Ray Casting of Trimmed NURBS Surfaces on the GPU

Hans-Friedrich Pabst[1] Jan P. Springer[1] André Schollmeyer[1] Robert Lenhardt[1] Christian Lessig[2] Bernd Froehlich[1]

[1]Bauhaus University Weimar
[2]University of Toronto

Abstract

We propose a conceptual extension of the standard triangle-based graphics pipeline by an additional intersection stage. The corresponding intersection program performs ray-object intersection tests for each fragment of an object's bounding volume. The resulting hit fragments are transferred to the fragment shading stage for computing the illumination and performing further fragment operations. Our approach combines the efficiency of the standard hardware graphics pipeline with the advantages of ray casting such as pixel accurate rendering and exact normals as well as early ray termination.

This concept serves as a framework for the implementation of an interactive ray casting system for trimmed NURBS surfaces. We show how to realize an iterative ray-object intersection method for NURBS primitives as an intersection program. Convex hulls are used as tight bounding volumes for the NURBS patches to minimize the number of fragments to be processed. In addition, we developed a trimming algorithm for the GPU that works with an exact representation of the trimming curves. First experiments with our implementation show that real-time rendering of medium complex scenes is possible on current graphics hardware.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, Surface, Solid, and Object Representations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: intersection shader, ray casting, parametric surfaces, NURBS, Iterative Bézier Clipping, high-quality rendering, programmable graphics hardware

1 Introduction

Parametric surfaces are widely used in Computer Aided Design (CAD) and, to a lesser degree, in Digital Content Creation (DCC) systems. Non-Uniform Rational B-Spline (NURBS) surfaces are the most common representative of this type of surface. NURBS provide local and intuitive control, have a compact representation, and they are well suited for a large number of problems. In most existing systems the parametric representation of NURBS is converted into triangle meshes for interactive rendering on graphics processing units (GPU). Trimming complicates the tessellation due to the required fine sampling of trim boundaries. This process is time consuming — nightly builds are commonly used to generate high quality meshes for CAD models. In addition, the triangulated representations have high memory requirements. On-the-fly tessellation approaches on the CPU [Guthe et al. 2002] and recently on the GPU [Guthe et al. 2005] have been introduced, which recompute a high quality approximation of the surface whenever necessary. However, if a pixel-accurate representation is required, the number of generated triangles can be quite high and may demand level-of-detail techniques.

In this paper we present the concept of an extended graphics pipeline, which allows the rendering of complex primitives such as parametric surfaces and implicit surfaces. Our pipeline adds an intersection stage right after the rasterization stage and before the fragment program. The rasterization unit generates the fragments (candidate pixels) corresponding to the bounding volume of an object to be displayed. The intersection stage reconstructs the corresponding ray from the viewpoint for each fragment and computes the intersection with the surface contained in the bounding volume. The intersection point and normal are passed on to the fragment program. This seamless integration into the graphics pipeline combines its high efficiency with the advantages of ray casting. In particular it enables pixel-accurate contours and intersection curves as well as accurate normals. A single pass implementation, front-to-back sorting of the objects, and direct access to the current depth buffer enables a two-stage early ray termination check, which leads to an efficient and scalable ray casting implementation.

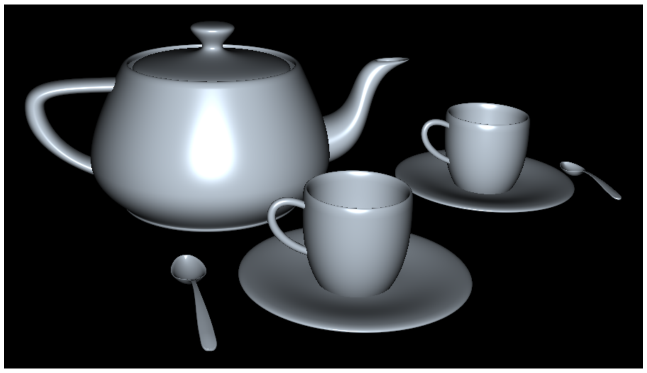


Figure 1: Perfect patch contours and intersections: ray casted scene of NURBS surfaces running with 20 Hz at a resolution of 1280 $\times$ 1024 pixels on a current GPU.

One of the most challenging graphics primitives for the implementation with our extended graphics pipeline are trimmed NURBS patches. Their high algebraic degree requires a numerical intersection approach and thus a complex intersection program. Our implementation uses Newton Iteration, a method sensitive to the chosen initial value. On the CPU the coherence of neighboring rays can be exploited to provide initial values. On the GPU rays are processed in parallel without access to the neighboring pixels' information. Instead of using ray coherence we make use of the geometrical context within the Newton Iteration. In addition to the computationally expensive ray-object intersection, high quality trimming based on NURBS trimming curves needs to be supported. However, if rays can be intersected with NURBS patches on the GPU, they certainly can be intersected with NURBS curves as well. We use the idea of a point-in-polygon test to classify the parametric coordinates of an intersection point with respect to the Bézier representation of the original NURBS trimming curves.

This paper demonstrates that direct real-time rendering of trimmed NURBS patches has become possible with current graphics hardware. We show how to efficiently implement ray casting for these types of surfaces in a single pass algorithm, which also includes pixel-precise trimming against an exact representation of the trim curves. For the implementation of our system, we introduce the concept of an extended graphics pipeline, which allows the direct treatment of graphics primitives that can be intersected with a ray. The actual implementation maps the intersection stage to the first part of the fragment program. The experiences with our system convinced us that our approach will be an ideal solution for the next generation of CAD and other NURBS-based systems. No approximations are necessary any more, neither for NURBS patches and their normal vectors nor for the frequently used trimming curves. The direct support of trimmed NURBS models and other higher order primitives are a powerful addition to the standard hardware rendering pipeline, since they greatly simplify the implementation of many graphics applications.

2 Related Work

Ray tracing and ray casting of parametric surfaces [Kajiya 1982; Toth 1985; Martin et al. 2000] and implicit surfaces [Blinn 1982; Hanrahan 1983] have been demonstrated but these approaches do not provide real-time performance on commodity hardware. Recently, several approaches for ray tracing and ray casting on GPUs have been proposed [Purcell et al. 2002; Foley and Sugerman 2005; Thrane and Simonsen 2005]. However, these techniques limit themselves to triangulated geometry, and the results reported so far do not show the speedup expected when comparing the raw performance of CPUs and GPUs. In contrast to these techniques, our approach uses the available resources of current GPUs efficiently.

2.1 Ray Tracing of Parametric Surfaces

Algorithms for ray-patch intersection can be divided into four broad categories: subdivision-based algorithms, solely iterative, numerical techniques, algebraic methods, and Bézier Clipping. Subdivision-based algorithms [Whitted 1980; Rogers 1985; Woodward 1989] use the convex hull property of parametric patches to determine the ray-patch intersection by subdividing a polygonal approximation of the surface up to a given precision. Numerical root finding methods have been employed, e. g., by Toth [1985] and Martin et al. [2000]; both were using Newton Iteration. Kajiya [1982] and Manocha [1994] used algebraic techniques, i. e. resultant-based elimination, to simplify the ray-patch intersection.

One of the first algorithms for computing the intersection between a ray and a surface patch was presented by Kajiya [1982]. He demonstrated that a ray-patch intersection test can be reduced to the problem of finding the roots of a univariate polynomial. Laguerre's method was employed for finding all roots because of its cubic convergence and the guarantee of finding an intersection. Unfortunately, because of its algorithmic complexity, this algebraic technique cannot be employed for higher order surfaces on current graphics hardware.

Toth [1985] presented results in terms of solving a general nonlinear system of equations using multivariate Newton Iteration. He solved the problem of finding an initial value by employing techniques from Interval Analysis. Unfortunately computing interval extensions results in subdivision of the surface, which is too complex to be implemented efficiently for a whole scene on current graphics hardware.

The Bézier Clipping algorithm for ray-patch intersection was first demonstrated by Nishita et al. [1990]. Bézier Clipping integrates subdivision-based and numerical methods to compute the intersection between a ray and a Bézier patch. The algorithm and its recent enhancements by Efremov et al. [2005] uses the convex hull property of parametric patches to determine intervals which are guaranteed to not include any intersection point. This improves the convergence rate compared to subdivision-only techniques while it also allows for finding the nearest intersection point. Nishita et al. [1990] also used Bézier Clipping for trimming patches. We have adapted and extended this subdivision method for ray-curve intersection and show an implementation on current GPUs.

The approach for ray-patch intersection of Martin et al. [2000] is very similar to ours. An axis aligned bounding box hierarchy was employed to reduce the number of intersection tests and to provide initial guesses for the Newton Iteration. We do not use a hierarchy and we use convex hulls as bounding volumes. Martin et al. [2000] use a polygonal approximation of the trim region to simplify the classification of intersection points.

Streaming SIMD extensions on current CPUs combine the power of single instructions executed on multiple data with the conventional programming model of CPUs, i. e. a mature tool chain supporting the development. Benthin et al. [2004] show that using streamlined arrangement of data along with the advantages of ray tracing exhibits faster rendering times of cubic Bézier and Loop subdivision surfaces than conventional CPU approaches. Geimer and Abert [2005] show interactive ray casting of trimmed bi-cubic Bézier surfaces. Both approaches exhibit lower memory requirements compared to highly triangulated models.

2.2 GPU-based Techniques

Guthe et al. [2005] propose a GPU-based algorithm for rendering trimmed rational bi-cubic Bézier patches. NURBS and T-Spline surfaces of arbitrary degree are supported by approximating them with bi-cubic Bézier patches. The two-pass algorithm seamlessly integrates into the standard graphics pipeline and generates vertices on the surface in the vertex shader through evaluation of the patch at a uniform grid in parameter space, with grid size adapted to the current view and the curvature of the patch, from which a mesh is easily generated. The individual tessellation of the patches requires an explicit treatment of cracks, which introduces further complexity. For trimming, textures containing the discretized trim curves are employed, which are updated based on the current viewing parameters. Their approach is based on rendering triangles and is therefore vertex limited, though the view dependent re-tessellation generates high quality images with very little artifacts. The approximation of trimming curves based on texture grids requires very high resolution if discontinuities of the trimming curves need to be reproduced accurately. In addition, the intersection of curved surfaces requires very fine tessellation along intersection curves to avoid visible artifacts. These situations are difficult to detect.

Purcell et al. [2002] were the first demonstrating how to implement a full ray tracing system for static scenes entirely on the GPU. They showed that consumer-level graphics hardware, as it is available today, can be used for ray casting and ray tracing of triangle-based models. In their work they employed a uniform grid as a GPU-based acceleration structure which has been built in a preprocessing step on the CPU. Such a 3D structure is not necessary for the limited case of ray casting on the GPU, since rasterization can be used efficiently to identify a good candidate set of rays for each object.

Gumhold [2003] renders ellipsoids on the GPU. He projects a quad for each ellipsoid and interpolates the parameters for the ray-ellipsoid intersection using the rasterization unit. Thus the ray-ellipsoid intersection for each pixel requires only very few simple operations, which results in high frame rates. A similar approach was implemented in older graphics hardware to support the scan conversion of spheres. This work is conceptually similar to our technique, since we also project a proxy geometry for generating the candidate rays for our objects. However we cannot use scan conversion for computing the intersection with NURBS surfaces.

The introduction of an intersection stage into the graphics pipeline as well as the corresponding intersection program is mainly a conceptual contribution of our work. The intersection stage requires that the ray through each pixel of the projection of a bounding volume is reconstructed and intersected with an object. The depth value

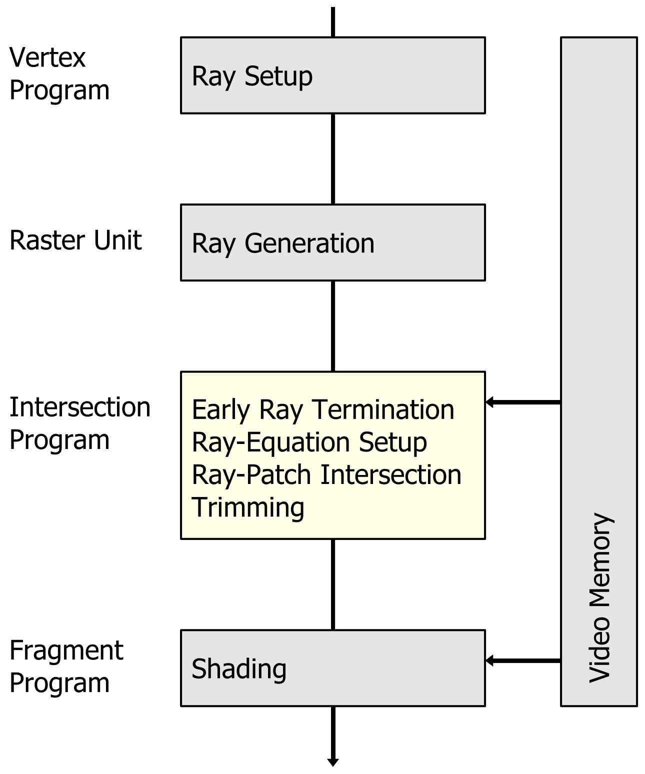


Figure 2: *Overview of the algorithm stages and their relation to stages in the extended graphics pipeline.*

of the actual intersection is written into the depth buffer instead of the depth value of the rasterized bounding volume. Krüger and Westermann [2003] and later Hadwiger et al. [2005] have also reconstructed the ray equation for each pixel from a rendered bounding box of volume data. They use this information to loop through the volume in the fragment program. They have not generalized the concept to multiple different objects and they did not introduce the concept of an intersection program or an intersection stage.

3 Algorithm

The algorithm starts with a preprocessing phase where the parametric data for each NURBS surface is stored into textures. This includes knot vectors, control points, and other auxiliary information. Next, vertex and fragment programs are generated dependent on the degree of each NURBS patch. The intersection program is implemented as part of the fragment program. Code for trimming is included if the surface domain is trimmed. The control points of the trimming curves, in our case Bézier curves derived from the original NURBS representations, are also stored into textures. In order to provide a bounding volume for each surface, sub-patches are generated by subdividing each surface patch. For each sub-patch the convex hull of its control points is computed. The corresponding parameter range is used to generate initial guesses for the Newton Iteration. The preprocessing phase would be able to run incrementally frame by frame because of low computational cost.

Figure 2 provides an overview of the stages of our algorithm, which are executed for each primitive per frame. Rendering a NURBS patch is initiated by rendering the convex hulls of its sub-patches. If early ray termination is enabled the convex hulls are sorted front-to-back. The ray originating at the front face of the convex hull and its direction is reconstructed in object space for each vertex in the vertex program. These values are interpolated in the raster unit yielding the correct ray equation for each fragment. The intersection program performs the early ray termination, a test if the intersection with the convex hull is farther away than the nearest intersection point encountered so far. If this test fails the intersection computation with the surface is performed. After that, the patch is trimmed. The intersection found in parameter space is classified against the trim region by counting the intersections of a ray from this point with all trim curves. A variation of Bézier Clipping is used for the ray-curve intersection test. All fragments that are not discarded by the trimming stage are passed on to the fragment program for shading computation. This includes the surface normals which are already calculated during the ray-surface intersection.

GPU Limitations Current GPUs only provide a subset of the functionality commonly found on CPUs. The most important limitations, which makes the implementation of complex single-pass programs a difficult task, are:

- The instruction count is effectively limited to 4096 instructions.
- There is only a small number of registers available, e. g. current graphics boards from Nvidia have 32 registers and there is no further writable random access memory.
- Because of the parallel nature of the fragment processing units coherence of initial values for the Newton Iteration between neighboring pixels/rays is difficult to exploit.

4 Surface Rendering

4.1 Notation

A non-uniform rational B-spline (NURBS) curve is defined in homogeneous form over $t \in [0,1]$ by

$$C_\tau(t) = \sum_{i=0}^{n} P_i \cdot N_{\tau,i}^{p}(t)$$

where $P_i \in \mathbb{R}^3 \times \mathbb{R}$ are homogeneous control points and $N_{\tau,i}^{p} : \mathbb{R} \mapsto \mathbb{R}$ are B-spline basis functions of degree p defined over a knot vector $\tau = (t_i)_{i=0}^{n+p+1}; t_i \in [0,1]$. For Bézier curves the knot vector simplifies to $t_0 = \ldots = t_p = 0$ and $t_{p+1} = \ldots = t_{2p+2} = 1$. The order o of the curve is defined as $p+1$; see [Piegl and Tiller 1997] for details. Control points consists of a three dimensional position and a weight. Only the position is used for convex hull computations.

A tensor product NURBS surface is defined in the following way:

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} \cdot N_{\tau,i}^{p}(u) \cdot N_{\tau,j}^{q}(v),$$

with two knot vectors τ_u and τ_v for the parameter axes and a control point mesh $P_{i,j}$ of size $(n+1) \times (m+1)$.

Following Kajiya [1982] we represent a ray in its implicit form by the intersection of two planes P_1 and P_2. The ray is therefore given by

$$P_1 = (n_1, d_1) \quad \text{and} \quad P_2 = (n_2, d_2),$$

where n_1 and n_2 are the normal vectors and d_1 and d_2 are the distances of the planes from the origin. The ray-patch intersection equation is then given by

$$F(u,v) = \begin{pmatrix} n_1 \cdot S(u,v) + d_1 \\ n_2 \cdot S(u,v) + d_2 \end{pmatrix} = 0.$$

4.2 Evaluation of Curves and Surfaces

In order to solve $F(u,v) = 0$ the surface and its partial derivatives have to be evaluated. This can be done through direct evaluation of the defining function. However, other algorithms such as the de Casteljau for Bézier curves and surfaces as well as the de Boor algorithm for B-spline curves and surfaces exist. They exhibit several properties which make them more suitable for our purposes [Farin 1990]. In particular, these algorithms are numerically robust since they mainly consist of repeated convex combinations. From a computational point of view the algorithms allow to calculate the partial derivatives while incurring only little additional computational cost; in our GPU implementation de Boor's algorithm involves a binary search to find the knot vector range for the respective parameter when evaluating a curve. When evaluating a curve $C(t)$, $t \in [0,1]$ at t_0, one step of the de Casteljau algorithm subdivides C into two sub-curves $C_l(t)$, $t \in [0,t_0]$, and $C_r(t)$, $t \in [t_0,1]$, and calculates the corresponding control points. This property is exploited for trimming, as described in section 5.

To implement the NURBS evaluation on the GPU, the number of required registers is the critical factor. The evaluation of a curve needs only N registers with N being the order of the curve. Evaluating a tensor-product surface with curves of order M and N requires $M+N$ registers. $M+2N$ $(N \le M)$ registers are necessary to additionally compute the partial derivatives during surface evaluation.

4.3 Ray-Patch Intersection

Most algorithms for intersecting a ray with a parametric patch cannot be implemented on current GPUs as they either use an amount of memory which is not bound by the degree of the patch, e. g. subdivision methods, or have excessive resource requirements, e. g. algebraic elimination methods for degrees higher than bi-cubic.

We chose Newton Iteration because only evaluation of function values and partial derivatives are required. Additionally, the memory requirements are low because only the parameter values of one iteration step need to be kept. It converges quadratically if the initial guess is close enough to the solution. This makes it superior to other iterative methods such as Bisection which converges at a linear rate. However, if the initial guess is not close to the solution neither the quadratic convergence nor finding the root are guaranteed. We will discuss this in more detail in section 4.4.

The roots of $F(u,v)$ are the parameter values of ray-patch intersections. They can be found with Newton Iteration by starting with a guess $(u_0,v_0)^T$. Each step then takes the form

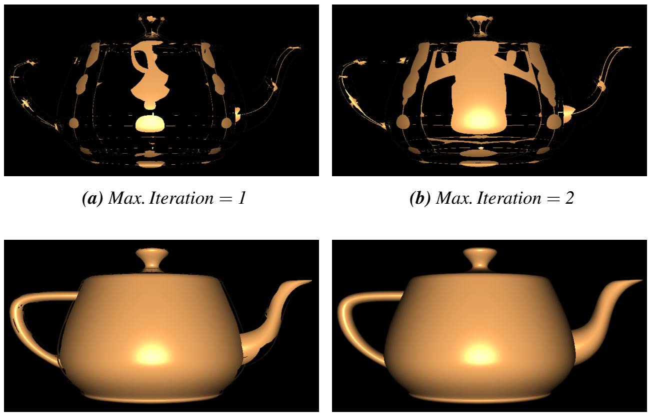
$$(u,v)^T_{k+1} = (u,v)^T_k - J^{-1} F(u_k,v_k).$$

The Jacobian J of F at $(u,v)^T$ is given by

$$J = \begin{pmatrix} n_1 \cdot S_u & n_1 \cdot S_v \\ n_2 \cdot S_u & n_2 \cdot S_v \end{pmatrix}$$

where S_u and S_v are the partial derivatives of S at $(u,v)^T$. The inverse of the 2×2 matrix J can be computed directly. The Newton Iteration terminates if the absolute value of F falls below a tolerance ε, i. e. $|F(u,v)| < \varepsilon$ or the maximum number of iterations is exceeded. The image sequence in figure 3 exemplary visualizes the influence of the number of iterations during the Newton Algorithm.

For an efficient implementation, in particular for making use of early ray termination, Newton Iteration has to be implemented in a single rendering pass. Newton Iteration only adds a constant amount of registers to the complexity of the surface evaluation. Therefore computing the partial derivatives simultaneously with a surface point is still bound to $M+2N$ registers, with $N \le M$.



(a) *Max. Iteration = 1* ***(b)*** *Max. Iteration = 2* ***(c)*** *Max. Iteration = 3* ***(d)*** *Max. Iteration = 4*

Figure 3: *Convergence of the Newton Iteration for different maximum numbers of iterations; from (a) to (d) the number of iterations increases from one to four (initial values were generated using view-independent uv-texturing and 2×2 uniform subdivision).*

4.4 Initial Values

A good choice of the initial value is situated close to the parameter value corresponding to the first intersection point of a ray with the surface. The choice of the initial values directly affects visual quality and performance. We developed two complementary approaches to generate initial values: subdivision of the convex hull of a NURBS patch and uv-texturing of the bounding volume.

Subdivision of the Convex Hull Following the approach of Martin et al. [2000] multiple bounding volumes are used; each being the convex hull of a sub-patch and each providing an initial guess for its enclosed part of the NURBS surface. For each NURBS surface the control mesh is subdivided by knot insertion [Piegl and Tiller 1997] until a given depth is reached or the sub-patch becomes sufficiently flat. For each sub-patch the convex hull of its control mesh is computed using the Quick Hull algorithm [Barber et al. 1996]. The parameter range for each sub-patch is restricted to the parameter range between the inserted knots. The center of this parameter range is a suitable initial guess because the union of the convex hulls of all sub-patches for a surface patch approximate the patch with quadratically decreasing distance error as the subdivision increases [Piegl and Tiller 1997]. The convex hull is also used by the uv-texturing methods described below.

The number of rays missing the enclosed surface is reduced with the subdivision due to the tight approximation of the patch through the convex hulls of its sub-patches. Thus the convex hulls employed in our work provide an additional advantage over the axis-aligned bounding boxes used in [Martin et al. 2000]. Although convex hulls require a higher number of triangles than bounding boxes, processing vertices is not our limiting factor.

***uv*-Texturing** We have developed a view-independent and a view-dependent technique for computing an initial guess for the Newton Iteration. The view-independent method interpolates parameter values corresponding to the vertices of the convex hull of the control mesh. Since the convex hull is an approximation of the NURBS surface, the parameter values interpolated along its surface provide a better approximation of the intersection than the fixed midpoint approach described earlier.

In order to include the ray direction in the calculation of the initial guess the view-dependent method uses the vertex program to intersect a ray from the viewpoint through each vertex of the convex hull of the contained patch. Newton Iteration is also used for this ray-patch intersection. The start values for the vertex rays are obtained from parameter values of the corresponding control points similar to the view-independent method. The computed parameter values for the vertices are then interpolated across the surface of the convex hull and provide the initial guess for each generated fragment.

4.5 Early Ray Termination

The current hardware rendering pipeline uses the early-Z test to avoid unnecessary shading operations. Fragments are discarded before they reach the fragment processors if they will fail the Z-test [Montrym and Moreton 2005; ATI 2005]. In the extended graphics pipeline as proposed in this paper, a similar test, performed before the intersection shader, could be even more beneficial. The minimal possible depth value of a ray-surface intersection is given by the intersection point with its convex hull. A fragment can be discarded if the depth value generated by rasterizing the convex hull is greater than the one stored in the depth buffer. In this case the intersection test of the corresponding ray with the contained object is not necessary. Even if this approach is not identical to the early ray termination used with CPU-based ray tracing systems, its effect is similar. Thus, we will use this term also in the context of the extended rendering pipeline.

Unfortunately, when implementing the intersection program as part of a fragment program, the early-Z test available on current hardware cannot be used: the depth value of the ray-object intersection is returned from the fragment shader instead of the depth value generated by the rasterization of the convex hull. Modifying the depth value in the fragment program disables the early-Z test on current graphics hardware. An alternative is to implement the functionality of early-Z through the programmable pipeline. This requires that the current depth buffer is also used as a texture and the early ray termination test is implemented as a branch in the fragment program, skipping the whole program if any possible intersection is behind the already encountered ones. However, the result of reading from a texture which is part of the active frame buffer is neither defined in OpenGL nor in Direct3D. In practice, current graphics hardware supports reading from the active depth buffer even if there is no guaranty that the values retrieved from a texture fetch are the currently correct values of the depth buffer. The parallelism of the processing units in the GPU as well as the missing texture-cache-memory synchronization may yield this inconsistency. Fortunately, in the case of early ray termination this may only incur some additional intersection tests but the regular depth test, performed after the fragment program, ensures that the generated image is artifact free. In particular, the value retrieved by a texture lookup into the depth buffer is always greater or equal to the currently correct depth value. By discarding rays whose minimal possible intersection depth is greater than the depth value retrieved from the texture lookup into the depth buffer, our implementation of early ray termination is conservative and the standard depth buffer test ensures depth consistency in the rendered scene.

A second early depth test can be used after the ray intersection is computed. If a ray hits a surface, the exact depth of the intersection point is computed and compared to the depth value already fetched from the depth buffer for the first early-Z test. If the fragment is hidden then the trimming and shading can be skipped.

5 Direct Trimming of NURBS Surfaces

Trimming is the process of removing parts of a surface patch by specifying one or more non-intersecting trim curves in the parameter domain of the surface. These curves are typically NURBS curves and they form the boundary of the trim region. Parameter values

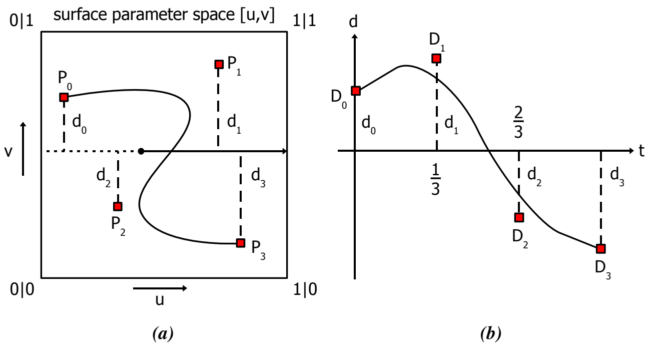


Figure 4: *Nishita transformation.*

inside the trim region do not belong to the domain of the patch and corresponding surface points are excluded from the surface.

We employ a technique similar to the point-in-polygon test for classifying points in the (u,v) domain with respect to the trim region. A ray is generated in the parameter domain of the patch originating at (u_p, v_p) and pointing in an arbitrary direction. This ray is intersected with all relevant trimming curves and if the total number of intersections is odd, the surface point $S(u_p, v_p)$ is trimmed away.

We developed an iterative version of the Bézier Clipping algorithm [Nishita et al. 1990] for the ray-trim curve intersection. The original version cannot be implemented on current graphics hardware due to its recursive nature. The NURBS representation of the trimming curves needs to be converted into Bézier form for Bézier Clipping, which provides an exact representation of the original curves. Our Iterative Bézier Clipping has the capability to determine all ray-curve intersections in the order of an increasing curve parameter. It uses the curve evaluation already described in section 4.2.

5.1 Bézier Clipping

Bézier Clipping is a robust root finding algorithm used in [Nishita et al. 1990] for trimming and rendering of Bézier surfaces. For computing ray-curve intersections the algorithm requires that each parametric Bézier curve C is transformed into a Bézier polynomial d. The roots of d are located at the ray-curve intersections as shown in figure 4a. The graph $D(t) = (t, d(t)) \in \mathbb{R}^2$ (figure 4b) can be thought of as an unwound version of the curve along the parameter t with evenly spaced Bézier points $D_i = (^i/_N, d_i)$ where N is the degree of the curve and d_i is the distance of the control point P_i perpendicular to the ray.

The roots of the Bézier polynomial d are determined by repeatedly clipping away parameter regions which do not contain any root. Both Bézier Clipping and Iterative Bézier Clipping start by clipping from the interval bounds of the original curve's domain, resulting in a connected sub-interval in each iteration. However, if this interval does not shrink by a specified amount, it is split and Bézier Clipping starts recursively for each part. Iterative Bézier Clipping has been developed to avoid the recursion and to work with constant memory requirements for the interval split.

```
// find the largest slope
for (i = 1..N)
  slope = slope( D[i], D[0] )
  if ((sign(d[i]) != sign(d[0])) and (slope < current))
    current = slope
    k = i

// find the predecessor with smallest slope
for (i = 0..k)
  slope = slope(D[i],D[k])
  if ((sign(d[i]) != sign(d[k])) and (current < slope))
    current = slope
    l = i
```

Listing 1: *Finding the first intersecting line segment of the convex hull with the t-axis.*

5.2 Convex Hull Intersection

Bézier Clipping requires that the parameter axis t of the Bézier polynomial d is intersected with its convex hull. The smallest and largest intersection parameters t_{min} and t_{max} define the boundary segments $l = [0, t_{min}]$ and $r = [t_{max}, 1]$, which cannot contain any root of d. The explicit computation of the convex hull using e. g. Jarvis March or Graham Scan is too expensive in both memory and time. We use a simpler algorithm, which searches on the fly for only those edges of the convex hull intersecting the t-axis (see listing 1 for finding t_{min}).

Starting from D_0, which is part of the convex hull, all control points are searched for point D_k with a different sign than D_0 and the maximum slope with respect to D_0. There exists a line segment L of the convex hull intersecting the t axis that contains D_k and another point D_l of $D_0 \dots D_k$ which is the other endpoint of L. D_l is found by searching $D_0 \dots D_k$ for the point with different sign than D_k and minimum slope with respect to D_k. The parameter t_{min} is then found by intersecting L with the t axis. Accordingly, t_{max} can be found by starting with D_N and searching analogously. For example, in figure 5 the line segment $[D_1, D_2]$ contains t_{min} and the line segment $[D_2, D_3]$ contains t_{max}.

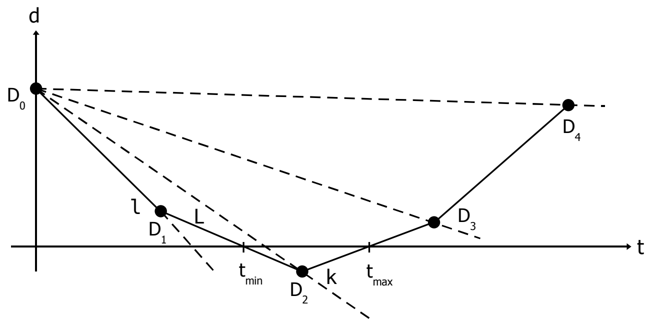


Figure 5: *Finding the intersecting line segment L of the convex hull with the t-axis, e. g. the line segment $[D_1, D_2]$ in the drawing.*

5.3 Iterative Bézier Clipping

As mentioned before, it is necessary to split the current interval in cases where it contains more than one root or the clipping operation does not shrink the interval l enough. In this case each of the sub-intervals is separately searched. Since this must be supported with constant memory cost only two intervals at a time are used: the interval l currently searched for roots and the interval r to be searched later. The interval s extends from the left boundary of l to the right boundary of r. The interval s contains all the possible roots and it serves as a parameterization of the sub-curves. The left interval is initialized with $l := s$ and searched first. The right interval r is initially empty. If l needs to be split, s is split instead and l is assigned the left part of s while r is assigned the remainder of s. Splitting will shrink l and enlarge r to ensure l and r remain a partition of the interval s. When l becomes small enough a root is found and the search is continued in r, i. e. $l := r$ and r empty. Control points for the curves restricted to the respective intervals are computed from the original control points every time they are needed. With this technique all roots in an interval can be found.

Iterative Bézier Clipping favors re-computation over storing values in order to maintain a small fixed-sized state through iterations. While our algorithm is conceptually simple, the requirement to avoid code duplication due to missing function call mechanisms and limited program memory leads to complex state management in our implementation.

In figure 6 an exemplary interval contraction is shown alternating between clipping and subdivision; note the growing of interval r when a subdivision step is applied.

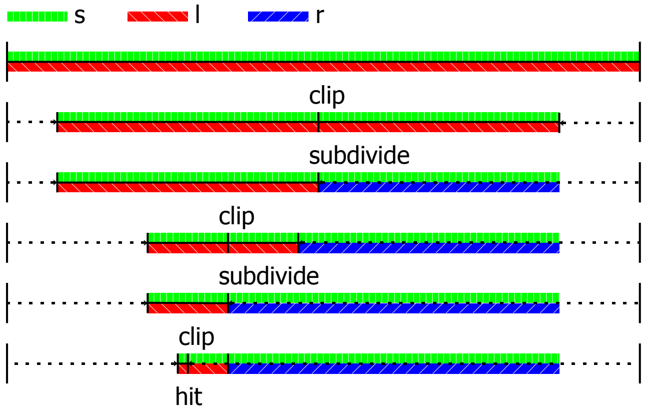


Figure 6: *Interval contraction during the Iterative Bézier Clipping algorithm.*

The algorithm's requirements in terms of register usage is mainly determined by the curve evaluation. Thus $N + C$ registers are needed, where N is the order of the curve and C is a constant amount of additional registers for holding the state inside the loop. The intervals are expected to converge quadratically if they contain only one root.

Alternatively the algorithm can be implemented by clipping only the left side of the left interval l. Thus the root is always approached from the left side and only t_{min} has to be determined in the convex hull intersection test. This saves half of the number of convex hull intersections at the expense of additional subdivisions. We found that this version performs nearly twice as fast as the original version.

6 Implementation and Results

In our implementation we employ the runtime generation of shader programs using the Sh framework [McCool and Du Toit 2004]. Sh allows the programmatic driven creation of GPU programs, also known as shader metaprogramming. We evaluated our implementation for a number of scenarios including trimming. The images were computed with a screen resolution of 1280 × 1024 pixels and partially cut out from the screen shot. The objects were positioned such as that 80% of the screen width was covered. The number of iterations for the Newton solver varied between four and 16. The original patches were uniformly as well as adaptively subdivided at different levels generating up to 70000 triangles for the convex hulls. For each surface an individual vertex program and fragment program was generated and bound to that surface. All tests were run on a PC with an AMD Athlon 64 3000+ CPU and Nvidia Geforce 7900 GT GPU.

The performance for rendering NURBS surfaces depends on the degree of a surface patch. As can be seen in table 1 the draw time for rendering a single Bézier patch at 100 % screen coverage is directly related to its degree. This is in accordance with the computational complexity of the de Boor algorithm as well as the de Casteljau algorithm which grow quadratically with respect to the degree of the surface patch under evaluation. Because of the very low number of vertices used to drive the fragment programs and the high arithmetic intensity in the fragment programs our approach is clearly fill limited. For this reason we prefer to show fill rates in MPixel per second instead of patches per second.

Degree	2×2	3×3	4×4	5×5	6×6
Draw Time (ms)	15	27	56	94	139
Fill (MPixel/s)	87	48	23	14	9

Table 1: *Peak performance for Bézier patches (single patch per degree, resolution:* 1280×1024*, screen fill:* 100 %*, max. iterations:* 4*).*

Table 2 shows the performance data for various scenes, trimmed and untrimmed, with increasing uniform subdivision. The results show that we achieve between 17 and 36 frames per second (FPS) using four iteration steps. Adding more iterations does not affect draw times dramatically, e. g. the teapot in figure 7b was rendered with 36 FPS at four iterations, it still achieves 18 FPS when rendered at a maximum of 16 iterations. Draw times decrease for sufficiently subdivided patches since less rays are created that do not intersect the surface. At some point draw times start to increase if the subdivision creates too many triangles. Adaptive subdivision helps to automatically find a good subdivision limit since a finer triangulation of the convex hulls is driven by a flatness criterion.

Figure	Triangles	Subdivision	Draw Time (ms) at Iteration Step		
			4	8	16
7a	3732	1×1	55	67	92
	15648	2×2	51	61	81
	63602	4×4	51	62	86
7b	3092	2×2	30	41	62
	12698	4×4	28	36	56
	51160	8×8	33	40	55
7b, 11a	3092	2×2	45	57	82
	12698	4×4	41	53	71
	51160	8×8	45	55	72
7b, 11c	3092	2×2	56	67	92
	12698	4×4	52	62	82
	51160	8×8	58	67	85

Table 2: *Performance data for various scenes with increasing uniform subdivision and number of triangles generated for the bounding volume.*

Figures 8 and 9 show the convergence of the Newton Iteration for a bi-cubic Bézier patch at different numbers of maximum iterations. Initial values were generated by view-independent *uv*-texturing as described in section 4.4. Figures 8a and 9a also show a color coding of the parameter domain with $RGB = (u, v, 0)$. Figure 8 uses the midpoint heuristic as described in [Martin et al. 2000]. Note that we generate the convex hull of the control point mesh as a surface approximation instead of using axis-aligned bounding boxes. In figure 9 the control point mapping described in section 4.4 was used. As can be seen very clearly the control point mapping converges faster than the midpoint heuristic. Additionally the different shapes of the convergence area suggest that the midpoint heuristic may still have visible artifacts for even more Newton steps.

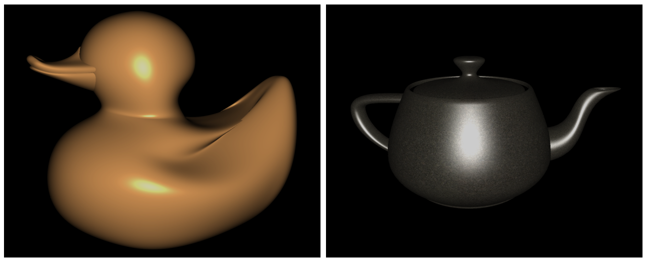

(a) *161 Patches* ***(b)*** *32 Patches*

Figure 7: *NURBS models used for performance data in table 2.*

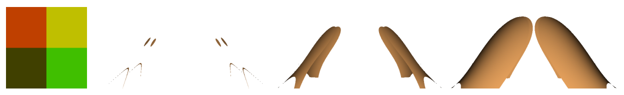

(a) ***(b)*** ***(c)*** ***(d)***

Figure 8: *uv-texturing using midpoint of parameter range (uniform subdivision:* 2×2*). (a) shows the uv-texture and (b) to (d) show the result for up to two, three, and four iterations, respectively.*

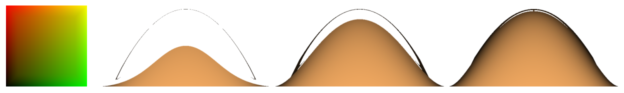

(a) ***(b)*** ***(c)*** ***(d)***

Figure 9: *uv-texturing using control point mapping (uniform subdivision:* 2×2*). (a) shows the uv-texture and (b) to (d) show the result for up to two, three, and four iterations, respectively.*

In figure 10 details of a teapot are shown for different subdivision methods. The methods generate the same amount of triangles for the convex hull of the whole model, 12500 in that case. The inset on the lower right of each image shows the convex hull textured with the initial values used for the Newton Iteration. In figure 10a adaptive subdivision was used while in figure 10b a uniform subdivision was employed. With the adaptive subdivision a triangle distribution towards critical regions is achieved driven by the flatness criteria. This leads to earlier convergence than using a convex hull generated by a uniform subdivision. Also, it generates less artifacts at the same iteration level as shown in the marked regions of figure 10b, where the upper right regions are missing pixels and the lower left region exhibits penetration problems between the body and the nozzle of the teapot.

Table 3 presents the results of operating without the early-Z test, using hardware early-Z, or the manual early-Z in the fragment program. The scene is a set of quads stacked in depth shaded by a compute intensive fragment program. We used a perspective view, which explains the nonlinear increase in the draw times. The results show that the hardware early-Z is very efficient, but the manual early-Z behaves also quite well in comparison to doing without early-Z at all. We employ the manual early-Z for early ray termination. The results in table 3 indicate that the dependency on the actual depth complexity in our case is quite low although hardware-supported early-Z cannot be used.

Table 4 shows the results of rendering a bi-cubic Bézier patch along with three different trim regions (see figure 11). The draw times clearly exhibit a strong dependency on the number of trimming curves, which is due to the fact that we are not using an acceleration approach. But even the current implementation would allow for

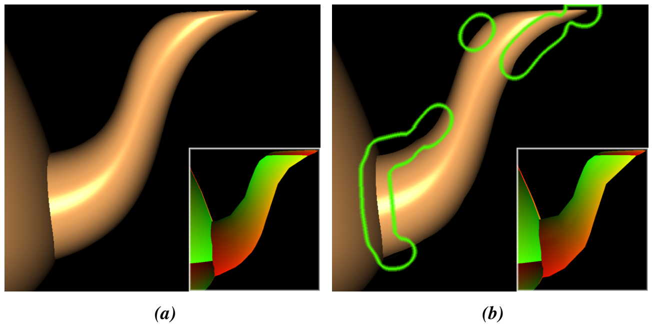


Figure 10: Adaptive subdivision using flatness criteria vs. uniform subdivision (teapot with 12500 triangles, 3 iterations). Inset on lower right shows the convex hull textured with the initial values. The marked regions show the differences, which are hardly visible. Along the spout of the teapot there are a few pixels missing. The intersection curve of spout and body of the teapot shows also a few pixel differences, which are due to z-fighting. The artifact markers in (b) were generated by image postprocessing.

Early-Z	Draw Time (ms) for No. of Surfaces						
	2	4	8	16	32	64	128
NURBS/off	33	52	80	135	183	205	250
NURBS/on	21	27	30	34	39	45	55
Tri/hardware	10	10	11	12	13	13	16
Tri/manual	13	16	20	24	30	38	42
Tri/none	18	32	56	86	116	160	200

Table 3: Draw times for different primitives with respect to early-Z and early ray termination (resolution: 1280 × 1024 pixels, screen fill: 80 %).

interactively changing trim curves because only the textures containing the control points would have to be updated. This feature is important for various application scenarios such as CSG-operations in a CAD context.

Figure	Curves	Draw Time (ms) at Resolution (100 % screen fill)			
		400^2	600^2	800^2	1000^2
no trim	0	6	11	17	25
11a	2	14	26	44	66
11b	8	26	53	88	134
11c	9	28	56	93	141

Table 4: Trimming baseline for a single bi-cubic Bézier patch (max. iterations: 4, uniform subdivision: 4 × 4).

7 Discussion

Our current implementation can handle patches up to an order of $M+2N \leq 19$. With 32 registers available in the fragment shader patches of degree six by six are supported. The already announced Direct3D 10 standard [Blythe 2006] requires 4096 registers for the fragment shader, which should be enough for all practical purposes. Moreover the storage is supposed to be provided by an indexable register file, which would enable unified programs for different degrees of surfaces.

An iterative algorithm like Newton's method may introduce pixel artifacts due to non-convergence, which was also reported from other implementations, e. g. [Martin et al. 2000]. A reference image generated with a large number of iteration steps and a very finely subdivided convex hull can be seen in figure 12a. The image in figure 12b results from a rendering using uniform subdivision, four by four sub-patches, and up to four iteration steps. The regions marked in figure 12b show the difference between both images. The single pixel error on the duck's bill was the only one we could find on this model. There are also a few pixel differences at the interpenetration of surfaces due to z-fighting. It is our experience that the adaptive subdivision of the convex hulls in combination with a maximum of four to eight iteration steps results in rendering with very few or no artifacts as can be seen in the images. To further reduce the amount of artifacts simply increasing the number of maximum iteration steps at runtime or generating finer subdivision in the preprocess is effective.

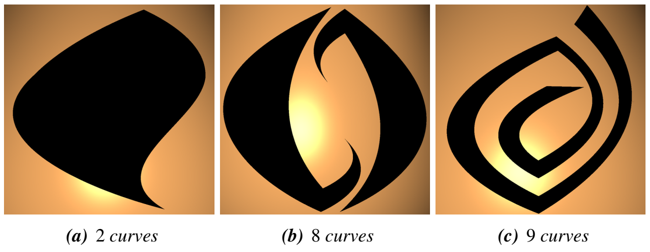


Figure 11: Images for trimming baseline performance for table 4.

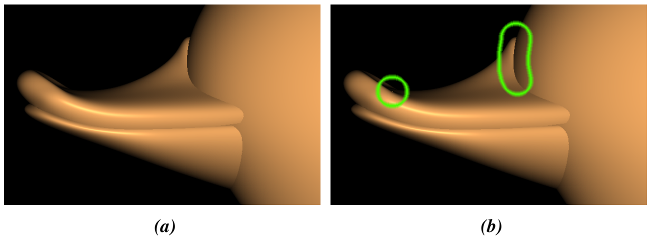
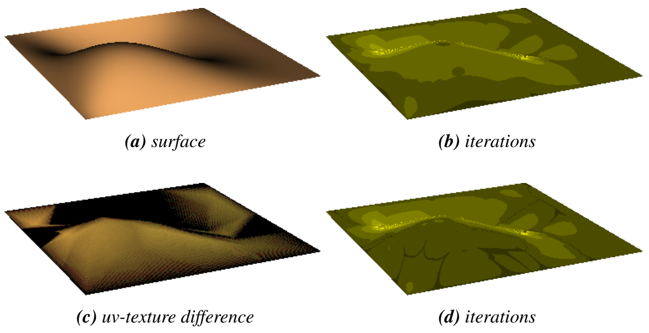


Figure 12: (a) Reference image vs. (b) default quality (max. iterations: 4, subdivision: uniform, 4 × 4). The artifact markers in (b) were generated by image postprocessing.

First experiments with view-dependent *uv*-texturing for creating improved guesses for the initial values are shown in figure 13. Figure 13a shows the surface used for the experiments. Figures 13b and 13d show the number of actual iteration steps (i) encoded in the pixel color as $RGB = (i,i,0)$ for view-independent and view-dependent *uv*-texturing, respectively. Brighter pixel colors denote a higher number of iterations (up to ten). In figure 13c the difference of the *uv*-textures from both methods is shown. Darker regions mean no difference, which results in the same number of iteration steps, while brighter regions show the difference of the initial values encoded in the color as $RGB = (\Delta u, \Delta v, 0)$ for the current view. With view-dependent *uv*-texturing the generation of initial values by intersecting the boundary control points of a patch in the vertex program in general leads to less iteration steps in the fragment programs.

Our approach for direct trimming of NURBS is pixel accurate and robust. Compared to texture-based methods, such as [Guthe et al. 2005], the memory requirements for our algorithm are negligible; only the control polygons of the Bézier curves have to be stored. Furthermore Iterative Bézier Clipping is not restricted to direct rendering of parametric surfaces. It can also be used to trim triangulated surfaces. Our implementation could also be used to trim regions which are changing on a frame-by-frame basis resulting



(a) *surface* ***(b)*** *iterations* ***(c)*** *uv-texture difference* ***(d)*** *iterations*

Figure 13: *(a), (b) view-independent uv-texturing vs. (a), (d) view-dependent uv-texturing (max. iterations:* 10*, subdivision: uniform,* 4×4*). (c) is the difference between the initial values of both methods.*

from CSG-operations at runtime. Even the present implementation which classifies every pixel of the patch with respect to all trim curves performs reasonably well. For more complex trim regions an acceleration approach needs to be implemented considering only the relevant trim curves with respect to a given point in the parameter domain.

The shader program creation is driven by classifying each surface in the scene. Depending on the scene the amount as well as the complexity of these shader programs varies. There is a trade-off between creating a specialized shader program for each encountered surface patch and using a single general program. Using one shader program per surface patch effectively limits the number of surfaces which can be rendered efficiently to a few thousand. On the positive side many parameters can be directly encoded into the program saving registers and limiting the number of instructions, which makes the implementation of complex primitives such as NURBS surfaces possible. However switching shader programs is an expensive operation on current graphics hardware. Alternatively shader programs can be build per surface type, e. g. based on the degree of the surface and shared between instances of that surface type. This limits the amount of shader programs that have to be generated and switched in the hardware. The drawback of this method is that the front-to-back sorting of the scene, required for early ray termination, conflicts with the sorting with respect to the shader program type. Trade-offs between both sorting orders are possible but it is not clear what the impact on the performance would be.

Our algorithm is mostly limited by the fragment processors. Solutions like tightly connected dual graphics cards and next generation graphics cards with their unified vertex and fragment processors (cf. Direct3D 10 standard [Blythe 2006]) will alleviate this problem considerably. First experiments with a dual graphics card solution using Nvidia's SLI hardware showed a speedup of almost a factor of two. Also, with the Sh framework we are able to generate GPU programs for surfaces with arbitrary degrees. However current hardware limits us to degree six by six.

Our approach is difficult to compare directly to triangulation-based techniques, but there are a few related issues. The numerical intersection approach leads to precision issues if the number of iterations is quite small. These problems are particularly present at interpenetrations of surfaces and at the silhouette. In other regions the normals and the shading computations seem to be less dependent on the number of iterations. Triangle-based renderings have precision problems at exactly the same locations as our ray casting approach if the subdivision is not sufficiently high. One of the most important differences is that ray casting is mainly an image resolution-dependent technique whereas the triangle-based methods depend heavily on the number of patches. However our method has also some dependency on the number of patches, since we are using the convex hulls for finding the candidate sets of rays. The early-Z test saves the expensive intersection operations for occluded patches. For triangle-based methods occlusion culling approaches could help to keep the dependency on depth complexity low.

8 Conclusions and Future Work

In this paper we have presented a conceptual extension of the rendering pipeline by adding an explicit intersection stage. This extended pipeline combines the efficiency of the standard hardware graphics pipeline with the advantages of ray casting enabling high-performance, pixel-accurate rendering of a large class of objects, for which the intersection with a ray can be computed. The extended rendering pipeline can be mapped onto available graphics hardware by implementing the intersection test as part of the fragment program. The applicability and efficiency of our approach has been demonstrated by the implementation of ray casting for trimmed NURBS surfaces on current graphics hardware. Newton Iteration is employed for the intersection test and Iterative Bézier Clipping for exact trimming. Early ray termination based on a double manual early-Z test results in a mainly resolution-dependent algorithm with only slight dependency on the actual depth complexity. We have shown that interactive rendering of medium sized scenes is possible with current graphics hardware.

We need to further investigate methods for the generation of good initial guesses for the Newton Iteration, since they could speed up the computations significantly and avoid artifacts due to incorrect or failing convergence. One promising idea is the use of a very low resolution 4D structure similar to a light field [Levoy and Hanrahan 1996]. In a potentially GPU-based preprocess a set of rays is intersected with the patch and the *uv*-values of the intersection points are stored in the 4D structure. At runtime, an interpolating look-up is performed, which considers the actual origin and direction of the current ray. Different methods for computing the intersection test, such as Branin's method or Bézier Clipping, need to be further explored as well.

Rendering of patches with a complex set of trimming curves requires an acceleration structure for fast access to the relevant Bézier segments. In most cases a 1D structure subdividing the axis orthogonal to the test ray direction should be sufficient. The subdivision could be a regular structure or adapted to the boundaries of the convex hulls. The first approach requires only indexing, while the second approach needs to use a binary search for accessing the relevant segment. In each segment, the Bézier curves are sorted along the ray, which allows to exclude curves, which lie behind the start point of the ray. Here a binary search may be used as well, but it might not be necessary since the number of curves per segment is typically very small. These acceleration approaches have low memory requirements, but the challenge is to fit this additional program code on the GPU.

We believe that GPU-based ray casting of trimmed NURBS surfaces will be a viable alternative for CAD and other NURBS-based systems since it greatly simplifies the implementation of such systems. In addition to parametric surfaces the extended rendering pipeline allows to seamlessly integrate implicit surfaces as well as other objects that can be intersected with a ray into the hardware rendering pipeline. Moreover the introduction of an intersection stage and an explicit intersection program keeps the ray-surface intersection separated from the actual shading program, which allows arbitrary combinations of primitives and shading algorithms. The straightforward mapping onto existing graphics hardware lets us to believe that these are the first indications that triangles as the primary rendering primitive will be complemented by a powerful set of higher order primitives in the near future.

Acknowledgments

We thank the Nvidia Professional Solutions Group Europe for providing us with sample hardware. We also thank Sebastian Knödel for help and discussions. This research was partly supported by the DaimlerChrysler Virtual Reality Competence Center.

References

Radeon X800 3D Architecture White Paper. http://www.ati.com/products/radeonx800/RadeonX800ArchitectureWhitePaper.pdf, 2005.

C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

C. Benthin, I. Wald, and P. Slusallek. Real-time Rendering: Interactive Ray Tracing of Free-Form Surfaces. In *AFRIGRAPH '04: Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, pages 99–106, 2004.

J. F. Blinn. A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982.

D. Blythe. The Direct3D® 10 System. *ACM Trans. Graph.*, 25(3): 724–734, 2006.

A. Efremov, V. Havran, and H.-P. Seidel. Robust and Numerically Stable Bézier Clipping Method for Ray Tracing NURBS Surfaces. In *Proceedings of the 21st Spring Conference on Computer Graphics SCCG '05*, pages 127–135, May 2005.

G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., New York, NY, USA, 1990.

T. Foley and J. Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 15–22, 2005.

M. Geimer and O. Abert. Interactive Ray Tracing of Trimmed Bicubic Bézier Surfaces without Triangulation. In *WSCG '2005: The 13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 71–78, 2005.

S. Gumhold. Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings 8th International Fall Workshop on Vision, Modelling and Visualization*, pages 245–252, November 2003.

M. Guthe, J. Meseth, and R. Klein. Fast and Memory Efficient View-dependent Trimmed NURBS Rendering. In *Proceedings of Pacific Graphics 2002*, pages 204–213, 2002.

M. Guthe, Á. Balász, and R. Klein. GPU-based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Trans. Graph.*, 24(3):1016–1023, 2005.

M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In *Proceedings of Eurographics 2005*, volume 24, pages 303–312, 2005.

P. Hanrahan. Ray Tracing Algebraic Surfaces. In *SIGGRAPH '83: Proceedings of the 10th Annual Conference on Computer Craphics and Interactive Techniques*, pages 83–90, 1983.

J. T. Kajiya. Ray Tracing Parametric Patches. *SIGGRAPH '82: Proceedings of the 9th Annual Conference on Computer Craphics and Interactive Techniques*, pages 245–254, 1982.

J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003*, pages 287–292, 2003.

M. Levoy and P. Hanrahan. Light Field Rendering. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Craphics and Interactive Techniques*, pages 31–42, 1996.

D. Manocha. Solving Systems of Polynomial Equations. *Computer Graphics and Applications*, 14(2):46–55, March 1994.

W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphical Tools*, 5(1): 27–52, 2000.

M. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. A K Peters, Ltd., Wellesley, MA, USA, 2004.

J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25 (2):41–51, 2005.

T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics*, 24 (4):227–345, August 1990.

L. Piegl and W. Tiller. *The NURBS Book*. Monographs in Visual Communication. Springer-Verlag New York, Inc., New York, NY, USA, 2nd. edition, 1997.

T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Craphics and Interactive Techniques*, pages 703–712, 2002.

D. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, New York, NY, USA, 1985.

N. Thrane and L. O. Simonsen. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus, July 2005.

D. L. Toth. On Ray Tracing Parametric Surfaces. In *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pages 171–179, 1985.

T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):96–102, 1980.

C. Woodward. Ray Tracing Parametric Surfaces by Subdivision in Viewing Plane. In W. Strasser and H.-P. Seidel, editors, *Theory and Practice of Geometric Modeling*, pages 273–290. Springer Verlag, 1989.