Interactive Ray Tracing of NURBS Surfaces by Using SIMD Instructions and the GPU in Parallel

Diploma Thesis

Presented by

Oliver P. Abert

Centre
Graphics & Media
Technology

Centre for Graphics and Media Technology
Nanyang Technological University

Institute for Computational Visualistics
Working Group Computer Graphics

Examiner: Prof. Dr.-Ing. Stefan Müller
2^{nd} Examiner: Assoc. Prof. Dr. Ing. Wolfgang Müller

September 2005

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt Übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben kann.

Singapur, den 25. September 2005 Oliver Abert

Danksagung

Diese Diplomarbeit entstand im Centre for Graphics and Media Technology an der Nanyang Technological University in Singapur. Daher möchte ich mich bei Prof. Dr.-Ing. Stefan Müller und Assoc. Prof. Dr.-Ing. Wolfgang Müller bedanken, dafür, dass sie mir den Aufenthalt hier ermöglicht haben und das sie als Prüfer zur Verfügung stehen.

Desweitern gilt mein besonderer Dank Dipl. Inform. Gerrit Voss und Dipl. Inform. Markus Geimer, die beide stets gute Anworten auf alle meine Fragen bereit hielten. Ohne diese Hilfe, wäre diese Arbeit nicht zu dem geworden, was sie ist.

Weiterhin danke ich auch Matthias Biedermann, der so hilfbereit war diese Arbeit für mich auszudrucken, binden zu lassen und einzureichen, während ich zu dieser Zeit noch in Südost Asien war und keinen Zugang zu einem Drucker in Koblenz hatte.

Ebenfalls danken möchte ich Matthias Zumpe, der als mein Bürokollege mich vor der Vereinsamung bewahrt und immer ein offenes Ohr für meine neuesten Ergebnisse hatte.

Nicht zuletzt möchte ich auch meinen Eltern danken, die es mir ermöglicht haben dieses Studium zu beginnen und zu einem erfolgreichen Ende zu führen.

Oliver Abert

Zusammenfassung

Ziel dieser Diplomarbeit ist es, komplexe Freiformflächen, d.h. B-Splinesowie NURBS Flächen, mit interaktiven Frameraten direkt, mit Hilfe des Raytracing Verfahrens, zu rendern, ohne dabei vorher eine sonst übliche Triangulierung durchzuführen. Um dieses Ziel zu erreichen ist die Verwendung von Vektoreinheiten (SIMD Einheiten) moderner Prozessoren, sowie parallel dazu, die Nutzung des Graphikprozessors vorgesehen. Interessant ist das direkte Raytracing von NURBS Flächen aus mehreren Gründen. Zum einen sind Flächen dieser Art zum Standart für CAD/CAM Systeme geworden und werden vielfach in der Industrie verwendet. Oftmals wird für eine Darstellung solcher Modelle in Echzeit eine aufwendige und meist fehlerbehaftete Triangulierung durchgeführt. Weiterhin ist der Speicherverbrauch von NURBS Szenen im Vergleich zu hochauflösenden Dreiecksmodellen geringer und zudem ist die Entfernung des Betrachters zur Szene beliebig, da bei Nahaufnahmen störende Dreieckskanten an gekrümmten Flächen gar nicht erst auftreten können. Nicht zuletzt bietet das Raytracing Verfahren eine Simulation von realistischen Effekten, wie Schatten und Reflektionen oder globale Beleuchtung, die mit den herkömmlichen Rastererisierungsverfahren nicht, oder zumindest physikalisch unzureichend, dargestellt werden können. All dies macht das direkte Raytracing von komplexen Freiformflächen ein sehr interessantes Feld und vor allem unter Berücksichtigung der ungebrochenen rasanten Leistungssteigerung der Prozessoren insbesondere zukunftsträchtig.

Im Rahmen dieser Arbeit wurden zwei Bibliotheken, *libNURBSIntersectionKernel* und *libSIMD*, sowie zwei Applikationen, *iges2dsd* und *crianusurt*, entwickelt. *libSIMD* stellt eine Abstraktionsschicht für SIMD Befehle dar, die je nach Architektur, auf SSE, AltiVec oder die FPU umgesetzt werden. *libNURBSIntersectionKernel* stellt die eigentliche Funktionalität für die Schnittpunkt- und Normalenberechnung, sowie einen Trimmingtest und Oberflächenevaluationsmethoden zur Verfügung. Darüber hinaus gibt es eine hochperformante Unterstützung für die weniger mächtigen, aber rechnerisch einfacheren bikubischen Bézierflächen. Alle Berechnungen nutzen das SIMD Potentzial voll aus, d.h. es werden stets Packete von vier Werten parallel berechnet. Die Anwendung *iges2dsd* übersetzt IGES Dateien in ein binäres Format, dass später von der eigentlichen Renderapplikation *crianusurt* gelesen werden kann. Während des Übersetzens einer Datei werden zahlreiche Daten vorberechnet um die Performanz während des Renderings zu maximieren. So wird beispielsweise eine hocheffiziente Bounding Volume Hierarchie erzeugt, die zugleich, wie üblich, die Anzahl der zu testenden Objekte reduziert, gleichzeitig aber auch einen guten initialen Schätzwert für die Newton Iteration liefert, welche eingesetzt wird um den Schnittpunkt zu ermitteln. Weiterhin werden alle Basis Funktionen

der NURBS Flächen vorberechnet und als einfache Polynome bei den jeweiligen Flächen gespeichert. Dies ermöglicht eine Flächenevaluation durch simples auswerten eine Anzahl Polynome. Trimmingkurven, falls vorhanden, werden in eine optimierte Bézier Darstellung gebracht und Bounding Boxen werden soweit möglich vorklassifiziert (d.h. komplett getrimmt oder nicht).

Die vorgestellten Bibliotheken und Anwenungen sind allsamt Cross-Plattform entwickelt worden und sind somit, wie getestet, auf Mac OS X/PowerPC, Linux/x86 und Linux/Itanium lauffähig. Es wurden interaktive Frameraten für nicht triviale Szenen auf einem einzelnen handelsüblichen PC erreicht. Die erwarteten Eigenschaften, wie geringerer Speicherverbrauch und Erhalt der Bildqualität bei extremen Nahaufnahmen, wurden erreicht. Die optimierte Bézier Repräsentation erweist sich dabei als deutlich schneller. Weiterhin ist das in dieser Arbeit präsentierte polynombasierte Verfahren zum evaluieren von NURBS Flächen mehr als 100 mal schneller als ein üblicher brute force Ansatz, der einfach die Cox-de Boor Rekursion auswertet. Einschränkenderweise muss gesagt werdem, dass die Verwendung von hochkomplexen NURBS Flächen zwar mit dem Vorgestellten Verfahren möglich ist, doch dabei die Interaktivität verloren geht. Komplexe Flächen mit vielen Kontrollpunkten überfordern die numerische Genauigkeit der vorberechneten Basisfunktionen. In diesem Falle muss ebenfalls die Cox-de Boor Rekursion verwendet werden. Weiterhin hat sich die GPU als noch nicht mächtig genug erwiesen um in diesem Kontext verwendet zu werden. Einzelne Schnittpunktberechnungen auf der GPU sind zwar möglich, aber die beschränkte Anzahl der ausgeführten Kommandos, sowie der temporären Register, lassen nicht genügend Spielraum für die erforderlichen komplexen Berechnungen.

Contents

Chapter 1

Introduction

1.1 Motivation

Recently interactive ray tracing became reality even on a single commodity PC due to the fast pace CPU performance was, and still is, increasing. However, most implementations can solely handle triangles as the only geometric primitive. By carefully exploiting the available resources of today's computers it was lately shown, that it is possible to render even simple free form surfaces, like Bézier surfaces, at interactive frame rates for non-trivial scenes (see [5] and [25] or [1] for a more detailed essay) on a single PC.

On the one hand, using Bézier surfaces instead of triangles has several advantages, including but not limited to lesser memory consumption and higher precision, especially on boundaries of curved surfaces, as free form surfaces will always stay perfectly curved, independent of the distance to the viewer. On the other hand, some drawbacks are apparent: the intersection test is much more complex and takes more time to compute. Also often NURBS (Non Uniform Rational B-Spline) surfaces are used during modeling, thus there is still a (lossy) conversion of data needed. Further advantages include all the benefits a ray tracing approach offers in general, i.e. ray tracing is a lot more physically correct than raster graphics are. Effects like reflections, transmission and even global illumination can be computed in a very straight forward manner. Due to this fact, ray tracing has become especially popular in visualization contexts (i.e. CAD/CAM), rather than real time applications like computer games. Although the NURBS representation has become the standard in computer aided design and digital content creation, they usually have not been used in conjunction with ray tracing. This is basically due to the fact, that ray tracing is considered an expensive algorithm and the ray-NURBS intersection is even more expensive. Until today the most common way to render NURBS surfaces, is to tessellate them into triangle meshes beforehand, which can unfortunately

introduce artifacts and consumes much more memory. Preprocessing time is also increased, notably when tessellating very complex surfaces.

Now it is time for the next logical step: Direct rendering of NURBS surfaces. In this work it is presented, how it is possible to render nontrivial scenes based on NURBS primitives at interactive frame rates, by benefiting from the experience gathered in the previous work on Bézier surfaces [1]. The most serious problem that needs to be addressed, is the much higher complexity of NURBS surfaces compared to their Bézier counterparts. Basically the most important operation for Bézier intersection tests is only the multiplication of three matrices for point evaluation, which is well suited for execution using SIMD (single instruction multiple data) instructions (see [16] for x86 or [28] for ppc details on SIMD instruction sets). With NURBS we now face recursively defined basis function descriptions, which can neither be computed using SIMD nor be processed by the GPU straight away. Finding a most efficient way to avoid recursive computation is essential for this work, in order to achieve interactive frame rates. As ray tracing is an ideal candidate for parallel computation, because of the independency of one pixel to another, it seems like a natural choice to use both the power of SIMD instructions as well as the brute force of the GPU.

1.2 Requirements

The Newton based iterative intersection algorithm (see 5.2.2 for details), which is employed to perform intersection tests, along with the evaluation of points on NURBS surfaces are quite complex compared to a simple ray-trinagle intersection test. In order to be able to achieve interactive frame rates a recent machine should be used, for example a Macintosh with a G4 processor running at 1+ GHz or an Pentium 4 class PC with 1.5+ GHz. As multithreading is supported dual core and/or multiprocessor systems will definitively increase performance considerably. Memory consumption is not very high unless loading extremely large and complex models, so 512 MB of main memory is sufficient most of the time.

The algorithms, that were developed on the GPU, make plenty use of floating point textures, dynamic branching and loops which are inevitable, thus the choice of usable graphic boards is quite limited. First of all, the shader for intersection test is written in OGSL (OpenGL ShadingLanguage - see 2.3.2), which actually is not a real restriction to the hardware. However, the bottom line of cards that supports all needed features is NVidia's GeForce 6800[1].

[1] Upon completion of this work the successor GeForce 7800 was released, but it was not available in time to be used in this work

The application itself runs on Linux as well as on Mac OS X[2]. The appropriate SIMD instruction set (if available) for the current architecture is detected automatically during compile time.

1.3 Structure of this Document

First of all Chapter 2 gives a general introduction on free form surfaces, in particular on NURBS curves and surfaces, how they are defined and what impact this has on implementing a CPU/GPU based algorithm.

Chapter 3 explains what steps are necessary during preprocessing in order to convert data to a suitable format, for achieving a maximum of performance during rendering. This includes the generation of an advanced bounding volume hierarchy in section 3.2, which pays special attention to the needs of free form surfaces. Also the preparation of trimming curves is discussed in section 3.3, as well as important techniques to improves the speed of surface evaluations in section 3.4, which is crucial to the intersection test. Finally, in section 3.5 it is explained in what way the previously computed data is stored for CPU and GPU use.

Chapter 4 shortly outlines the API of the two stand-alone libraries *libSIMD* and *libNURBSIntersectionKernel* which can also be used independently in any other application as well.[3]

Chapter 5 is about the actual rendering application called *crianusurt* (**Cr**oss-platform **in**teractive **NU**RBS **su**rface **r**ay **t**racer). The first section 5.1 explains the basic architecture and design of the application, where section 5.2 shows in detail, how the intersection test of a ray and NURBS surface is solved, which actually is the most important task. The trimming test is dealt with in section 5.4 and section 5.6 is about limitations and problems of the algorithms presented in this chapter.

The usage of both applications *iges2dsd* (converter application) and *crianusurt* are explained in chapter 6. Chapter 7 shows various results obtained with the algorithms presented here, whereas finally chapter 8 concludes and suggests future areas of research.

[2] Apple Mac OS Tiger recommended, Panther should work though, too

[3] Note that the intersection functionality is explained and discribed in chapter 5, although it is actually implemented in *libNURBSIntersectionKernel* - Chapter 4 only deals with the API

Chapter 2

Basics

This Chapter describes how free form surfaces are defined. Special attention is paid to B-Spline and NURBS curves and surfaces. The mathematical background is explained in section 2.2 as much as required for comprehension of this work. A complete discussion on the complex topic of NURBS would exceed the scope of this work. Please refer to [34] if a more complete essay regarding free form surfaces in general is desired. Furthermore the basic principles of GPU programming are described in section 2.3 where section 2.4 is about using SIMD instructions for increased performance on the CPU. Trimming is outlined in section 2.5 where the concept of ray tracing in general is introduced in section 2.6. This chapter concludes with section 2.7 which states the basic software design decisions taken.

2.1 Free Form Curves and Surfaces

Mathematically there are basically three different ways to define curves or surfaces, which are *explicitly*, *implicitly* and *parametrically*. In other contexts well known and very useful, explicit representations (i.e $y = f(x)$) do not play any major role in computer graphics in general. Implicit representations (i.e. $f(x, y) = 0$), too, are not widely used, although they are more common. In difference to the former, implicit surfaces are able to have multiple-valued functions.

Whereas parametric curves and surfaces are an ideal candidate for computer graphics, since these are not axis dependent, it is possible to represent multiple-valued functions and last but not least it is easy to define bounds of the curve/surface by limiting the parametric domain. The following equation is a simple example for a parametric line in 2D space:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c_{x1} \\ c_{y1} \end{pmatrix} t + \begin{pmatrix} c_{x2} \\ c_{y2} \end{pmatrix}$$

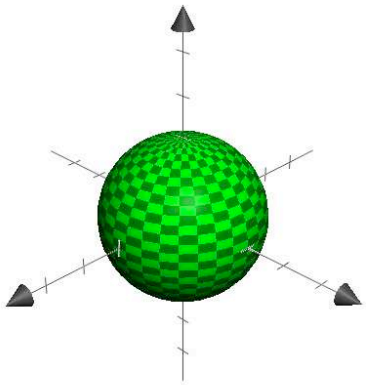

Figure 2.1: Sphere defined by parametric equation 2.1

where c are numeric constants defining the slope and position of the line and t is the parameter. For every value $t \in \mathbb{R}$ the resulting point lies on the line. The parametric domain is usually defined between 0 and 1, however it is also possible to use arbitrary values.

The line from the last example can be extended to a surface easily by adding another parameter where the degree can also be raised without much effort. The next example shows the parametric equation for a 3D sphere. Figure 2.1 displays the result.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \sin u \sin v \\ \sin u \cos v \\ \cos u \end{pmatrix} \tag{2.1}$$

with $0 \leq u \leq \pi$ and $0 \leq v \leq 2\pi$. Although parametric surfaces are quite flexible and powerful, there still are an unlimited number of surfaces that can not be expressed analytically by a single surface. The body of an aircraft, For example, is not modeled with a single surface, but with a number of piecewise surfaces similar to a patchwork. These patches are defined in a way that they join each other along their edges.

2.1.1 Continuity

In relation to parametric curves and surfaces there are two different kinds of continuity: *geometric* and *parametric*. The latter is more restrictive than the former. Both are an indicator for the smoothness of two curves or surfaces joining each other.

Geometric Conitinuity

Geometric continuity is, first of all, simply physical continuity. When two curve segments are joined at their end-points that is called G^0 continuity. If additionally the tangent vectors have the same direction at the join, then this is called G^1 continuity.

Parametric Continuity

Basically parametric continuity is the same as geometric, however additionally not only the direction, but the magnitude of the tangents as well have to be equal at both segments. In that case this is C^1 continuity[1]. In general C^n continuity requires equal direction and magnitude of both tangents at the join after differentiating n times. Always C^n implies G^n but not necessarily the other way round, except for $n = 0$.

Continuity is obviously quite important, if surfaces are composed out of several patches. Although G^0 continuity is essential to avoid visible holes and cracks between patches, often a higher continuity is needed for more advanced lightning effects.

2.2 B-Splines and NURBS

NURBS curves are parametric curves which are more powerful than Bézier and B-Spline curves. However, the only difference to the latter is, that NURBS offer an additional degree of flexibility by introducing control point weights as a fourth dimension. Mathematically a NURBS curve is given by

$$C(t) = \sum_{i=1}^{n+1} B_i^h N_{i,k}(t) \tag{2.2}$$

The B_i^h are control vertices given in four-dimensional homogeneous coordinate space. Thus, a NURBS curve actually is a non-rational (i.e. polynomial) B-Spline curve in four-dimensional space, which has to be back projected into conventional three-dimensional space. The $N_{i,k}$ are the recursively defined basis functions as stated in equation 2.3 (known as the Cox-de Boor recursion formulas - see [8] and [7]) - they are the correspondent to what the Bernstein basis functions are for Bézier curves.

$$N_{i,k}(t) = \frac{(t - x_i)N_{i,k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t)N_{i+1,k-1}(t)}{x_{i+k} - x_{i+1}} \tag{2.3}$$

[1] C^0 is equal to G^0 requiring only the same end-points

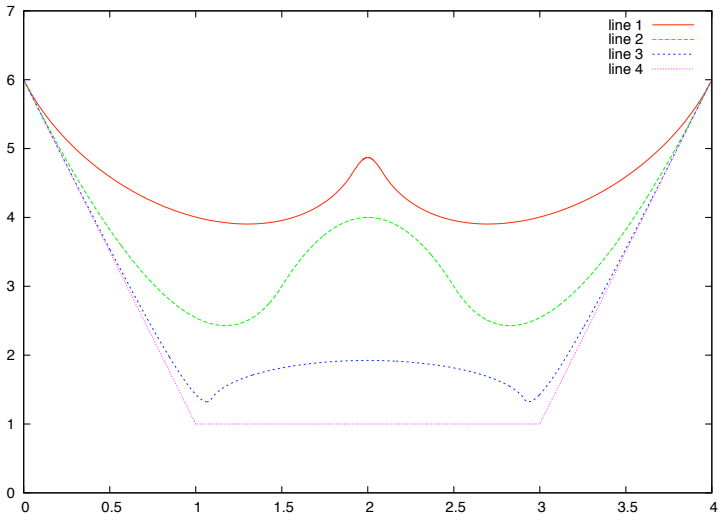

Figure 2.2: The same NURBS curve with different weighting factor at B_3 $(2/5)$. From top to bottom the weights are: 1000, 1, 0.1, 0

The recursion will stop when k equals 1. In this case equation 2.4 applies.

$$N_{i,1}(t) = \begin{cases} 1 & \text{if } x_i \leq t < x_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

This recursive definition requires the convention $\frac{0}{0} = 0$ to be applied. It is mandatory to avoid division by zero. Figure 2.2 shows four exemplary NURBS curves, which have the same control points and identical knot vectors but differ in the weighting factor of a single control point.

2.2.1 Knot Vectors

The x_i found in equation 2.3 are elements of a knot vector. The choice of the knot vector has a direct influence on the basis functions and therefore on the curve itself. The number of elements of a knot vector is determined by the sum of the number of control points and the order of the basis function. Furthermore the knot vector values have to be monotonically increasing, i.e $x_i \leq x_i + 1$. There are two different kinds of knot vectors *periodic* and *open*, where both of them can either be *uniform* or *non-uniform*. A uniform knot vector has equidistant values usually beginning at zero like the following example shows

$$[0\ 1\ 2\ 3\ 4\ 5\ 6]$$

Often knot vectors are normalized

$$[0\ 0.33\ 0.66\ 1]$$

Curves with open or periodic knot vectors define slightly different curve shapes, however the biggest difference is to be found in the start- and endpoints. The beginning of a curve with an open knot vector lies always exactly on the very first control point, respectively the end is found on the last. Periodic knot vector curves unfortunately do not start and end at a specific control point, so they are more complicated to handle for modeling purposes. Periodic knot vectors are not considered any further in this work, nevertheless there is not an architectural issue that would prevent an easy and straightforward integration of these in the library/application presented here.

Open knot vectors **always** have a multiplicity of knot values at both their ends, equalling the order k of the basis function (k =degree+1). The internal knot values are either evenly spaced or not depending wether the vector is *uniform* or *non-uniform*. The first example shows an open uniform knot vector for $k = 3$ and the second shows an open non-uniform knot vector for $k = 4$

$$[0\ 0\ 0\ 1\ 2\ 3\ 4\ 4\ 4]$$

$$[0\ 0\ 0\ 0\ 0.5\ 1.2\ 3.8\ 4\ 4\ 4\ 4]$$

2.2.2 Basis Function Dependancy

A basis function of some given order k depends on lower basis functions of order $k-1$, with the exception for order 1 which is either 0 or 1. Note that the Cox-de Boor formula is defined recursively (see equation 2.3). The dependancies reveal a triangular pattern as follows

$$\begin{array}{lllllll} N_{i,k} & & & & & & \\ N_{i,k-1} & N_{i+1,k-1} & & & & & \\ N_{i,k-2} & N_{i+1,k-2} & N_{i+2,k-2} & & & & \\ \vdots & & & & & & \\ N_{i,1} & N_{i+1,1} & N_{i+2,1} & N_{i+3,1} & \cdots & N_{i+k-1,1} \end{array} \tag{2.5}$$

It is obvious, that even for low order NURBS curves the basis functions can become quite expensive to compute. The efficient evaluation of all involved basis functions is one of the most important challenges, since this is an essential task which has to be performed several million times in a second in order to achieve interactive frame rates. In addition, the basis function evaluation must be as exact as possible, unfortunately strictly excluding approximative approaches, since the Newton iteration (5.2.2), used for the intersection test, is itself a numeric method. It is not advisable to feed it with imprecise data since errors would accumulate fast.

2.2.3 Weighting Factors

Basically the only difference between NURBS and non-uniform B-Splines is the extended flexibility of the former by introducing weighting factors. Usually control points with weighting factors are represented as homogeneous coordinates in four dimensional space. If all weights of a given NURBS surface equals 1, then the surface is just the same as a non-uniform B-Spline surface with identical control vertices. However, increasing a weight at any control point bends the surface towards that point as long as the weight is greater than one. Respectively negative weighting factors will push the surface away from the control point. A value between zero and one indicates a weaker influence of the appropriate control point, while still not pushing the surface away. Finally, assigning zero as a weight causes this control point to have no influence on the curve at all.

In order to have a curve defined in common 3D space the curve has to be back projected from homogenous coordinates into common spacial dimensions. Equation 2.2 yields

$$C(t) = \frac{\sum_{i=1}^{n+1} B_i h_i N_{i,k}(t)}{\sum_{i=1}^{n+1} h_i N_{i,k}(t)} \tag{2.6}$$

As mentioned before, negative weighting values can also be used as well, although they barely appear in real life circumstances, since they are quite unpredictable in their behavior and additionally introduce some more or less serious problems. In nearly any case it is more feasible to describe a surface by a more complex one, that uses positive weighting values (or even none at all). The problems mentioned are as follows:

- The convex hull property may be destroyed
- Singularities may occur
- The surface shape will become quite unpredictable

The convex hull property can be important for the generation of bounding boxes (see section 3.2.2), depending on the employed technique. Although their usage is really not suggested they are fully supported by the work presented here.

2.2.4 NURBS Properties

Since NURBS are a generalization of non-uniform B-Splines, they largely have the same characteristics. The most important are shortly outlined here

- All basis functions are zero or positive for all valid parameter values

- For any parameter value the sum of all basis functions is precisely one for that value
- The maximum order equals the number of control points in that parametric direction
- For all weights $h > 0$ the curve or surface lies in the convex hull which is formed by the union of k successive control point vertices (see next section 2.2.5)

2.2.5 Convex Hull of B-Splines and NURBS

The convex hull property can be exploited for determining exact bounding volumes which are, of course, very important for any ray tracer (also see 2.6). However, convex hull determination is a bit more sophisticated compared to the same process for Bézier surfaces, where these can be created simply by taking the minimum and maximum coordinate values of all control points of a single surface. This could be applied to B-Splines as well, although this potentially will yield wrong results on some special occasions. Both approaches are investigated later in section 3.2.2.

The convex hulls for B-Splines are dependent on the degree of the basis function of a specific curve or surface. The higher the degree, the larger the hulls will become. Figure 2.3 illustrates the effect of the basis function order on the convex hull. The two most extreme cases are

- $k = 2$: If the basis functions are linear segments, the convex hull is completely identical to the control polygon.
- $k = \#ControlPoints$: If the basis functions have maximal degree (i.e. order is equal to the number of control points) the convex hull is just the same as it would be in case of a Bézier curve/surface.

Note, that the curve or surface is usually not bounded by a single convex hull, which would only happen in the latter case. The number of convex hulls is equal to the intervals defined by the knot vector. For example a B-Spline curve with open knot vector $[0\ 0\ 0\ 1\ 2\ 3\ 4\ 4\ 4]$ would be bound by four different convex hulls.

The convex hull properties for NURBS are nearly identical to the B-Splines properties. Positive weighting factors will never conflict with the properties described above. Negative values instead may indeed do so, since singularities can occur, which naturally will break any convex hull. In section 3.2.2 an improved approach for bounding box generation will be presented, which is not dependent on the convex hull properties, thus offers a robust bounding volume generation for all kinds of surfaces.

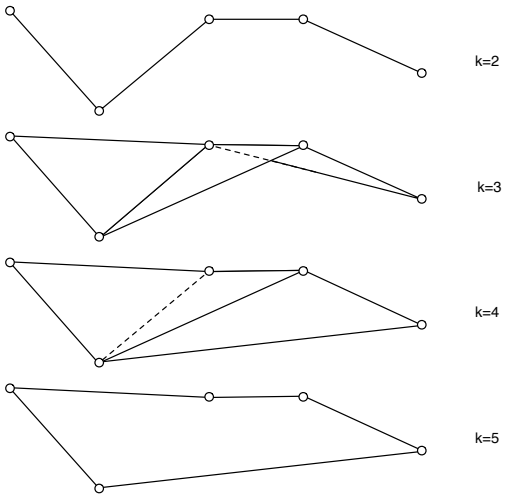

Figure 2.3: The convex hull of a NURBS curve which is defined by five control points. Varying order of the basis functions shows the effect on the size and number of convex hulls. The smaller the order the closer the convex hull bounds the actual curve

2.2.6 Curve Refinement

As described in the previous section, bounding boxes can be created by using a the convex hulls. However, depending on the surface, more bounding boxes than that are needed most of the time (the reason is explained in section 5.2). By subdividing a curve/surface it is possible to generate more convex hulls which in turn creates more boxes.

Refinement of NURBS requires to add more control points which increases the number of knot vector intervals. The surfaces' shape remains identical, of course, by offering a higher degree of flexibility. Basically there are two common different ways to refine a NURBS curve.

First, the flexibility of NURBS can be increased by raising the degree of the basis functions, called **degree elevation**. This, however, will produce larger convex hulls and even reduces their number, which is contra productive in this context! In addition to that, the order of the curve can not exceed the number of control points, thus their number might have to be increased as well which again raises complexity further, resulting in longer computation time for any operation on the curve/surface.

The second approach is called **knot refinement**. The idea is, to simply split a polynomial segment into two piecewise new polynomial segments

for that given interval. This will create more parametric intervals, which in turn can be used to create more bounding boxes. However, in many cases it will be necessary to initially refine over the whole curve/surface not only a specific parametric interval, thus multiple knots have to be inserted at once. Here the Oslo algorithm (see [10] for more detail) is shortly outlined, as this algorithm is capable to insert multiple knot values in a single step, which is to favor over the method developed by Boehm et al. [40], which can only insert one knot value at a time. Recall equation 2.2 now with the following assigned knot vector

$$[x_1 \;\; x_2 \;\; \dots \;\; x_{n+k+1}]$$

Generally by inserting an arbitrary number of new knot values the new knot vector will become

$$[y_1 \;\; y_2 \;\; \dots \;\; y_{m+k+1}]$$

where m has to be greater than n. Thus the new curve can be described as

$$D(t') = \sum_{j=1}^{m+1} C_j^h N_{j,k}(t')$$

As stated above the shape of the curve should be exactly the same afterwards, so the C_j^h need to be computed such that $C(t) = D(t')$. The new C_j^h are given by

$$C_j^h = \sum_{i=1}^{n+1} \alpha_{i,j}^k B_j^h$$

with $1 \leq i \leq m$ as shown in [33]. The $\alpha_{i,j}^k$ are given recursively, similar to the basis function recursion

$$\alpha_{i,j}^1 = \begin{cases} 1 & x_i \leq y_j \leq x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_{i,j}^k = \frac{y_{i+k-1} - x_i}{x_{i+k-1} - x_i} \alpha_{i,j}^{k-1} + \frac{x_{i+k} - y_{j+k-1}}{x_{i+k} - x_{i+1}} \alpha_{i+1,j}^{k-1}$$

Here, too $\sum_i^{n+1} \alpha_{i,j}^k = 1$ does hold.

If uniform knot vectors should be preserved as such, some special atten-

tion has to be payed: To maintain these, it is necessary to add one new knot value midway in each parametric interval. The knot values are usually multiplied by two beforehand when using integer values to avoid fractions.

For example the knot vector

$$[0\ 0\ 0\ 1\ 2\ 2\ 2]$$

is first multiplied by two (optional)

$$[0\ 0\ 0\ 2\ 4\ 4\ 4]$$

and then the new values are inserted

$$[0\ 0\ 0\ 1\ 2\ 3\ 4\ 4\ 4]$$

A curve subdivided this way has the same degree as before, but more control vertices and a longer knot vector with more intervals. Recall, that the number of knot vectors intervals specifies the number of convex hulls bounding the curve. Effectively every subdivision step doubles the number of intervals, thus unfortunately only offering a quadratically increase in the number of intervals, thus bounding boxes to be generated.

A surface can be subdivided respectively by subdividing all splines of one parametric direction the same way as described above. First subdividing along u parametric direction and afterwards along v parametric direction will yield the same result, as when swapping the order in which it is subdivided.

2.2.7 Derivatives

The derivative for non-rational B-Spline curves at a given point can be obtained by formally differentiating, thus equation 2.2 will yield

$$C'(t) = \sum_{i=1}^{n+1} B_i^h N'_{i,k}(t) \tag{2.7}$$

Obviously only the basis functions need to be differentiated, which again, for B-Splines can be done straight forward yielding

$$N'_{i,k}(t) = \frac{N_{i,k-1}(t) + (t - x_i)N'_{i,k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t)N'_{i+1,k-1}(t) - N_{i+1,k-1}(t)}{x_{i+k} - x_{i+1}} \tag{2.8}$$

However, by having a closer look at equation 2.4, it becomes apparent that now $N'_{i,1}(t) = 0$ for all t. The recursion is now terminated for $k = 2$, so that

$$N'_{i,2} = \frac{N_{i,1(t)}}{x_{i+1} - x_i} - \frac{N_{i+1,1}(t)}{x_{i+2} - x_{i+1}} \tag{2.9}$$

With the new recursion formulas above, the values for the derivatives at any given point can be calculated. Also tangents, of course, can now be computed, which are needed for the previously mentioned Newton iteration as well. The second derivative could be computed respectively, but as it is not needed in this context, it is not discussed here.

For NURBS curves the idea is basically the same, albeit it is a bit more sophisticated, due to the fraction that occurs with rational B-Splines. By rewriting equation 2.6 into the shortcut notation $\frac{n}{d}$ with n denoting the nominator and d the denominator, the first derivative can be expressed as

$$C'(t) = \frac{n'd - nd'}{d^2}$$

where n' was already derived above (see equation 2.7) and d' can be computed just the same way.

2.2.8 NURBS Surfaces

NURBS surfaces are, obviously, the logical extension of their curve counterpart. Represented as a cartesian product, the following equation describes a NURBS surface (compare equations 2.2 and 2.6)

$$S(u,v) = \frac{\sum_{i=1}^{n+1}\sum_{j=1}^{m+1} B_{i,j}h_{i,j}N_{i,k}(u)M_{j,l}(v)}{\sum_{i=1}^{n+1}\sum_{j=1}^{m+1} h_{i,j}N_{i,k}(u)M_{j,l}(v)} \tag{2.10}$$

Here again, the $B_{i,j}$ are the three dimensional control vertices, $h_{i,j}$ are the corresponding weight values, $N_{i,k}$ are the basis functions in the u parametric direction and $M_{j,l}$ likewise for the v direction. The properties of NURBS surfaces are analogue to the properties for NURBS curves, so they need not to be repeated here.

NURBS cover both Bézier and B-Splines where the latter are simply NURBS with all weighting factors equaling one. Bézier curves and surfaces are indeed a special case of NURBS, when the basis functions actually reduce to the Bernstein polynomials, which is the case, whenever the order of the basis functions equals the number of control points and an open knot vector is employed, for instance

$$[0\ 0\ \ldots\ 0\ 1\ 1\ \ldots\ 1]$$

The multiplicity of zeros and ones is corresponding to to the order of the basis. In contrast to Bézier surfaces, NURBS are much more flexible and powerful, thus allowing a more complex geometry description with less surfaces, which in turn means lesser memory consumption.

Subdivision or bounding box creation for surfaces is quite similar to the same process for curves, only that the computations have to be done for two parametric directions for a single surface. For example a NURBS surface with a 5x5 control point net defines actually five NURBS splines in both u and v parametric direction. Subdividing such a surface requires to subdivide every of the five splines in one parametric direction and then subdividing the resulting surface again in the other parametric direction. Each spline is subdivided accordant section 2.2.6, only that is has to be done ten times for such a surface.

2.2.9 NURBS Surface Derivatives

Although NURBS derivatives are computed in a similar way as their non-rational counterparts, the process is a bit more complex, thus it might be worthwhile to discuss it in more detail. When computing the derivative of a surface which is defined in two parametrical directions the partial derivatives are required. The partial derivative of a NURBS surface in u parametrical direction can be expressed as

$$S_u = \frac{n}{d}\left(\frac{n_u}{n} - \frac{d_u}{d}\right)$$

where again d denotes the denominator and n the nominator of equation 2.10. Respectively

$$S_v = \frac{n}{d}\left(\frac{n_v}{n} - \frac{d_v}{d}\right)$$

where

$$n_u = \sum_{i=1}^{n+1}\sum_{j=1}^{m+1} B_{i,j} h_{i,j} N'_{i,k}(u) M_{j,l}(v)$$

and

$$d_u = \sum_{i=1}^{n+1}\sum_{j=1}^{m+1} h_{i,j} N'_{i,k}(u) M_{j,l}(v)$$

Respectively

$$n_v = \sum_{i=1}^{n+1}\sum_{j=1}^{m+1} B_{i,j} h_{i,j} N_{i,k}(u) M'_{j,l}(v)$$

and

$$d_v = \sum_{i=1}^{n+1}\sum_{j=1}^{m+1} h_{i,j} N_{i,k}(u) M'_{j,l}(v)$$

Here the derivatives of the basis functions N' and M' are identical, of course, to the derivatives as they were introduced for single curves in section 2.8.

Algorithm 1: Cg example code

```
void main( float2 inCoord : TEXCOORD0,
   out float4 outColor : COLOR,
   uniform samplerRECT : texture)
{
   outColor = texRECT(texture, inCoord);
   outColor.a = 0.5;
}
```

2.3 Shading Languages

Since this work targets to present a cross-platform solution, one of the well known shading languages can not be considered, which is Microsoft's DirectX [9], because it naturally runs on x86 Windows machines only. This basically leaves a choice between Nvidia's Cg (2.3.1) and the OpenGL Shading Language (2.3.2), which both are available for Mac OS X, Linux as well as Windows.

2.3.1 Cg

Cg (C for graphics) [24] was 2003 introduced by NVidia as a complete developing platform for GPU programming. In principle all programmable graphic boards are supported (for example GeForce FX and ATI Radeon 9700 generation and above), but for ATI cards support is limited to ARB multivendor specifications. This means for more advanced vertex programs or fragment shaders a current NVidia card is required. For example, floating point textures are not available within the ARB specification.

Cg offers a C like syntax so it is very convenient to start with. The code snipped 1 shows a basic fragment program that assigns a color value sampled from a texture and assigns a 50% transparency.

By using a high level language like Cg it is no longer necessary, even not worthwhile in most cases, to write assembler code. Another general positive aspect of NVidia's Cg is, that it can compile code for both major graphic APIs OpenGL as well as DirectX.

2.3.2 OpenGL Shading Language

With the release of OpenGL specification 1.5, support for programmable vertex and fragment shaders was added by introducing the OpenGL Shad-

Algorithm 2: OGSL example code

```
const uniform sampler2DRect texture;

void main()
{
   gl_FragColor = texRECT(texture, gl_TexCoord[0]);
   gl_FragColor.a = 0.5;
}
```

ing Language [30] (GLSL). Since OpenGL is installed already on nearly every modern system, usually nothing particular has to be done in order to develop vertex and fragment shaders. Cg, however, requires the *Cg Toolkit SDK* which has to be downloaded and installed. Shaders written in Cg need to link against this library. Similar to Cg, OGSL has a C like syntax. The GLSL example 2 will have the same effect than the Cg example. Comparing the example code 2 with 1, the most striking difference is the missing parameter list in the latter. Textures and the like are referenced as global variables so they do not need to be passed to every function within the program. Other values like texture coordinates and output colors are referenced as OpenGL global build-in variables which is quite comfortable. However, the real advantages of OGSL are to be found in excellent support for both NVidia and ATI cards, so by choosing OGSL as the language of choice, there is a broader range of graphic cards that will support the full feature set. Additionally OGSL offers SIMD commands similar to the ones that will be used in the code for the CPU. Of course Cg supports SIMD operation as well, but it lacks some important features. For example the *bool4* data type is missing in Cg, however OGSL offers not only that, but a full set of vector relational functions as well, like

```
bvec4 lessThan(vec4 a, vec4 b)
```

or

```
bvec4 any(bvec4 a)
```

This makes it easier to port the CPU code to the GPU as the commands nearly map one-to-one. The intersection code was partially developed in both shading languages. There was no real difference in performance, however the code with OGSL is much cleaner, more maintainable and also easier to synchronize with the code implemented for the CPU.

2.4 SIMD Instruction Set

SIMD is an extension offered by most of the current CPU models and architectures. Intel's implementation found on Pentium II processors and above is called *SSE* and *SSE2* on Pentium IV class processors respectively[2]. Motorola's G4 as well as IBM's G5 processors feature *AltiVec* (i.e. named *VelocityEngine* by Apple) which is also a SIMD implementation. Basically both different instruction sets offer roughly the same features, but one may lack some specific instruction of the other and vice versa. For example AltiVec offers an instruction that will multiply a and b and adds c to the result which is perfectly suited for dot products, but SSE lacks such a command and has to combine the common multiply and add commands. However, there is no simple multiply with AltiVec, which means a multiplication has always to be implemented as a *MultiplyAdd* where c equals zero. Both have in common that they do not work on common data types like *int* or *float*, but on vector data with a fixed witdh of 128 bits. Dependent on the processor the multiplication of two float values may take a specific number of cycles. Apparently four multiplications will take four times as long (at least when only one FPU is available). By using SIMD instructions these four multiplications can be carried out in parallel by storing four values each in a SIMD vector ($4 * 32$ bit = 128 bit), then one multiplication command performs the operation on the whole 128 bit data vector, effectively yielding four results in parallel (see 2.4). However, the expected speed up of four can not be reached in real life circumstances because not every computation can be carried out in parallel and additionally often a small (or not so small) overhead computation is required. For example, writing values into a SIMD register is expensive, because data need to be aligned in memory along 16 byte boundaries.

2.5 Trimming

All parametric surfaces tend to have a rectangular-like shape, which is due to the rectangular parametric domain. By carefully choosing a certain knot vector, it is possible to overlay multiple control vertices at the same location in space, which can reduce the above mentioned effect slightly. However, it is much more efficient and easier to model non rectangular like shapes by using so called *trimmed* surfaces. Certain regions in the parametric domain are simply defined as invalid, thus all surface points defined by these invalid parametric values are considered not to be part of the surface, effectively cutting the regions out.

[2] SSE3 can now be found on some of the most current processors from Intel

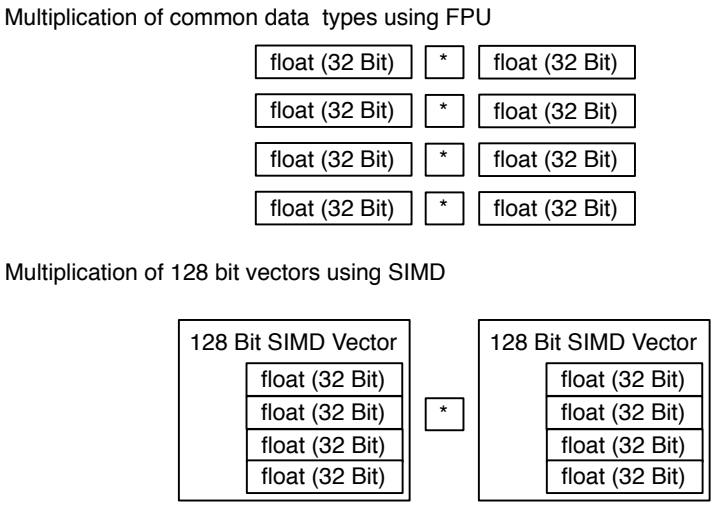

Figure 2.4: Four multiplications of common float data type compared with SIMD instruction multiplying of vector data

A very common example is the window of a car. Usually it is simply modeled as a curved, rectangular surface, where the actual shape of the window is defined by trimming curves, thus cutting out the dispensable regions at the borders. Of course it would also be possible to model this window without using trimming curves, however a higher resolution of the control point grid would be required, as at every sharp edge a multiple of control points equalling the basis function order is necessary, making the surface in its whole a lot more complex than necessary. Figure 2.5 is illustrating the trimming process.

Because trimming is very powerful and convenient it has become a very popular feature in all CAD/CAM and 3D modeling package applications. However, when it comes to real time rendering of these models there are several issues. The biggest problem possibly is to generate a robustly tessellated mesh from trimmed surfaces to use with modern graphic cards for real time rendering. Until today there is no algorithm or concept which assures an artifact free tessellation of high quality. Often visual glitches like gaps and cracks occur. Additionally depending on the size and resolution of the model such a tessellation can be quite costly regarding resources and time.

Naturally, these problems will not appear with the ray tracer presented in this work. A tessellation is, of course, not necessary at all. Instead after a

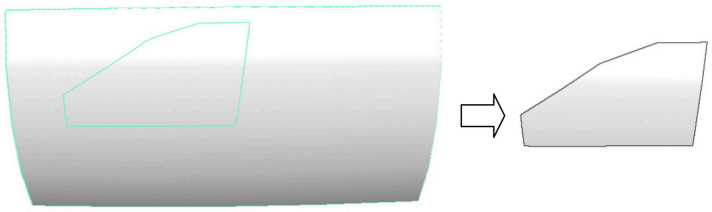

Figure 2.5: Trimming of a window of a car

potential hit point has been found, it is only necessary to check wether the u and v values are within a valid parametric region or not. In the latter case the hit point is simply discarded and tracing continues as if no hit point was found at all. This test is basically performed by an optimized point in curve test. Trimming will be discussed in detail in section 5.4

2.6 Ray Tracing

This topic alone fills whole books, so in this context only some very basic concepts can be taken into account. For a more detailed discussion refer to [35], for example.

Ray tracing is, next to radiosity and scanline rendering, one of the most popular methods to generate images. The first ray tracing algorithm was developed in the late 1960s. Since then it was, of course, dramatically enhanced, so that today, there are a lot of different variants for different purposes. By using the ray tracing technique it is possible to render images which are physically extremely correct, for instance, it is possible to compute the exact amount of incident light on a virtual model at a specific point. Ray tracing is nowadays even used in the automobile industry to test wether a car prototype will have irritating reflections on the windscreen or not, before it has been build at all.

2.6.1 Basic Idea of Ray Tracing

Ray tracing is a (global) illumination rendering technique. A ray is shoot from the virtual camera ,i.e. the eye, through each picture element, which usually is a single pixel of the virtual image plane. Each ray is tested for intersections with all objects in the scene. Whenever not a single object is hit, this pixel is shaded in a defined background color. Most of the time several objects are hit, but obviously only the intersection with the closest distance

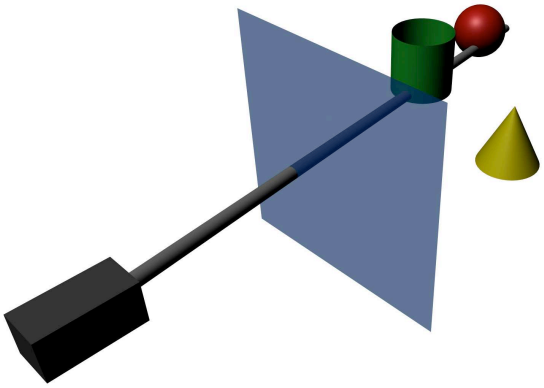

Figure 2.6: Basic principle of the ray tracing algorithm

to the camera is valid (unless that one is transparent). The incoming light at that location will be computed afterwards, finally yielding a color value based on the material properties and the light that effectively is transmitted from all light sources in the scene to that point. The range of different methods that can be applied for these individual computation steps are ranging from very simple and fast to highly complex. On the on hand, often simple *Phong shading* [4] is used for computing the color value which is not physically correct at all. On the other hand, sophisticated algorithms which, for example, take the BSSRDF [20] (Bidirectional Scattering Surface Reflectance Distribution Function) of the surface into account, can be used as well. Whatever method is used for shading, the ray tracing framework lying underneath does not need to be changed.

Ray tracing systems generally are able to handle shadows, multiple reflections, refractions and texture mapping with ease in a very straight forward manner. Additionally global illumination can also relatively easy be implemented by different variants of Monte Carlo ray tracing [32]. Figure 2.6 shows the basic principle of ray tracing. The ray emerges from the camera (black box on the left side) and intersects two objects in the scene, the green cylinder and the red sphere. However, the intersection point with the green object is the closest to the camera, so the color green is assigned to that picture element in the blue plane that is hit by the ray. This is simply done for every picture element in the blue image plane. Extending this approach to also incorporate effects like shadows and reflections for instance, is quite easy. If a surface, that has been hit, is reflective then a new ray has to be

cast from that point by calculating the reflected direction and recursively calling the trace routine with the newly computed ray. The result of this trace is not used as a color value for the image plane, but scaled according to the reflectivity of the previously hit surface and added to its color. Transparency can be treated in just the same way by eventually computing two rays, one refracted and one reflected according to Snell's law. Shadows are even computationally easier. From a hit point a ray is casted towards the light source. If any objects lies in between then no light from this source is transmitted to the intersection point. If that is the case for all light sources in the scene, this point will be shaded black (respectively in an ambient color).

2.6.2 Ray Tracing Acceleration Techniques

Ray tracing has the reputation of being very slow. However, by carefully implementing several acceleration techniques, ray tracing can become fast enough for an interactive setting nowadays. In turn this would mean, that without these, there would not be the slightest chance to become fast enough.

Bounding Volume Hierarchy

By reconsidering what was said in the last section, it becomes obvious that the computation time to render a single frame would increase linearly with the number of objects (i.e. triangles, NURBS surfaces etc.) in the scene as every single object has to be tested for an intersection with the ray. This can be overcome by the usage of bounding volumes, where a single volume is represented by an easy geometry like a box or a sphere. This volume contains a number of objects and instead of the individual objects, the bounding volume itself is tested for an intersection. If the volume is missed, so are all contained objects. These volumes themselves can be organized in larger volumes thus creating a bounding volume hierarchy. For example if the topmost volume, containing the whole scene, is missed, then the trace routine for the current picture element will terminate after only one intersection test with a box!

There are a lot of different approaches and techniques for more or less efficient bounding volume hierarchies. A short overview as well as the approach taken in this work can be found in section 3.2.

Parallelism

Ray Tracing is extremely well suited for parallel execution, since every picture element that is being processed, is totally independent on all others. Basically it would be possible to have a single processor for every picture element. Unfortunately such machines are not available yet, but still it is very attractive for multi core and/or multi processor systems. Even a cluster can be feasible, although network latencies can be a serious problem when targeting at interactive frame rates. The application *dream*, which is the foundation of this work, has indeed been extended with network rendering capabilities. The scalability is still a subject to improvement, however, first results justify further effort in that field. More details can be found in [3].

Fast Intersection Test

Obviously the intersection test itself is quite important, as it will be called several million times for a single frame, depending on the frame size and scene itself. Regarding triangles, there are lots of different intersection algorithms, each with their own advantages and disadvantages in performance, memory consumption, preprocessing time and other factors. However, for NURBS surfaces the choices are rather limited, as this field was not so much worked on, compared to the ray tracing of triangles. Any improvement in the intersection test usually has a direct impact on overall performance of the whole system.

2.7 Basic Application Design Decisions

The software employs a range of different modules which might either be used or not, in order to be able to compare results with other approaches of ray tracing applications and also for the purpose of running it on older hardware as well as uncommon architectures like the Itanium processor which does not feature a SIMD instruction set. The most important modules are the intersection computation modules which there are three of them, namely a *FPU* (Floating Point Unit) emulation mode, *SIMD* and *GPU*. Figure 2.7 shows how these modules collude. The FPU emulation mode is a lot slower compared to SIMD powered execution. However, the FPU module is also a good candidate to compare results with completely different ray tracers as most of them use the FPU only. Last but not least by using the emulation mode, the application will run on every of the rack computer that has been sold in the last ten years as it's only requirement is a floating point unit, though performance might be far from interactive in that case, but still it would be possible to use the application as a fast, but non-interactive, NURBS ray tracer anyway.

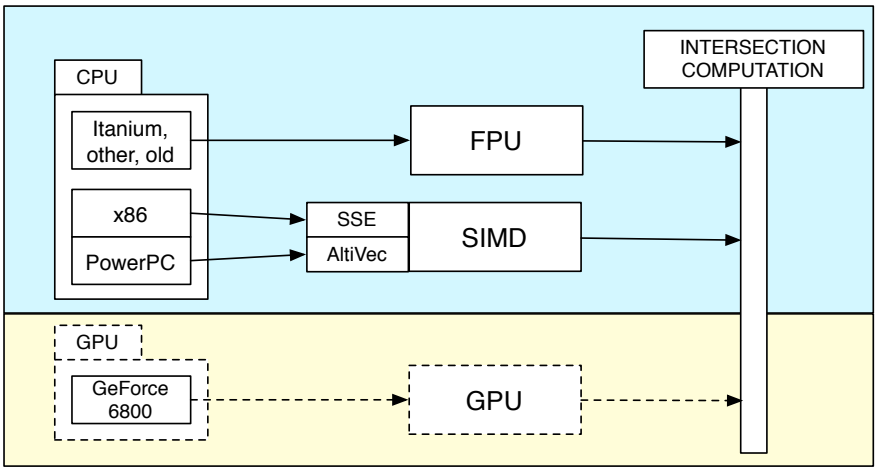

Figure 2.7: The upper blue part shows module choices for the CPU. The lower yellow part for the GPU is optional. The dashed lines indicate, that the feature set is not yet comparable to the CPU implementation

If available, the SIMD unit takes advantage of either SSE or AltiVec depending on the processor type. Usage of SIMD instructions boosts the performance by more than 400%[3] compared to the emulation mode and should be used if SIMD extensions are available to the processor (beginning with Pentium II or G4 class processors). Note that, of course, only either FPU or SIMD can be used at a time, not both!

Finally, usage of the GPU is optional and additional to the above mentioned modules. However, as stated earlier at least a GeForce 6800 graphics board is needed, as older cards do not offer some of the needed functionality which is inevitable for the employed algorithm.

Figure 2.8 shows how the individual components work together. Basically any software that can handle NURBS data and supports the IGES file format can be used to create scenes for crianusurt. Alias's Maya for instance is a software offering these features[4]. Both applications *iges2dsd* and *crianusurt* (chapter 5) rely on functionality offered by the libraries *libNURBS* (section 4.2) and *libSIMD* (section 4.1). Note that libSIMD uses only one of

[3]Theoretically the maximum speedup is factor 4, but the emulation mode is an emulation as the name says, so there is some additional overhead involved. An optimized FPU based ray tracer would be a bit faster than 25% of SIMD performance

[4]The IGES importer/exporter module has to be activated in the plugins preference panel. By default this plugin is deactivated

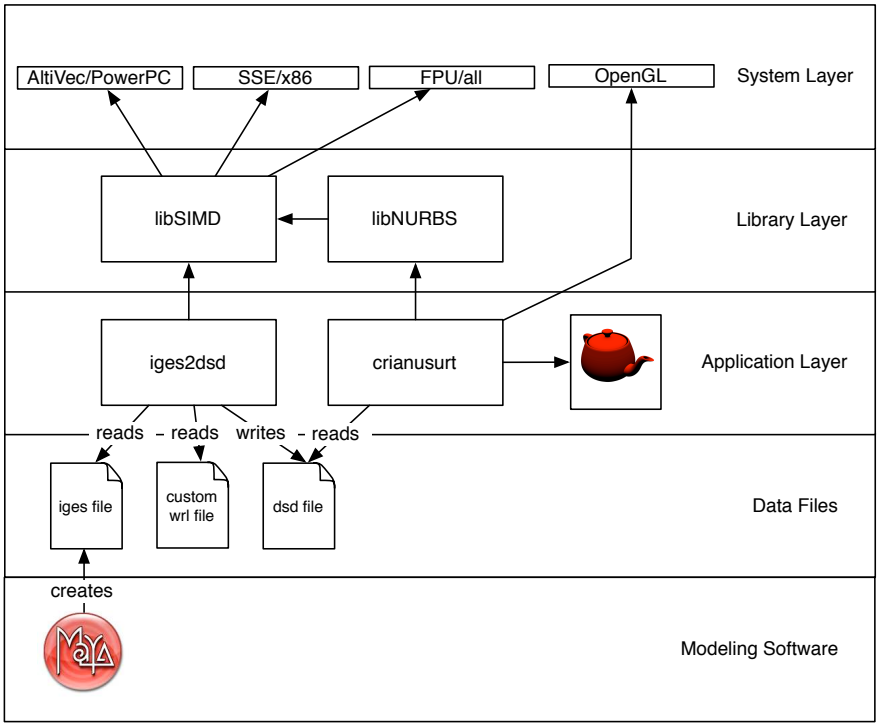

Figure 2.8: The system components and their relation to each other

the three modules (AltiVec, SSE, FPU) at once. In order to change the used module, the library has to be recompiled. Both, *libSIMD* and *libNURBSIntersectionKernel* can be used by any other application as well.

Chapter 3

Preprocessing

The task of rendering NURBS surfaces need quite some preprocessing computation in order to get the intersection test as fast as possible. The following sections will give a overview about different ways on how the evaluation of NURBS surfaces can be performed and motivates the approach taken in this work. The application *iges2dsd* is responsible for the preprocessing. *Iges2dsd* is an application on its own and totally independent from the main rendering application. As input data it can take IGES files (see section 3.1.1) or custom VRML-like files (section 3.1.2), as they are used with the *Open Ray Tracing* (OpenRT) [31] system of the realtime ray tracing working group in Saarbrücken. After preprocessing is completed a custom binary *dsd* file is written. This file can be read later by the render application, thus the preprocessing step is, of course, necessary only once, unless for example the parameters for the bounding box volume generation are changed.

After giving a general introduction to different bounding volume hierarchies and space subdividing approaches in section 3.2, two variants which were implemented are discussed in more detail. Section 3.3 investigates the necessart pre-processing for trimming curves, where section 3.4 is about the improvement of the surface evaluation methods, which is very important for this work. Finall section 3.5 is about the memory layout for both, CPU and GPU, which differ from each other naturally.

3.1 File Loader

Currently the loader can read two different input files. However certain restrictions apply, because NURBS surfaces are the only supported primitive[1]. VRML and especially IGES offer a lot of functionality which is not

[1] Bicubic Bézier surfaces are also supported as a special case which speed up the intersection test significantly

needed in this context. Any unknown entity encountered during loading will simply be ignored, thus adding triangles to the scene, for example, will not stop the loader from working, but it will not change the rendered scene in any way. The following sections give a quick overview of the capabilities.

3.1.1 IGES File Format

The IGES file format is a good candidate for data exchange from any modeling application like Maya [2] to *crianusurt*. It is possible to store NURBS surfaces lossless, where most other file formats only store triangular approximations of free form surfaces. IGES entity type 128 is reserved for storing rational B-Spline surfaces. This is the only geometric entity parsed by the loader[2], except for trimming curves. As mentioned above, every other entity found will be ignored.

3.1.2 Customized VRML File Format

The above mentioned working group of Philipp Slusallek extended the VRML file format with a new tag to store bicubic Bézier surfaces. These files can be read by *iges2dsd* to test the optimized rendering of bicubic Bézier surfaces. However an explicit representation by NURBS can be enforced. Additionally the light sources and the camera stored in the file are also interpreted.

3.2 Bounding Box Volume Hierarchy Generation

Basically there are currently two different, concurrent classes of methods on how to reduce the number of objects which have to be tested for intersections. Bounding volume hierarchies are used since the very early days of ray tracing, whereas different space partitioning methods are becoming more popular in recent years.

3.2.1 Space Partitioning

Some of the better known variants that fit into this class are *Oct-Trees* [13], *BSP-Trees* [14] and *kd-Trees* [17].

Oct-Trees

Oct-trees are able to adapt to the scene. In the beginning the whole scene is regarded as one voxel. By employing a heuristic function which may take

[2] Bicubic Bézier surfaces, as a special case of B-Spline Surfaces, are also included in type 128

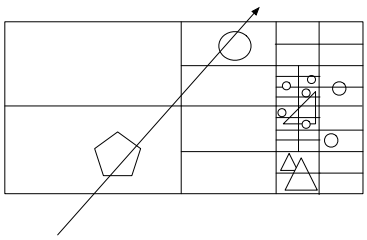

Figure 3.1: Exemplary Oct-Tree subdivision in 2D space. Regions with more objects have a higher density of voxels. The ray has to test four voxels and two objects, however a smart implementation would stop after the first voxel, because it is not possible to find a closer intersection

the number of objects and the depth of the tree into account, the voxel can be split up into eight new voxels of the same size each. This is repeated recursively until no further subdividing is necessary or possible. Regions in space with many objects will have a higher resolution voxel grid, whereas large empty regions are bounded only by a few big voxels. However, it can occur that still some objects lie within more than one voxel so that the tracing algorithm has to keep track of which objects have already been tested to avoid multiple intersection tests on those. Figure 3.1 shows an 2D example of an Oct-Tree (which actually reduces to a Quad-Tree in a 2D case)

BSP-Trees

BSP-Trees (**b**inary **s**pace **p**artitioning) divide the space into two equal sized half spaces by using a plane which separates them. It is possible to use an arbitrary plane for that, but due to a much easier intersection test it is advantageous to use a plane that is perpendicular to one of the coordinate axis. The resulting hierarchy will bound the scene objects a bit closer than an Oct-Tree. However, even better result can be achieved when the dividing plane is not placed in the geometric middle, but in such a way, that a cost function will be minimized. This kind of tree is referred to as k-dimensional trees (k-d trees).

kd-Tree

K-d trees adapt to the structure of the scene extremely well. If two new half spaces are created, then each one will contain the same number of objects[3], thus the number of objects to test against is halved with every subdivision step. This is recursively repeated until a maximum depth has been reached

[3] +/-1 if number of objects in parent space was odd

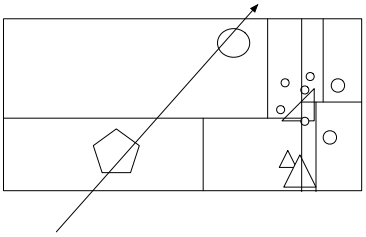

Figure 3.2: Exemplary kd-Tree subdivision in 2D space. The geometric pivot is used to decide wether an object belongs to the one or antother half space. Here two voxels need testing as well as two objects

or a half space has the predefined minimum number of objects. K-d trees have recently become very popular for computer graphics purposes, for example for photon mapping [19], but as well for conventional ray tracing. Even *dream*, the application this work is based on, was also extended by a photon mapping implementation. See [22] for more details.

3.2.2 Bounding Volume Hierarchy

There are a lot of different shapes that can be used as a bounding volume, however, preferably the intersection test should be as fast and easy as possible and the volume is supposed to bound the enclosed objects as tight as possible. With respect to the first requirement, this basically leaves only boxes and spheres as potential candidates, though many others have been employed as well. Spheres have a very fast intersection test, but they tend to bound the objects only relatively loose in most circumstances. In this context axis aligned bounding boxes will be used, because of their formidable intersection test properties. Additionally the subdivision of the parametric domain is much more comfortable and computationally cheaper with axis aligned boxes.

Space subdividing approaches are top down methods, they begin with the whole scene, dividing this space until some termination criteria is met. In theory, bounding volume hierarchies are basically the opposite as they usually begin by looking for objects that are very close by each other and then span a box around them. After all objects have been bounded, several boxes are then bounded themselves by a new box, until the whole scene has finally a single box. However, the more efficient algorithms also take a top down approach, for example the Goldsmith/Salmon algorithm [18] which is employed for this work. More details about this algorithm can also be found in [12]. Figure 3.3 shows a common hierarchy as they are used in many ray tracing systems.

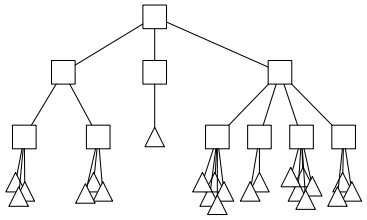

Figure 3.3: An example for a common bounding volume hierarchy. The boxes represent bounding volumes, where each triangle represents a geometric primitve, which is a triangle itself in most cases

All children can be skipped, if the parent bounding volume was not hit. Although in many circumstances space subdividing methods, especially kd-trees, are considered to be faster, bounding volumes were used for this work. One of the main reasons is, that bounding boxes can have a closer relationship to the objects they are actually bounding. As explained later in section 5.2.3 good initial guesses for starting the Newton iteration are required. These can be provided by the bounding volume hierarchy easily. Using a space partitioning scheme however, is not an adequate solution for obtaining such guesses, since there is no direct correlation to the surfaces, only to space itself. Good initial guesses are mandatory, thus bounding volumes, which can provide these, are the only option in this context.

Bounding Volume Hierarchy for NURBS

Basically the hierarchy employed in this work is quite similar to the common hierarchies used in other ray tracing systems, including *dream*, however there is one very important difference. Usually a single bounding volume has a few children which are either other bounding volumes or geometric primitives. In this work however, a single NURBS surface needs to be bounded by a couple up to a few hundred bounding volumes, depending on the complexity of the surface (15 are an average value for low-complexity surfaces). Figure 3.4 shows two examples of scenes, where only the bounding boxes were rendered roughly[4].

A high number of bounding boxes is necessary to achieve good initial guesses for the Newton iteration, but also to avoid unnecessary intersec-

[4]Actually always four neighboring pixels will always yield the same result, i.e. color value, because a specific bounding box render mode was node implemented. Basically this is an interesting side effect, of an extremely large threshold parameter used during the Newton Iteration

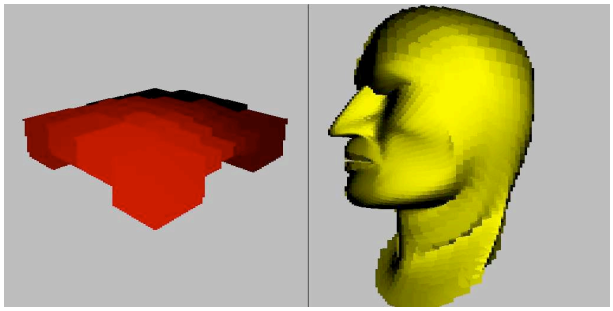

Figure 3.4: The surface on the left side is bounded by 59 boxes whereas the face on the right has 13842. In the latter case the bounding boxes are so small that actually the shape of the object is visibly approximated by them.

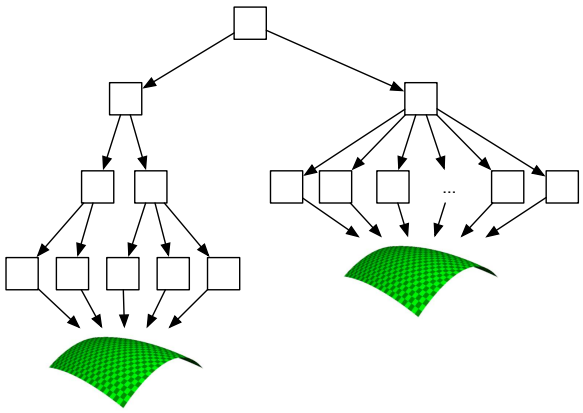

Figure 3.5: An example for the bounding volume hierarchy employed in this work. Every surface is bounded by several volumes.

tion tests with NURBS surfaces. A single test with a NURBS surface is far more computationally expensive compared with a triangle intersection, so it is important to avoid as many unnecessary operations on NURBS as possible, even if some additional bounding box tests are required in turn.

In the first place it seems that this will produce a huge amount of bounding volumes. However, the actual number of generated volumes will still be comparable or even less to common triangles scenes as a single NURBS surface can describe geometry that otherwise would have been approximated by possibly hundreds or thousands of triangles, requiring a large number of bounding boxes themselves.

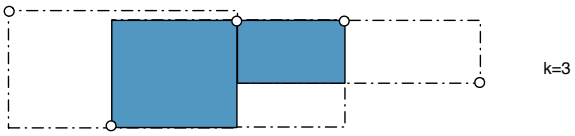

Figure 3.6: Five control points define a spline in 2D space which has cubic degree (spline itself is not shown). Three boxes are bounding the spline, but the two regions colored in blue are bounded twice, which is not optimal

Every single bounding volume that has a NURBS surface as a child will bound a specific parametric region. If the volume is small enough, so is the parametric region, such that for every point on the surface in that region, the Newton iteration will most likely converge. If that is the case, simply the median values for u and v parameter values can be used as initial guess. The problem however, is to find criteria which will assure convergence of certain regions of a patch. The following sections explain two different approaches that were investigated.

Bounding Volume Generation Based on Convex Hull Property

It is possible to generate bounding volumes on behalf of the convex hull property as mentioned in 2.2.5. As long as negative weighting factors are not used, the surface is known to stay within that hull. From the convex hull an axis aligned bounding volume can easy be calculated by computing the minimum and maximum values of all related control vertices for that hull. The problem is, that there will be a lot of bounding boxes that overlap each other within a single surface. The convex hulls themselves will overlap each other, unless the surface's order is equal to the number of control points in that direction, which is, of course, usually not the case[5]. Figure 3.6 illustrates this problem.

Basically there are two major issues. First a specific region may be bound by several volumes which later during ray tracing have to be tested all, thus the same hit point will be reported more than once. This will not produce errors or artifacts, but is an obvious waste of processor cycles! A possible solution to that problem is, to compute the space where bounding boxes overlap and to create a new single volume in that space, replacing the overlapping region and then reducing the original ones. Most of the time the old volumes have to be split up into several new ones. However, this is not trivial and will produce even more bounding boxes, some of them

[5]If additionally the knot vector is open then the basis functions will be equal to the Bernstein functions, which form the basis for Bézier

which might even become useless, as the do not bound a part of the surface anymore. Although this can be done during preprocessing, it will still require a great amount of computation power and leaves another problem: The correct computation of the newly bound parametric domain. Second there is the lack of flexibility of the generation process. The surface can be refined by adding new values to the knot vector, which in turn will increase the number of knot vector intervals, i.e. the number of convex hulls. However this process is quite expensive (with respect to runtime) and it can not be controlled very well. Each time the number of convex hulls will increase super linearly. Other refinement methods, like control point insertion, suffer from similar problems. Subdivision was discussed in section 2.2.6. Whenever any kind of subdivision is performed, the original surface has to be kept, of course, because the refined surface is describing exactly the same surface in a more complex way.

Considering all this, a different approach seems to be (and will prove to be - see results section 7.5) much easier and a lot better in performance.

Bounding Volume Generation Based on Surface Flatness Criterion

The basic idea of this approach is, that a surface, which is very flat, is optimal for convergence with an arbitrary starting guesse somewhere on the surface and thus need not to be refined further. However a surface with strong curvature is most likely a good candidate for further subdivision. Flat surfaces will have normals which are roughly equal in direction to each other. By computing several normals at different positions on the surface it is possible to estimate a measure for the curvature of that surface (or subsurface). In this work eight normals are taken into account as shown in figure 3.7, simply because a multiple of four can take the greatest advantage of the SIMD implementation, i.e. eight normals are roughly computed as fast as normally two would with a FPU implementation!

It is possible to construct special surfaces that would not be handled correctly by this technique - for example when a surface would have an extremely sharp peak in the middle of the surface. However, these surfaces occur rarely in real life and tests of several different scenes have shown no problems. If visible artifacts occur, which are caused by this approach, then still the convex hull based generation mode can be used.

The surface would be perfectly flat if

$$\Pi_{i=1}^{7} n_i * n_{i+1} = 1$$

does hold, where the n_i are the normal vectors. However, whenever $n_i * n_{i+1} < 0$ occurs, the process is interrupted and the result is reported as 0

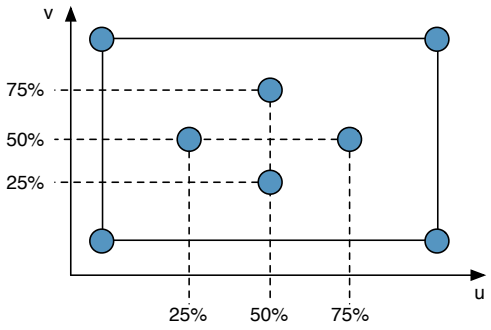

Figure 3.7: The blue circles mark the positions where normals will be evaluated

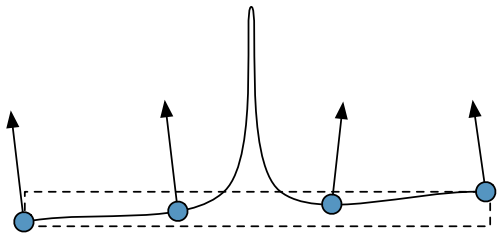

Figure 3.8: The normal samples taken may satisfy the flatness criterion (depending on the chosen constant) but the peak in the middle of the spline is not recognized properly and the generated bounding box (dashed box) will not include it

immediately, as in this case the surface is bend so hard, that a subdivision is necessary anyway. Considering this, the result of the product will always be between 0 and 1 with values close to 1 denoting a relatively flat surface.

If a surface is considered flat, then the coordinate values at the four edges are used to determine the bounding volume. At this point the above mentioned error might occur. Figure 3.8 shows an example in 2D where this technique may fail.

Any surfaces that do not satisfy the flatness criterion will be further recursively subdivided. In order to keep it simple and fast, the original surfaces parametric domain will be divided into four (parametric) equally sized sub-surfaces. The one point shared by all four subpatches is therefore to be found at $S(0.5, 0.5)$, assuming normalized knot vectors. The process will be repeated on them unless a maximum depth is reached or the sur-

face is considered flat enough. By using the parametric middle, the surface will not necessarily be divided into parts with equal face area (for surfaces with non-uniform knot vectors to be exact), but the advantage is that the new parametric domains for the sub-surfaces come for free.

With this approach it is now possible to seamlessly adjust the number and quality of bounding volumes generated.

- Choosing a constant f with $f \in [0,1]$ so that every surface flatness criterion result p with $f < p \leq 1$ denotes a surface that does not need further subdividing
- Choosing a maximum subdivision depth to limit the possible maximum number of volumes
- Increasing the number of normals used for flatness estimation to improve accuracy further (for very complex surfaces)

These properties offer a configurability that will suite all scenes in a convenient and efficient way. For most scenes, values for f between 0.8 and 0.9 will yield good results, increasing that value will create more and smaller bounding boxes. The maximum subdivision depth should not be more than 8, although often 6 will also be sufficient. Using more than 8 normals for flatness estimation has brought no improvement so far, but could be useful with unusual or extremely complex scenes.

3.3 Trimming Curves

Most CAD/CAM applications today are able to render an image of the current scene by using the ray tracing technique. However, of course, none of these do not target interactive frame rates. This also is a reason why often the trimming curves are represented as ordinary B-Spline curves. In order to achieve interactive frame rates, it is useful or even mandatory to reduce the complexity of these curves. Otherwise most of the rendering time would be spend computing the trimming test rather than the actual intersection point. In the previous work (see [1]) it was shown that using cubic Bézier curve segements as trimming curves is very well suited regarding accuracy and performance. The approach taken can be perfectly ported to the more complex NURBS surfaces, since it does not matter what kind of parametric surface is employed, as long as it is using a two dimensional parametric domain.

Basically a trimming test has to be performed for any potential hit point found on a surface which has trimming curves assigned to it. However,

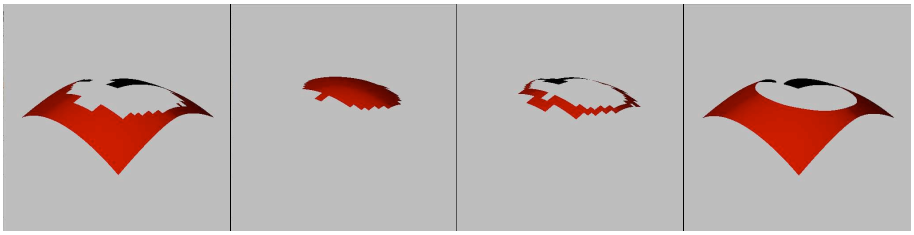

Figure 3.9: The different cases of pre-classification from left to right: 1) area where no trimming test has to be performed 2) region that is cut out completely 3) region that needs trimming 4) complete surface rendered correctly

often large surfaces have cut out only a small minor part and most of the trimming tests will yield the very same result. In order to avoid as many unnecessary tests as possible a pre-classification method is presented in the following section.

3.3.1 Pre-Classification

As mentioned in the previous section (3.2.2), there are a number of small bounding boxes for each individual surface. This is actually a great advantage for the trimming test, since a great number of the bounding boxes can be pre-classified. During the preprocessing it is checked which of the three following cases applies to each individual box

- Complete enclosed domain is untrimmed, thus trimming test can skipped during rendering
- Complete enclosed domain is trimmed, thus bounding box can be deleted
- Domain is partially trimmed, thus trimming test has to be performed during rendering

The first case means, that all bounding boxes with a completely untrimmed domain are actually treated like rendering an untrimmed surface, because any hit point found, would obviously lie automatically within a valid parametric range. The second case is especially interesting, as this reduces the complexity of the scene by removing whole bounding boxes, because every hit point found within that bounding box would be marked as invalid anyway. This only leaves the last case, where a pre-classification is not possible, thus a trimming test has to be performed during runtime. Figure 3.9 shows the three individual cases on the basis of a simple single surface.

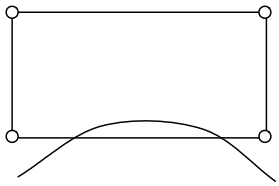

Figure 3.10: A bounding box and a single trimming curve (part of a complete set of trimming curves) are shown here. The dots at the corners mark the locations where the classification test is performed. All of them will be classified as not trimmed and thus disabling trimming test for this box, which in this particular case is wrong. Another test halfway between all corners would correct that error, as the lower middle point would have been classified as trimmed then

To classify a bounding box, simply all four corner values of the parametric domain are tested to check wether they are trimmed or not. Although possibly errors can occur like shown in figure 3.10, these have not been seen in real life models so far, which might be due to the fact that cubic Bézier splines tend to be most trivial in terms of complexity[6] (i.e. no sharp, unpredictable curves) and the bounding boxes are quite small as required for the intersection test. Considering that, this approach is reasonable and fast. Additionally it would be possible to increase the accuracy of this technique by simply using more than four points for the classification process. Note that a more detailed discussion on how the trimming test is actually being performed, can be found in section 5.4.

3.3.2 Trimming Curve Requirements

A trimming curve may be composited out of an arbitrary number of individual curve segments. However, it is mandatory that this composited curve is closed. If there are holes in between neighboring segments, the application will still run, but it will yield wrong trimming results. Additionally only cubic Bézier curves are supported, where any B-Spline curve with cubic basis function degree will be converted automatically to a number of piecewise cubic Bézier curves. Lines are currently represented by Bézier curves as well, which of course, should be optimized in the future.

[6] Actually often a single trimming curve segment is so small that they are nearly linear

Algorithm 3: Naive approach to basis function evaluation

```
float basisFunction(int i, int k, float t, float* knot){
if (k == 1){
   if ((knot[i] <= t)&&(t < knot[i+1])) return 1;
   else return 0;
}

float z1 = (t - knot[i]) * basisFunction(i,k-1,t,knot);
float n1 = knot[i+k-1] - knot[i];
float z2 = (knot[i+k] - t) * basisFunction(i+1,k-1,t,knot);
float n2 = knot[i+k] - knot[i+1];

float b1,b2 = 0;
if (!(z1==0)&&(n1==0)) b1 = z1/n1;
if (!(z2==0)&&(n2==0)) b2 = z2/n2;
return simd_add(b1,b2);
}
```

3.4 Improving Surface Points Evaluations

The evaluation of surface points (including their partial derivatives) is of very special importance, since it is an essential operation of the ray tracer. Depending on chosen parameter values for the intersection test and also depending on the individual surface the number of evaluations necessary may vary. As shown later (see 5.2.3) in average around 11.5 evaluations and partial derivative evaluations are needed per intersection test, so it is obvious that the evaluation method is a natural candidate for a maximum of optimization! The following sections will first show an easy brute force approach, which is rather slow, but will be improved incrementally, finally yielding a fast polynomial representation of NURBS surfaces.

3.4.1 Naive Brute Force Approach

The most straight forward approach would be a brute force evaluation of the basis functions. The code snippet 3 shows an exemplary implementation. Although this is very easy and fast to implement, it is totally unsuitable for an interactive environment as the results in section 7.2 attest.

There are several serious performance issues with such an implementation. First, by looking at the basis function dependancies (see 2.5), it is obvious, that for each surface evaluation quite a number of basis functions are com-

puted several times each, especially when using high order surfaces with many control vertices. The waste in computation power increases linearly with the number of control points but quadratically with the degree of the basis function. Additionally a recursive function call cannot be inlined[7] by the compiler, which is an additional loss in performance when a function is called several million times per second. Finally the numerous if-then-else constructs can slow down computation speed further, since whenever a branch prediction fails the CPU has to be stalled while the pipeline runs empty.

3.4.2 SIMD Improved Approach

Some of the issues mentioned in the last section can be addressed by using SIMD instructions which will compute four values in parallel, but additionally eliminate the need for some dynamic branching. Algorithm 4 shows another exemplary implementation. Although this is an improvement by around factor four it still can not address the problem of exponential increasing computation time of more complex surfaces. A solution is presented in the next section.

3.4.3 Avoiding Recursion

In order to improve performance by more than an order of magnitude, it is obviously necessary to avoid the multiple computation of the same basis function values over and over again, which basically means avoiding the recursion at all in some way. On the one hand, a recursive function normally looks more like the original formula and is easy to implement, but more computation effort is needed. On the other hand, an iteration is sometimes difficult to derive from an recursion, but most of the time it performs significantly better. In general it is always possible to turn any recursion into an iteration. However, this often requires a stack data structure, which is not optimal for use with SIMD instructions and not even possible to implement on today's GPUs without expensive tricks and workarounds. Since this work targets to use both the CPU and GPU in the best way possible it is more reasonable to look for a way that is useful to both processing units.

By having a closer look at the resulting basis function formulas it becomes evident, that these are in fact common (non rational) polynomials with a maximum degree corresponding to the given NURBS basis function de-

[7] For inlined functions the function call we be replaced by a copy of the code. This avoids expensive functions calls but also increases the size of the binary. Recursive functions can not be inline as this would result in an endless loop.

Algorithm 4: Improved evaluation by using SIMD instructions

```
float4 basisFunction(int i, int k, float4 t, float4* knot){
   //check if recursion has to stop
   if (k == 1){
      bool4 sel = simd_and(simd_cmple(knot[i],t),
                  simd_cmplt(t,knot[i+1]));
      return simd_select(mOne, mZero, sel);
   }

   //simple Cox-de Boor recursion
   float4 z1 = simd_mul((simd_sub(t,knot[i])),
               basisFunction(i,k-1,t,knot));
   float4 n1 = simd_sub(knot[i+k-1], knot[i]);
   float4 z2 = simd_mul((simd_sub(knot[i+k], t)),
               basisFunction(i+1,k-1,t,knot));
   float4 n2 = simd_sub(knot[i+k], knot[i+1]);

   float4 b1,b2;

   //every zeroMask component is true for every n1 comp. = 0
   bool4 zeroMask = simd_cmpeq(n1,mZero);
   //SIMD zero division does not cause application to crash
   b1 = simd_div(z1,n1);
   //every comp. in b1 with corresponding zero comp. in zeroMask
   //will be replaced by zero, since convention 0/0=0 is applied
   b1 = simd_select(b1,mZero,zeroMask);

   //analog to the previous
   zeroMask = simd_cmpeq(n2,mZero);
   b2 = simd_div(z2,n2);
   b2 = simd_select(b2,mZero,zeroMask);

   return simd_add(b1,b2);
}
```

gree, but often these even reduce to simply zero[8]! In case of rational surfaces the basic functions can be expressed as a fracture of non rational polynomials. However, due to the recursive nature of the basis function definition the computation of these polynomials is not very straightforward and requires some preprocessing. Fortunately the resulting polynomials can be stored and used for the next run of the application without needing to recompute these, as long as the NURBS properties do not change. Modifications in the control point coordinates are possible, however, changes in the knot vector or basis function degree would require a re-computation of all basis functions.

The benefits are obvious: not only is the recursion avoided but additionally the required computation effort is reduced remarkably, because only the simple and fast evaluation of polynomials is necessary. This can be done very efficiently using Horner's scheme [38] on the CPU or by brute force on the GPU, since the *pow()* command maps to a native instruction on the GPU.

3.4.4 Basis Function Precomputation

The number of basis functions that need to be computed is dependent on the NURBS surface. Unfortunately the basis functions are also dependent on the parameter intervals of their corresponding knot vectors which means that a number of basis function polynomials equalling the function order is needed for every interval defined by the knot vectors of each parametric direction. The following equation shows how many functions are needed for a specific NURBS surface

$$c = (\#itKnotsU + 1) * \#(degreeU + 1) + (\#itKnotsV + 1) * \#(degreeV + 1) \tag{3.1}$$

Here $\#itKnots$ denote the number of internal knot values either in u parametric direction or v, whereas $\#degree$ is denoting the degree of the basis functions in the according direction. For example, a NURBS Surface with four control points in each direction and with both knot vectors having no internal knot (thus degree 3), 8 basis functions would need to be computed and stored. The challenge is to do this computation efficiently and precisely. An analytic approach might be more complex than some numerical approximation, but as numerical errors can be avoided here, the analytic computation of the basis functions is definitively more reasonable, since there will be numerical errors made inevitable during the Newton's iteration, that will be employed for the intersection tests. Also compare the

[8] A specific basis function is always completely zero for a given interval if the current control point has no influence on the surface in that region at all - this does happen especially often for surfaces with a large number of control points and a low degree

results on issues with limited numerical accuracy shown in section 7.9.

The approach for analytic basis function pre-computation developed in this work is basically a two pass algorithm. As mentioned earlier, each basis function is dependent on two lower order functions, except for those with order one, which are either one or zero, depending on the deployed parameter value. First, a tree will be expanded by recursively expanding every basis function according to equation 2.3, yielding a broad tree structure with a depth equalling the NURBS basis function degree (see next section 3.4.4 for more details). Second, for each parametric interval and for both parametric directions a separate basis function will be computed by inserting one representative value of each interval into the tree, which will yield a simple polynomial after evaluating each element of the tree bottom up wise (see 3.4.4).

1^{st} Pass: Tree Construction

By having a closer look at the basis function definition (equation 2.3) it becomes obvious, that these are actually the sum of two very similar fractions each of which has a real number as denominator and a product of a real number and a lower order basis function as numerator. The fraction can be turned into a product as well by taking the reciprocal value of the denominator leaving the sum of two three-component products. This scheme is the same for all basis functions, except for order one functions, which have to be handled separately later. Figure 3.11 shows the first two steps during the tree construction phase. The final depth of the tree corresponds to the order of the initial basis function (5 in this example, but not fully shown). The tree representation is needed during preprocessing only and can be discarded afterwards, thus memory consumption is not a concern here.

There are three different kinds of elements that appear in the tree:

- *First order basis functions* - These can not be resolved further, so they remain as this until the second pass. Higher order basis functions will be resolved recursively, effectively yielding polynomials, operators and first order basis functions, thus they do not appear in the final tree at all

- *Polynomials* - During the first pass there are only order two polynomials like $(t - x_i)$ and constant numbers which are represented as order one polynomials

- *Operator* - either multiply or add. They have an arbitrary number of children, which can be of any of these three types.

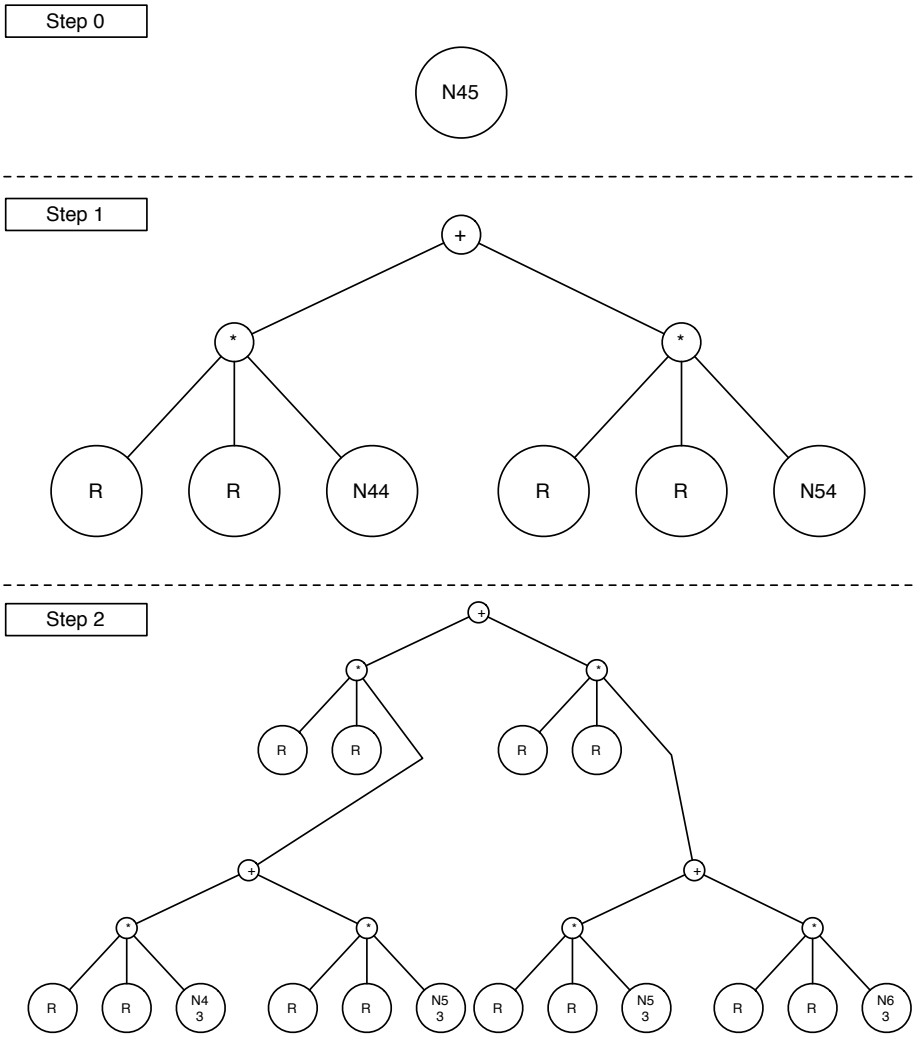

Figure 3.11: First two expanding steps during buildup of the computation tree that describes a specific basis function

The basis functions of order one are a special case as mentioned earlier. The value of these is simply either one or zero, dependent on the parameter value t, which is unknown in the first pass. The creation of the tree is finished when every basis function with a degree higher than one has been recursively expanded, finally leaving only polynomials, operators and first order basis functions.

2^{nd} Pass: Basis Function Computation

As stated earlier, there is one basis function for each parametric interval for both parametric directions. For example the knot vector [0 0 0 1 2 3 4 4 4] defines four intervals, namely [0;1],[1;2],[2;3] and [3;4]. Recall from equation 2.4, that $N_{i,1}(t) = 1$ if $x_i \leq t < x_{i+1}$ and 0 otherwise. By picking a representative value for each interval (the median for convenience) the final basis function for this specific interval can be computed.

The operators can, of course, only work on polynomials, however this can be ensured by employing a bottom up algorithm. All first order basis functions now reduce to first order polynomials by employing the chosen representative value of the current interval. It is now guaranteed that all leafs of the tree are in fact polynomials. Every operator that has polynomial children only, can perform their appropriate action (i.e. multiply or add) on them, yielding a new polynomial as result. This is continued bottom up, until only one polynomial is left, which is the complete basis function for this parametric interval. Figure 3.12 shows such a step exemplarily. According to this figure the following example by numbers will show the very first step, operating on the leaf nodes: *poly(1)* (i.e. polynomial of order 1) nodes are the reciprocal of the denominator (recall equation 2.3), thus $\frac{1}{x_{i+k-1}-x_i}$ where *poly(2)* denotes part of the nominator which is $(t - x_i)$. The third component is $N_{i,1}(t)$, which in this example evaluates to 1. Employing now the appropriate knot vector values, the result of the first component is assumed to be $\frac{1}{2}$ ($x_{i+k-1} = 4$ and $x_1 = 2$ for instance). Now all three components are multiplied (i.e. the parent node),which yields $\frac{1}{2}t - 1$, a simple order two polynomial.

After all basis functions of a NURBS surface have been computed, the tree can be deleted. All functions are stored along with their corresponding surfaces. For GPU usage they are additionally stored in a texture (see section 3.5.2). In the first place it might seem that the expensive pre-computation of all possible basis functions consumes a lot of memory. Section 5.5 has some numbers on memory consumption, which shows that this is not the case.

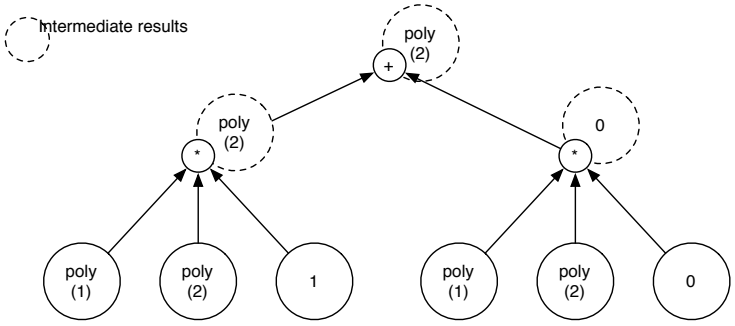

Figure 3.12: Bottom up resolving pass of the tree. Note that the intermediate results are not stored within the tree structure. These are only needed for this specific basis function defined by the current parametric interval and are deleted directly afterwards

3.5 Memory Layout

As the CPU and GPU handle memory quite in a different way, it is necessary to store most of the data in two variants. Data can be stored more or less in a conventional way for the CPU, however for GPU usage a texture has to be created that will act as a random access memory. What first might sound as a waste of memory, storing the same data twice, is of course not, because the data textures are stored on the RAM of the graphics board anyway, which would not be used at all otherwise. The following sections will show individually the design of memory usage for SIMD (3.5.1) and GPU (3.5.2).

3.5.1 Storage for SIMD Use

Data storage for SIMD is a bit more sophisticated than for common FPU operation (refer to section 2.4 for more details on SIMD). First of all, the basic NURBS data (i.e. number of control points, degree of the basis functions, etc) are packed into SIMD *float4* variables. For example one *float4* variable stores the number of vertices in u and v direction as well as both degrees of the basis functions. The control points with their four components fit perfectly into a $float4$ variable so not surprisingly these are stored in a dynamical array of $float4$, which size is equal to the number of control points.

SIMD										
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	0	0
z	0	0	0	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	0

Figure 3.13: Storage of both knot vectors, with ten respectively eight components. Each column represents a single *float4* variable. The unused components have actually undefined values, but they could be used to store other data related to the corresponding patch

Knot Vector Storage

Knot vectors are handled slightly different than a simple storage in an array. On the one hand, random access to the knot vectors is needed (i.e. access to the ith element), but access to the components of a SIMD register has to be known at compile time, thus only static access is possible. On the other hand, the knot vectors must have SIMD format, in order to be used by the SIMD unit[9]. There are basically three options to solve this

- Do all computation on common float values and write the result into a SIMD register
- Store one knot vector value in all four components of a SIMD register
- Store the u knot vector always in the first components respectively the v knot vector in the second components of a SIMD register array

Filling a SIMD register with new non-vector data is very expensive, due to memory allocation and alignment, thus the second approach is more reasonable, particularly because the memory consumption is very low for knot vectors at all compared to the storage of the basis functions (see section 5.5). However, packing both vectors together reduces the wasted memory remarkably, although still two components remain unused. However, these free components could be used in the future to store other data (like number of control points etc.) as well, but as memory consumption was not a major concern it was not implemented so far. Figure 3.13 shows the packing of knot vectors.

[9] On the fly conversion is possible, of course, but memory needs to be allocated and even aligned to 16 bit boundaries, which is simply to costly to be done during rendering in an inner loop

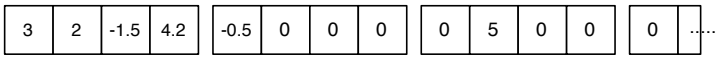

Figure 3.14: Each block of four boxes denotes a single $float4$ vector. In this example each basis function occupies two float4 elements. The first two store a fourth order (third degree) function: $2x^3 - 1.5x^2 + 4.2x - 0.5$ and the next two (not fully displayed) describe a first order (degree zero) function which is a a contant value of 5

Basis Function Storage

Finally the basis functions are stored in $float4$ arrays as well, that is, all basis functions of a single NURBS patch are stored in one array, where each coefficient is stored as a component of a $float4$ variable. The very first component stores the degree of the specific basis function that follows, because the degree may vary. The amount of memory assigned to each basis function is defined by the basis function degree of the NURBS surface, where

$$space = (floor(degree/4) + 1) * size\_of(float4) \tag{3.2}$$

hold. Here the result *space* denotes the memory needed by each basis function. Note that there might some components be wasted, for example, if the degree is three, then three components are wasted. However, this fixed width is necessary due to performance reasons. As mentioned above, it is not possible to access the individual components dynamically. Figure 3.14 shows an example of how basis functions are stored within the $float4$ variables.

How to store all basis functions in an efficient way is another important issue. As stated in equation 3.1 there are several basis functions for a single NURBS surface. Basically the functions span three dimensions, which are *parametric direction*, *parametric interval* and *index i* (running from 1 to the order of the basis function) - refer to equation 2.3 if necessary. However, three dimensional access is not very efficient at all, so the functions are stored sequentially. In order to access the correct functions a single integer offset has to be computed. This is best explained by an simple example: Consider a NURBS patch with four control points and order three (degree two) in both parametric directions. The knot vectors therefore will be $[0\ 0\ 0\ 1\ 2\ 2\ 2]$. The number of generated basis functions will be 12 (one internal knot per direction). There will be six for each parametric direction with each three functions per parametric interval of which there are two. Figure 3.15 shows the layout in memory. Note that the *mBFOffset* variable is not required, since it could also be computed from the given information.

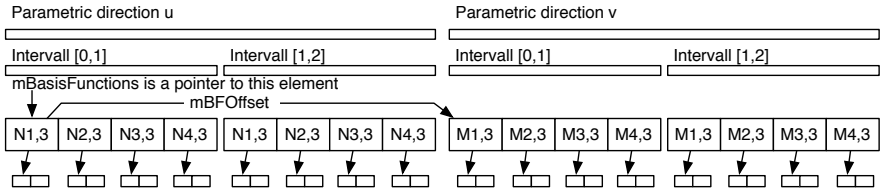

Figure 3.15: Storage of all associated basis functions. Every NURBS surface stores an entrance pointer to the first function. The small rectangles at the bottom represent a *float4* vector each

However, by pre-computing and storing this value a couple of operations can be saved for every basis function evaluation. Considering the fact that this is done million of times per second, the overall performance indeed benefits from such small measures. Also note that each of the shown basis functions corresponds to one or more *float4* variables depending on the degree, i.e. $floor(degree/4)+1$ *float4* variables are needed per function, which in this example is 2

3.5.2 Storage for GPU Use

Compared to common data storage, the memory layout for the GPU is quite a bit different of course, due to the graphics centric architecture of the GPU. However, each pixel of a RGBA float textures is actually very similar to a *float4* vector, as both store four distinct values and do not allow dynamic access on their individual components. Thus the same basic layout design can be applied to the textures, as it was used with the SIMD approach in the previous section.

The biggest difference lies in the general organization of the associated data. As with SIMD (or FPU) use, every NURBS patch stores it's own information like number of control points, knot vectors and the like. It would be possible, although not very efficient, to create a texture for every NURBS patch that stores exactly these information. Depending on the scene, this could result in more than several thousand textures, which can't be processed by OpenGL in an optimal manner, thus it is more reasonable to rather use a small number of big textures than a big number of small textures. Nvidia's GeForce 6800 graphics card, for instance, can access eight different ones in a single pass.

The data of **all** NURBS surfaces of the whole scene is stored in only three different textures (see figure 3.16). That is, all control points of all surfaces

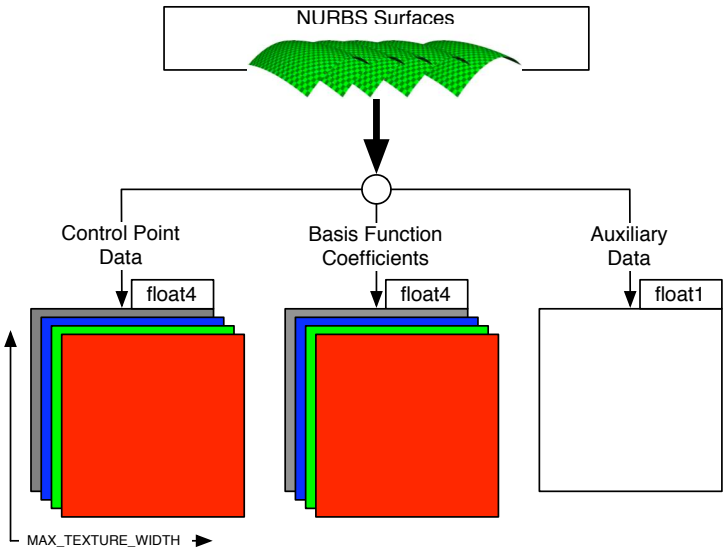

Figure 3.16: Distribution of data into three different textures

are stored in a single texture. The same way all knot vectors are stored in a second one and finally all remaining information (like number of control points, degree, several offset values) are stored in a third. This limits the number of supported surfaces for a scene, dependent on the amount of memory available on the graphics card. However, if most patches stay within reasonable ranges for the number of control points and degree, more than 50.000 surfaces can be uploaded to the graphics card, which should be sufficient for even very complex scenes.

The mentioned textures are uploaded to the graphics board upon launch of the application. During runtime it is not necessary to transfer new data, if the scene is static, except for the camera. The organization is as follows

- *Control point texture* - stores all control points of all NURBS surfaces. Used format is *float4* (i.e. RGBA)[10], which means each pixel can store a complete control point (x, y, z, w)

- *Basis function texture* - stores all basis function coefficients of all NURBS surfaces. Data format again is *float4*, where each pixel stores four coefficients. If the order is not a multiple of four, the remaining space is filled with zeros.

[10] OGSL features a datatype called float4 - this is not the same float4 as it is used by the C++ code

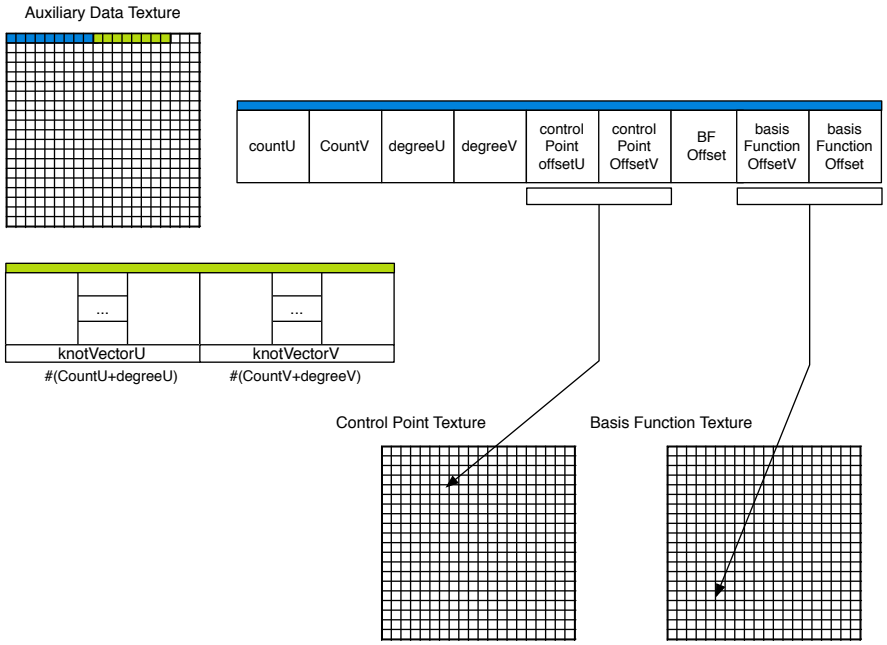

Figure 3.17: Three different textures store all information of all NURBS surfaces of the entire scene

- *Auxiliary data texture* - stores several different information, for example the knot vectors and also references where to find data in both other textures. Data format is *float1*

The maximum texture size can be changed by modifying a constant. However, the default size of 2048^2 is reasonable and should not be reduced. Reducing texture size reduces the maximum number of NURBS patches that can be uploaded to the graphics board. If the graphics card has more than 256 MB of RAM, the size can even be increased. With the above mentioned configuration and a maximum size of 2048 the memory consumption is as follows

$$2048^2 * 4[RGBA] * 4[float] * 2 + 2048^2 * 1 * 4[float] \\ = 64MB * 2 + 16MB = 144MB$$

Using 144 MB of memory for NURBS storage still leaves enough for whatever amount the operating system might need, since the GeForce 6800 (and follow-up models as well) usually come with not less than 256 MB of RAM.

Figure 3.17 illustrates how the three textures collude. Note, that in contrast to the CPU variant, it is necessary to store a number of offsets which

will point to the appropriate data for a specific NURBS surface. For example *controlPointOffsetU* and *controlPointOffsetV* will point to the texture-coordinates of the first control point for a given patch. By extracting the number of control points (*countU* and *countV*), the correct number of control points can be read from there on. Also note, that the data that corresponds to a NURBS patch is always stored in one row only, i.e. data is not distributed on two subsequent rows. Basically this is to avoid expensive pipeline stalls resulting from checking if the end of the row is reached. The amount of wasted memory is small anyway and again the application is not memory limited, but performance is the major concern.

Control Point Texture

As mentioned earlier, the four dimensional control points fit perfectly into a RGBA pixel each. The control points of each surface are simply stored in in a row major (v parametric direction) manner. After the control points of one surface, the control points of another follow immediately. If there are less pixels left in a row than the surface has control points, the data is stored at the beginning of the next row. As mentioned before data is never divided on multiple rows. When using the suggested maximum texture size of 2048^2 a maximum number of more than 4.1 million control points can be stored which is enough for more than 40.000 patches, with each 10^2 control points, which is rather large.

Basis Function Texture

Storage of the basis functions data is actually also very similar to SIMD data storage. Here as well, the usage of a RGBA float texture is quite comparable to the usage of *float4* data types. The coefficients, along with a leading number that specifies the degree, are stored component wise in a predefined number of pixels. Unused components can be filled with zeros or left undefined. All basis function data of a single patch has to be stored in a single row, again like the control point texture demands, too. The following table shows some examples on how many NURBS surfaces can be supported for different configurations (number of control points, degree) on basis of the maximum texture size of 2048^2.

control points	degree	supported surfaces
16	3	262144
100	3	14979
100	5	20971
100	9	69905
400	5	3495

Note, that the storage used per basis function is $3 * sizeof(float4)$ instead of $2 * sizeof(float4)$, whenever the degree of the basis functions is between eight and eleven, as explained in equation 3.2. It seems like there is only a minor number of surfaces supported, when the number of control points is high along with a relatively low degree, however, such a surface is extremely flexible and powerful so that they (if they occur at all) can describe a surface, which would have been approximated by several thousand triangles otherwise. Also note, that this limit as such applies to the GPU powered variant only!

Auxiliary Data Texture

The first elements are fixed for each surface, whereas the last two entries, the knot vectors, are variable in size depending on the NURBS surface. These are as follows (see figure 3.17)

- *countU* Number of control points in u direction
- *countV* Number of control points in v direction
- *degreeU* The basis function degree in u direction
- *degreeV* The basis function degree in v direction
- *controlPointOffsetU* u texture coordinate where to find the corresponding control points
- *controlPointOffsetV* analog to controlPointOffsetU
- *BFOffset* Offset between u and v basis function data
- *basisFunctionOffsetU* u texture coordinate where to find the corresponding basis function data
- *basisFunctionOffsetV* analog to basisFunctionOffsetU
- *knotVectorU* the knot vector in u direction. Number of elements corresponds to $countU + degreeU + 1$
- *knotVectorV* analog to knotVectorU

As only very little information is stored, compared to the basis function texture, this one is not limiting the maximum number of supported NURBS surfaces.

Chapter 4

Standalone libraries

Both, *iges2dsd* and *crianusurt* rely on functionality offered by *libNURBS*, which itself depends on *libSIMD*. These libraries can be used independently of the work presented here, so existing applications could easily benefit from the ray-surface intersection test or normal evaluation on Bézier and NURBS surfaces.

4.1 libSIMD

The cross-platform library *libSIMD* introduces an abstraction layer of SIMD commands, which maps either onto SSE or AltiVec commands depending on the current architecture or even on the FPU if no SIMD instruction unit is available (or desired). It compiles and runs on Mac OS X, Linux and Windows[1]. The modular design even allows to easily extend the framework to other architectures than x86 and PowerPC.

Originally the functionality was developed by Markus Geimer in the context of his PhD thesis [12], where it was integrated into the interactive raytracer *dream*. The corresponding code was moved into a standalone library, basically because both applications *crianusurt* as well as *iges2dsd* need the NURBS related methods, which heavily rely on SIMD operations. So the doubling of code is avoided and the code remains easily maintainable.

4.1.1 Data Types

Two new data types are introduced by this library: *float4* and *bool4*, which both store four values of their appropriate type. However, normal operators are not defined on them, thus SIMD commands have to be used in order to add two *float4* vectors for example. The following code fragment sets two variables and compute the sum of them.

[1] some modifications for the windows platform might be necessary

```
float4 a = simd_set(4.0f);
float4 b = simd_set(1.0f, 2.0f, 3.0f, 4.0f);

float4 c = simd_add(a,b);
```

The value of variable c therefor is $(5,6,7,8)$. Depending on the featured SIMD instruction set, either the appropriate SSE, AltiVec or FPU commands are used to compute the sum.

4.1.2 Abstraction Layer

As mentioned before in section 2.4, the two different instruction sets are similar but still different, of course. Some commands differ only in their names like the instruction for adding two registers.

```
libSIMD:
    simd_add(a,b);
SSE:
    _mm_add_ps(a, b);
altiVec:
     vec_add(a, b),
```

Some differ also in their list of arguments, for instance, there is no multiply command with AltiVec, thus a multiply-add command has to be carried out, where the last argument is simply zero.

```
libSIMD:
    simd_mul(a,b);
SSE:
    _mm_mul_ps(a,b);
altiVec:
    vec_madd(a, b, (vector float)(0,0,0,0))
```

Finally there are commands that are not known by one of the instruction sets. The select command, which either selects the contents of argument *a* or argument *b*, depending on the contents of argument *c* is known with AltiVec but has to be constructed out of three individual SSE commands.

```
libSIMD:
    simd_select(a,b,c);
SSE:
    _mm_or_ps(_mm_and_ps(c, b), _mm_andnot_ps(c, a))
altiVec:
    vec_sel(a, b, c);
```

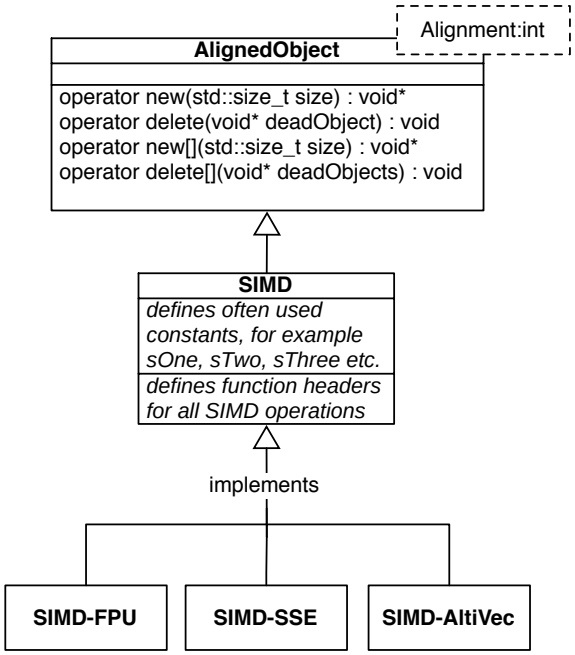

Figure 4.1: Basic design of *libSIMD*

libSIMD unifies the different syntax under an easy to use API for all basic arithmetic and logical operations, as well as some access and data conversion methods. Figure 4.1 shows an UML diagram of the basic design of *libSIMD*. Note that every class that makes use of dynamically allocated SIMD memory (i.e. *float4*) has to be derived from *AlignedObject* and every newly allocated memory has to be acquired by the following helper function, which assures correct alignment, required by SIMD operation.

```
void* newAligned(std::size_t size, std::size_t alignment);

//for example
mVar = (float4*)newAligned(arraySize * sizeof(float4),16);
```

4.2 libNURBSIntersectionKernel

This library offers the core functionality for all ray-surface and normal evaluation related methods needed by *crianusurt* and also *iges2dsd*. There are two different groups of commands that can be evoked. Before any of these methods can be used, the kernel has to be configured, which basi-

cally means providing all necessary information for the command to follow. Performance is a critical key feature for interactive ray tracing and thus no checks wether provided data is correct and complete or not will be performed. The application using the kernel has to make sure the kernel is set up properly. The following sections 4.2.1 and 4.2.2 show which information are needed and what actions can be performed. Figure 4.2 is showing the basic design of *libNURBSIntersectionKernel*, however, with respect to the page size, only the most important elements, methods and member variables are taken into account.

4.2.1 Evaluation Methods

The kernel can be used to evaluate points, partial derivatives and normals on NURBS and Bézier surfaces for any valid u/v parameter values. Additionally a pair of u,v values can be checked for trimming. As, of course, SIMD commands are used for all calculations, it is necessary to provide vectors of each four u and v parameter values. If, for some reason, only one single evaluation is requested, a whole SIMD register has to be filled with the same value or any other valid value. Again, due to achieve a maximum of performance no checks of the input will be performed. If the provided u/v parameter values are invalid (i.e. out of bounds) then the kernel will return undefined results or may even crash, so it is important to pass only legal values!

Required data for evaluate() / evaluateNormals() function calls on NURBS surfaces

- cfgActiveComponents(bool4 const active)
- cfgBasicDataVector(float4* const basicData)
- cfgControlPoints(float4* const controlPoints)
- cfgKnotVectors(float4* const controlPoints)
- cfgBasisFunctions(float4* const basisFunctions)
- cfgBFOffset(float4* const offset)

Bicubic Bézier surfaces are also supported by the kernel as a high performance alternative. The configuration in that case is even shorter since only the active components and control points have to be passed to the kernel. Note that the methods have the appendix *BS* added, i.e evaluateBS() for example. Refer to appendix A for a detailed overview.

Checking a value pair for trimming is very easy as only one item needs to be configured, which are the trimming curves themselves.

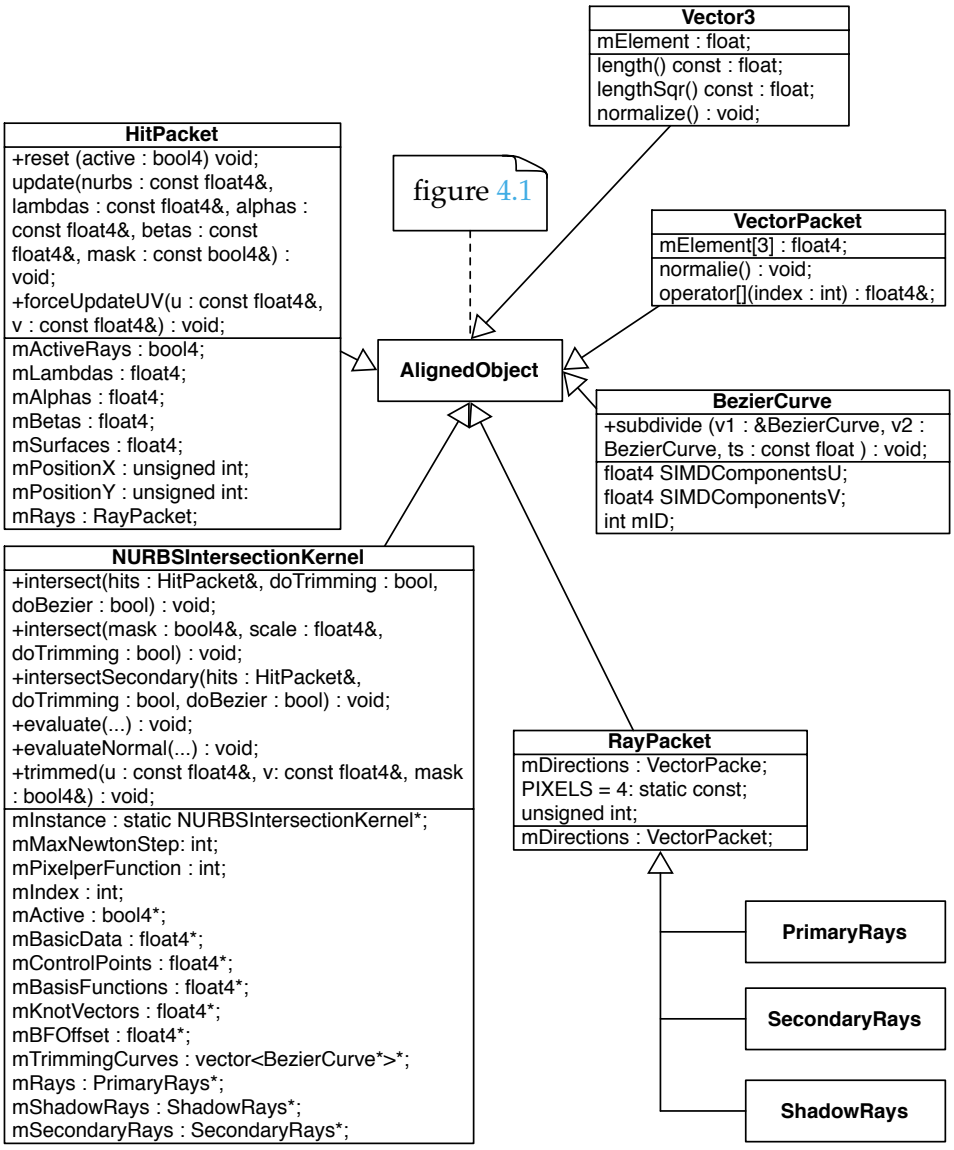

Figure 4.2: Basic design of *libNURBSIntersectionKernel*

Algorithm 5: NURBSIntersectionKernel evaluation function calls

```
NURBSIntersectionKernel *ik = NURBSIntersectionKernel::Instance();

//configure the kernel with all necessary data

float4 u = simd_set(0.0f, 0.33f, 0.66f, 1.0f);
//sets all four values to 0.5
float4 v = simd_set(0.5f);

//these will store the results
float4 resX, resY, resZ;

//normal evaluation of surface points
evaluate(u, v, false, false, resX, resY, resZ);

//evaluation of partial derivative in v direction
evaluate(u, v, false, true, resX, resY, resZ);

//evaluation of normals
evaluateNormal(u, v, resX, resY, resZ);
```

- cfgTrimmingCurves(vector<bezierCurve*>* const curves)

The evaluation call needs two *float4* vectors containing the u and v parameter values, two boolean values indicating if partial derivatives in u and/or v direction shall be computed and finally three *float4* vectors, which will store the results. The following example shows the appropriate call for computing four points on the surface, as well as the command for computing the partial derivative in v parametric direction. Computing the normals (i.e. evaluateNormal()) is nearly the same command, only the boolean values for the derivatives do not need to be specified. However, a boolean flag as last parameter switches between NURBS and Bézier normal evaluation. The default value is *false*, i.e. by not providing this value the NURBS variant will be used. Basically the computation of the normals is a cross product of the partial derivatives, i.e. $S_u x S_v$ where S_{dir} denotes their partial derivative in *dir* parametric direction. Again, for a complete overview on all related configure and computation instructions for both NURBS and Bézier, have a look at appendix A.

4.2.2 Intersect Methods

There are three different intersection methods, one each for *primary* rays, *secondary* (i.e reflected) rays and *shadow* rays. The required configuration commands for NURBS surfaces are as follows

- cfgIndex(const int index)
- cfgActiveComponents(bool4 const active)
- cfgBasicDataVector(float4* const basicData)
- cfgBoxMinMaxUV(float4* const minU, maxU, minV, maxV)
- cfgSurfaceMinMaxUV(float4* const minMaxUV)
- cfgControlPoints(float4* const controlPoints)
- cfgKnotVectors(float4* const controlPoints)
- cfgBasisFunctions(float4* const basisFunctions)
- cfgBFOffset(float4* const offset)
- one of these:
  - cfgRays(PrimaryRays const* rays);
  - cfgRays(ShadowRays const* rays);
  - cfgRays(SecondaryRays const* rays);
- optionally: cfgTrimmingCurves(vector<bezierCurve*>* const curves)

For Bézier surfaces this effectively reduces to

- cfgIndex(const int index)
- cfgActiveComponents(bool4 const active)
- cfgBoxMinMaxUV(float4* const minU, maxU, minV, maxV)
- cfgControlPoints(float4* const controlPoints)
- one of these:
  - cfgRays(PrimaryRays const* rays);
  - cfgRays(ShadowRays const* rays);
  - cfgRays(SecondaryRays const* rays);
- optionally: cfgTrimmingCurves(vector<BezierCurve*>* const curves)

Note: The trimming curves for both variants are optional, but if no trimming test should be performed, the kernel has to be configured by providing a *NULL* argument. If this is ignored, the old value will be kept and used!

After successful configuration of the kernel, the appropriate intersection method (depending on the rays provided) can be called. The interface for the intersection test for primary and secondary rays is identical. However, the shadow test interface differs, since it is uninteresting to know the exact hit location, but only if the rays were hitting anything or not.

```
//Primary intersection test
void intersect(HitPacket& hits);

//Secondary
void intersectSecondary(HitPacket& hit);

//Shadow
void intersect(bool4& mask, float4&scale);
```

The *HitPackets* will store all information related to the found intersection points, that is, the unique identifiers of the hit surfaces as well as the u and v parameter values and the distances from the hit points to the origin of the rays.

Whereas the *mask* parameter will only store wether a surface was hit at all or not, which means if a specific point in space is shadowed or not. The *scale* parameter can be used during transparency calculations.

Chapter 5

Application *crianusurt*

This chapter is about the actual rendering application *crianusurt*. Although the intersection and trimming tests are implemented within *libNURBSIntersectionKernel*, they belong logically to this application and thus are described in section 5.2 and section 5.4. Some in depth considerations about the memory consumption can be found in section 5.5 and section 5.6 concludes this chapter with problems and limits of ray tracing NURBS.

5.1 Basic Architecture

Detailed information about the basic architecture can be found in the dissertation of Markus Geimer [12]. Figure 5.1 shows the topmost architectural design of the original application *dream* which basically also applies to *crianusurt*. The biggest differences are to be found in the *PrimaryThread* and *ShadingThread*. The functionality to trace and shade triangles was replaced by the appropriate parts of *libNURBSIntersectionKernel*. Most other parts of the system needed only minor adaptions or none at all. The following sections will give an overview of the new concepts which extended or replaced important parts of the original application *dream*.

5.2 Intersection Test Ray-NURBS Surface

One of the most important tasks of a ray tracer is, not surprisingly, the intersection test of rays and scene objects, which are NURBS and Bézier surfaces only in this implementation. Depending on the scene, as well as camera position, roughly 30% to 60% of the computation effort is spend in the ray-surface intersection test. A major part is spent traversing the bounding volume hierarchy where the rest will be used for shading, generating new rays, composing the final image and so on. The bounding volume hierarchy traversal was already highly optimized in *dream*, which leaves the

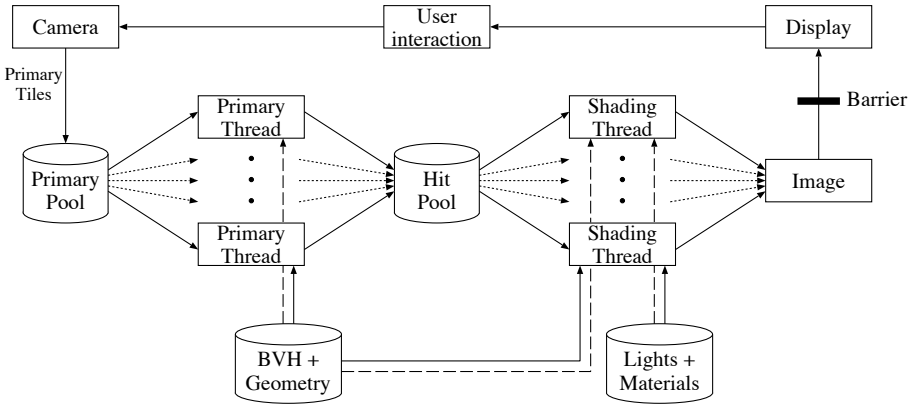

Figure 5.1: Basic architectural design of *dream/crianusurt*. Adapted from [12]

surface intersection test as the optimal candidate for optimization.

5.2.1 General Idea

Basically there are several different approaches on how an intersection test can be computed. These can be categorized into three distinct classes

- Algebraic methods
- Recursive subdivision methods
- Numeric methods

Analytic Methods

An algebraic solution for the intersection of a ray and a bicubic parametric surface was presented by James T. Kajiya [21] in 1982. He represented the ray by the intersection of two planes. The surface possibly defined a intersecting curve in the parametric domain of each plane. The intersection, in turn, of these two curves can be computed by the real-valued roots of a polynomial of 18^{th} degree. Needless to allude, that this is not very reasonable for an interactive environment. Additionally this approach would be applicable to bicubic surfaces only and not to arbitrary NURBS. Considering that such an attempt would be much more complex (if possible at all), algebraic methods seems to be an inadequate class of solution for interactive ray tracing of NURBS.

Recursive Subdivision Methods

One of the most common ways NURBS surfaces have been ray traced so far, was by actually avoiding the direct intersection test with NURBS surfaces at all. Nearly every ray tracing application computes only the intersection between rays and triangles, thus the idea is to triangulate the surfaces in a way that they are very close to the original shape defined by the NURBS surface. Again there are different approaches, but basically the choice is wether to triangulate in a preprocessing step or to triangulate during runtime on the fly. These *Subdivision Surfaces* are better known in their variants introduced by Catmull-Clark in [11] or Loop in [23], for example.

The other approach is easier to implement, but surely will suffer from the usual issues polygonial representations have, i.e. visible polygon edges within a surface, high memory consumption and the like. Additionally for high resolution models, like they are used in the automobile industry for example, the preprocessing step for triangulation can take up to a whole day (which also makes an good triangulation during runtime impossible). Finally, this work is about rendering real parametric free form surfaces and thus avoiding the problem by triangulating them is not an option[1].

Numeric Methods

Whenever an analytical solution is not available or not applicable, it is a good idea to consider employment of numeric methods. The *Newton Iteration* has proven itself very successful for the rendering of Bézier surfaces (see [1]) and the same basic algorithm can be applied to NURBS surfaces as well. The idea is outlined in the next section.

5.2.2 Newton Iteration

Newton's iteration (see [29] further details) , also called Newton-Raphson method, is often used in a 2D environment, for instance to compute the root of a function. It is very popular because of it's (most of the time) quadratic convergence, as well as an very easy and fast implementation. The drawback is, that an initial guess is required to start the iteration with. This value has to be close to the real root. How close a guess has to be, is dependent on the function, i.e. a high degree function with large coefficients most likely needs a closer guess than a simple quadratic function[2]. Another drawback is, that the first derivative is also required as well. Equation 5.1 shows Newton's iteration for the 2D case. In spite of these drawbacks the Newton Iteration seems like an ideal candidate, since the first derivative

[1] Additionally triangles are currently not supported in this version
[2] For linear functions the starting guess does not matter at all

is relatively easy to compute (see 2.2.9) and a good starting value can be ensured by the bounding box hierarchy (see 3.2.2)

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5.1}$$

Assuming a good starting value, every iteration will bring the result closer to the real root, effectively reducing the error quadratically, except whenever $f'(x_n) = 0$ holds, in which case the error only reduces linearly.

Obtaining a good initial guess is one of the major problems when employing the Newton iteration. As mentioned before, in this work the bounding volume hierarchy will provide the initial starting value. For launching the Iteration the median of the bounded parametric domain of the last bounding box that was tested, will be used. This approach was first proposed in [26]. Another method was introduced by Hanan Samet in [27], where the Oslo algorithm (refer to 2.2.6 if necessary) was used to subdivide the surface, until the real surface was approximated closely. An intersection with the control point net then yielded the initial guess. The oslo algorithm is employed in *iges2dsd* as well during pre-precoessing, however, such an approach, used during rendering, is much to expensive regarding required computation effort.

Other numerical approaches, like the *secand method* or *Banach fix-point theorem* among others, would also be more or less employable, but all of them are not really as qualified as the Newton iteration is. The secand method for instance looks well suited in the first place, since no evaluation of the first derivative is needed and with a fixed point approach (i.e. secand method first order) only one evaluation of the surface after the first iteration (requiring two evaluations) would be needed, but the convergence rate is linear only. The same applies basically for the fix point theorem. For the latter a single iteration step is additionally far more complex compared to the Newton iteration. Considering this, it can be assumed that both approaches will perform significantly slower, but still a further investigation on that in the future might be interesting.

5.2.3 3D Extension to Newton's Iteration

Since the ray tracing is performed in a three dimensional environment, the Newton Iteration needs to be extended, which renders the iteration a bit more sophisticated. The basic idea is to represent the ray in three dimensional space as an intersection of two planes, as showed in figure 5.2. Wang et al. have a similar approach in [36]. However they are using the Newton iteration only when the rays are coherent. If that is not given they employ Bézier clipping (as they are dealing with Bézier surfaces only) which is not

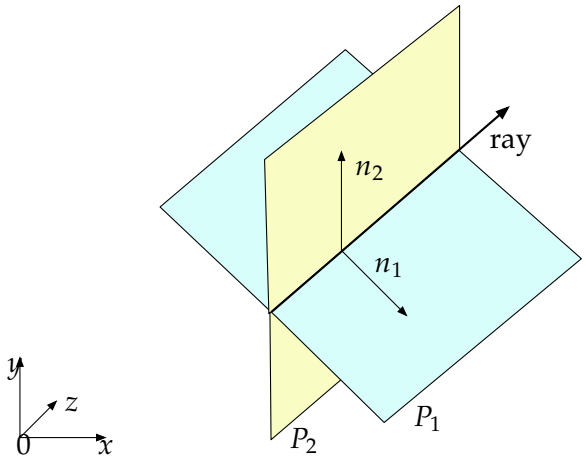

Figure 5.2: A ray described by the intersection of two perpendicular planes N_1 and N_2 as they are used for the Newton Iteration

applicable for NURBS surfaces[3]. William et al. are employing a similar approach with respect to NURBS surfaces in [26] too, however they are not targeting an interactive environment. Basically the implementation of the Newton iteration is akin to the works of these two groups, but the implementation of the surface evaluation including the first derivative is completely different, since both groups did not target interactive frame rates.

As mentioned earlier, the ray is represented by two planes. They do not have to be orthogonal to each other, but convergence is slightly faster and more robust if they are. In any case their normal vectors must not be collinear! The planes are represented by $P_1 = (N_1, d_1)$ and analogous $P_2 = (N_2, d_2)$, where the N_i are the normal vectors with unit length, which are perpendicular to the ray direction. The d_i indicate the distance to the origin of the coordinate system which can be computed as follows

$$d_i = -N_i O \quad i = 1, 2$$

where O is the origin of the ray. Figure 5.2 shows the ray with it's two planes. In order to find the intersection between a ray and the NURBS surface the roots of equation 5.2 have to be found

$$R(u,v) = \begin{bmatrix} N_1 S(u,v) + d_1 \\ N_2 S(u,v) + d_2 \end{bmatrix} \tag{5.2}$$

[3]This approach heavily relies on the convex hull properties of Bézier surfaces

where $S(u,v)$ is the point on the surface at the given parameter values. The conversion of the Newton Iteration into the three dimensional variant yields the following

$$\begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} u_n \\ v_n \end{bmatrix} - J^{-1}R(u_n, v_n)$$

where J is the Jacobian Matrix of R as given in in the following equation

$$J = \begin{bmatrix} N_1 S_u(u,v) & N_1 S_v(u,v) \\ N_2 S_u(u,v) & N_2 S_v(u,v) \end{bmatrix} \tag{5.3}$$

$S_u(u,v)$ and $S_v(u,v)$ are the partial derivatives of their corresponding parametric direction. The inverse of the Jacobi matrix is as follows

$$J^{-1} = \frac{1}{det(J)} \begin{bmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{bmatrix} \tag{5.4}$$

The Newton Iteration is executed until one of three termination criteria is met. An intersection is found if, and only if, the distance ($R(u_n, v_n)$) of the approximated intersection point to the real root is below some user defined threshold, i.e. $|R(u_n, v_n)| < \epsilon$. In any other case the iteration will be continued until either the distance increases from one step to the next (indicating a possible divergence), i.e. $|R(u_{n+1}, v_{n+1})| > |R(u_n, v_n)|$ or a predefined maximum number of steps have been computed, which possibly could indicate a loop or a very slow convergence. Figure 5.3 explains the control flow of the Newton iteration as it is implemented within this work.

Runtime Considerations

The evaluate (including partial derivatives) method is called three times during a single newton step and an additional evaluation is executed at the end of the intersection routine. The maximum number of iteration steps can be set by the user, however, more than five steps are seldom necessary or useful at all. Although an iteration can converge after the very first newton step it usually takes in average around 3 to 4 steps, depending on the complexity of the scene, the setup of the camera and the user defined parameter for a threshold that defines a valid intersection. Rays that hit a surface orthogonally, for example, are most likely converging faster than those that hit tangentially. For a single intersection test in average, roughly $3.5 * 3 + 1 = 11.5$ evaluations are necessary. Taking into account that for every pixel most likely more than one intersection test is required and reflections and shadows even increase this number further, it becomes clear, that several million evaluations per second are necessary to achieve interactive frame rates (assuming an image size of 512^2 pixels), which render it very important to optimize this method to the maximum extend possible.

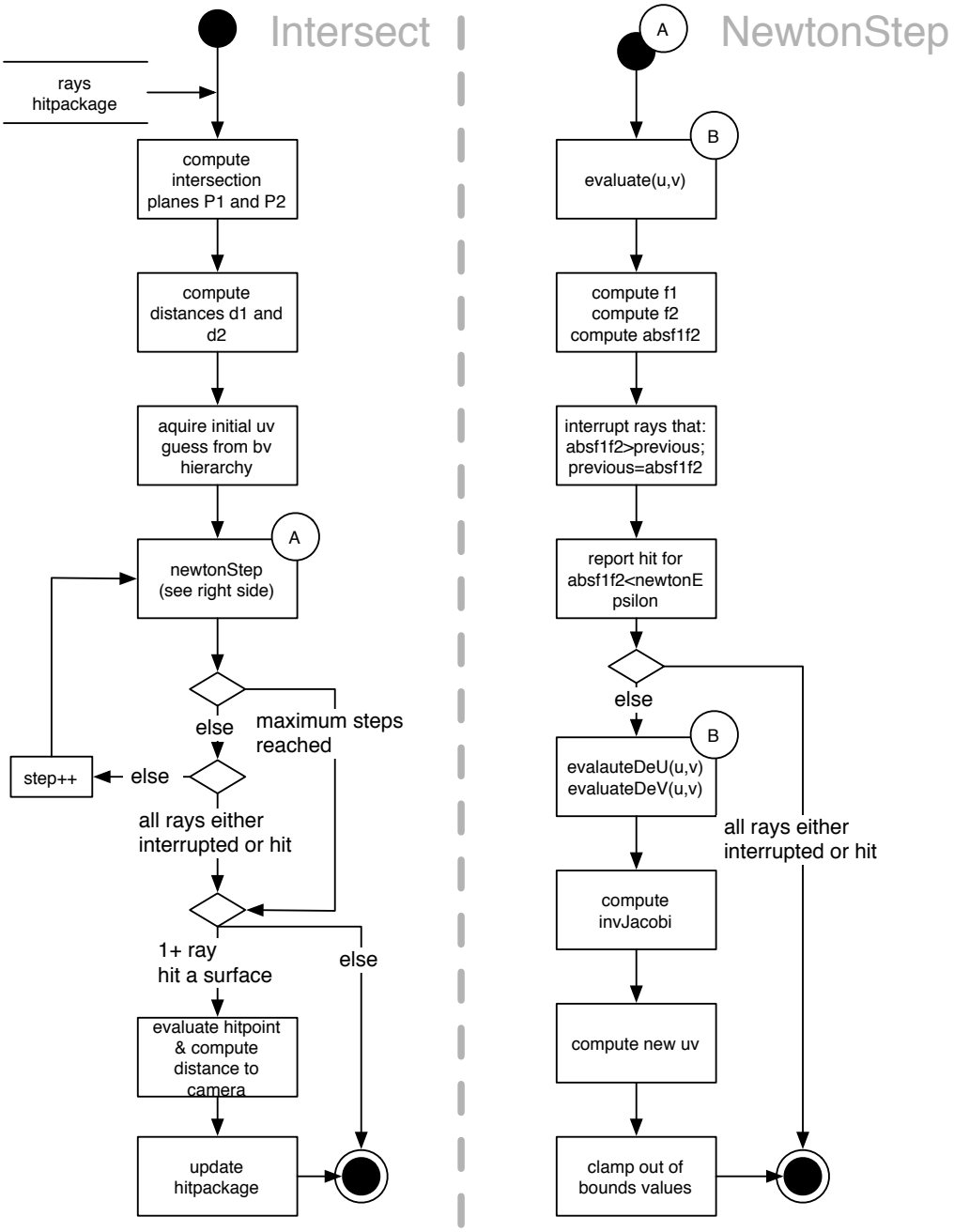

Figure 5.3: Control flow diagram of the intersection method. This diagram is applicable for any kind of rays, however, for computing shadow rays a faster approach has been implemented as it is sufficient to know if there is a hit point at all, rather than knowing the exact point. Evaluate resp. partial derivative are shown in figure 5.4

5.2.4 Evaluation of Surface Points

Every performance improvement during the surface evaluation will speed up the whole application significantly. As mentioned in section 3.4.3, the most efficient way is to pre-compute the basis functions and evaluate mere polynomials instead of the Cox-de Boor recursion. Depending on wether SIMD and/or GPU are used or not, the way of evaluating these polynomials will be different. Basically the same method with some modifications can be used for either evaluating a point on a surface or the partial derivative in u or v parametric direction.

The issue here is, that the derivative of a B-Spline Surface is rather simple, but when considering real NURBS surfaces, that have weights assigned to them, the derivative is becoming a bit more sophisticated to compute, since the quotient rule has to be applied. However, most of the time all weights will be one, because as mentioned earlier, the rational part of NURBS is not used very often[4]. Because of that several modules where implemented which will suit different purposes.

- highperf_nonrational: Offers the fastest evaluation of non-rational B-Spline surfaces (weights are all considered to be one, even if they are not) by using pre-computed basis functions as described in 3.4.4

- highperf_rational: Same as above, but adding support for rational surfaces as well. This variant is slower then the non-rational variant (see section 7.2 for comparison)

- baseline_nonrational: Also for evaluating non-rational B-Spline surfaces, but by using the most straightforward approach as shown in section 3.4.1. This module is usable for benchmarking and comparison.

- baseline_rational: Same as above with rational surface support

Because even a single *if* condition (to check wether the surface is rational or not) will be expensive when employed in the most inner loop, it is also possible to compile the rational module completely out, for a maximum of performance.

Figure 5.4 shows the general control flow chart of the evaluate method. It is more explained in detail in the following section.

[4]For instance, even Alias' Maya is not able to modify weights by default. This is only possible by writing a script

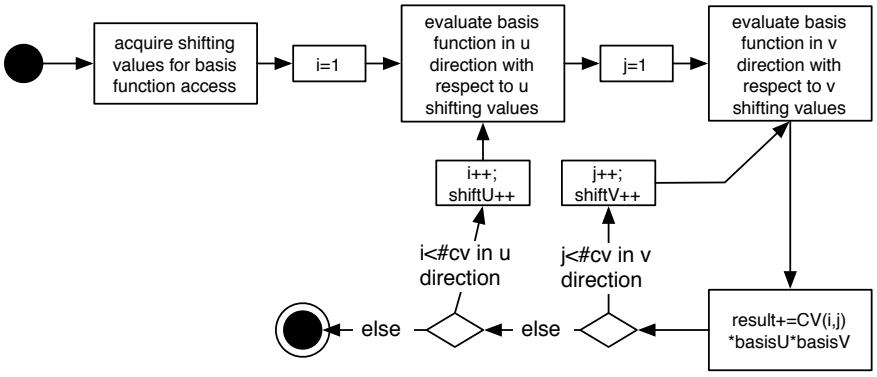

Figure 5.4: Control flow diagram of the evaluation method. Special attention has to be paid to the shifting values to access the correct pre-computed basis functions

Evaluating using SIMD

Basically the evaluation of a point on a NURBS surface is the result of equation 2.10 for any given u and v parameter value pairs. For non-rational surfaces this reduces to:

$$S(u,v) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B_{i,j} N_{i,k}(u) M_{j,l}(v)$$

The biggest different to a straight forward approach is probably the correct access and evaluation of the pre-computed basis functions. Every NURBS surface stores a pointer to the first basis function, which is describing the first knot vector interval in u parametric direction. Refer to section 3.5.1 for a more detailed discussion about that.

However, there are some issues when employing SIMD instructions with pre-processed basis functions. Recall, that always pairs of four values are processed in parallel. The basis functions are dependent on the current u or v parameter values, which means, that it can occur, that two or more different basis functions have to be used for a single evaluation. This can not be done efficiently in parallel, thus it has to be done sequentially. In the worst case this can result in four sequential evaluations, if all values within a single SIMD vector lie in different knot vector intervals. However, as the rays are usually more or less coherent, all lie most of the time within the same interval or seldom in two adjacent intervals when the rays are crossing a boundary. Still, the worst case is simply comparable to FPU computation.

Furthermore there is another issue with the way data is stored in SIMD registers. It is not possible to randomly access single components of one SIMD register, as the specific access to a single of the four float components has to be known at compile time, as mentioned before. In order to be able to use the efficient Horner scheme to evaluate the basis functions, the data has to be copied (i.e. splatted) into SIMD vectors. For example the SIMD vector $a = \{1,2,3,4\}$ will be copied into four vectors $v_1 = \{1,1,1,1\}, v_2 = \{2,2,2,2\}$ and so on. In the former case, random access to the individual components of a during runtime is not possible, where v_i can be accessed at will, of course. The following code snipped shows the SIMD code for Horner's scheme

```
float4 result = data[0];
result = simd_madd(data[0],t,data[1]);
float4* dp = &(data[1]);
for (int i = 1; i < degree; i++)
    result = simd_madd(result,t,*(++dp));
```

where t denotes the parameter value which is employed and *data* is an array of *flaot4* vectors holding the splatted values as described above. Note, that if the derivative is requested, the *data* array is modified accordingly and the degree is reduced by one beforehand.

Evaluating using the GPU

On the GPU the polynomials are evaluated by using brute force, that is by simply using the *pow()* built-in function. Usually this would not be very clever, since raising to the power of an arbitrary number is extremely expensive to compute and thus can not compete in any way to the Horner's scheme used on the CPU. However, due to it's graphics centric architecture, the GPU lacks some important abilities, like dynamic access on an array during runtime (i.e. the access index has to be determined during compile time), similar to the component access of a SIMD vector. Implementing Horner's scheme on the GPU is only possible with a considerable amount of workarounds, which would drop performance significantly. Luckily the *pow()* command does not mean a performance loss on the GPU, as it is mapped to a hardware function and thus brute force evaluation is not only necessary but also reasonable.

Algorithm 6 shows the corresponding GLSL code for evaluating basis functions. *PIXELPERFUNCTION* denotes the constant, which specifies how many pixels are used to store the basis function. Furthermore, *basisFunctions* is a global shader constant referring to the texture storing all basis functions and *accessU*/*accessV* denote the coordinates where to find the correct basis function.

Algorithm 6: Function evaluation on the GPU

```
float evaluateBasisFunctionFloat4
     (const float accessU, const float accessV,
      const float t, const int PIXELPERFUNCTION){
    float result = 0;
    //this is the first pixel storing the degree in its first
    //component and the three coefficients with the highest
    //degree
    const vec4 p = texture2DRect(basisFunctions,
                                 vec2(accessU, accessV));
    const int degree = p.x;

    //independent of the degree the first pixel will always
    //be read and processed. Note that components might be
    //zero if they are not used and thus do not influence
    //the result
    result = p.y*pow(t, degree)
           +p.z*pow(t, max(degree-1,0.f))
           +p.w*pow(t, max(degree-2,0.f));

    //process more pixels depending on the degree
    for (float i = 1; i < PIXELPERFUNCTION; i++){
     //read next pixel, storing four more coefficients
        const vec4 p = texture2DRect(basisFunctions,
                                     vec2(accessU+i, accessV));
        result = result + p.x*pow(t, max(degree-(i*4+1),0.f))
                        + p.y*pow(t, max(degree-(i*4),0.f))
                        + p.z*pow(t, max(degree-(i*4-1),0.f))
                        + p.w*pow(t, max(degree-(i*4-2),0.f));
    }
    return result;
}
```

5.3 Intersection Test Ray-Bézier Surface

As mentioned before, bicubic Bézier surfaces are also supported, as a high performance alternative to NURBS surfaces. An in depth discussion would exceed the scope of this work, thus in the following only a short summary will be presented. Refer to [1] for more details, if desired.

The main difference between NURBS and Bézier surfaces during the intersection test lies principally in the evaluation of points and partial derivatives only. Everything else, like the Newton Iteration and bounding volume hierarchy is fully compatible and fortunately works in the very same way. The bicubic Bézier representation is a lot faster than NURBS (see section 7.6 for results), however, it is not as flexible, since such a surface always has 16 control points and a cubic degree in both parametric directions. The performance advantage is mostly due to a very powerful matrix representation of the surface. Every bicubic Bézier surface can be expressed as a matrix product

$$\mathbf{P} = \mathbf{UNBNV}$$

where for bicubic patches

$$\mathbf{N} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Furthermore $\mathbf{U}$ is a matrix composed of the potencies of the parameter value u, where the surface shall be evaluated

$$\mathbf{U} = \begin{bmatrix} u^n & u^{n-1} & \dots & u & 1 \end{bmatrix}$$

analog to that

$$\mathbf{V} = \begin{bmatrix} v^n \\ v^{n-1} \\ \vdots \\ v \\ 1 \end{bmatrix}$$

Finally $\mathbf{B}$ is the control point matrix, i.e. every component holds a control point, thus a three component vector. Note that the inner product of $\mathbf{P} = \mathbf{NBN}$ can be pre-computed, yielding a single 4×4 matrix with 3D vector components. The necessary computation during runtime is only the product of three matrices, i.e. $\mathbf{UPV}$, which is, compared to the operations required for NURBS, rather simple.

The partial derivatives are also easy to compute. The derivative in u direction requires only $\mathbf{U}$ to be derived thus straightforward yielding

$$\mathbf{U}' = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix}$$

Derivative in v direction is analog.

5.4 Trimming Test

A trimming test has to be performed, if at least one ray in a *RayPacket* has hit a surface and the bounding box, which started the Newton iteration, has the *doTrimming* flag set (compare section 3.3.1 on pre-classifying). Obviously, rays intersecting a non trimmed surface report their hits immediately, without performing any trimming operations. Rectangular parametric regions, which are always trimmed have been removed during the preprocessing (see 3.3.1) and thus can not be hit at all.

As mentioned earlier in section 3.3.2 trimming curves are required to be closed. If that is assured then the following method can be applied to test wether a specific point lies within a trimmed region or not.

By shooting a 2D ray (in the parametric domain) in any direction, it is possible to count the number of intersections with trimming curves. If this number is odd, then the point from where the ray was casted, is enclosed by trimming curves (i.e. trimmed) otherwise the point is not trimmed. Figure 5.5 gives an example. Although this sounds very simple, it is still necessary to make it run efficient and fast, for example, it is not reasonable to cast a ray and perform real full featured ray tracing in 2D. An efficient solution is presented in the next section.

5.4.1 Classification of Hit Points

The following technique for fast hit point classification is based on an idea initially proposed by Nishita in [39], called *Bézier Clipping*, which got modified to suit the needs of interactive ray tracing.

First of all, Nishita is proposing to cast a ray in a direction, that the distance to the border of the parametric domain will be minimized. However, seems not reasonable in this context, as the computation to find the shortest axis itself takes a few cycles and after all it potentially does not really reduce the number of curve segments that have to be tested. Additionally the algorithm would get more complicated, as it would have to handle more general cases, so it is actually more feasible to cast the ray always along one parametric direction, which is the positive u axis here (although any other

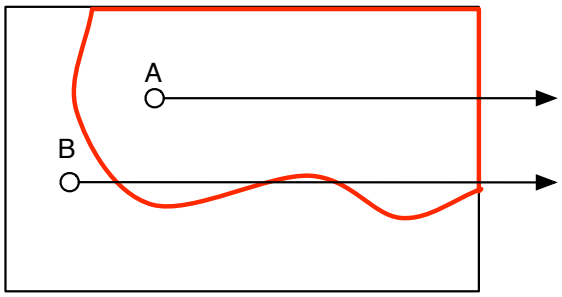

Figure 5.5: 2D ray tracing - red lines indicate trimming curves. The ray originating from A intersects once (odd - so A is trimmed) whereas the Ray from B intersects four times which is even, thus B is not trimmed

axis would be fine as well).

Furthermore *Bézier Clipping* was designed to find an intersection between a line and a Bézier curve. Here however, the question reduces to wether there is *any* intersection at all, or not. Thus, the complexity reduces as outlined in the following.

The point, which is being tested, defines the origin of a new local coordinate system by using the u and v parametric directions as axes. Four quadrants are defined, the same way they are usually defined (i.e counter clockwise and top right quadrant is the first). Figure 5.6 is showing an example where all four control points of a Bézier curve lie in the first quadrant. Due to the convex hull property of Bézier splines[5] there can not be any intersection between the curve and the positive u axis, which is actually the ray being casted along positive u parametric direction. Even if all control points lie within quadrants II and III, there can not be any intersection, as the complete curve lies behind the ray. However, an intersection is guaranteed if all control points are located in quadrants I and IV and the fist and last control points are not in the same quadrant. The following table lists all possibilities and the action necessary to take.

[5]The convex hull property of a Bézier curve is actually identical to a NUBRS curve with open knot vector and the order equalling the number of control points - compare section 2.2.5

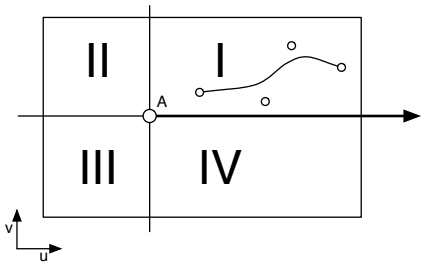

Figure 5.6: Point A is tested for trimming. A ray is cast originating from A along positive u axis. All control points of this trimming curve lie in quadrant I thus there can not be an intersection due to the convex hull property

Control points	
All in I	no Intersection -> next Curve
All in II	no Intersection -> next Curve
All in III	no Intersection -> next Curve
All in IV	no Intersection -> next Curve
All in I & II	no Intersection -> next Curve
All in II & III	no Intersection -> next Curve
All in III & IV	no Intersection -> next Curve
All in I & IV	$q(p_1) \neq q(p_4)$ & Intersection -> invert in/out status
else	undefined -> subdivide curve & repeat for both

It becomes obvious, that in most cases the classification for a single curve segment is already done after a single step. Only if a curve, for example, is passing quadrants I, II and III a subdivision is necessary, in order to solve the problem for both smaller curves. As mentioned before, a hit is guaranteed, if the first and last control point lie in quadrants I and IV, but not within the same, whereas the other two have to be in I and IV as well. In that case it is actually only known, that there is an intersection at all, but it is unknown how many. However, as trimming curves have been restricted to cubic degree in the beginning, it can be either one or three intersections and this in turn does not play any role for the trimming test, because it is only interesting to know, wether there is an even or odd number of intersections. In the case of cubic curves it is always an odd number of intersections, if there is any at all. Thus, whenever any intersection occurs, it is sufficient to flip a boolean value indicating if the point is trimmed or untrimmed.

However, if none of the definite cases is applicable, the curve has to be subdivided, since it is not possible to decide the problem on basis of this curve. Nishita introduces in [39] an approach with which it is possible to

divide the curve into three segments, where two of them are assured to be one of the definite cases and the third (i.e. middle) segment comes under another scrutiny. Tests have shown that in the context of interactive ray tracing it is more efficient to employ a rather brute force approach, which simply subdivides the curve in the parametric center. Otherwise, following Nishita's approach, it would be necessary to compute two subdivision parameters which is considerably expensive on its own. Theoretically it can happen that an (nearly) endless loop of subdivisions occur, but this is most unlikely anyway, so that a proper solution to that problem is to simply interrupt subdivision after, for example, 25 steps. Tests have shown that this happens with a probability of less than $10^{-8}\%$ and even then it only results in up to four wrong pixels for one frame.

As mentioned before, a trimming curve is actually composed of a number of segments. Each segment has to be tested individually as described above, however, the boolean flag storing the trimming status (i.e. in or out) has to be carried over, of course. After all trimming curves segments have been processed sequentially, the boolean flag indicates the final status of the point that was being tested.

5.4.2 Subdivision of Trimming Curves

Obviously two important requirements for trimming curve subdivision are performance as well as accuracy (i.e. the two resulting curves concatenated should be identical to the original one). As only cubic curves are considered, the well known *de Casteljau* algorithm [6] is perfectly suited for that purpose. The idea is outlined in the following.

In order to subdivide a Bézier curve of arbitrary degree at some parameter value t with $0 \leq t \leq 1$, it is necessary to recursively compute the vectors between two neighboring points. These will be scaled by factor t and the whole process is repeated for the resulting points until only one point is left, which is the point on the curve at that parameter value. In every step the number of resulting points is reduced by one, thus when subdividing cubic curves, three steps are necessary, which is still quite reasonable, but might be too costly if higher degree curves were considered. The interesting thing is, that the intermediate points, that were computed during the process, are the new control points for the two refined curves, which makes de Casteljau's algorithm very valuable.

For the following example points P_i denote the control points of the original curve, where A_i and B_i refer to the two curves resulting after subdivision ($0 \leq i \leq 3$). Figure 5.7 is showing a simple example where a curve is subdivided at $t = 0.75$. It is known, that the first and last control point of a

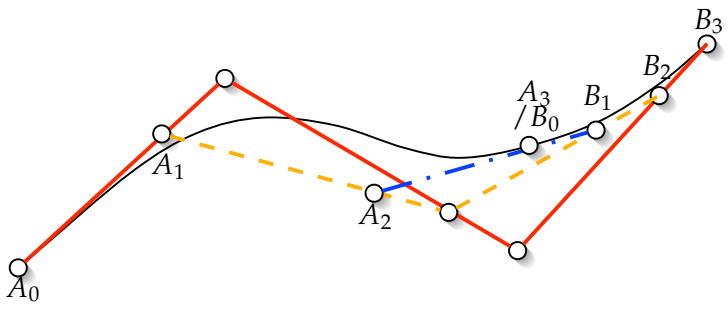

Figure 5.7: de Casteljau's algorithm for a cubic spline being evaluated at $t = 0.75$

Bézier curve lie always on the curve itself, furthermore the point evaluated at t is also to be known on the curve and so is the value, where the two are actually connected to each other. Thus for the new segments A and B, it is already know that

$$\begin{aligned} A_0 &= P_0 \\ A_4 &= eval \\ B_0 &= eval \\ B_3 &= P_3 \end{aligned}$$

where eval denotes the evaluated point (i.e. A_3/B_0 in figure 5.7). The other four missing control points (two per curve) are, as mentioned, the intermediate results of the de Casteljau algorithm.

$$\begin{aligned} A_1 &= t * (P_1 - P_0) \\ A_2 &= t * (t * (P_2 - P_1) - A_1) \\ B_1 &= t * (P_3 - P_2) \\ B_2 &= t * (t * (P_2 - P_1) - B_1) \end{aligned}$$

Note again, that only cubic curves are considered, so it is possible to implement a SIMD powered subdivision algorithm which due to some necessary overhead is not optimally fast, but still faster than an implementation using the FPU. For example in the first step of the algorithm it is possible to compute all vectors from P_i to P_{i+1} with $(0 \leq i \leq 2)$ in parallel using SIMD instructions which is illustrated in figure 5.8. In the next two following steps the parallelism is reduced by one each time.

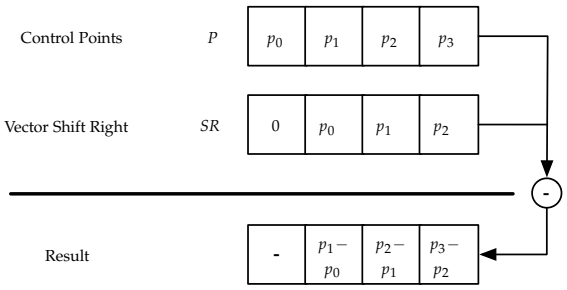

Figure 5.8: First step of the de Casteljau algorithm using SIMD

5.5 Memory Consumption

Memory consumption for a single patch is obviously higher than for a triangle, but overall memory usage can compete against triangular meshes quite well, especially if those are of high resolution. As mentioned before, a single free form surface can describe a shape that would have been approximated by several thousand triangles otherwise, if a high resolution representation is desired. The following tabular lists the memory needed by each object type and their components. The abbreviations used are: Control Points (CP), Knot Vectos (KV), Basis Functions (BF) and Trimming Curves (TC).

Object	Memory (bytes)
NURBS patch	104+CP+KV+BF+TC
Bézier patch	224 + TC
CP	16 per Vertex
KV	$max(degreeU + countU + 1, degreeV + countV + 1) * 16$
BF	$c * PIXELPERFUNCTION * 16$ (for c see 3.1)
TC	#Curves * 32
Bounding Box	52
NURBSKernel	92
Triangle[6]	48

The following example shows the memory consumption for the *NURBShead* scene. The head consists of 915 NURBS surfaces, each having 4×4 control points, degree 3 in both directions, yielding $[0\ 0\ 0\ 0\ 1\ 1\ 1\ 1]$ as the knot vectors. This will result in eight basis functions per patch, requiring two *float4* vectors per function. Furthermore 13842 bounding boxes are generated and no trimming curves are used. Memory consumption therefore is

[6] as used in *dream*

- NURBS surfaces: 95160 bytes
- Control Points: 234240 bytes
- Knot Vectors : 117120 bytes
- Basis Functions: 234240 bytes
- Bounding Boxes: 719784 bytes

The total memory consumption for this scene is only around 1.33 MB for a virtual unlimited high resolution model. Comparing this to a triangle based ray tracer and assuming the same memory usage for the bounding volume hierarchy, around 680 KB would be left for triangle storage. This is enough to store more than 14000 triangles. With that number of triangles the head could be modeled, of course, but it might not be sufficient for high detailed close up shots.

If the same scene would be represented as Bézier surfaces instead of NURBS the memory consumption is reduced further to only 1.04 MB (0.83 MB for bounding boxes) would be needed, leaving only an equivalent of 4270 triangles, which is really not sufficient for a sophisticated head model.

5.6 Problems and Limits

NURBS surfaces can easily become very complex. Naturally there are some limits within which the interactive ray tracing of NURBS surfaces is feasible and reasonable.

In section 7 it is shown, that it is possible to render thousands of NURBS surfaces with interactive frame rates, even on a single modern commodity PC. However, this requires that the surfaces are within a certain reasonable range. On the one hand, the *baseline* algorithm can render any complexity, but interactive frame rates are out of sight pretty fast, whereas on the other hand, the *highperf* algorithm keeps it's speed quite well, but loosing accuracy pretty fast, because of the limited precision of floating point values. One possible workaround is to use double precision when enabling the FPU emulation mode, however this of course is reducing performance a lot, as well. A possible solution to that problem is outlined in chapter 8.

5.6.1 Insufficient Floating Point Precision

Numerical inaccuracy becomes a serious problem, when there are a large number of control vertices with a low or medium degree of the basis functions. The reason is, whenever this situation occurs, a great number of basis

functions has to be generated, which each cover only a very small region of the parametric domain, This in turn means, that their coefficients will become rather large. Unfortunately single precision floating point values (after IEEE-754 [37]) are only able to store seven different figures. The following example shows two of the basis functions, that are used in the a simple B-Spline Patch, taken from [34], but instead constructed out of 13^2 control vertices with degree 5 in each parametric direction.

$$M^s_{9,5}(v)_{0.875<v\leq 1} = 3117.51v^5 - 14734.2v^4 + 27761.8v^3 - 26055.1v^2 + 12174.2v - 2264.18$$

$$M^s_{11,5}(v)_{0.875<v\leq 1} = 43614.8v^5 - 193896v^4 + 344130v^3 - 304803v^2 + 134803v - 23810.4$$

However, when these basis functions are computed using double precision floating point values, then the result is quite different, as follows

$$M^d 9,5(v)_{0.875<v\leq 1} = 3117,5111111111v^5 - 14734.2222222222v^4 + 27761.7777777778v^3 - 26055.1111111111v^2 + 12174.2222222222v - 2264.1777777778$$

$$M^d 11,5(v)_{0.875<v\leq 1} = 43614.8148148148v^5 - 193896.2962962963v^4 + 344130.3703703703v^3 - 304841.4814814815v^2 + 134802.9629629629v - 23810.3703703704$$

By comparing the different coefficient values, the discrepancy between the two precisions become obvious. Even more serious is the fact, that this applies to many of the basis functions of such a surface. When evaluating a point on the surface or a normal, then these errors accumulate, which in turn can lead to rather inaccurate results. For example, evaluating the surface with single precision at $u = 0.95$ and $v = 0.95$ results in

$$(15.1864, 1.80162, -14.681)$$

whereas double precision yields

$$(14.6165208918, 1.7701131564, -14.1254754908)$$

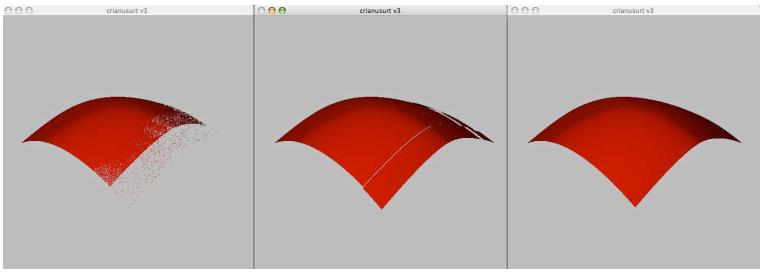

Figure 5.9: Rendered surface with 13x13 control points and degree 5 in both parametric directions. From left to right: Rendered with single precision (FPU and SIMD yield same result), double precision (only FPU), baseline

The absolute distance between the two different results therefor is 0.32575!

Obviously, this causes problems during the intersection tests, since the iteration will converge to a point in space, that is not on the ray and thus the intersection test wrongly will report a miss. An easy workaround is to soften the threshold *epsilon* value for intersection acceptance, however this in turn may cause hits to be reported where there are none. Figure 5.9 shows the mentioned cases. For very complex surfaces only the baseline variant yields a correct result, however unfortunately with far from interactive speed. By using double precision, it is possible to handle more complex surfaces, but 64 bit floating point values are not supported by libSIMD[7], so this mode is only available when using the slow FPU emulation mode. Tabular 7.3 shows the rendering times for the different methods. The rendering times for the *teapot* and *NURBShead* can also be found for comparison. Note that both latter scenes are rendered correctly by any of the mentioned methods.

5.6.2 GPU Issues

It turned out, that there are some severe issues when using the graphics card for ray tracing of NURBS, which in the end, made it not worthwhile to use it yet. The three major problems are as follows

- Limited number of temporary registers
- Limited execution length
- Performance

[7] SSE is supporting double precision floats with half performance, where AltiVec does not support it at all

The available amount of memory however, which seemed to be a problem in the first place, is not a real concern, unless extremely large models should be rendered. Unfortunately there are currently no solid official numbers available, which state how many temporary registers the GeForce 6800 card really has nor what the limit of executed commands is. Even dedicated tests showed an unexpected behavior by actually allowing a greater number of temporary registers then in the intersection code. This effect is most likely due to some optimizations in the compiler, which makes it rather hard to estimate the number of registers really being used. However, it seems that the following statistics are true for the GeForce 6800

- Number of constant registers: 224
- Number of temporary registers: 32
- Number of lines of executed code: 65000+

The actual problems are the number of temporary registers and the limit of executed code. The latter is unlimited by hardware design, but it is in fact limited by the driver.

Temporary Registers

The effect of exceeded temporary registers is unfortunately a simple instant termination of the application, with the error message *floating point exception* displayed on the console, which, in the first place, does not help much. Because of this, a lot of unfavorable workarounds had to be implemented and features like trimming had to be skipped completely. Another possible workaround would have been a multipass approach, i.e. to render these temporary values into textures where they can be read again later, but this doesn't even enter the equation as it is much to slow in an interactive context.

Execution Length

From the hardware design the GeForce 6800 can executed an unlimited number of commands, unless the predecessors. However, it would be unfavourable to send the graphics card into an endless loop, effectively freezing the screen, thus the driver does employ a limit, which seems to be 2^{16} commands. Unfortunately this is, depending on the surface, only enough for one or two steps of the Newton iteration, which is simply not sufficient. Usually four or more steps are needed to obtain a result with an error below 10^{-3}. Also here a multipass approach could help in theory (every step of the Newton iteration as one pass), but as mentioned above, this would reduce performance considerably, making it useless in an interactive setting.

Performance

Ignoring all issues encountered so far, general performance is, by far, not as high as necessary for interactive frame rates. Even a single surface, which did not even render correctly due to the mentioned problems above, took more than a minute, when the rendering was done on the GPU only. Comparing these to the results achieved with the CPU solely, it is obvious that, when using them in parallel, the GPU would compute less then 1% of the pixels - and again these were even not correct results because the computation could not be completed because of to the execution limit. Although there might be room for runtime optimization, the complexity still remains an severe issue.

Chapter 6

Usage

The usage of *crianusurt* is very easy, but beforehand all required libraries and applications have to be installed. Follow these commands for installing everything needed. The given examples will work on both Linux/x86 as well as MacOS/PowerPC.

Installing *libSIMD*

```
//change to libSIMD directory
./bootstrap.sh
./configure --enable-includedir=/usr/local/include/SIMD
make
//change to superuser
make install
```

Installing *libNURBSIntersectionKernel*

```
//change to libNURBSIntersectionKernel directory
./bootstrap.sh
./configure --enable-includedir=
                /usr/local/include/NURBSIntersectionKernel
make
//change to superuser
make install
```

Installing *iges2dsd/crianusurt*

```
//change to iges2dsd/crianusurt directory
./bootstrap.sh
./configure
make
```

The order is important, as both, *iges2dsd* as well as *crianusurt* are dependent on *libNURBSIntersectionKernel*, which itself is dependent on *libSIMD*!

It is possible to append a –enable-fpu option to **all** configure commands, which will force the usage of the FPU emulation mode, rather than using the appropriate SIMD instruction set. This however, should only be used for speed comparisons, since it is a lot slower than the SIMD enabled variant. On systems that don't offer SIMD functionality the FPU mode will be used automatically. The option –enable-debug can be employed to generate unoptimized code that can be debugged. Any of these options have to be applied to all libraries and applications, especially applying the –enable-fpu option to only one package will prevent the other libraries/applications from working at all!

6.1 *iges2dsd* Usage

As discussed before, *iges2dsd* is responsible for converting *.iges* or special *.wrl* files into the binary *.dsd* format used by *crianusurt*. While converting, all necessary preprocessing is done, like precomputing the basis functions and generating the bounding volume hierarchy.

The application is called by

```
iges2dsd [options] -i inputfile -o outputfile
```

Options can be any of the following

- -d maximum subdividing depth during bv-hierarchy generation
- -w swap the y and z coordinate axis
- -b use the dynamic bounding volume generation (see 3.2.2)
- -t must be between 0 and 1. The closer to one, the more and more exact bounding boxes will be generated
  [requires: -b](default: 0.85)
- -c read the configuration file *inputfile*.cfg to sets various model parameters (i.e. like camera, lights and scale)

The time needed for converting is usually only a few seconds. For large scenes with a lot of subdividing the process can take up to several minutes. Some preprocessing times can be found in section 7.1.

6.2 *crianusurt* Usage

Crianusurt is the actual rendering application. It needs a *.dsd* file as input. Besides the two libraries *libSIMD* and *libNURBS*, it also requires an OpenGL environment, which should be part of the operating system anyway. *Crinsusurt* is launched with

```
crianusurt [options] -f inputfile
```

Options can be any of the following

-w width	width of the output window (default:512)
-h height	height of the output window (default:512)
-t num	number of threads for rendering/shading (see 3.2.2)
-p speed	set panning speed (default:1.0)
-P steps	set panning steps (default:0.1)
-r speed	set the rotation speed (default: 10.0)
-R speed	set the rotation steps (default: 1.0)
-e epsilon	sets the epsilon value for the intersection test (Default:10^{-3})
-n	disable trimming (default: enabled)

Some of these parameters can also be changed during runtime using the following keys

+	increase panning speed by *panning step*
-	decrease panning speed by *panning step*
Shift +	increase rotation speed by *rotation step*
Shift -	decrease rotation speed by *rotation step*
w	enable/disable trimming
q	increase epsilon for intersection test by factor 10
Q	decrease epsilon for intersection test by factor 10

Additionally there are several other keys which influence the behavior of the application. These are listed in the following tabular for reference

Esc	terminates the application
D	increase recursion depth for secondary rays
d	decrease recursion depth for secondary rays
b	activate/deactivate benchmarking mode
n	force rendering of new frame
j	activate/deactivate antialias mode
s	activate/deactivate shadows
r	activate/deactivate reflections
e	activate shadows and reflections
x	activate/deactivate rotation around x axis
y	activate/deactivate rotation around y axis

Finally you can use the arrow keys to move the camera (look-at point stays as is) and while holding shift, the arrow keys will pan the camera (look-at point also moves)

Chapter 7

Results

In this chapter results can be found, that were achieved with the application presented in this work. After introducing the test systems, section 7.1 describes the scenes used in several benchmarks. Sections 7.2 thru 7.10 show results under different aspects, including comparison between FPU and SIMD, single and double precision, different architectures, different bounding box generation approaches and more. Throughout this chapter some shortcuts are used for convenience. These are listed here for reference.

- *non-rationl*: support for rational surfaces is compiled out, i.e. rendering non-uniform B-Spline surfaces
- *rational*: support for rational surfaces is compiled in, i.e. real NURBS surfaces
- *highperf*: high performance evaluation method is used
- *baseline*: low performance, straight forward evaluation method is used
- *singleprec*: single precision floating point values are used, default for all SIMD operations
- *doubleprec*: double precision floating point values are used, can only be used with the FPU emulation mode

For achieving the results, one or more of the following four test systems were used with configurations listed below.

PowerMac

This Macintosh desktop computer is equipped with two G4 (aka. PowerPC 7455 revision 3.2) processors each running at 1.25 GHz with 166 MHz bus speed. The system has 1024 MB PC2600 of main memory. Each processor

has its own L1 (64 KB), L2 (256 KB) and L3 (2 MB) cache, but the bus to the processors is shared between both. The Operating system is Mac OS X 10.4.2.

PowerBook

This portable Macintosh computer sports a G4 processor running at 1.67 GHz, also with a bus speed of 166 MHz. In difference to the desktop system, this is a PowerPC 7447A revision 1.2, which is optimized for mobile operation. There is no L3 cache, the L1 cache has 64 KB of memory, whereas the L2 cache comes along with 512 KB. The main memory is 1.5 GB of PC2700 DDR RAM. The Operating system is the same as on the PowerMac.

PC

A desktop computer featuring a 3.0 GHz Pentium 4 processor with 1024 MB of main memory. Operating system is Gentoo Linux with a 2.4.26 Kernel. The processor supports Intel's hyper threading technology.

Prism

SGI's Prism in the configuration used, features six Intel Itanium 2 64 bit processors each running at 1.5 GHz connected with a 400 MHz bus. Every processor has it's own caches, that is 32 KB level 1, 256 KB level 2 and 4 MB level 3 cache. Unfortunately the Itanium processor does not feature a SIMD unit, thus computation has to be done using the FPU emulation mode. However, each processor features two full floating point units, which both are capable of computing two operations in parallel. The machine is equipped with 16 GB of RAM. The operating system is Fedora Core Linux.

G5

Another Macintosh Desktop Computer, that was used to test the performance on Apple's newest machines. This particular one was equipped with dual G5 (64 KB lvel1 and 512 KB level 2 cache) processors running at 1.8 GHz, connected over a 900 MHz bus to the rest of the system. Available memory was 1GB DDR RAM. Operating system is Mac OS X 10.3.9.

7.1 Test Scenes

Before going into detail, the used test scenes are shortly described. To compare results with each other, it matters what kind of surfaces are being rendered, i.e. the number of control points and the degree of the basis func-

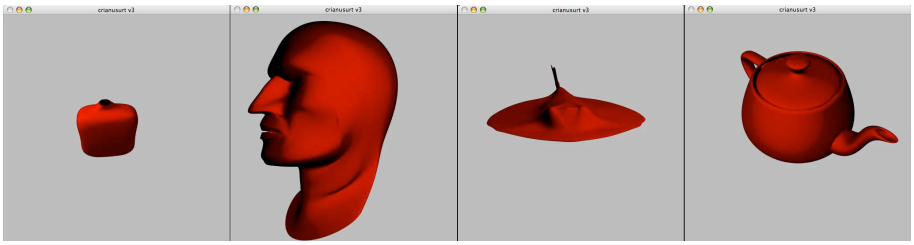

Figure 7.1: Test scenes from left to right: parfum, head, stingray, teapot

tions that are employed. Unless otherwise stated the following parameters are always passed to the converter *iges2dsd*

```
iges2dsd -b -d 6 -t 0.85 -c [-w] -i inputfile -o outputfile
```

Have a look at section 6.1 for more details about the individual parameters. Note that *-w* (swapping Y and Z axis) is only used where needed and has no performance impact at all. Furthermore, all results were achieved by using two *float4* values for basis function storage, effectively limiting the maximum degree to six. Also unless otherwise stated, all scenes use the improved bounding box generation algorithm (surface flatness criterion) corresponding to the parameter -b. By default, all scenes are represented by NURBS surfaces, not Bézier!

All results shown in figure 7.1 are actually bicubic Bézier models represented by uniform B-spline surfaces (unless otherwise stated). The following table states of how many patches the individuals scenes consist and also how many bounding boxes will be generated when using the parameters given above.

Scene	Surfaces	Bounding boxes	approx. Preprocessing time
Parfum	110	3090	2 sec.
Head	915	13842	7 sec.
Stingray	1160	26268	14 sec.
Teapot	32	12985	7 sec.

From this result it seems that the preprocessing time is heavily dependent on the number of bounding boxes generated rather than on the number of surfaces in a scene.

7.2 Comparsion: Rational vs. non-rational

The following results show, that the non-rational variant is more than 20% faster than the rational variant, even if all weights are 1, which effectively

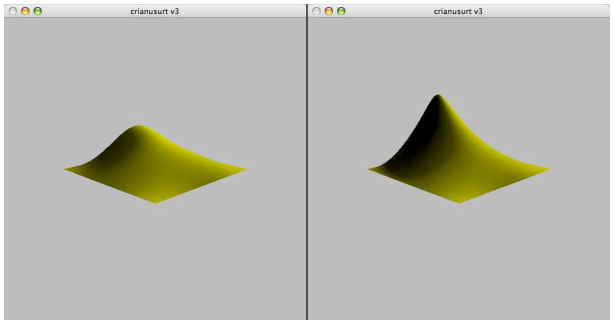

Figure 7.2: Example NURBS surface from Roger's book (see [34]). On the left all weights are 1, which is a simple non-rational B-Spline surface. On the right $h_{4,3}$ is set to 5 thus defining a real NURBS surface

yields the same surface. However, especially the derivatives are computationally more expensive, since the quotient rule has to be applied. Figure 7.2 shows both surfaces with and without using weighting factors. The weight $h_{4,3}$ corresponds to the uppermost control vertex $B_{4,3}$ and thus the surface is pulled up towards that control point. Tabular 7.2 shows some results. Note that the two results which have a weighting assigned at $h_{4,3}$ actually modify the geometry in a way, that more pixels are rendered and tested for intersections (compare the images) and thus the frame rate is slower also due to this fact.

Kind	Weighting at $h_{4,3}$	variant	fps	efficiency
non-rational	no	highperf	2.75	2291%
rational	no	highperf	2.26	1883%
rational	yes	highperf	1.47	1225%
non-rational	no	baseline	0.12	100%
rational	no	baseline	0.10	83%
rational	yes	baseline	0.05	41%

Similar speed impacts were obtained when rendering the Utah teapot. Tabular 7.2 shows some more results.

Kind	variant	fps	efficiency
non-rational	highperf	1.35	2596%
rational	highperf	1.01	1942%
non-rational	baseline	0.052	100%
rational	baseline	0.043	82.6%

Note that the teapot consists of very low complex NURBS surfaces, which are actually bicubic Bézier surfaces represented as B-Splines. However, the

performance gap between the high performance and the baseline evaluation variants becomes even larger, the more complex the surfaces will become. This becomes clear in the next section.

7.3 Comparison: SIMD vs. FPU(single and double)

Theoretically the SIMD unit is four times faster than the FPU, however most of the times the factor is not reached due to some overhead involved when using SIMD instructions. As the following results will show, in this implementation the computation using SIMD is even faster than four times. This can be explained by taking into account, that the FPU mode is only an emulation mode, which means that actually arrays of four rays are being computed. This of course causes a certain amount of overhead. An highly optimized FPU based ray tracer would be able to outperform the emulation mode. Table 7.3 shows the results for different computation times depending on the method which was employed. Especially interesting is the huge performance gap when comparing *baseline* and *highperf* computation modes on the B-Spline Surface. Here the slow down factor from *highperf* to *baseline* is even roughly ten times larger as the slow down that occurs with both other scene. This is because of the more complex surface: the baseline algorithm complexity increases super linear with every added control vertex whereas the *highperf* algorithm does not!

Comparing double and single precision, the results show that performance drops between 25% and 40% where the latter of course is faster. This observation was expected, because double precision values (i.e 64 bit length) are always more costly to handle, especially for 32 bit processors (which the Macintosh and PC test systems are).

Method	Single(13x13/5x5)	Teapot	NURBShead
FPU single precision	11.5s	4.5s	12.4s
FPU double precision	15.5s	6.68s	19.9s
SIMD basline	479s	18.6s	55s
SIMD highperf	1.66s	0.74s	1.8s

7.4 Comparison: Architectures

In this section some scenes are rendered on different machines. Tabular 7.4 shows the results, where the following abbreviations correspond to the test machines described in section 7. For each system the optimal number of threads was used to take advantage of multiple processors respectively hyper threading.

Scene	PowerMac	PowerBook	P4	G5
Teapot (NURBS)	1.08 fps	0.70 fps	0.93 fps	1.09 fps
Teapot (Bézier)	3.65 fps	2.73 fps	4.56 fps	5.72 fps
Single (NURBS)	2.28 fps	1.58 fps	2.40 fps	2.43 fps
NURBShead (NUBRS)	0.41 fps	0.27 fps	0.33 fps	0.38 fps
NURBShead (Bézier)	2.63 fps	1.87 fps	2.55 fps	3.81 fps

By comparing the results gathered by the PowerMac and PowerBook, they are close to what is expected. Performance problems due to memory shortage can be ignored as the scenes fit into a fraction of the available free memory for all systems. Ray tracing is basically a very processor intense algorithm as long as not the bus marks a bottleneck. Taking into account the raw processor power only, we expect roughly the following:

$$fps_{powermac} = \frac{fps_{powerbook}}{1.336} * 1.85$$

The difference between the processor clocks yield the factor 1.336 (i.e $\frac{1.67GHz}{1.25GHz}$), where 0.85 is the assumed scalability factor for dual processors (see also next section 7.7). For the teapot in NURBS representation this means: $0.70/1.336 * 1.85 = 0.969$ which is quite close to the actually achieved 1.08 fps. The small difference might be due to architectural differences in both processors or similar reasons.

However, comparing the results on the PC with those achieved on a Mac, some unexpected surprises arise. The results for the teapot in NURBS representation are actually what is expected. The 3 GHz Pentium IV performs 32.8% better than the 1.67 GHz G4, but 13.8% slower than the dual 1.25 GHz G4. Considering, that PowerPC processors have a better GHz/Performance ratio and AltiVec is mostly superior to SSE the result seems to be reasonable. However, the same scene in Bézier representation yield a different view. In this case the Pentium outperforms both Macintoshs by 24.9% respectively by 67%, which might lead to the conclusion that Bézier surfaces run faster on the x86 architecture. Unfortunately this assumption can not be hold as the NURBShead in Bézier representation is actually slightly faster on the PowerMac. The reason for that is not really clear yet. Most likely it is a combination of several reasons, like SIMD instruction set differences, clock and bus speed discrepancies as well as other causes.

By having a closer look at the G5 results even more surprises arise. Usually the performance of a G5 is superior to a G4, also at (theoretically) same clock speeds. However, the scenes in NURBS representation on both Macintosh desktop computers are nearly the same. Although no specific G5 optimizations were employed so far for NURBS surfaces, this result seems strange in the first place. By taking into account that the AltiVec unit build

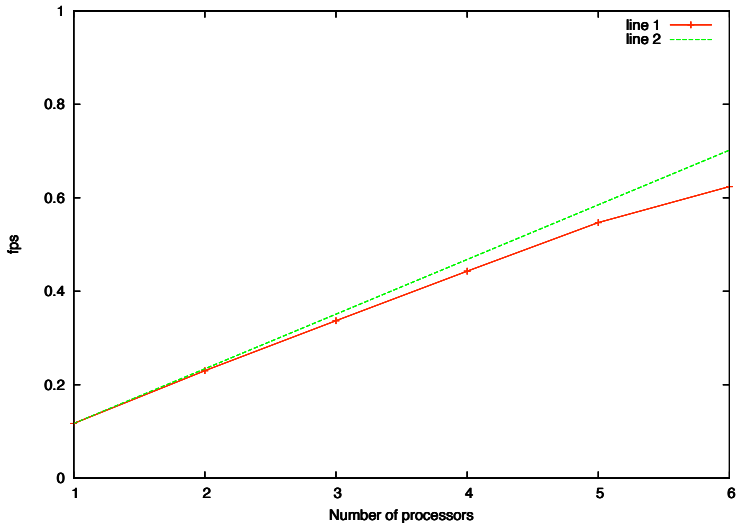

Figure 7.3: *crianusurt* performance on a SGI Prism. The dashed green line shows an ideal linear scale, where the red line shows, how the application scales with increasing number of processors

in G5 processors is not as powerful as compared to the G4, the results seem more reasonable. Looking at the Bézier representation the G5 is a lot faster, which is most likely due to specific optimizations implemented in [1], where the functionality is inherited from, for this kind of processor.

The Prism was tested at the very end, upon completion of this thesis, thus time was not sufficient to fully port the application. Because of this, only the teapot scene was benchmarked using the FPU emulation mode, since a SIMD unit is not available on Itanium processors. Additionally, double precision was used, as single precision floating point values caused some problems on the native 64 bit architecture. The following figure 7.3 shows, that the scalability is extremely good, but overall performance is quite poor compared to the other test systems. The maximum frame rate that was achieved using all six processors was 0.624 fps, which is even beaten by the PowerBook, that reached a top speed for the same scene of 0.7 fps. If the PowerBook is not using its SIMD unit and additionally also computes double precision values, the frame rate drops to 0.061 fps, which is nearly a factor 10 slower than the Prism. Anyway, the conclusion is, that it is a lot more feasible (especially if funds are limited) to use a multi processor machine with support for a SIMD unit rather than a SGI computer.

7.5 Comparison: Bounding Box Hierarchy Creation

As mentioned in section 3.2.2, two different variants were implemented to generate a bounding volume hierarchy. The generation on behalf of the convex hulls (CH) is expected to be less efficient regarding runtime and memory consumption than the variant that creates the hierarchy based on a surface flatness criterion (FT). Table 7.5 shows the performance of both approaches with a different maximum subdivision depth. Additionally it is stated wether the resulting image is rendered without artifacts or not. Other test scenes yield results that do not differ much relatively, so only the teapot is taken as an example.

Depth	Convex Hull	Flatness	Aftifacts (CH/FT)
6	0.071 fps	0.603 fps	none/none
5	0.077 fps	0.604 fps	none/none
4	0.103 fps	0.551 fps	none/none
3	0.088 fps	0.594 fps	none/none
2	0.067 fps	0.599 fps	some/some
1	0.116 fps	0.613 fps	many/some
0	0.312 fps	0.615 fps	lots/many

It is clearly visible that the FT method is superior to CH in both performance and quality. Even with a subdivision depth of 0 the FT method is roughly twice as fast and still yielding a better result. Beginning with depth 3, both approaches yield an artifact free result, but FT is about 6.75 times faster! Overall, the FT method remains quite constant with increasing depth, indicating that the number of bounding boxes do not have much impact on performance (but on image quality). Tabular 7.5 shows the number of bounding boxes generated for both approaches.

Depth	CH BB Count	FT BB Count
8	2604231	13067
7	651883	13067
6	164453	12981
5	41302	11963
4	10373	7100
3	2593	2824
2	623	767
1	165	195
0	49	195

The number of generated boxes for the CH method is constantly growing, roughly by factor four for each new subdivision step. However, FT stops

growing at depth 7, which is due to the other parameter bounding the generation - the flatness criterion. This indicates that according to the value (0.85 used here) all surfaces are sufficient flat enough and no further subdivision is necessary.

7.6 Comparison: dream, dreamBSE, crianusurt

The application *crianusurt* descended from *dream*, which provided the general SIMD ray tracer framework, and *dreamBSE* where the Bézier representation was taken from. Naturally quite interesting is a comparison between the original triangle based ray tracer *dream*, the Bézier based ray tracer *dreamBSE* and this application *crianusurt*. The following tests were performed on the PowerMac using both processors. The scene rendered is the teapot, since this is the only scene that was available for all three systems at this point of time. Figure 7.4 shows the three results rendered by these applications. Note, that the result rendered by *dream* is a bit different, because the original Bézier data could not be fed into the loader, since there is no triangulation routine for free form surfaces yet. However, the rendered plane should not make a big difference to the frame rate. The following table shows the frame rates for the different systems.

System	Frame Rate	Objects	Bounding Boxes
dream	1.58 fps	41050 Tris	19660
dreamBSE	4.5 fps	32 Bézier	12326
crianusurt	3.45 fps	32 Bézier	18875
crianusurt	1.07 fps	32 NURBS	12981

The NURBS representation is not surprisingly the slowest compared to Bézier and triangles, however it is only around a third slower than the latter. Taking into account that generally the intersection test of ray-NURBS is far more complex than a ray-triangle test, this result is quite impressive. Additionally the image quality of both ray tracers is comparable. The quality of *dreamBSE* however, is inferior to the results rendered by using the Bézier representation[1] of *crianusurt*, which explains the slower frame rate in that case. Several enhancements were implemented which improve accuracy during the tracing and shading, which were not integrated in *dreamBSE* yet. In figure 7.5 the difference is visible, i.e. the object boundaries are a lot sharper in *crianusurt*, where the are relatively inaccurate in *dreamBSE*.

Note, that any free form surface representation uses lesser bounding boxes and only a fraction of surfaces compared to the number of triangles, thus

[1] *crinusurt* image quality is the same for Bézier and NURBS representation

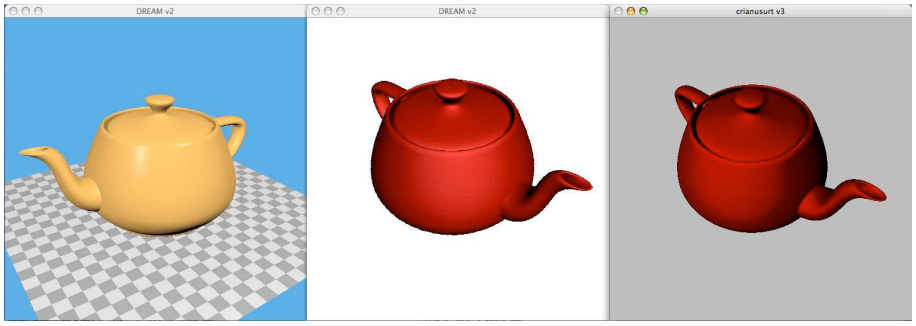

Figure 7.4: From left to right: *dream*, *dreamBSE* and *crianusurt*

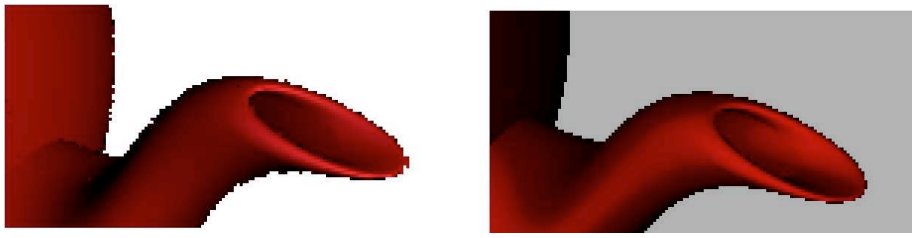

Figure 7.5: Magnification of the teapot. The object boundary is a lot more exact in *crianusurt* (right side)

memory consumption is lower for all of these. Additionally this has an positive effect on loading time and cache performance.

7.7 Scalability

As shown in the following *crianusurt* is scaling quite well especially for the NURBS representation. The application can be run with an arbitrary number of threads which enables support for any number of processors. For each thread an instance of the *NURBSIntersectionKernel* class will be created, but this is actually not a real limit to the maximum number of executable threads, as the kernel only stores pointers to the NURBS data. This is easily possible, as all associated data is read-only, so there is no need to employ locks or barriers. Every instance consumes 92 bytes of memory, assuming 4 byte long pointers. Regarding this, a few kilobytes are enough to support more processors than anyone will be able to buy within the next couple of years.

The following table 7.8 shows the achieved results with three different test scenes using the default parameters for conversion and rendering.

Scene	1 CPU	2 CPUs	Speedup factor
Teapot	0.576 fps	1.071 fps	1.85
NURBS head	0.215 fps	0.416 fps	1.93
Trimmed	1.100 fps	1.937 fps	1.76

The scalability for all three scenes is quite good and lies within reasonable range, as speedup factors above 1.95 can nearly not be achieved in real world applications due to the overhead involved when using multiple processor systems. However, the speedup factor always stays clearly above 1.5 which leads to the conclusion that there is no real bottleneck involved like the system bus speed for example or locks in the code that prevent individual threads from working. Also note the very good scaleability on the Prism in figure 7.3.

The difference in the speedup factors might be based on a lot of factors, but from the observed results it seems most likely, that more complex scenes scale better, which might be due to the fact of better cache performance, since more operations are performed on the same sub-surface by both processors.

7.8 Hyper Threading Efficiency

Although not a real parallel technique, Intels *Hyper Threading* technology enables the processor(s) to execute two threads simultaneously. To the operating system a hyper threading enabled processor appears as two physical processors. As there is not a second computation unit, it is really not comparable to multi processor nor multi core systems. Basically only the I/O of the processor is optimized, thus reducing processor stalls due to cache misses for example. However, this technique is a mixed bag regarding performance. Of course, single threaded applications can not take any advantage of it, but there are also cases when performance actually decreases when using hyper threading with multiple threads (some kinds of video encoding are known for that phenomenon). Further information on Intel's hyper threading technology can be found in [15]

The application presented in this work is optimized for cache coherency, so only a minor speed up factor is expected, as the results in tabular 7.8 approve. Note, that these results were achieved with the PC system mentioned above.

Scene	HT disabled	HT enabled	Speedup factor
Teapot	0.847 fps	0.89 fps	1.05
NURBS head	0.308 fps	0.327 fps	1.061
Trimmed	1.856 fps	1.870 fps	1.007

Obviously, the performance gain is quite moderate or for the last scene even nearly non existent. It seems that due to the high optimization of the application and especially the intersection test and bounding hierarchy box traversal, there is not much room for hyper threading left.

7.9 Newton Iteration Accuracy

As mentioned in section 5.2.3, there are three different termination criteria for the Newton iteration, where $|R(u,v)| < \epsilon$ was indicating a successful intersection. The ϵ value can be chosen by the user as an application parameter. The closer ϵ is to zero, the more accurate the actual intersection point will be calculated, however, more iteration steps will be necessary to achieve the higher precision. Additionally there is a limit to the maximum accuracy, because of the limited precision of floating point values. Tabular 7.9 displays the effect of different ϵ values on render times when using SIMD instructions, whereas tabular 7.9 shows the improved accuracy range by using double precision floating point values. Note that the latter requires FPU emulation mode, thus the frame rate is a lot slower compared to the former. Figure 7.8 shows the different results for the teapot achieved with the SIMD enabled variant.

ϵ	Frames/s	Quality of result
10^{-7}	0.48 fps	> 20% intersections missed
10^{-6}	0.93 fps	< 1% intersections missed
10^{-5}	0.98 fps	faultless
10^{-4}	1.03 fps	faultless
10^{-3}	1.15 fps	faultless
0.01	1.43 fps	some wrong intersections at borders
0.1	2.5 fps	predominant bounding box intersections
≥ 1	3.1 fps	bounding box intersections only

Figure 7.6: Results using SIMD

ϵ	Frames/s	Quality of result
10^{-16}	0.028 fps	> 50% intersections missed
10^{-15}	0.059 fps	< 5% intersections missed
10^{-14}	0.067 fps	faultless
$\vdots$	$\vdots$	$\vdots$
10^{-3}	0.085 fps	faultless
0.01	0.112 fps	some wrong intersections at borders
0.1	0.237 fps	predominant bounding box intersections
≥ 1	0.464 fps	bounding box intersections only

Figure 7.7: Results using FPU emulation with double precision

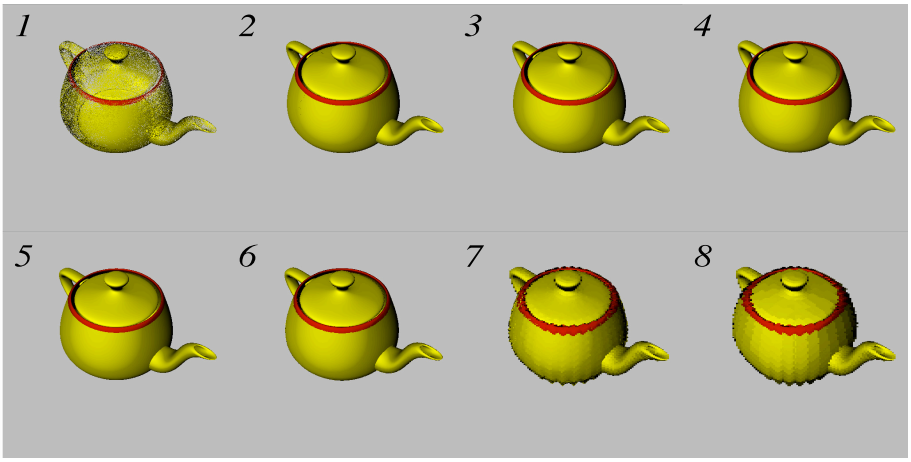

Figure 7.8: Different results for different ϵ values. The images correspond to the entries in tabular 7.9 from left to right and top to bottom. Basically from this point of view results 3 to 5 yield exactly the same result, but image 5 was rendered with 1.15 fps which is around 17% faster than render time for image 3 (0.98 fps)

Interestingly the results achieved with double precision are sill faultless with $\epsilon = 10^{-14}$ which is actually more than double the precision reached with floats, where 10^{-5} was the smallest value to yield correct results with.

By using double precision floating point values, it is possible to compute more exact intersections which, however, does not seem to be very worthwhile, as the image quality does not benefit from smaller values than 10^{-5} and as mentioned in section 5.6.1 numerical problems with complex surfaces prohibit the use of extremely small ϵ values anyway. Finally, as also mentioned before, double precision is available only with the FPU emulation, which is rather slow. Considering this, the double precision mode is basically only a minor extension to the range of surfaces that can be rendered by sacrificing performance.

Finally, the values shown here are also dependent on the scale of the scene, so that the *epsilon* value has to be chosen carefully for each scene. The teapot is defined in 3D space with control points values ranging roughly from -50 to 200. If another scene is defined in a hundred times larger scale the ϵ value has to be adjusted accordingly[2].

[2]The ϵ value can also be adjusted during runtime

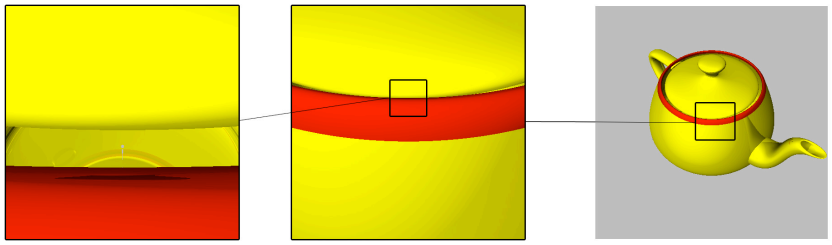

Figure 7.9: The same model rendered from three different views, all with the same parameters (i.e. $\epsilon = 10^{-6}$)

7.10 Close-up Accuracy

One of the big advantages of rendering free form surfaces directly, rather than performing a tessellation beforehand is, that there is basically no zoom-in limit, as curves will always stay curved. Usually even with an extreme high tessellation there is always a limit how close one can go, until the triangle structure will become visible.

Additionally the negative effects of high memory consumption due to a large number of triangles do not occur with free form surfaces, since there is no need to modify or subdivide them in any way. Merely only the parameter defining the intersection test accuracy, i.e. the Newton-ϵ is required to be a bit stronger, which means closer to zero, in order to get correct results at short distances.

Figure 7.9 is demonstrating some flawless, extreme close-up views. Note the reflections inside the teapot, which are still perfectly visible when viewing through the narrow gap, which borders are still consummately curved.

7.11 GPU results

As mentioned before, the GPU is unfortunately not powerful enough to perform the ray-NURBS intersection test yet, especially not with the performance desired. Nevertheless very simple intersection test do work, depending on the surface and size of the parametric domain, as well as the direction of the incident ray. As figure 7.10 shows, the example B-Spline surface is not rendered correctly, however, a large number of pixels is indeed correct. These result does not look very promising yet, but even more deflating is the rendering time, which is around 80 seconds (when using the GPU only). When the CPU and GPU are used in parallel only a very

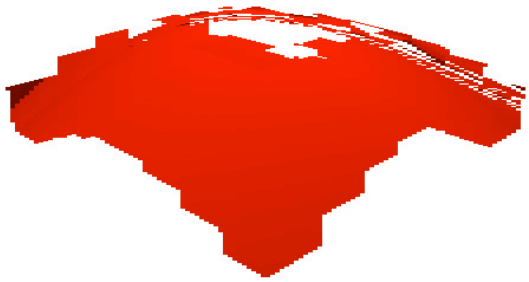

Figure 7.10: B-Spline Surface rendered by the GPU only

small number of *RayPackets* are computed by the GPU, since in the same time it takes the GPU to complete a tile (i.e. 16 *RayPackets*) assigned to it, the CPU has completed the rest of the image due to the extremely large difference between both approaches. For this specific surface comparing the GPU render time taken with the corresponding value achieved on the P4 test system, the CPU is roughly more than 190 times faster!

Additionally a lot of features the CPU variant offers could not be taken into account for the GPU, since the computational abilities as stated in 5.6.2 where already reached. Features could not be implemented on the GPU are

- Checking, if the iteration leaves the parametric domain (results in wrong, i.e. to large, surface boundaries)
- Rational surfaces are not considered
- Bézier representation is not considered
- Maximum of two Newton iteration steps (results in holes)

With future generations of graphic cards these limitations might be annihilated one after another.

Chapter 8

Conclusions & Future Work

In this work it was shown, that it is possible and worthwhile to use NURBS surfaces as the geometric primitive for interactive ray tracing on single commodity PCs. Compared to the common triangle ray tracer, frame rates are naturally not as high. Nevertheless there are several advantages when ray tracing NURBS directly, including lower memory consumption, especially for more sophisticated scenes, as well as artifact free rendering of extreme close up shots, without any additional effort. Furthermore, required preprocessing time is reduced and artifacts, due to errors during tessellation of trimmed surfaces, are avoided.

It was also shown, that it is advisable to render low complex NURBS surfaces with a reasonable number of control points and a basis function degree with less than seven. By exceeding these soft limits (depending on model scale and complexness) errors will occur in form of missed hits. This is due to the limited accuracy of single precision floating point values. Using double precision is not a real solution, but relaxes this limitation slightly, however performance dramatically decreases, since this is not compatible with SIMD execution yet. By using the baseline evaluation method any surface can be processed, regardless of control point grid resolution or degree, however frame rates are far from interactive in that case.

The additional usage of a GPU turned out not to be worthwhile yet. The performance achieved was slower by more than an order of magnitude compared to CPU computation on even only mediocre processors. The main problem is the complexity of the intersection test in contrast to the relatively limited computational abilities of recent GPUs. Problematic is the maximum number of temporary registers as well as the number of executions performed, which was effectively limiting the process to only one or two Newton iteration steps, which is definitively not sufficient. On the one hand it might be possible to implement the algorithm by using a mul-

tipass approach, but on the other hand, this will slow down performance even further making it rather useless for interactive rendering. However, performance as well as the abilities of the instruction sets for graphic cards are developing extremely fast, so it might be still interesting to keep an eye on the development.

It turned out, that one of the biggest issues actually is the limited accuracy of *floats*, when dealing with NURBS that have many control points or a high degree. Besides that, the frame rate goes down considerably with such kind of surfaces. Thus it would be worthwhile for future work to find a solution to both problems, especially because often real life data sets contain geometries that are not represented in an optimal manner. A possible and very promising approach would be to integrate another step into the preprocessing pipeline, where large and/or high degree surfaces will be split up into several less difficult patches, i.e. each with only a small number of control points and a basis function degree of preferably three or five. For these patches the accuracy will be sufficient and performance will remain good.

Additionally it might be interesting to re-integrate the work from [12] into *crianusurt* by adding support for triangles from the original ray tracer *dream*. Currently even flat surfaces like walls and floors have to be constructed by using NURBS, which is of course not optimal as for flat surfaces triangles are well suited. By combining triangle/Bézier/NURBS functionality it would be possible to employ some kind of level of detail hierarchy, where patches close to the viewer are rendered using the NURBS representation, in middle distance the Bézier representation is used and in the far distance only the control net is rendered using triangles for example.

Last but not least, a very promising development is the upcoming Cell-Processor, a specialized chip for vector operation designed by IBM and Toshiba. This chip will be first seen in Sonys new Playstation 3, but it will hopefully also be available by other means. A single Cell processor features one POWER based processor element (PPE) and eight *Synergistic Processor Elements* (SPE), each of which is capable of full SIMD operation on 128 bit registers. Basically this sums up to eight SIMD units for parallel use. Together these will reach a performance peak of 256 GFLOPS for single float precision operation. Considering that *crianusurt* is perfectly designed for this kind of execution, it would be very interesting and promising to port it to the CELL architecture[1], when available.

[1] Even if the CELL will not be available for PCs, it still might be possible to port *crianusurt* to the Playstation 3 itself, as a linux kit is expected to be realeased for it, just the same way it was for the predecessor

Appendix A

libNURBSIntersectionKernel Commands Overview

In the following the API of the *NURBSIntersectionKernel* is explained. Some functions are only applicable to either NURBS or Bézier, in that case these methods have a leading [N] or [B] before their header. Those that have not are useable for both free form surface types

A.1 Acquiring an Instance

static NURBSIntersectionKernel* Instance(int ID);

param ID: has to be between 0 and MAX_NUMBER_OF_THREADS (default:8)
return: An instance of the kernel that can be used by **one** thread

A.2 Kernel Configuration

void cfgIndex(const int index);

param index: the unique index of the surface the kernel will be configured for

void cfgActiveComponents(bool4* const active);

param active: Specifies active components. Computation will be executed for every element that is true. If all are false no computation will be performed at all

[N] void cfgBasicDataVector(float4* const basicData);

param basicData: sets the basic data for the NURBS patch.
Format: (Number of control points in U direction, number of control points in V direction, basis function degree in U direction, basis function degree in V direction)

void cfgBoxMinMaxUV(float4* const minU, float4* const maxU, float4* const minV, float4* const maxV);

param minU: minimum u parameter value for this current intersection test. This value is provided by the last bounding box that was hit, i.e. local valid parametric domain
Format: (minU, minU, minU, minU)
param others analog to *minU*

[N] void cfgSurfaceMinMaxUV(float4* const minMaxUV);

param minMaxUV: minimum and maximum u/v parametric values for the surface being tests, i.e. global valid parametric domain
Format: (minimum U, maximum U, minimum V, maximum V)

void cfgControlPoints(float4* const controlPoints);

pram controlPoints: the control points for this surface
Format: $(x_1, y_1, z_1, w_1), \ldots, (x_n, y_n, z_n, w_n)$

[N] void cfgBasisFunctions(float4* const basisFunctions);

param basisFunctions: each basis function is represented by a number of float4 variables according the value of PIXELPERFUNCTION which is set in NURBSIntersectionKernel.h. First all basis function in u direction are stored after which the function in v direction follow. For each parametric direction all intervals are stored in an ascending order, where each interval stores a number of basic functions equalling the order of the basis function, also in ascending order according to the i loop value of the sum. Refer to 3.5.1 or figure 3.15 for a more detailed discussion.

[N] void cfgKnotVectors(float4* const knotVectors);

param knotVectors: stores both knot vectors. Knot vector values in u direction are stored in the first components, respectively v knot vector values are stored in the second components of each float4 variable. Third and fourth components are not being used. Refer to 3.5.1

[N] void cfgBFOfsset(float4* const offset);

param offset: the number of basis functions in u parametric direction, i.e. the offset to the first basis function in v direction

cfgTrimmingCurves(vector<BezierCurve*>* const curves);

param curves: all trimming curve segments for this surfaces stored in a STL vector in no particular order

void cfgRays(PrimaryRays const* rays);

param rays: to be set if the intersection test shall be computed with a packet of primary rays, i.e. rays having an identical origin

void cfgRays(ShadowRays const* rays);

param rays: to be set, if a shadow test shall be performed

void cfgRays(SecondaryRays const* rays);

param rays: to be set if a intersection with a packet of secondary rays shall be computed, i.e. rays with different origins

A.3 Global Kernel Configuration

void cfgMaxNewtonStep(const int maxNS);

param maxNS: the maximum number of newton steps that will be performed, before the intersection test will be interrupted. Default:6

[N] void cfgPixelPerFunction(const int ppf);

param ppf: the number of float4 variables that will be used to store one basis function. Maximum degree of a function: ppf*4-2. Default:2

void cfgNewtonEpsilon(const float4& eps);

param eps: the epsilon value that defines the maximum distance to the real root that will still be regarded as an intersection. Values closer to 0 will yield more exact results but more likely missed hits. Default:0.001

A.4 Evaluations and Intersections

void intersect(HitPacket& hits, bool doTrimming, bool doBezier = false);

The previously configured surface data will be used to perform an intersection test with a primary ray packet
param hits: stores the results of the intersection test
param doTrimming: if true, trimming test will be performed
param doBezier: use high performance Bézier intersection test

void intersect(bool4& mask, float4& scale, bool doTrimming);

The previously configured surface data will be used to perform a shadow test with a shadow ray packet
param mask: Stores the result of the test with false indicating a shadowed pixel. If components were set to false before calling this method they will always stay false, i.e. the shadow test will not be performed on these components
param scale: Currently not used in this implementation. Is planned to be used for computing transparencies *param doTrimming*: if true, trimming test will be performed

void intersectSecondary(HitPacket& hit, bool doTrimming);

Performs an intersection test with secondary rays (reflected rays for example) Note: Secondary reys for Bézier surfaces not supported yet
param hit: stores the results of the intersection test
param doTrimming: if true, trimming test will be performed

[N] void evaluate(const float4& u, const float4& v, const bool derivativeU, const bool derivativeV, float4& resX, float4& resY, float4& resZ) const;

Evaluates the NURBS surface at given u and v parameter
param u: four u parameter values
param v: four v parameter values
derivativeU: if true, partial derivative u will be computed
derivativeV: if true, partial derivative v will be computed
resX: stores the four x values of the resulting points
resY: stores the four y values of the resulting points
resZ: stores the four z values of the resulting points

[B] void evaluateBS(float4 &u, float4 &v, float4 &resX, float4 &resY, float4 &resZ) const;

Evaluates the Bézier surface at given u and v parameter
param u: four u parameter values
param v: four v parameter values
resX: stores the four x values of the resulting points
resY: stores the four y values of the resulting points
resZ: stores the four z values of the resulting points

[B] void evaluateBSDerivativeU(const float4 &u,const float4 &v, float4 &resX, float4 &resY, float4 &resZ) const;

Evaluates the partial derivative u of the Bézier surface at given u and v parameter
param u: four u parameter values
param v: four v parameter values
resX: stores the four x values of the resulting points
resY: stores the four y values of the resulting points
resZ: stores the four z values of the resulting points

[B] void evaluateBSDerivativeV(const float4 &u,const float4 &v, float4 &resX, float4 &resY, float4 &resZ) const;

Evaluates the partial derivative v of the Bézier surface at given u and v parameter
param u: four u parameter values
param v: four v parameter values
resX: stores the four x values of the resulting points
resY: stores the four y values of the resulting points
resZ: stores the four z values of the resulting points

void evaluateNormal(const float4& u, const float4& v, VectorPacket& normals, bool doBezier = false) const;

evaluates the normal at the given parameter values u and v
param u: four u parameter values
param v: four v parameter values
normals: vector packet soring the four resulting normals
doBezier: if true, use high performance Bezier mode

trimmed(const float4 &u, const float4 &v, bool4 &mask) const;

performes a trimming test on current surface on given u and v parameter values

param u: four u parameter values
param v: four v parameter values
mask: every value that is false denotes a trimmed point

Bibliography

[1] Oliver Abert. Interaktives raytracing von getrimmten bikubischen bézier flächen unter verwendung von simd instruktionen, December 2004. 1, 2, 36, 65, 74, 97

[2] Alias maya,
http://www.alias.com. 28

[3] Robert Bärz. Paralleslisierung von ray tracing berechnungen in einem heterogenen netzwerk, October 2004. 24

[4] Phong Bui-Tuong. Illumination for computer genereated pictures. In *Communications of the ACM*, volume 18(6), pages 311–317, June 1975. 22

[5] Philipp Slusallek Carsten Benthin, Ingo Wald. Interactive ray tracing of free-form surfaces. In *ACM AFRIGRAPH*, pages 99–106, 2004. 1

[6] de casteljau's algorithm,
http://en.wikipedia.org/wiki/de_casteljau's_algorithm, September 2005. 78

[7] M.G. Cox. The numerical evaluation of b-splines. In *Journal of Institutional Mathematical Application*, volume 10, pages 134–149, 1972. 7

[8] C. de Boor. on calculating with b-splines. In *journal of Approximation Theory 6*, pages 50–62, 1972. 7

[9] Microsoft directx,
http://www.microsoft.com/windows/directx/default.aspx. 17

[10] T.F. Riesenfeld E. Cohen, T. Lyche. Discrete b-splines and subdivision techniques in computer aided geometric design and computer graphics. In *Computer Grahpics and Image Processing*, volume 14, pages 87–111, 1980. 13

[11] James Clark Edwin Catmull. Recursevely generated b-spline surfaces on arbitrary topological meshes. In *Computer-Aided Design*, volume 10(6), pages 350–355, September 1978. 65

[12] Markus Geimer. Interactive ray tracing, August 2005. 30, 55, 63, 64, 108

[13] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984. 28

[14] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988. 28

[15] Hyper threading technology, http://www.intel.com/technology/hyperthread. 101

[16] Intel corp. ia-32 architecture software developers manual. 2004. 2

[17] Frederik W. Jansen. Data structures for ray tracing. *Data structures for raster graphics*, pages 57–73, 1986. 28

[18] John Salmon Jeffrey Goldsmith. Automatic creation of object hierarchies for ray tracing. In *IEEE Computer Graphics and Applications*, volume 7, pages 12–20, May 1987. 30

[19] Henrik Wann Jensen. *Realistic image Synthesis Using Photon Mapping*. 2001. 30

[20] Henrik Wann Jensen. State of the art in monte carlo ray tracing for realistic image synthesis. In *SIGGRAPH 2001 Course Notes CD-ROM*. ACM SIGGRAPH, 2001. 22

[21] James T. Kajiya. Ray tracing parametric surfaces. In *Computer graphics*, volume 16(3), pages 245–254. ACM SIGGRAPH, July 1982. 64

[22] Andreas Langs. Photon mapping unter verwendung eines simd ray tracers, June 2005. 30

[23] Charles T. Loop. Smooth subdivision surfaces based on triangles. In *Master's thesis, University of Utah*, 1987. 65

[24] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark. J Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics (TOG)*, 22(3):896–907, 2003. 17

[25] Oliver Abert Markus Geimer. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *WSCG'2005*, 2005. 1

[26] William Martin, Elaine Cohen, Russell Fish, and Peter Shirley. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools: JGT*, 5(1):27–52, 2000. 66, 67

[27] Richard Bartels Michael Sweeney. Ray tracing free-form b-spline surfaces. In *IEEE Computer Graphics and Applications*, volume 6(2), pages 41–49, February 1986. 66

[28] Motorola, inc. altivec technology programming interface manual, 1999. 2

[29] Newton's iteration, http://mathworld.wolfram.com/newtonsiteration.html. 65

[30] Opengl shading language, http://www.opengl.org/documentation/oglsl.html. 18

[31] Openrt, http://www.openrt.de. 27

[32] Changway Wang Peter Shirley. Direct lighting calculation by monte carlo integration. In *Proceedings of the second Eurographics Workshop on Rendering*, june 1991. 22

[33] H. Prautzsch. A short proof of the oslo algorithm. In *Computer Aided Geometry Design*, volume 1, pages 95–96, 1984. 13

[34] David F. Rogers. *An Introduction to NURBS with Historical Perspective*. Morgan Kaufmann Publishers, 2000. 5, 82, 94

[35] Peter Shirley. *Realistic Ray Tracing*. 21

[36] Ruei-Chuan Chang Shyue-Wu Wang, Zeng-Chung Shih. An efficient and stable ray tracing algorithm for parametric surfaces. *Journal of Information Science and Engeneering*, 18:541–561, 2001. 66

[37] ANSI/IEEE standard 754-1985. Ieee standard for binary floating-point arithmetic, 1985. 82

[38] R. Séroul. *programming for Mathematicians*, chapter 10.6 Evaluation of Polynomials: Horner's Method, pages 216–262. Springer Verlag, 2000. 42

[39] Masanori Kakimoto Tomoyuki Nishita, Thomas W. Sederberg. Ray tracing trimmed rational surface patches. In *Computer Graphics*, volume 24(4), pages 337–345, August 1990. 75, 77

[40] H. Prautzsch W.Boehm. The insertion algorithm. In *CAD*, volume 17, pages 58–59, 1985. 13