2}$ is the valence of the vertex.

After one subdivision of a quadrilateral mesh each face has at most one irregular vertex. This property reduces the number of special cases a lot and it is assumed that the input for the algorithm in the next section is in this form.

Floating Point Precision Ray Tracing of Catmull Clark Subdivision Surfaces

The algorithm and code seen in this section are just a proof of concept. It is not as far developed as the code shown before for Bézier surface patches and by no means optimized.

To be able to ray trace subdivision surfaces with the general algorithm 2.2.1 each surface patch of the limit surface has to be represented separately. For Catmull-Clark subdivision surfaces each quadrilateral has an associated limit surface patch that is completely defined by the face and its 1-neighborhood. This is illustrated in figure 2.30. The shaded area is the face that defines the surface patch. As noted above each face has at most one irregular vertex. So for each patch there are $2N + 8$ vertices stored where N is the valence of the irregular vertex. The general idea of the ray tracing algorithm is now to represent each surface patch by its neighborhood (either pre-calculated or generated on the fly) and subdivide each one into new smaller neighborhoods that describe the same limit surface.

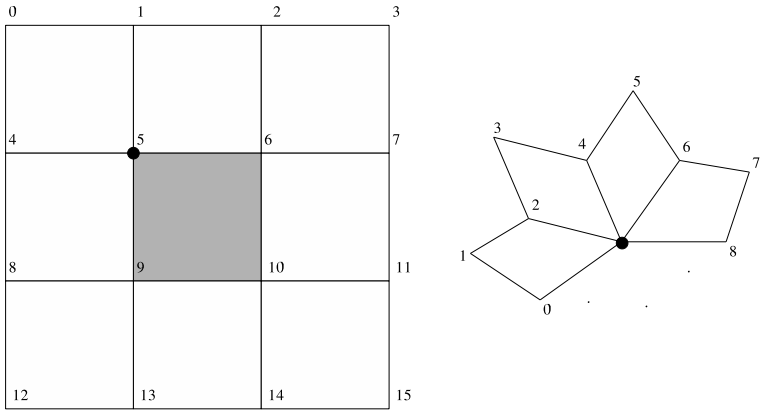


Figure 2.30: The left image shows the neighborhood for a regular patch and the vertex indices of the memory layout. If vertex 5 is irregular then the neighborhood is stored separately as shown on the right. The vertex 4 on the left corresponds with 0 on the right.

Since neighboring patches are treated independently in the ray tracing algorithm (each patch is subdivided on its own) it is important to make sure that exactly the same operations are applied to connected boundaries. Otherwise there would be cracks due to floating point imprecision. The numbering of the vertices in figure 2.30 is used for every patch. Each surface patch is rotated so that the irregular vertex has always index 5. For the Catmull-Clark surface patch creation it is necessary to have an adequate data structure for the base mesh to allow access to neighboring faces of edges and vertices. The faces must also be ordered in a consistent way to allow an ordered access. In the test implementation a brute force approach has been used that searches the complete mesh for the neighbors (either for an edge or a vertex) and orders them according to their indices as needed. After the preprocessing of the input mesh an array of Catmull-Clark surface patches (the data structure can be seen in code 2.9.1) is created and used for ray tracing.

Another important property of Catmull-Clark subdivision surface patches (and some other subdivision methods as well) that simplifies the implementation significantly is that after one subdivision there are three regular surface patches and one irregular with the same valence. This is shown in figure 2.31. The surface patch is subdivided two times and there is always only one patch with an irregular vertex.

For this reason the data structure of a Catmull-Clark surface patch contains a fixed array of size 16 array of vertices for a regular patch and an array to store additionally the neighbors of an irregular vertex if the valence is greater then four. Since the ray tracing algorithm does not need the original patch after subdivision the memory for the irregular vertices can be overridden and the subdivision stack already contains enough space for the 3 regular patches. This speeds up the recursive subdivision significantly.

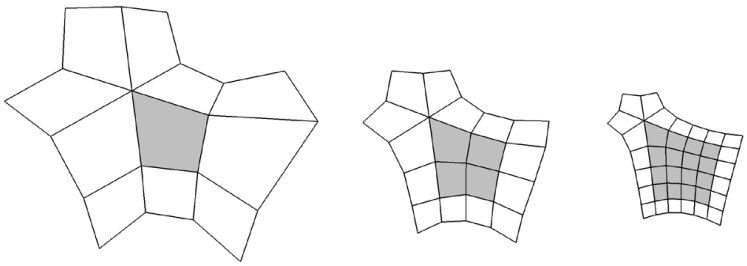


Figure 2.31: On the left a face (gray) with its neighborhood is shown. Two subdivisions steps are vizualized on the right with the resulting quadrialterals and their neighborhood. One vertex has valence 5 over all subdivision levels. All other patches created through subdivision are regular.

```
class SDSurfCC
{
    // used as a simple way to determine the type of the patch
    // like boundary, crease edge, crease vertex and
    int type;

    // the 16 control vertices of a regular patch
    Vector3d m_Data[16];
    // to keep track of the parameterization
    Vector3d u, v;
    // the valence of the irregular vertex
    int m_NumIrregularVerts;
    Vector3d* m_pIrregularVerts;
    // the index of vertex "4" in the m_pIrregularVerts array
    int m_IrregularStartVertex;
}
```

Code 2.9.1 *Data structure for a Catmull-Clark surface patch.*

The subdivision of such a patch is just an implementation of the subdivision rules seen in figure 2.29. A problem with this subdivision is that there are many special cases that have to be checked for each subdivision iteration. The main cases are regular patch, boundary, crease edge and crease vertex all with possibly one irregular vertex. In an optimized implementation it should be better to create different subdivision methods for each type to reduce the amount of branching.

The basic approach is to subdivide a patch into four new patches and place them onto the stack. Since the neighborhood of the new patchs share many vertices the set of new vertices is computed once and then copied. Each rule has been put into a macro. The rules for the regular interior case are shown in code 2.9.2. The control points of the `SDSurfCC` class are indexed and combined. The rules for the regular boundary cases are similar.

```
#define CC_FACE_VERTEX(i)
          ((m_Data[i+0] + m_Data[i+1] + m_Data[i+4] + m_Data[i+5])*0.25f)

#define CC_EDGE_RIGHT_VERTEX(i)
          ((m_Data[i+0] + m_Data[i+4] + m_Data[i+2] +
            m_Data[i+6])*0.0625f + (m_Data[i+1] + m_Data[i+5])*0.375f)

#define CC_EDGE_DOWN_VERTEX(i)
          ((m_Data[i+0] + m_Data[i+1] + m_Data[i+8] +
            m_Data[i+9])*0.0625f + (m_Data[i+4] + m_Data[i+5])*0.375f)

#define CC_OLD_VERTEX(i, a)
          ((m_Data[i+0-5] + m_Data[i+2-5] + m_Data[i+10-5] +
            m_Data[i+8-5])*0.0625f + (m_Data[i+1-5] + m_Data[i+4-5] +
            m_Data[i+6-5] + m_Data[i+9-5])*0.0625f + m_Data[i+0]*0.5f)
```

Code 2.9.2 *Subdivision rules for the regular case.*

The 25 new control vertices are now computed and stored in an array (`cct`). The computation is done by using the macros with the appropriate index. This can be seen in for the regular interior case in code 2.9.3.

```
if (patch is regular)
{
    cct[0]  = CC_FACE_VERTEX(0);
    cct[1]  = CC_EDGE_RIGHT_VERTEX(0);
    cct[2]  = CC_FACE_VERTEX(1);
    cct[3]  = CC_EDGE_RIGHT_VERTEX(1);
    cct[4]  = CC_FACE_VERTEX(2);
    cct[5]  = CC_EDGE_DOWN_VERTEX(0);
    cct[7]  = CC_EDGE_DOWN_VERTEX(1);
    cct[9]  = CC_EDGE_DOWN_VERTEX(2);
    cct[10] = CC_FACE_VERTEX(4);
    cct[11] = CC_EDGE_RIGHT_VERTEX(4);
    cct[12] = CC_FACE_VERTEX(5);
    cct[13] = CC_EDGE_RIGHT_VERTEX(5);
    cct[14] = CC_FACE_VERTEX(6);
    cct[15] = CC_EDGE_DOWN_VERTEX(4);
    cct[17] = CC_EDGE_DOWN_VERTEX(5);
    cct[19] = CC_EDGE_DOWN_VERTEX(6);
    cct[20] = CC_FACE_VERTEX(8);
    cct[21] = CC_EDGE_RIGHT_VERTEX(8);
    cct[22] = CC_FACE_VERTEX(9);
    cct[23] = CC_EDGE_RIGHT_VERTEX(9);
    cct[24] = CC_FACE_VERTEX(10);

    cct[6]  = CC_OLD_VERTEX(5, 0);
    cct[8]  = CC_OLD_VERTEX(6, 2);
    cct[16] = CC_OLD_VERTEX(9, 10);
    cct[18] = CC_OLD_VERTEX(10, 12);

    CC_ASSIGN(s[0], 0);
    CC_ASSIGN(s[1], 1);
    CC_ASSIGN(s[2], 5);
    CC_ASSIGN(s[3], 6);
}
else ...
```

Code 2.9.3 *Subdivision of a regular interior* `SDSurfCC` *patch.*

The `CC_ASSIGN` macro copies the correct vertices into the new subpatches `s[0]` to `s[3]`. The code for the regular boundary cases looks again similar. For the case of one irregular vertex the affected contol points are computed within a loop using the `m_pIrregularVerts` array of the patch data structure. As written above only one of the four new patches is irregular and since the subdivision is done in place the memory location can be reused. The cases for crease edges and crease vertices can be handled similar to the boundary cases.

A nice property of Catmull-Clark subdivision is that the weights for the vertices during subdivisions for regular patches are all numerically "nice". The weights in decimal representation used in the regular subdivision are $0.5, 0.25$ and 0.0625 which are all powers of two. Since in the limit an irregular patch is just a point there is no need for a special termination criterion. The one used for integral Bézier patches is sufficient. Figure 2.32 shows a comparison of the shadow cast by a subdivision surface. One image was tessellated by Maya using an adaptive method and the other was rendered with the algorithm described above.

A property of subdivision surfaces is that generally there exists no simple parametric representation

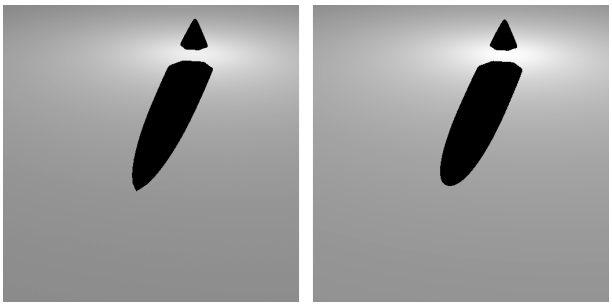


Figure 2.32: Shadow of a tessellated (left) subdivision surface and one rendered with the method described above.

of the limit surface. This is especially a problem for the calculation of the surface normal once a point of intersection was found by the recursive subdivision. Calculating the normal from the patch of the last iteration in the subdivision algorithm is not an option because either the control points are collapsed to a point or they are so close together that no meaningful calculation can be done. The easiest solution would be to use the natural parameterization ([Ze99]) of the subdivision surface to interpolate the vertex normals of the original patch. Images computed with this method can be seen in figure 2.33. A better approach would be to use a method developd by Jos Stam ([Sta98]). He uses a large set of precomputed eigenbasis functions to directly evaluate Catmull-Clark subdivision surfaces at arbitrary parameter values.

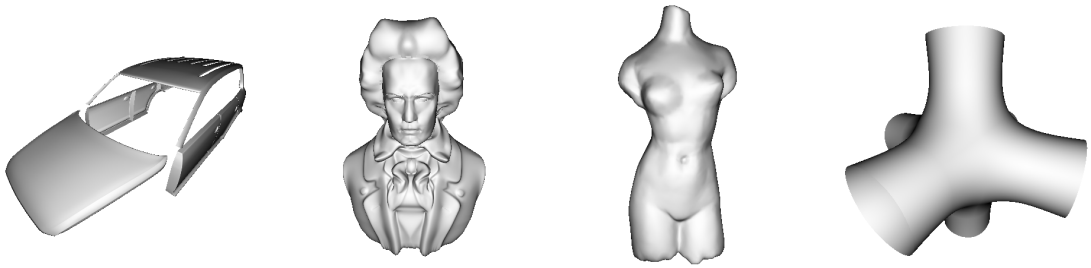


Figure 2.33: Four different subdivision models that were rendered with the ray tracing algorithm described above. The car body is part of a Maya subdivision model from Dosh design (www.doschdesign.com) the other three models are sample models of the Subdivide 2.0 package from Henning Biermann and Denis Zorin (mrl.nyu.edu/ biermann/subdivision).

Chapter 3

Integration into a Ray Tracing System

In the previous chapter the intersection of a ray with a single surface patch was discussed. When calculating an image the scene is generally composed of (at least) hundreds of patches that have to be managed efficiently. Each patch may have a different surface property and may reflect or refract light. Additionally there are light sources in the scene and the image should be generated as fast as possible. In this chapter an integration of ray tracing kernels that handle free form surfaces into a more complete ray tracer is described. So besides the ray-patch intersection the handling of secondary rays and accelerating the search for a possibly hit patch are the most important things considered here.

3.1 The Ray Tracing Kernel

Generally a ray tracer for image synthesis has to support full scenes that are composed of different types of primitives. The surface properties are described through a complex set of shaders and different light sources and special effects have to be handled. The basic operation needed for this is tracing a ray through the scene and it is useful to encapsulate this functionality in an own module. The ray tracing kernel is here the part of the ray tracing software that holds a representation of the scene (or part of it) and allows to trace arbitrary rays through it, returning the closest point of intersection and some additional information of the surface that was hit.

Since scenes are composed out of different types of primitives, a ray tracer has either to support all of them directly or convert them into another representation. The fastest ray tracers use triangles as a primitive of choice and free form surfaces are tesselated as described in section 1.3.3. As the free form surfaces are ray traced directly another approach is chosen similar to the one described by Kirk in [KA88]. It is assumed that each object is built only with one type of primitive. For example the object consist only of NURBS surfaces of same degree. For each supported primitive exists a ray tracing kernel that can build an own acceleration structure for it and allows to trace rays. All objects get an axis aligned bounding box that is managed in a global bounding volume hierarchy. When a ray is traced through the scene, the traversal starts with the global BVH tree and when a leaf is hit the appropriate ray tracing kernel is called. With this form instances of objects can be supported directly. The creation and traversal of the tree is similar to the one described in the next section.

3.2 Acceleration Structure

Acceleration structures are a crucial part of every ray tracer to speed up the search for the closest point of intersection. The most common forms are 3d grids, octrees, kd-trees (sometimes called BSP trees) and bounding volume hierarchy (BVH) trees [Gla89]. State of the art triangle ray tracers use kd-trees. But for the kernels that ray trace patches the usage of a (binary) BVH has been the fastest up to now. There are various reasons for this. Surface patches are generally much larger than triangles and the intersection calculation is far more expensive than for triangles. Each patch can be oriented arbitrarily and has

generally numerous different control points. So an axis aligned split plane as used in the kd-tree cannot separate two adjacent patches in most cases. Patches of degree higher than two can not be separated by an arbitrary plane in the general case since they have at least 4 control points on a boundary curve. So to make sure an arbitrary ray hits a leaf with few patches the kd-tree has to be quite deep.

In contrast to this, a BVH tree is of small depth because there are few surface patches compared to a triangle scene and it is easy to assure that there is only one patch per leaf. But the bounding volumes in the tree can overlap. The disadvantage of this is that the search cannot be stopped with the first point of intersection found as it is possible with kd-trees. To ameliorate this problem the tree is traversed in a way that the child that is closer to the ray origin is traversed first. So when a point of intersection is found, the ray can be already shorter and miss the away BVH nodes completely.

When using a bounding volume hierarchy the first decision to make is what bounding volume to use. There is always the trade off between the tightness of fit and the cost for intersection calculation. If the build/update time of the tree is also important (for example for animations) the complexity of calculating the bounding volume should also be considered. Another factor might be the amount of memory needed to store a node of the tree. The bounding volume chosen here are axis aligned bounding boxes because they are very easy to compute, need only 6 floats to be stored, and can be intersected quite fast (see code 2.3.7).

The next decision to make is how to create the tree. The general principles are either bottom up or top down. Bottom up means to start with the bounding box of each patch and group two neighboring bounding boxes together. This is repeated until only a single bounding box for the complete scene is left. The quality of the tree depends on how good the neighboring bounding boxes are selected. To build the binary tree top down it is started with the scene bounding box and a list of all patches. Now the list is split recursively in two parts and a new bounding box is calculated. The quality of the resulting tree depends on the procedure used to split the list respectively on how to group the patches together. There exist various methods to do this. One simple way is the median cut from Kay and Kajiya [KK86] and this is the one used here. The list of the bottom bounding volumes is sorted along one axis and split in the middle. This process is repeated recursively cycling through each axis. Another approach is to create a heuristic cost function that is minimized by the split. Goldsmith and Salmon [GS87] describe such a heuristic witch takes $O(n \log n)$ but the result is not nessecarily a binary tree and may become rather unbalanced.

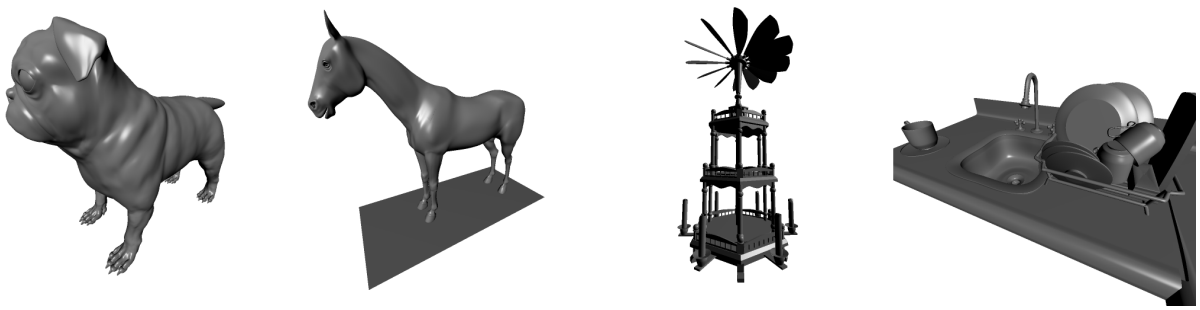


Figure 3.1: The four test scenes used for comparison between BVH and BSP. The dog and the horse are Alias Maya scenes from the patch modeling DVD that is part of Maya 5.0. The pyramid is from www.porrey61.de and the kitchen sink from Marjorie Graterol (www.emediez.com).

Table 3.1 shows a comparison between BVH and kd-tree on four different test scenes that are shown in figure 3.1. The data in the table was collected at a resolution of 256x256. The scenes are composed entirely of cubic Bézier patches. The kd-tree traversal used here is described in [HKBZ97]. No mailboxing was used in the traversal. Mailboxing is a method to avoid multiple intersections with the same primitive (see for examples [Wal04]) by storing each intersection found during one traversal. Even though the intersection calculation with a patch is very expensive, mailboxing has not made any significant difference in ray tracing speed. One reason for this is that for mailboxing a patch has to be intersected with the

full ray length. In normal kd-tree traversal the ray can be clamped to the voxel that is considered.

The BVH tree traversal used here is just an iterative implementation of the straight forward recursive tree traversal. Both trees were created top-down. For the BVH construction a median cut has been used. The BSP splitting plane was created by taking the side of a patch bounding box that is closest to the middle of the voxel. This ensures that at least one patch is split off per recursion step. It is also interesting to note that the same kd-tree algorithm applied to triangle scenes is much faster then the BVH algorithm. As can be seen in the table the BSP tree is much slower then the BVH tree. The speed

Scene (num patches)	BSP Depth	# BSP Nodes	fps (BSP)	# BVH Nodes	fps (BVH)
Dog (24878)	10	1165	0.150	49755	1.393
	20	66173	0.302		
	30	250641	0.303		
Pyramid (6244)	10	955	0.553	12487	1.634
	20	23851	0.640		
	30	44611	0.640		
Horse (4213)	10	611	0.677	8425	1.836
	20	11177	0.707		
	30	15931	0.709		
Stuff (2535)	10	237	0.114	5069	0.927
	20	4692	0.115		
	30	9093	0.115		

Table 3.1: Speed comparison of BVH and BSP. The data was created at a resolution of 256x256 and the test scene can be seen in image 3.1.

does not change much as the depth of the tree is increased. The reason for this was already noted above. The axis aligned planes cannot separate the patches very well and the rather slow ray-patch intersection (compared to ray-triangle intersection) dominates the time for the image generation since there are many leafs with more than one patch.

The more interesting comparison between the direct rendering of Bézier patches and triangulation is done in section 3.4

3.3 Instances

Instances are a way of reducing the memory usage in scenes where many objects have the same geometry but a different position, scaling or rotation. Many modeling applications produce scenes with instances and sometimes instances are the only way to handle vast scene complexity. As described in section 3.1 the organization in ray tracing kernels directly supports this. The idea is that for one object the geometry and its acceleration structure are stored only once and in object space. The instances represent the actual objects in the scene and they contain a reference to the geometry and a transformation matrix for ray, normal, and point of intersection transformation. In addition each instance has a bounding box in world space so it can be treated like a regular ray tracing kernel.

Let M be the 4×4 matrix that holds the transformation from world space to object space. The new ray can be calculated as:

```
instanceRay.m_Origin = M * ray.m_Origin;
instanceRay.m_Direction = M * ray.m_Direction;
```

Since `m_Origin` is a point ($w = 1$) it is also translated whereas `m_Direction` is a vector($w = 0$) that is only scaled and rotated. This new ray can now be sent to the ray tracing kernel to retrieve the hit information in object space. Let `MI` $= M^{-1}$ and `MT` $= M^T$. The hit information should now be transformed back into world space.

Since there is a hit interval (hit box) each of its eight points has to be transformed and a new axis aligned bounding box must be computed. But the use of instances inherently has the problem of floating point inaccuracy due to the transformations. So a slightly faster way of avoiding the problems described

in section 2.4 is used. The hit box is left in object coordinates and used only for the secondary ray when it is intersected with the associated instance. For the other instances the transformed point of intersection (the middle of the intersection interval) is used:

```
hitInfo.m_HitPoint = MI * instanceHitInfo.m_HitPoint;
hitInfo.m_HitNormal = MT * instanceHitInfo.m_HitNormal;
```

In figure 3.2 an image of a simple test scene is shown that consists of 40000 sun flower instances. The sun flower model was build from 60 cubic patches and was distributed on a jittered grid. Each instance was also randomly scaled. The image was rendered with shadow rays at a resolution of 4640x1920 with 13 rays per pixel.

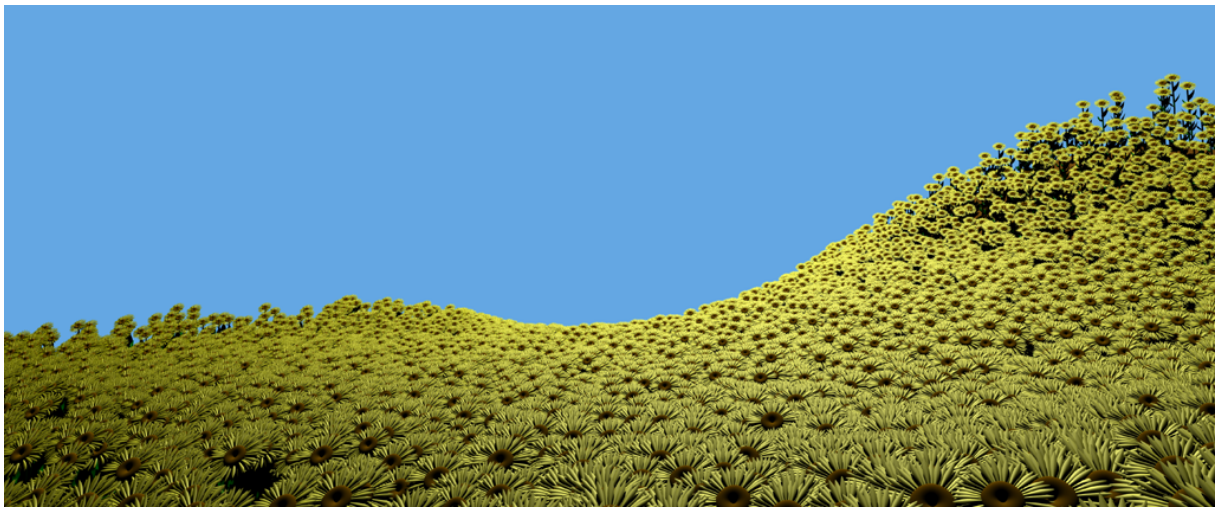


Figure 3.2: Instance test-scene with 40000 sun flowers. The image was rendered at 4640x1920 and the sun flower is modeled with 60 patches.

3.4 Results

In this section the direct ray tracing of surface patches is compared with the tessellation into triangles. As basis for testing the modeling program Alias Maya 5.0 has been used. The triangle renderer was mental ray 3.2.2.1 that is integrated into Maya 5.0. For rendering with the parametric surface ray tracer (PSRT) the scenes have been directly exported from Maya into the mental ray file format and were loaded by the ray tracer. All images were rendered with a single primary ray per pixel on an Intel Pentium 4 at 2.8 GHz running Windows XP. The settings for the surface tessellation of mental ray were left at the default values for the first pass (*MR P1*). The second pass was ray traced with the preset "Angle Detailed High Quality" (*MR P2*) and the third pass with "Pixel Area High Quality" (*MR P3*). All other settings and the scenes were left on default. Especially there were no adjustments made to the models like for example adding connectivity information for single groups. All images were rendered with these settings at 512x512 resolution. Since the "Pixel Area" method is a view and resolution dependent method the image was also rendered at 1024x1024. The rendering times given for mental ray is the wallclock time that is output by the ray tracing core at the end of rendering. Since mental ray creates the BSP while rendering the scene, this time is also contained but the scene parsing and geometry approximation (tessellation) is done before. The rendering time of the parametric surface ray tracer consists only of the image generation (ray tracing) without scene parsing, conversion into Bézier patches and acceleration structure construction. This time is given separately for each scene. There are two images of each test scene shown. One was computed with the PSRT and the other with mental ray 3.2.2.1. The "Pixel Area High Quality" gives the best visual results for a non-animated scene, so this is shown here.

Scene 1: Correct Shadows

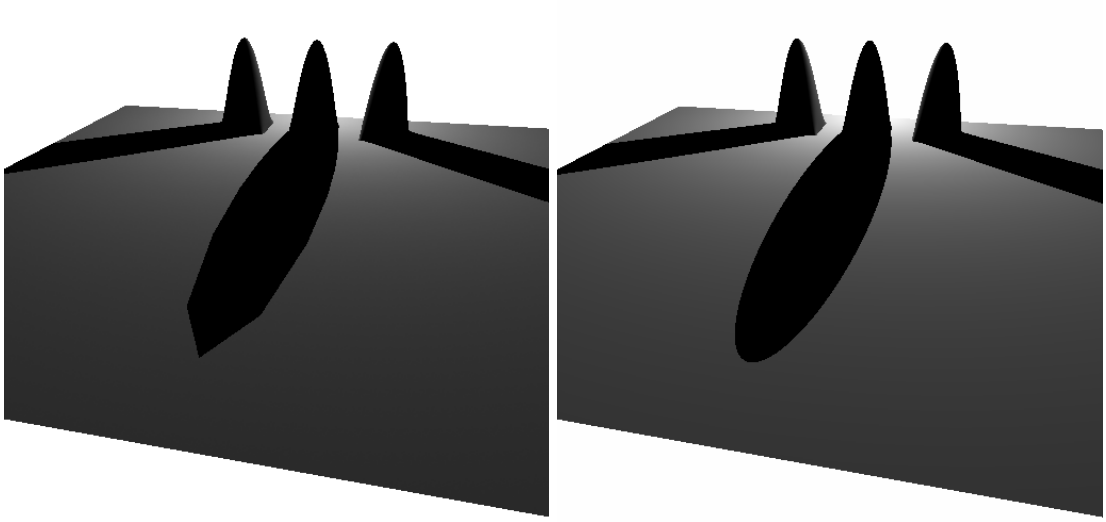


Figure 3.3: Shadow test scene. Left rendered with mental ray 3.2.2.1 (Pixel Area High Quality) and right with PSRT.

Method	Resolution	# of primitives	Size of Acceleration Structure (kb)	Rendering Time (s)
MR P1	512x512	388	14	13.65s
MR P2	512x512	1540	70	13.47s
MR P3	512x512	1540	70	14.17s
PSRT	512x512	4	0.52	4.82s
MR P3	1024x1024	1540	70	53.41s
PSRT	1024x1024	4	0.52	19.24s

Table 3.2: Measurements for the shadow test scene. File size (when exported to .mi): 11 kb; Number of Materials: 1; Number of NURBS surfaces: 4; PSRT time for parsing and acceleration structure construction: 0.01s.

Figure 3.3 shows two images of the simplest of the test scenes used here. It is the same that was already shown in chapter 2. The scene consists only of four NURBS surface patches that are already in Bézier representation. The light source is close to the top of the middle patch so this is the critical region because if it is not tessellated enough the approximation will be visible in the shadow. The patches are quite large so the non-default tessellation settings are at the limit of the predefined maximum tessellation and produce the same number of triangles. Of course better results could be achieved by hand-tuning the tessellation parameters but this setting illustrates problems that the PSRT does not have.

Scene 2: Correct Reflections

The reflection test scene that can be seen in figure 3.4 is the same as shown in chapter 2. It consist of 20 red cylinders in the sky and one large reflecting patch. The pixel area tessellation that was used in the image on the left looks much better than the other tessellation methods but the corners in the reflection can still be seen. The tessellation of the 20 red cylinders has no effect on the quality of reflections in this test case because the cylinders are straight but it effects of course the memory needed for the scene and acceleration structures.

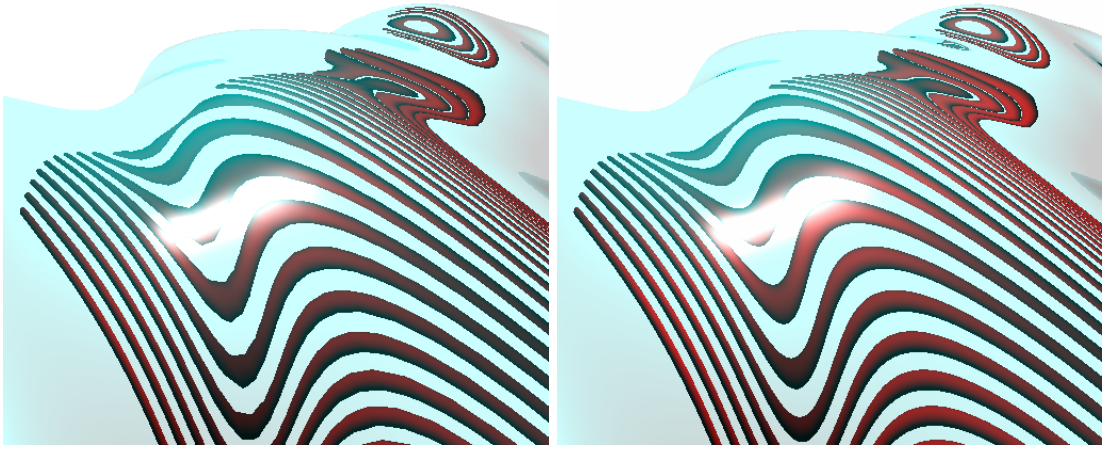


Figure 3.4: Reflection test scene. Left with mental ray 3.2.2.1 (Pixel Area High Quality) and right with PSRT.

Method	Resolution	# of primitives	Size of Acceleration Structure (kb)	Rendering Time (s)
MR P1	512x512	66695	6663	25.43s
MR P2	512x512	12480	1618	23.87s
MR P3	512x512	13904	5317	25.72s
PSRT	512x512	182	18	11.96s
MR P3	1024x1024	14944	5369	93.18s
PSRT	1024x1024	182	18	48.15s

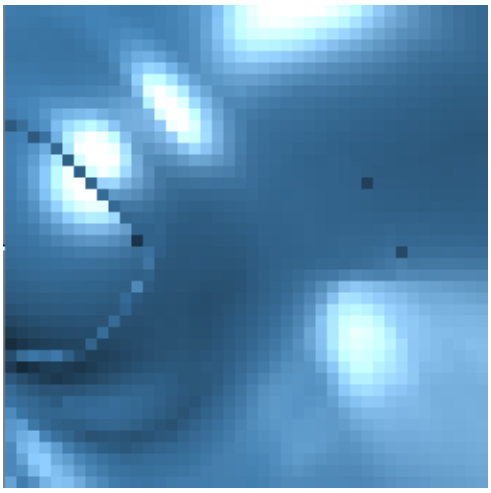
Table 3.3: Measurements for the reflection test scene. File size (when exported to .mi): 72 kb; Number of Materials: 1; Number of NURBS surfaces: 21; PSRT time for parsing and acceleration structure construction: 0.01s

Scene 3: No Pixel Errors and Correct Phong Highlights

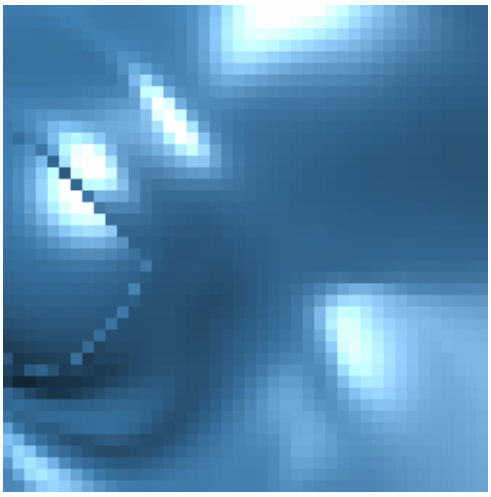
The dog model is from the "Patch Modeling for Visual Effects" tutoring DVD that is part of Maya 5.0. It consists of several larger NURBS patches with varying curvature. The NURBS patches contain no connectivity information so there are some cracks in the triangulated renderings that are visible as false pixels. This is no problem in the PSRT because the boundary control vertices are the same of adjacent patches. Some of the cracks can be seen in the figure 3.5(c) where there is a closeup of a part of the dogs eye. Even though the dog is tessellated into many triangles there is also a visible difference in the phong highlights. In figure 3.5(e) and 3.5(f) a closeup of one highlight can be seen. This difference is even more noticable with the other surface approximation modes.

(a) Dog rendered with mental ray 3.2.2.1 (preset Pixel Area High Quality).

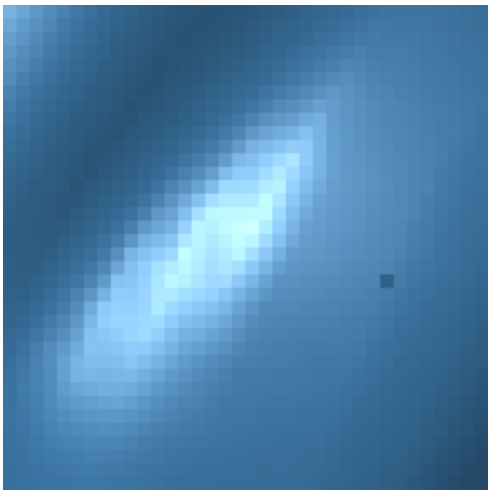
(b) Dog rendered with PSRT.



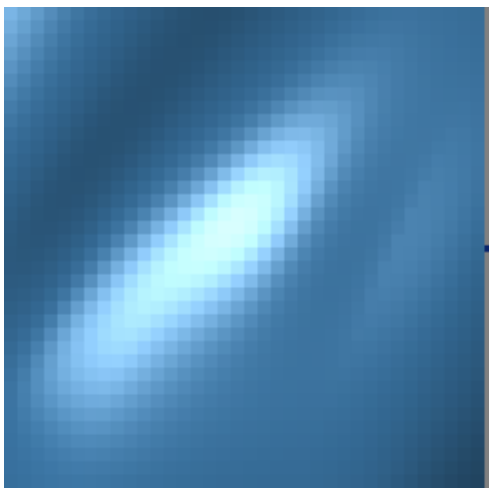
(c) Closeup of the dogs eye rendered with mental ray 3.2.2.1.



(d) Closeup of the dogs eye rendered with PSRT.



(e) Closeup of a phong highlight on the dog rendered with mental ray 3.2.2.1.



(f) Closeup of a phong highlicht on the dog rendered with PSRT.

Figure 3.5: The NURBS dog test scene.

Method	Resolution	# of primitives	Size of Acceleration Structure (kb)	Rendering Time (s)
MR P1	512x512	957260	28745	21.96s
MR P2	512x512	2140102	72894	40.05s
MR P3	512x512	71585	2303	9.42s
PSRT	512x512	24878	2350	2.78s
MR P3	1024x1024	136709	4360	31.29s
PSRT	1024x1024	24878	2350	10.97s

Table 3.4: Measurements for the dog test scene. File size (when exported to .mi): 2687 kb; Number of Materials: 1; Number of NURBS surfaces: 378; PSRT time for parsing and acceleration structure construction: 0.74s.

Scene 4: Complex Geometry

The car scene is the most complex of all test scenes. The car model is taken from the Cars 2004

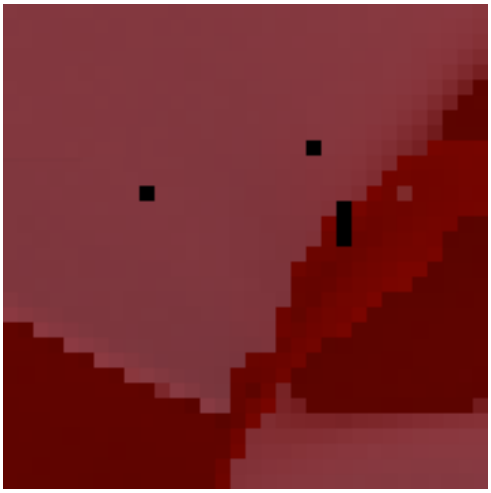
Method	Resolution	# of primitives	Size of Acceleration Structure (kb)	Rendering Time (s)
MR P1	512x512	6801488	136477	141.60s
MR P2	512x512	4593154	103275	114.72s
MR P3	512x512	439036	15097	40.02s
PSRT	512x512	261581	24709	16.70s
MR P3	1024x1024	487656	16854	121.22s
PSRT	1024x1024	261581	24709	86.02s

Table 3.5: Measurements for the car test scene. File size (when exported to .mi): 26854 kb; Number of Materials: 23 (some reflecting and/or refracting); Number of NURBS surfaces: 3968; PSRT time for parsing and acceleration structure construction: 13.42s.

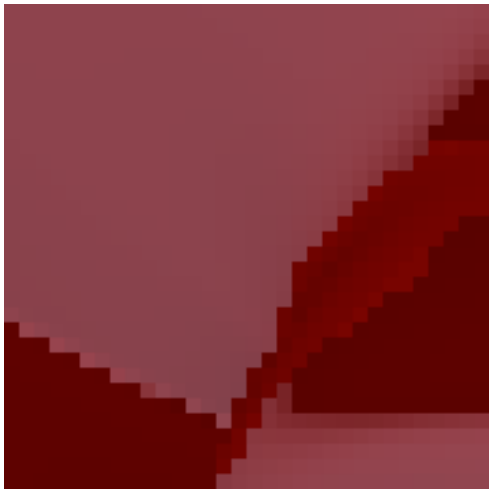
DVD from Dosch Desgin (www.doschdesign.com). As in the dog scene there are a some false pixels in the images rendered with mental ray. The false pixels above the right fender of the car can be seen in figure 3.6(c). The reason for this is again mainly due to the missing connectivity information (which is not neccessary for PSRT) and a non-uniform tessellation but there are also some false pixels in the middle of a single patch (possibly due to triangles that get too small during tesselation). Except this the resulting images look very similar from the chosen camera perspective. The reason for this is that the object consists already of many very small NURBS surfaces.

(a) Car rendered with mental ray (preset Pixel Area High Quality).

(b) Car rendered with PSRT.



(c) Closeup of the right fender rendered with mental ray.



(d) Closeup of the right fender rendered with PSRT.

Figure 3.6: The car test scene.

Chapter 4

Conclusion and Future Work

A practical method for ray tracing free form surfaces has been described that does not need any parameters to work correctly and obtains maximum floatingpoint precision, i.e. the best possible result. The accuracy of the result is much better than achieved by triangulations which can be directly seen in reflections and shadow silhouettes. The speed for ray tracing lower order polynomial surfaces is at least comparable to common triangle ray tracers. It has been shown that it is fast enough to compute images from scenes of medium complexity (about 500000 Bézier patches) with multiple light sources and different material properties.

Some optimizations have been described that sacrifice accuracy for the gain of speed. One interesting further optimization would be to update the bounding volume hierachy bottom up for animations. Since there are fewer primitives when using the surface description directly an incremental update of the acceleration structure should increase the speed of ray tracing complete animations. Other optimizations could consider secondary rays. For example extending the adaptive method to secondary ray would be usefull.

A problem of this ray tracing method is that it cannot be easily used with displacement maps that are often applied in the entertainment industry. In [HS98] Heidrich and Seidel describe a method to ray trace procedural displacement shaders directly. This could be used with the ray tracing algorithm for parametric surfaces. However displacement maps are not needed for CAD applications and product design.

Implementations for other types of subdivision surfaces are also possible. First experiments have shown that with the accuracy reduction method used in section 2.7.2 Loop subdivision surfaces can be ray traced efficiently.

Bibliography

[BWS04] C. Benthin, I. Wald, and P. Slusallek. Interactive ray tracing of free-form surfaces. In *Proceedings of Afrigraph 2004*, November 2004. 7, 8

[CC78] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978. 43

[CCC87] R. Cook, L. Carpenter, and E. Catmull. The REYES image rendering architecture. *Computer Graphics, ACM*, 21(4):95–102, 1987. 34

[CS] S. Campagna and P. Slusallek. Improving Bézier clipping and chebyshev boxing for ray tracing parametric surfaces. 7

[DBB03] P. Dutre, P. Bekaert, and K. Bala. *Advanced Global Illumination*. A K Peters, 2003. 5

[Far91] G. Farin. Rational B-splines. *Geometric Modeling - Methods and Applications, Springer-Verlag New York, Inc.*, pages 115–130, 1991. 38

[FB94] 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

[FvDFH96] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice, Second ed.* Addison Wesley, 1996. 5

[GA05] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic Bézier surfaces without triangulation. In *WSCG 2005 converence proceedings*, Plzen, Czech Republic, 2005. 8

[Gla89] A. Glassner. *An Introduction to Ray tracing*. Morgan Kaufmann, 1989. 5, 49

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. 13

[GS87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987. 50

[Her00] M. Herf. Robust epsilons in floating point, 2000. 28

[HKBZ97] V. Havran, T. Kopal, J. Bittner, and J. Zara. Fast robust BSP tree traversal algorithm for ray tracing. *Journal of Graphics Tools: JGT*, 2(4):15–24, 1997. 50

[HS98] W. Heidrich and H. Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface*, pages 8–16, 1998. 59

[JA90] D. Mitchell J. Amanatides. Some regularization problems in ray tracing. *Proceedings on Graphics interface '90*, pages 221–228, 1990. 27

[KA88] D. Kirk and J. Arvo. The ray tracing kernel. In *Proc. of Ausgraph '88*, pages 75–82, Melbourne, Australia, 1988. 49

[Kaj82] J. Kajiya. Ray tracing parametric patches. *ACM SIGGRAPH Computer Graphics*, 16(3):245–254, July 1982. 7

[KK86] T. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 206(4):269–278, August 1986. 50

[LG90] D. Lischinski and J. Gohczarowski. Improved techniques for ray tracing parametric surfaces. *The Visual Computer*, 6(3):134–152, 1990. 8

[MCFS00] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools: JGT*, 5(1):27–52, 2000. 8, 36

[MTF03] K. Müller, T. Techmann, and D. Fellner. Adaptive ray tracing of subdivision surfaces. *Computer Graphics Forum*, 22(3):553–562, 2003. 7

[NSK90] T. Nishita, T. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics, ACM*, 4(24):337–345, 1990. 7, 35

[PBP02] H. Prautzsch, W. Boehm, and M. Paluszny. *Bézier and B-Spline Techniques*. Springer, 2002. 23

[PT97] L. Piegl and W. Tiller. *The NURBS Book*. Springer, 1997. 5, 41, 42

[RW80] 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. 7, 12

[Sny92] J. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121–130, 1992. 14

[Sta98] J. Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. *Computer Graphics*, 32(Annual Conference Series):395–404, 1998. 48

[Tot85] D. Toth. On ray tracing parametric surfaces. *ACM SIGGRAPH Computer Graphics*, 19(3):171–179, 1985. 7

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 30, 50

[WBMS03] A. William, S. Barrus, R. Morley, and P. Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools: JGT*, 2003. 20

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980. 5, 7, 12

[Woo89] 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. 7

[WSC01] S. Wang, Z. Shih, and R. Chang. An efficient and stable ray tracing algorithm for parametric surfaces. *Journal of Information Science and Engineering*, 18:541–561, 2001. 8

[Ze99] D. Zorin and P. Schröder (editors). Subdivision for modeling and animation. *Course Nodes ACM Siggraph*, 1999. 43, 48