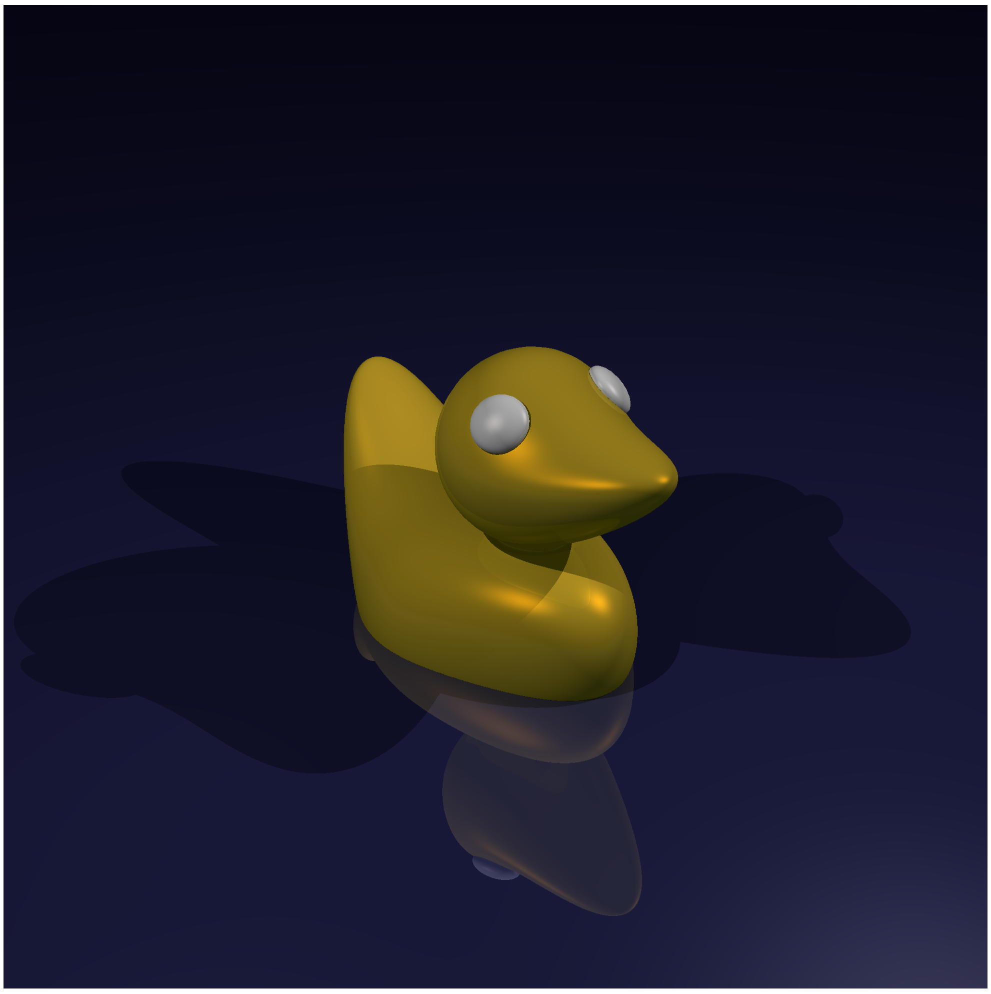
Ray Tracing NURBS Surfaces using CUDA

Master's Thesis (Version of 2nd February 2010)



Erik Valkering

Ray Tracing NURBS Surfaces using CUDA

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Erik Valkering
born in Limmen, the Netherlands

Computer Graphics and CAD/CAM group
Department of Mediamatics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: The ducky scene ray traced by CNRTS with shadows and reflections.

Ray Tracing NURBS Surfaces using CUDA

Author:	Erik Valkering
Student id:	1328913
Email:	erikvalkering@gmail.com

Abstract

This thesis presents CNRTS, a CUDA-based system capable of ray tracing NURBS-surfaces, directly. By coarsely subdividing the NURBS-surfaces in a preprocessing step, a more tight bounding volume hierarchy is generated which can be traversed first, thus heavily reducing the number of ray-patch intersection tests. Additionally, the smaller leaf-nodes provide better initial seed-points, which allows the root-finder to converge more quickly. For scene-traversal, two approaches have been investigated, a packet-based traversal scheme, employing a shared stack implemented using the GPU's fast on-chip memory. The second approach uses an optimized single-ray traversal scheme, in which the scene-traversal and root-finding are separated from each other, to help maximize the utilization during root-finding. The single-ray approach turns out to be up to $3\times$ faster than the packet-based approach.

Furthermore, a hybrid approach has been used in tracing primary rays, by employing the rasterizer. Besides that only one ray-patch intersection test is required per ray, the initial seed-points provided by the rasterization are very close. Using the hybrid approach, the performance will generally be increased. However, an artifact-free rendering is not always guaranteed, due to tessellation.

Preface

High-quality visualization of Non-Uniform Rational B-Spline (NURBS) surfaces, mostly found in industrial designs (such as cars, airplanes, ships, etc.), is usually performed through rasterization, after a costly preprocessing phase, in which the NURBS-model is converted to a triangular representation. In order to represent highly curved areas, the triangulation must be very fine. As a consequence, a lot of triangles will be generated in areas with high curvature. This results in a long preprocessing time, in which lots of triangles are generated.

In this thesis, a solution will be presented, which avoids the expensive preprocessing step, and works directly with the NURBS-data. By employing the ray tracing algorithm, the resulting visualization will always appear smooth, regardless of the viewing distance. Because the original NURBS-data is used for visualization, the memory requirements are very low. Furthermore, the ray tracing system is fully implemented using CUDA to exploit the processing power of recent GPUs.

I would like to thank a number of people who have been a great help and support in realizing this thesis. First of all, I would like to thank my supervisor Erik Jansen for the many interesting discussions which kept me motivated and for providing really helpful advice on how to improve this report.

Next, I would like to thank Jorik Blaas for all those clever ways on how to debug my software. I would also like to thank Menno Kuyper for always be willing to help me to test my software on his computer, despite of the many system crashes I have caused.

And last but not least, I would like to thank my parents, Bert and Ellen, for their continuous support, in every possible way, and giving me the opportunity to study Computer Science.

Contents

Contents **v**

List of Figures **vii**

1 Introduction **1**
1.1 Rasterization 1
1.2 Ray Tracing 2
1.3 General-Purpose GPU computing 3
1.4 Structure of this thesis 6

I Background **7**

2 NURBS Basics **9**
2.1 Definition 9
2.2 Cox-de Boor Recurrence 10
2.3 Knot-vector 11
2.4 Convex Hull 12
2.5 Knot-insertion 12
2.6 Surfaces 14
2.7 Derivative 14

3 NURBS Ray Tracing **17**
3.1 Scene Traversal 17
3.2 Packet-Based Ray Tracing 19
3.3 Ray-Patch Intersection 19
3.4 Adaptive subdivision 22
3.5 Evaluation Schemes 26
3.6 Overview 32
3.7 Discussion 33

4 NURBS Ray Tracing on the GPU **35**
4.1 GPU Ray Tracing 35
4.2 Hybrid Ray Tracing 37
4.3 Discussion 43

5 CUDA 45
5.1 Massive Multi-threading Architecture 45
5.2 Memory Hierarchy 46
5.3 Launching a kernel 48
5.4 Debugging 48

II Development 51

6 System overview 53
6.1 Primary-ray Accelerator subsystem 53
6.2 Ray Tracer subsystem 56
6.3 Summary 61

7 Preprocessing 63
7.1 Subdivision 63
7.2 Root-Finder Data 66

8 Kernel details 69
8.1 Inter-kernel Communication 69
8.2 Root-finding 72
8.3 Surface Evaluation 72
8.4 Primary-ray Accelerator subsystem 74
8.5 Ray Tracer subsystem 77
8.6 Kernel-launch configurations 86

III Results 89

9 Results 91
9.1 Experimental Setup 91
9.2 Image Quality 93
9.3 Performance 97
9.4 Analysis 104
9.5 Discussion 109

10 Conclusions and Future Work 111
10.1 Future Work 111

Bibliography 113

List of Figures

1.1 NURBS-model 1
1.2 Tessellation of a NURBS model. 2
1.3 A model of the VW Polo from different viewpoints. The car remains perfectly curved, even from the shortest viewing distance, thanks to the direct ray tracing of NURBS. . . . 2
1.4 Ray Tracing 3
1.5 Performance statistics for the CPU and GPU. 4
1.6 Simplified processing model. 5

2.1 Fine-tuning a NURBS-curve (a): By specifying different weights (b) and through the knot-vector (c,d). 10
2.2 Example B-Spline basis-functions: constant (a), linear (b), quadratic (c) and cubic (d). . 11
2.3 Example of polynomial segments of a cubic NURBS-curve. 12
2.4 Strong convex hull property: for $u \in [u_i, u_{i+1})$, $\vec{C}(u)$ is in the triangle $\vec{P}_{i-2}\vec{P}_{i-1}\vec{P}_i$ (a) and in the quadrilateral $\vec{P}_{i-3}\vec{P}_{i-2}\vec{P}_{i-1}\vec{P}_i$ (b). 13
2.5 Knot-insertion: knot inserted at $t = 0.5$. 13
2.6 NURBS-surface example: Utah Teapot consisting of 32 bi-cubic patches. The patches are visualized using a red and green color mapping for the parametric u and v direction, respectively. 14

3.1 Six levels in a Bounding Volume Hierarchy. 18
3.2 Newton's Iteration: each subsequent point is closer to the root. 20
3.3 Plane representation for a ray for which $|\vec{d}_x| > |\vec{d}_y|$ and $|\vec{d}_x| > |\vec{d}_z|$ holds, results in a plane rotated about the z-axis and a plane perpendicular to it (a) and for a ray for which it does not hold, the resulting plane will be rotated about the x-axis (b). The cyan line represents the ray and the yellow plane is the first plane $\vec{p}_1 = (\vec{n}_1, d_1)$. 21
3.4 Example of a NURBS-curve with exemplary associated bounding volumes. 23
3.5 Flatness criterion based on the normals of a sub-patch. 24
3.6 Splitting a sub-patch into two sub-patches obtaining two control-point grids. 25
3.7 Procedure to fix a crack in the tessellation. 25
3.8 Evaluating the NURBS basis-functions. 31

4.1 Multiple stages in the ray tracer of [PBMH05]. 36
4.2 Stackless BVH traversal encoding example. 37
4.3 Hybrid Ray Tracing phases. 38

4.4 Overview of the algorithm stages and their relation to stages in the extended graphics pipeline. 39
4.5 uv-texturing using midpoint of parameter range (uniform subdivision: 2×2). Figure 4.5a shows the corresponding uv-texture. Figure 4.5b to Figure 4.5d show the result for up to two, three, and four iterations, respectively. 40
4.6 uv-texturing using control point mapping (uniform subdivision: 2×2). Figure 4.6a shows the corresponding uv-texture. Figure 4.6b to Figure 4.6d show the result for up to two, three, and four iterations, respectively. 40
4.7 Figure 4.7a, Figure 4.7b view-independent uv-texturing vs. Figure 4.7a, Figure 4.7c view-dependent uv-texturing (max. iterations: 10, subdivision: uniform, 4×4). Figure 4.7d is the difference between the initial values of both methods. 40
4.8 Artifacts which are handled correctly. 42
4.9 Artifacts which cannot be handled correctly. 43

5.1 Thread blocks and their mapping to the multiprocessors. 46
5.2 Memory Hierarchy. 47

6.1 System overview of the rendering core of the CNRTS 53
6.2 Schematic overview of the rasterization phase of the Primary-ray Accelerator. The output provides some cues for finding the exact intersection-point: the left rasterized image contains the patch-ids and the right rasterized image contains the uv-values encoded in red for u-parameter range and green for v-parameter range. 54
6.3 The ray's direction is derived from the viewing parameters. In here, a denotes the field-of-view. By setting the image-plane's distance d from the camera equal to 1, the position in the image plane is easily converted to world-coordinates. 55
6.4 Artifacts appear when using the Primary-ray Accelerator solely. In Figure 6.4a, background-pixels appear instead of the surface. By forwarding all background-pixels to the *Ray Tracer*, this artifact will disappear. Figure 6.4b shows some gaps, appearing in-between two surfaces. Figure 6.4c shows an artifact which cannot be detected using the algorithm, the only solution is to tessellate more fine. 57
6.5 The ray tracing algorithm separated into different CUDA-kernels. The CPU part coordinates the algorithm by launching the kernels successively. The inter-kernel data propagation through global-memory is visualized by dashed lines, whereas the arrows indicate the flow/dependencies. 58
6.6 Separation of image-space into *tiles*: instead of applying the ray tracing algorithm onto the entire image, it is now applied successively to each tile. The tiles are visualized using different shades of grey, the pixels are separated by dashed lines. 59
6.7 Implementing recursion for CUDA using a stack. 60
6.8 Lifetime of a *primary-ray* processed by the *Primary-ray Accelerator* and the *Ray Tracer*. A primary-ray is spawn by rasterizing the tesselation; a fragment is further processed by the primary-ray accelerator's root-finder; a hit may skip the ray tracer's traversal and root-finder and immediately continue by casting a shadow ray; misses and background pixels are traced conventionally by the ray tracer;whereafter they are processed further by casting a shadow ray; shading is applied for non-blocked shadow-rays; and finally reflection and refraction rays are spawn and traced recursively. 62

7.1 Overview of subdivision results 63
7.2 The NURBS-surface is converted into smaller Bézier-patches. 64
7.3 Bounding boxes of a BVH. 65

8.1 Memory layout of a stack inside global-memory. This stack is for 4 threads and has a maximum depth of 3. The exemplary "virtual" stack of thread 2 shows the mapping of threads and stack-levels to global-memory addresses. 70
8.2 Arrangement of data for a cache inside shared-memory. This example uses an array of 3 elements for each thread, and a block size of 4 threads. 73
8.3 Mapping of pixels to threads. This example uses an image-resolution of 16×16, a tile-size of 4×4, and a block-size of 2×2. The tiles are visualized using a shade of grey, the blocks are separated by black lines, and the pixels are separated by dashed lines. . . . 87

9.1 Test scenes used. 95
9.2 teapot-scene: accelerator (left), full hybrid (middle), standard ray traced (right) 95
9.3 head-scene 96
9.4 This image shows a close-up of the ducky-scene: hybrid (left), and normally ray traced (right). It seems that the hybrid method can sometimes introduce artifacts, when zooming in too close. 96
9.5 Killeroo-closeup-scene rendering with a increasing *miniterations*-value. The images correspond to the *miniterations*-values in Table 9.4. 97
9.6 Average speedup of primary rays w.r.t. reference implementation (packet-based, no caching, no acceleration). 99
9.7 Average speedup of shadow rays (level 0) w.r.t. reference implementation (packet-based, no caching). 100
9.8 Average speedup of reflection rays (level 1) w.r.t. reference implementation (packet-based, no caching). 100
9.9 Average speedup of refraction rays (level 1) w.r.t. reference implementation (packet-based, no caching). 101
9.10 Durations of the variants using the 8800M test-system, as a function of the number of patches. The red line shows the curve corresponding to a logarithmic function fitted to the data. 102
9.11 Durations of the variants using the GTX 295 test-system, as a function of the number of patches. The red line shows the curve corresponding to a logarithmic function fitted to the data. 102
9.12 Number of BVH-nodes visited. 104
9.13 Amount of time spent idle of total durations for each thread. 107
9.14 Frequencies of the average number of rays participating in the root-finding. 108

Chapter 1

Introduction

In the field of CAD/CAM[1], designs are often modeled using curved, smooth-looking free-form surfaces. In modeling industrial designs such as cars, airplanes, ships, etc. the surface-type of choice is the Non-Uniform Rational B-Spline (NURBS) surface (Figure 1.1). This type of surface is favored, because it provides local and intuitive control, has a very compact representation and is very well suited for a large number of problems.

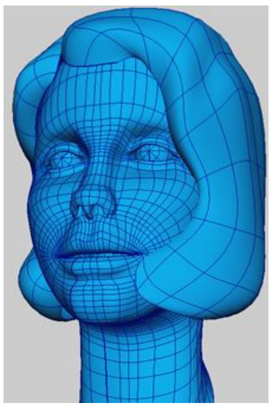


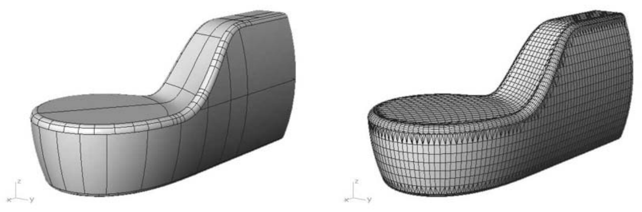
Figure 1.1: NURBS-model

In current CAD/CAM software, when visualizing a model, the curved surface is first tessellated into a polygonal mesh in order to obtain a tight approximation of the surface (Figure 1.2). Such a tessellation usually results in many polygons, taking up lots of memory. After the model has been tessellated, the original continuity is lost, zooming in on the tessellated model will reveal this. Besides memory requirements, the tessellation preprocessing step also takes some time. For complex models found in the automobile industry for example, this can take up to a whole day making it not very suitable for rapid prototyping [AGM06].

1.1 Rasterization

Usually, these polygon meshes are rendered using rasterization. Basically, this method first projects the objects to a two-dimensional image plane, after which the algorithm determines for each pro-

[1]Computer Aided Design/Computer Aided Manufacturing



(a) Original NURBS-model (b) Tessellated NURBS-model

Figure 1.2: Tessellation of a NURBS model.

jected object, which pixels of the image plane will be occupied by it. By taking the depth of the objects into account, the pixels will be given the color of the closest object for that pixel. While this method enables easy visualization of the model, it does not handle physically-correct lighting effects, such as shadows, reflections, refractions, etc. which are often needed in industrial prototyping. In order to include these effects, several methods have been devised to approximate them. Reflections, for example, can be obtained by first generating a texture onto which the environment surrounding the object is projected (usually using a spherical or cubical texture). During rendering, reflection is applied by following the reflection-vector to obtain the reflection-texel from the environment map. However, since only the environment is stored in the texture, self-reflection cannot be simulated using this method. Furthermore, since the rendering time of the rasterization method is linear in the number of polygons, eventually the models become too complex, and cannot be visualized interactively anymore. Although rasterization, when implemented on graphics hardware, quickly generates images having reasonable quality, the effects necessary to obtain a realistic visualization are non-trivial and methods to simulate them usually produce very rough approximations.

1.2 Ray Tracing

An alternative to rasterization is visualization using the *ray tracing* algorithm [WH80]. The relatively simple algorithm, compared to rasterization, implicitly embeds all these effects in a physically-correct way. Although this algorithm is computationally much more expensive, it is logarithmic in the number of objects. Therefore, increasing the tessellation-complexity will have less impact on the performance when compared to rasterization. Moreover, by using ray tracing, the resulting pixel values can even be computed directly without the intermediate tessellation step. In this way, the original model can be used instead, reducing memory requirements, and increasing the detail (Figure 1.3).

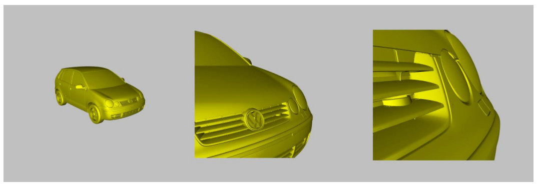


Figure 1.3: A model of the VW Polo from different viewpoints. The car remains perfectly curved, even from the shortest viewing distance, thanks to the direct ray tracing of NURBS.

The algorithm is actually quite simple. Instead of projecting each geometrical object to the two-

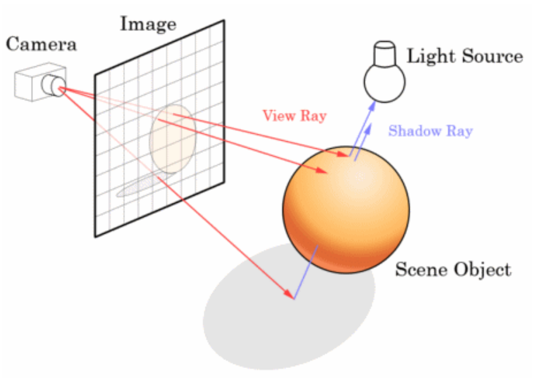


Figure 1.4: Ray Tracing

dimensional image plane, this algorithm works the other way around. For each pixel in the image plane, the color is determined by “shooting” a primary ray originating from the camera (the eyes of the viewer) passing through that pixel and finding the closest intersection with an object from which the color is obtained. These primary rays “look” into the scene and try to find an object it sees. If no object crosses the path of the ray, the ray will leave the scene non-intersected and the color of the corresponding pixel will be set to a default background color. This primary intersection stage will result in an image equivalent to the rasterization algorithm, and is known as Ray Casting.

If the ray does intersect an object, the lighting of the primary intersection is computed by spawning several *secondary rays* originating from the intersection point of the corresponding object: for each light-source in the scene a shadow ray pointing to this light-source; one reflection ray pointing to the mirror-reflection direction from a shiny surface; and one refraction ray, with a direction into the surface of the object (in case of a (semi)transparent surface). The shadow rays are used to determine if the primary intersection is illuminated by the light-sources. If any object happens to block the path of a shadow ray, the light of the corresponding light-source will not reach the primary intersection and will not contribute to the illumination of it. The other secondary rays are further traced recursively to compute secondary intersections which may contribute in the final lighting for the primary intersection. These secondary intersections will again spawn rays and the process is continued recursively until a certain traversal-depth has been reached or the color converges. This process can be seen in Figure 1.4.

Although the generated images by the Ray Tracing algorithm are much more realistic than the images generated by the Rasterization algorithm, the cost for generating them however, is very high. Algorithm 1 shows the basic ray tracing algorithm, which is executed for each pixel. As can be drawn from the algorithm, the complexity of a basic ray tracer depends on the total number of pixels N in the image plane, the number of geometrical objects M in the scene and the traversal-depth D for each ray and can be described as $O(NM^D)$. Later will be shown, how this complexity can be reduced to $O(N\log(M)^D)$, by introducing an acceleration structure which reduces the number of intersection computations to a relatively small subset of the objects in the scene.

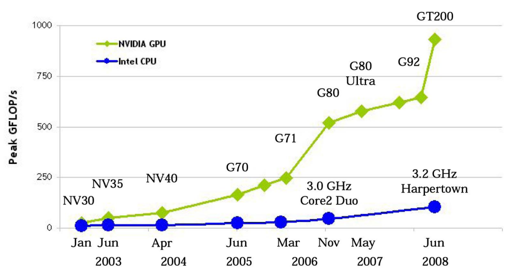
1.3 General-Purpose GPU computing

In the last few years the potential numerical performance of graphics processing units (GPUs) has grown dramatically. Modern GPUs usually offer multiple processor cores running in parallel. Furthermore the available memory bandwidth is usually also very high, typically 60 GB/s or more. This high memory bandwidth is needed to keep the cores busy. Figure 1.5 gives the potential numeri-

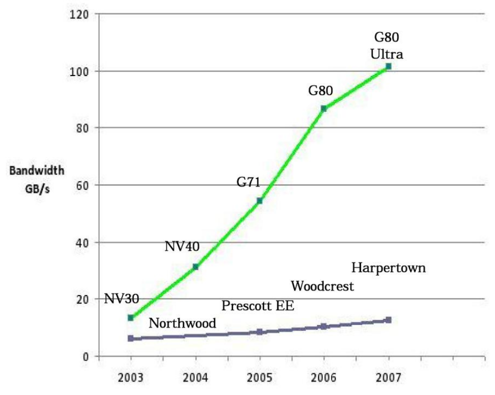
Algorithm 1 Ray Tracing algorithm

```
Require: Ray $R$, Traversal-depth $d$
Ensure: Color $C$ of ray
 1: $C \leftarrow$ Default background color
 2: if $R$ intersects an object in the scene then
 3:     $O \leftarrow$ Nearest object intersected by $R$
 4:     for all light-source $L$ in the scene do
 5:         $S \leftarrow$ Shadow-ray originating from $L$and pointing to $O$
 6:         if $S$ intersects no object then
 7:             $C \leftarrow C +$ Apply illumination model$(L, O)$
 8:         end if
 9:     end for
10:     if $d \leq$ maximum traversal-depth $D$ then
11:         $\{R_i\} \leftarrow$ Secondary reflection and refraction rays
12:         $C \leftarrow C + \sum_i$ Ray Trace$(R_i, d+1)$
13:     end if
14: end if
15: return $C$
```

cal performance of both the latest GPU and CPU families. The potential numerical performance is expressed in GFLOPS[2]. The main reason of the fast evolution of modern GPUs is that they are specially designed for compute intensive and highly parallel computation.



(a) Floating-Point Operations per Second



(b) Memory Bandwidth

GT200	**= GeForce GTX 280**	G70	= GeForce 7800 GTX
G92	**= GeForce 9800 GTX**	NV40	= GeForce 6800 Ultra
G80	**= GeForce 8800 GTX**	NV35	= GeForce FX 5950 Ultra
G71	= GeForce 7900 GTX	NV30	= GeForce FX 5800

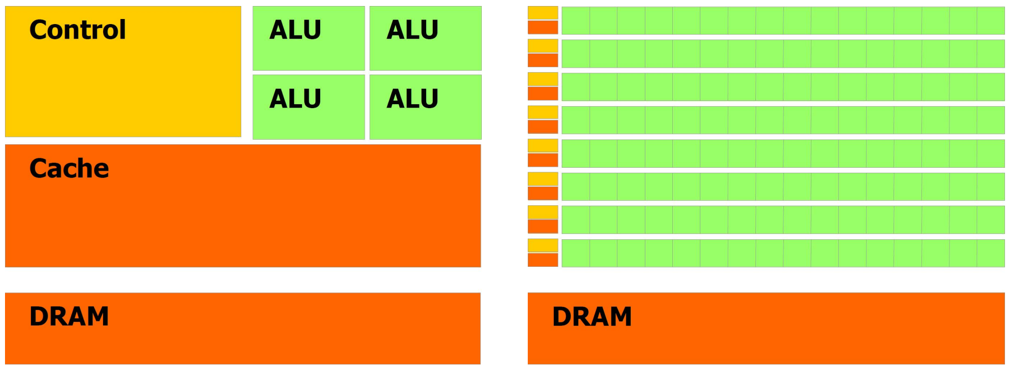
Figure 1.5: Performance statistics for the CPU and GPU.

A typical CPU has a very large cache memory and a very deep pipeline. The cache memory is required to hide long memory latencies; the pipeline is required to hide complicated arithmetic

[2]Giga FLoating-point Operations Per Second

operations. So the latency for most operations on a CPU is very short. But usually a CPU only offers a couple of fixed and floating point arithmetic and logical units (ALUs). So the number of operations that can be done in parallel is very limited. In other words a CPU is not really suitable for highly parallel computation.

Modern GPUs do not have very large data caches or very deep pipelines. Most of the transistors on the chip are devoted for data processing. The simplified processing model of a typical CPU and GPU is given in Figure 1.6.



(a) CPU (b) GPU

Figure 1.6: Simplified processing model.

A GPU is very well suitable for applications/algorithms that require the same set of operations for each data element. Because the same set of operations is performed for each element there is a lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

But using this large amount of potential numerical performance can be tricky. A typical GPU can only be programmed in special development environments using special programming languages (such as Microsoft's HLSL, OpenGL GLSL, NVIDIA Cg) and complicated (texture) memory fiddling. Furthermore, no recursion is supported, due to the lack of a stack. In other words, it involves a high learning curve. However, more recently several GPGPU[3]-languages have been devised in order to help the programmer to focus on the parallel problem, instead of all the stages in the graphics pipeline. These languages allow to program in a familiar language such as c or c++. The tools then generate code that run as shaders using the graphics pipeline (BrookGPU, Sh, RapidMind, CUDA). The latter of these languages, CUDA, is used throughout this thesis, and will be discussed in more detail in Chapter 5.

Ray tracing is an inherently parallel and highly compute intensive operation. Therefore, the GPU should be exploited as much in order to speed up the visualization process. However, due to the divergent nature of the algorithm, it's hard to keep all processor cores busy at all times. Some rays may require more intersection tests, while others are already finished. By employing a ray tracing algorithm which maximizes the number of active cores in addition with a fast intersection test, these problems can be overcome. In addition, a hybrid rasterization/ray tracing approach is taken in accelerating the ray tracing of primary rays.

[3]General-Purpose GPU

1.4 Structure of this thesis

This thesis will continue with part I, which focuses on the existing literature. In Chapter 2, the mathematical basics for NURBS and their algorithms will be given. Then, in Chapter 3, existing techniques are described for ray tracing NURBS-surfaces. Next, in Chapter 4, this discussion will extend to GPU-implementations. Finally, Chapter 5 will discuss the CUDA architecture, which is used in this thesis.

Part II of this thesis will present the CNRTS, the ray tracing system developed for this thesis. Chapter 6 will give a brief overview of the system's architecture. Next, Chapter 7 will describe the preprocessing steps required prior to visualization. And finally, Chapter 8 will give a more in-depth description of the system.

The final part of the thesis, part III, will discuss the results achieved with the system described in part II. Chapter 9 will discuss the methods used to evaluate and analyze the system, and Chapter 10 will conclude this thesis, by summing up the most important observations, and giving some hints on how the system can be further improved.

Part I

Background

Chapter 2

NURBS Basics

In geometric modeling, the two most common methods of representing curves and surfaces are implicit equations and parametric functions [PT97]. While both forms have advantages over each other, the parametric form is the preferred method to represent free-form surfaces in computer aided geometric design, due to its intuitive geometrical properties. In parametric form, the coordinates of a surface-point are represented separately as an explicit function of two independent parameters

$$S(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}, \qquad (u,v) \in [a,b] \times [c,d].$$

Thus, $\vec{S}(u,v)$ is a vector-valued function of the independent variables, u and v. For many types of free-form surfaces, the domain is normalized to let $0 \leq u,v \leq 1$, in order to simplify computations[PT97].

The most used type of parametric free-form surface is the *Non-Uniform Rational B-Spline*-surface or NURBS-surface. Before going into the details of this surface type, the NURBS-curve is discussed first, which is easily extended to a surface. Section 2.1 will start with some mathematical definitions about NURBS.

2.1 Definition

Geometrically, a NURBS-curve is defined by a set of *control-points*, along with a weight for each control-point, and a *knot-vector*. The control-points roughly specify the shape of the curve (see Figure 2.1a). The shape of the resulting curve can be fine-tuned in several ways: by including more control-points around an existing control-point, the curve will be drawn towards the cluster of control-points. However, this will require more storage for representing the curve, since more control-points are needed in order to obtain the desired shape. Alternatively, by specifying a weight for each control-point, the curve can be pulled towards the control-point by increasing this weight (see Figure 2.1b). Decreasing this value will push the curve away from the control-point. Using weights instead of extra control-points, a more compact representation of the curve is obtained. More important, using the weights, the required geometry can be specified very precisely, without them it is impossible to represent basic shapes such as circles, arcs and other conic sections. Therefore, using NURBS it is possible to specify analytical shapes as well as free-form shapes. Finally, the knot-vector is used to fine-tune the shape of the curve (Figure 2.1c, Figure 2.1d). The knot-vector will be discussed in Section 2.3. The power of a NURBS-curve lies in the fact that its complexity is independent on the number of control-points. Instead, it depends on its degree. Therefore, a NURBS-curve can be adjusted locally without changing the shape of the entire curve.

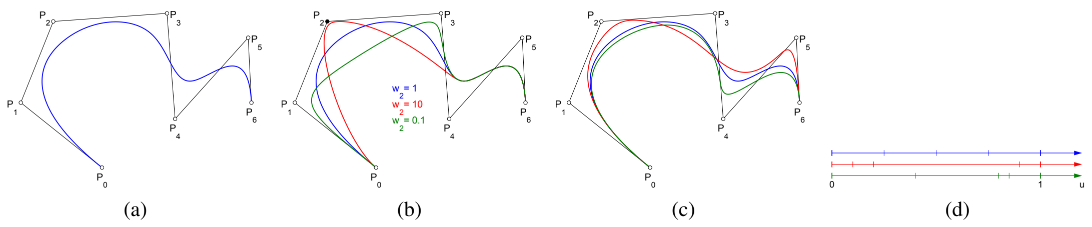


Figure 2.1: Fine-tuning a NURBS-curve (a): By specifying different weights (b) and through the knot-vector (c,d).

Mathematically, a NURBS-curve $\vec{C}(t)$ is a piecewise rational function defined by a set of n control-points $\{\vec{P}_i\}$ along with their weights $\{w_i\}$, its degree p (or its order $p+1$), and a non-decreasing sequence $\{t_0, ..., t_{n+p}\}$ called the *knot-vector* on the domain $[t_p, t_n]$:

$$\vec{C}(t) = \frac{\sum_{i=0}^{n-1} w_i \vec{P}_i N_i^p(t)}{\sum_{i=0}^{n-1} w_i N_i^p(t)} \equiv \frac{\vec{D}(t)}{w(t)}, \tag{2.1}$$

or more compact as:

$$\vec{C}^h(t) = \sum_{i=0}^{n-1} \vec{P}_i^h N_i^p(t) \equiv \left(\vec{D}(t), w(t)\right), \tag{2.2}$$

where the $N_i^p(t)$ functions are the B-Spline basis-functions (discussed in Section 2.2). The $\{\vec{P}_i^h\}$ control-points embed their weights using a *homogeneous space*-coordinate, by multiplying each coordinate with the weight w_i and appending the weight to the control-point: $\vec{P}_i^h \equiv (w_i \vec{P}_i, w_i)$. To obtain the corresponding curve-point as in Equation 2.1, $\vec{C}^h(t)$ only needs to be divided by its homogeneous coordinate $w(t)$.

In the special case of all weights equaling one, the equation reduces to:

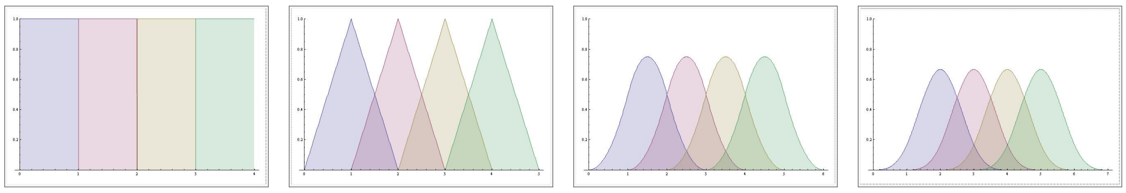
$$\vec{C}(t) = \sum_{i=0}^{n-1} \vec{P}_i N_i^p(t). \tag{2.3}$$

2.2 Cox-de Boor Recurrence

The B-Spline basis-functions, or *blending-functions*, in Equation 2.1 are used to smoothly blend together the control-points of the curve. They are derived from a knot-vector (Section 2.3) using the *Cox-de Boor* Recurrence formula [dB72, Cox72]:

$$N_i^p(t) = \begin{cases} 1 & \text{if } p = 0 \text{and } t \in [t_i, t_{i+1}), \\ 0 & \text{if } p = 0 \text{and } t \notin [t_i, t_{i+1}), \\ \left[\frac{t - t_i}{t_{i+p} - t_i}\right] N_i^{p-1}(t) + \left[\frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}}\right] N_{i+1}^{p-1}(t) & \text{otherwise.} \end{cases} \tag{2.4}$$

Here, $N_i^0(t)$ are step functions equal to zero except in the interval $[t_i, t_{i+1})$. For the basis-functions of higher degree, they are linear combinations of two lower-degree basis-functions (Figure 2.2). In the case the numerator and the denominator are both zero, the convention is used that $\frac{0}{0} \equiv 0$.



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

Figure 2.2: Example B-Spline basis-functions: constant (a), linear (b), quadratic (c) and cubic (d).

2.2.1 Local support

Following the dependencies of the recurrence, the following triangular structure appears:

$$\begin{array}{llllll}
N_i^p(t) & & & & & \\
N_i^{p-1}(t) & N_{i+1}^{p-1}(t) & & & & \\
N_i^{p-2}(t) & N_{i+1}^{p-2}(t) & N_{i+2}^{p-2}(t) & & & \\
\vdots & & & & & \\
N_i^0(t) & N_{i+1}^0(t) & N_{i+2}^0(t) & N_{i+3}^0(t) & \cdots & N_{i+p}^0(t)
\end{array}$$

As can be seen, basis-function $N_i^p(t)$ depends on basis-functions $N_i^0(t)\cdots N_{i+p}^0$, therefore $N_i^p(t)$ is zero if $t \notin [t_i, t_{i+p+1})$. This means that each control-point P_i has *local support* on the curve on the domain $[t_i, t_{i+p+1})$, moving the control-point does not change the shape globally.

2.3 Knot-vector

The *knot-vector* determines the resulting basis-functions and thus the shape of the NURBS-curve. A knot-vector $\vec{t}$ consists of a non-decreasing sequence of $n+p+1$ real numbers, the so-called *knots*, where n is the number of control-points of the corresponding curve, and p is the degree of that curve:

$$\vec{t} = \{t_0, t_1, \cdots, t_{n+p}\}. \tag{2.5}$$

These knots "tie together" the polynomial pieces of the basis-functions (Figure 2.3). Each successive pair of knots represents a parametric interval $[t_i, t_{i+1})$ called a *knot-span*. Knot-spans may be empty, increasing a knot's *multiplicity* (the number of successive knots which are equal). At a knot of multiplicity k, a basis-function $N_i^p(t)$ is C^{p-k} continuous, i.e. $p-k$ times differentiable. Obviously, within a non-empty knot-span, a basis-function $N_i^p(t)$ is C^∞ continuous, since in that region the basis-function is defined by only one single polynomial. In the special case of a knot t_i having multiplicity p, the only non-zero basis-function is $N_i^p(t)$, with $N_i^p(t_i) = 1$. Therefore, the curve interpolates the control-point $\vec{P}_i$ at $t = t_i$.

There are basically three flavors of knot-vectors: *periodic*, *non-periodic*, and *non-uniform*. The periodic knot-vectors have equidistant knots. Curves using periodic knot-vectors result in having the endpoints coincide, obtaining a *closed* shape. For example, a periodic knot-vector for a curve with $n = 4$ control-points and degree $p = 3$ could be:

$$\vec{t} = \{0, 1, 2, 3, 4, 5, 6, 7\}.$$

Non-periodic knot-vectors also have equidistant knots, except for the first and last knots, having a multiplicity of $p+1$. As a result, the curve interpolates both endpoints. For example, a non-periodic

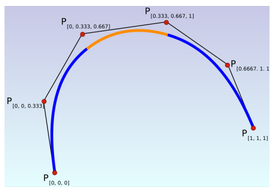


Figure 2.3: Example of polynomial segments of a cubic NURBS-curve.

knot-vector with $n = 7$ control-points and degree $p = 2$ could be:

$$\vec{t} = \{0,0,0,0.1,0.2,0.3,0.4,0.5,0.5,0.5\}.$$

Non-uniform knot-vectors provide more flexibility as they allow each knot-span to vary. For example, a non-uniform knot-vector with $n = 9$ control-points and degree $p = 2$ could be:

$$\vec{t} = \{0,0,0,0.5,0.5,0.75,2,2,2.5,2.5,2.5\}.$$

Finally, a very special knot-vector is the non-periodic knot-vector for which $n = p+1$. These knot-vectors result in basis-functions mathematically equivalent with *Bernstein*-polynomials, which are used by the well-known *Bézier*-curves [BM99]. For example, the following knot-vector results in a cubic Bézier-curve:

$$\vec{t} = \{0,0,0,0,1,1,1,1\}$$

2.4 Convex Hull

A very useful property of B-Spline basis-functions is the *partition of unity* which states:

$$\sum_{j=i-p}^{i} N_j^p(t) = 1 \quad \text{for all } t \in [t_i, t_{i+1}). \tag{2.6}$$

Which is proved easily by expressing the sum in terms of lower-degree basis-functions. For the full proof I refer to [PT97].

Combined with the *local support* property (Section 2.2.1), the following statement can be made, known as the *strong convex hull* property: if $t \in [t_i, t_{i+1})$, then $C(t)$ is contained within the convex hull of the control-points $\vec{P}_{i-p}, \ldots, \vec{P}_i$. This property is shown in Figure 2.4.

2.5 Knot-insertion

Another basic ingredient for many other algorithms is *knot-insertion*, in which a knot is inserted in an existing knot-vector without changing the shape of the curve [PT97]. However, since the size of the knot-vector is always equal to $n+p+1$, an extra control-point needs to be added (or the degree needs to be incremented, but this modifies the shape of the curve). Furthermore, the existing control-points need to be modified in order to preserve the shape of the curve.

When inserting a knot $t \in [t_i, t_{i+1})$, the *local support* property (Section 2.2.1) states that only basis-functions $N_{i-p}^p(t)..N_i^p(t)$ are non-zero. Therefore, the computation of the new control-points can be restricted to only these. The following scheme transforms the old control-points $\{\vec{P}_0, \ldots, \vec{P}_n\}$

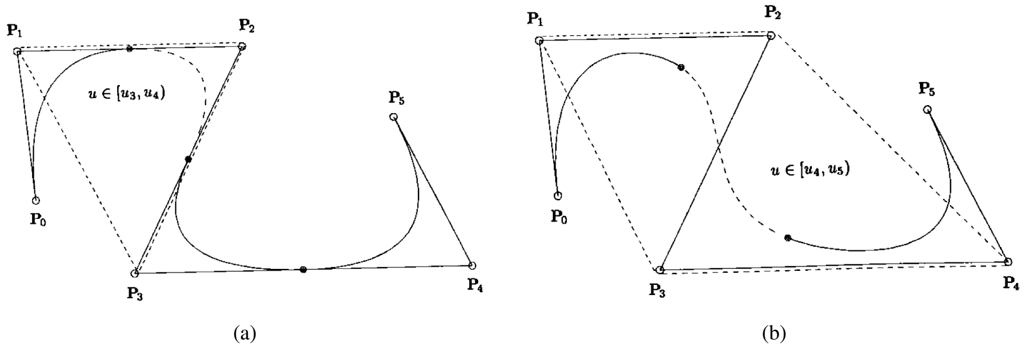


Figure 2.4: Strong convex hull property: for $u \in [u_i, u_{i+1})$, $\vec{C}(u)$ is in the triangle $\vec{P}_{i-2}\vec{P}_{i-1}\vec{P}_i$ (a) and in the quadrilateral $\vec{P}_{i-3}\vec{P}_{i-2}\vec{P}_{i-1}\vec{P}_i$ (b).

to the new ones $\{\vec{Q}_0, ..., \vec{Q}_{n+1}\}$ (Figure 2.5):

$$\vec{Q}_k = \begin{cases} \vec{P}_k & k \leq i-p \\ (1-\alpha_k)\vec{P}_{k-1} + \alpha_k\vec{P}_k & i-p < k \leq i \\ \vec{P}_{k-1} & otherwise, \end{cases}$$

where

$$\alpha_k = \frac{t - t_k}{t_{k+p} - t_k}.$$

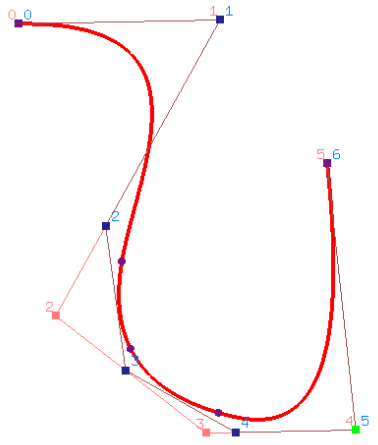


Figure 2.5: Knot-insertion: knot inserted at $t = 0.5$.

2.6 Surfaces

A NURBS-curve is easily generalized to a NURBS-surface $\vec{S}(u,v)$, by defining $\vec{S}(u,v)$ as a *grid* of $n \times m$ control-points $\{\vec{P}_{ij}\}$ along with their weights $\{w_{ij}\}$, the degrees p and q, and two sets of basis-functions for each parametric direction, together with their knot-vectors $\{u_0,\ldots,u_{n+p}\}$ and $\{v_0,\ldots,v_{m+q}\}$:

$$\vec{S}(u,v) = \frac{\sum_{j=0}^{m-1}\left[\sum_{i=0}^{n-1} w_{ij}\vec{P}_{ij}N_i^p(u)\right]N_j^q(v)}{\sum_{j=0}^{m-1}\left[\sum_{i=0}^{n-1} w_{ij}N_i^p(u)\right]N_j^q(v)} \equiv \frac{\vec{D}(u,v)}{w(u,v)}, \tag{2.7}$$

or analogous to Equation 2.2:

$$\vec{S}^h(u,v) = \sum_{j=0}^{m-1}\left[\sum_{i=0}^{n-1} \vec{P}_{ij}^h N_i^p(u)\right]N_j^q(v) \equiv \left(\vec{D}(u,v), w(u,v)\right). \tag{2.8}$$

which needs to be divided by its homogeneous coordinate $w(t)$ in order to obtain the surface-point. The domain of the surface is then being defined as $[u_p,u_n] \times [v_q,v_m]$. Just like the parametric domain of a NURBS-curve is divided into *spans*, the parametric domain of a NURBS-surface is divided into rectangular areas called *patches* (Figure 2.6). The patches are defined by each combination of the non-empty spans between the two parametric directions.

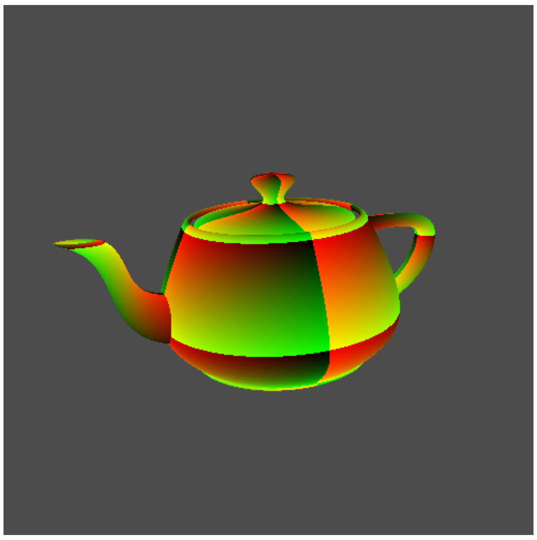


Figure 2.6: NURBS-surface example: Utah Teapot consisting of 32 bi-cubic patches. The patches are visualized using a red and green color mapping for the parametric u and v direction, respectively.

2.7 Derivative

The availability of the partial derivatives of a NURBS-surface is very important for many algorithms, such as finding the roots, determining the normal at a point on a surface, computing the curvature, etc. In this section, a very simple analytical formula is given for the derivative of a NURBS-curve, which is then extended to the partial derivatives computation of a NURBS-surface.

The first-order derivative of a NURBS-curve is obtained by applying the quotient rule to Equation 2.1:

$$\vec{C}(t)' = \frac{w(t)\vec{D}(t)' - \vec{D}(t)w(t)'}{w(t)^2}, \tag{2.9}$$

where

$$\vec{D}(t)' = \sum_{i=0}^{n} w_i \vec{P}_i N_i^p(t)', \qquad w(t)' = \sum_{i=0}^{n} w_i N_i^p(t)',$$

and

$$N_i^p(t)' = \left(\left[\frac{p}{t_{i+p} - t_i} \right] N_i^{p-1}(t) - \left[\frac{p}{t_{i+p+1} - t_{i+1}} \right] N_{i+1}^{p-1}(t) \right). \tag{2.10}$$

A proof for Equation 2.10 can be found in [PT97, Pro05].

The first-order partial derivatives of a NURBS-surface follow by applying the quotient rule to Equation 2.7:

$$\vec{S}_u(u,v) = \frac{w(u,v)\vec{D}_u(u,v) - \vec{D}(u,v)w_u(u,v)}{w(u,v)^2}, \tag{2.11}$$

$$\vec{S}_v(u,v) = \frac{w(u,v)\vec{D}_v(u,v) - \vec{D}(u,v)w_v(u,v)}{w(u,v)^2}, \tag{2.12}$$

where

$$\vec{D}_u(u,v) = \sum_{j=0}^{m} \left[\sum_{i=0}^{n} w_{ij} \vec{P}_{ij} N_i^p(u)' \right] N_j^q(v), \qquad w_u(u,v) = \sum_{j=0}^{m} \left[\sum_{i=0}^{n} w_{ij} N_i^p(u)' \right] N_j^q(v),$$

$$\vec{D}_v(u,v) = \sum_{j=0}^{m} \left[\sum_{i=0}^{n} w_{ij} \vec{P}_{ij} N_i^p(u) \right] N_j^q(v)', \qquad w_v(u,v) = \sum_{j=0}^{m} \left[\sum_{i=0}^{n} w_{ij} N_i^p(u) \right] N_j^q(v)'.$$

Chapter 3

NURBS Ray Tracing

As we discussed in Chapter 1, rasterization of the tessellation is not very memory-friendly, requires a long preprocessing time, and does not provide the desired realism. Furthermore, it will become a very slow visualization for too complex scenes. Although ray tracing solves these last two problems, it still depends on the quality of the tessellation.

Alternatively, the original NURBS-surface can be ray traced directly, by employing a root-finder to find intersections between a ray and the original mathematical representation of the NURBS-surface (Section 3.3.1). This method, while being more expensive, leads to an *always* exact visualization of the surface without any artifacts, independent of the view point. Additionally, since the surface does not need to be converted into a triangular mesh, the tessellation step can be avoided, resulting in a shorter start-up time and lesser memory requirements.

Before we discuss the ray-patch intersection algorithm, we start with some BVH-based methods to traverse through the scene more efficiently.

3.1 Scene Traversal

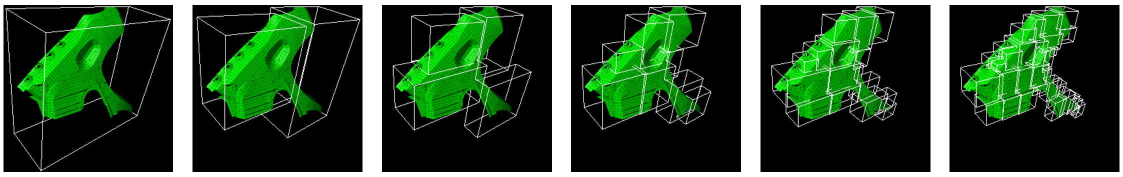
A naive implementation of the basic ray tracing algorithm will test every object for an intersection during ray traversal, resulting in many costly computations. Already since the very beginning of ray tracing, several ray traversal acceleration data structures have been developed, in order to reduce this number of ray/object intersection tests. The first one ever used in ray tracing was the *bounding volume hierarchy* (BVH) [WH80], after which many other more efficient data structures have been used, such as *uniform grids*, *octrees*, *kd-trees*, etc. In 2001, Havran investigated these data structures in his PhD. thesis, and concluded the *kd-tree* as the most efficient, and the *BVH* as the least efficient [Hav01].

More recently, the bounding volume hierarchy has gained more popularity, due to its simpler traversal algorithm, lesser memory requirements and being more appropriate for dynamic scenes [WBB08, WBS07, LYTM06]. Furthermore, BVHs built according to the *surface area heuristic [MB90]* seem to be quite competitive to kd-trees, when traced using *packets of rays* (Section 3.2) [WBS07].

The Bounding Volume Hierarchy, or *BVH*, is a hierarchical scene partitioning datastructure. It differs from spatial partitioning techniques (uniform grid, kd-tree, etc.) in that it partitions objects as opposed to space.

In general, a BVH starts with a *root-node*, holding a bounding volume corresponding to the entire scene (usually a sphere or a bounding box). The children of the root-node, and in general of every *internal-node*[1], partition their parent's objects by holding a bounding volume of a subset of

[1] A node with children is called an *internal-node*.



(a) Level 0 (b) Level 1 (c) Level 2 (d) Level 3 (e) Level 4 (f) Level 5

Figure 3.1: Six levels in a Bounding Volume Hierarchy.

Algorithm 2 Traverse-BVH

```
Require: Ray $R$, BVH-Node $N$
Ensure: Object $O$
 1: if $R$ does not intersect the BV of $N$ then
 2:     return null
 3: end if
 4: if $N$ is leaf-node then
 5:     return Object of($N$)
 6: end if
 7: for all child $C$ do
 8:     $O \leftarrow$ Traverse-BVH($R$,$C$)
 9:     if $O$ is not null then
10:         return $O$
11:     end if
12: end for
```

their parent's objects. Finally, a node without children is called a *leaf-node*, which wraps a single object and holds a bounding volume of it. Figure 3.1 shows an example of BVH built from a scene.

3.1.1 BVH Traversal

The basic ray traversal algorithm is modified by first testing for an intersection with the BVH. A ray missing the bounding volume of the root-node implies a ray missing the entire scene and is therefore discarded. Otherwise the ray is tested against the bounding volume of the child-nodes, and the ray is traversed further through the nodes the ray intersects, if any. This process continues recursively until the traversal reaches a leaf-node, where the basic algorithm is executed. Algorithm 2 shows the modified traversal using a BVH as the acceleration datastructure.

Using this method, many costly ray/object intersection tests will be avoided, since rays will terminate early when missing a bounding volume. On average, a ray traverses the tree until a leaf is reached or the ray is early terminated, because the ray misses all bounding volumes. Since the depth of the tree is logarithmic in the number of objects in the scene, traversing the scene is now of order $O(\log n)$, instead of $O(n)$.

3.1.2 BVH Construction

A careful construction of the BVH-tree is very important. Because the BVH determines how a ray traverses the scene, a badly constructed BVH will result in many ray/bounding box-intersection tests. For other meshes with no additional hierarchical information, the construction of the BVH is non-trivial. The BVH is usually constructed by recursively partitioning the scene objects using a split plane determined by some heuristic. Traditional heuristics determine the splitting plane to be the midpoint of the bounding box' axis with the longest extent (*spatial median*) or the midpoint through the median of the objects when sorted along the axis with the longest extent (*object median*). Although the spatial or object median methods lead to a very efficient construction algorithm, the BVH-traversal efficiency is generally not very high. A better way to determine where to partition the scene is by using the *Surface Area Heuristic*. This algorithm chooses from a set of splitting planes, the one with the lowest cost based on the surface area and number of primitives of the children. More information about this algorithm can be found in [GS87, MB90].

3.2 Packet-Based Ray Tracing

In [WSBW01], a modified version of the standard ray tracing algorithm is presented, which is called the RTRT system. By paying close attention to the coherence among the rays, a performance increase of an order of magnitude is achieved by tracing *packets of rays* in parallel.

A kd-tree datastructure is used to accelerate the ray traversal. Since the traversal algorithm is relatively simple to implement, the complexity of the code is reduced so the compiler can optimize the generated code.

A traditional ray tracer will trace all rays sequentially. However, since there is a lot of coherence among primary rays and to a somewhat lesser degree among shadow rays, the same memory locations will be accessed multiple times for rays that intersect the same primitive. In order to remedy this, multiple rays are grouped together to form a *packet of rays*. Now the whole packet is traced simultaneously. This way, the memory is only accessed once for the whole packet, instead of separately for each ray.

During traversal a packet has three choices: traverse the left child, the right child or both. Although the rays inside the packet are coherent, it is not guaranteed that they all will traverse the same child. Therefore some rays get disabled when traversing the non-intersecting child. This way, unnecessary calculations will be avoided.

By using SIMD instructions, the packets can be traversed, intersected and shaded in parallel.

Since the major bottle-neck in Ray Tracing is the memory bandwidth [WSBW01], it is necessary to exploit the caches as much as possible. By aligning the data to the cache lines, every data item can be fetched by loading one cache line. Items larger than the cache line width, are split over multiple cache lines and padded to half the width. In order to minimize the bandwidth requirements, only data needed for the actual computation is kept together. In this way, loading data is avoided that will not be used. Another problem is the relative high latency for accessing the main memory. By prefetching the data, instead of fetching it on demand, the data will be available instantly when they are needed.

3.3 Ray-Patch Intersection

The most important part of a NURBS Ray Tracer is the intersection test between a three-dimensional ray and the surface, since this test determines if the corresponding pixel should be colored using the surface or not. Normally, such a test would be trivial since a lot of fast analytical intersection tests are available for standard primitives, such as triangles, quadrilaterals, etc. However, for NURBS-surfaces, this is less obvious.

Algebraic methods Several approaches exist for computing an intersection between a ray and a NURBS-surface. Algebraic methods have been devised for the intersection of a ray and a bi-cubic parametric surface [Kaj82]. While this method computes the exact intersection, it requires the computation of the roots of a polynomial of 18^{th} degree. Solving such a polynomial will be very expensive. Furthermore, this method will be applicable to bi-cubic surfaces only and not to NURBS-surfaces.

Subdivision Another approach is on-the-fly subdivision [BBLW07]. By repeatedly subdividing the NURBS-patches, the mesh will converge to the actual surface. Due to the coherence among rays, same patches will be subdivided in exactly the same way. By paying attention to the coherence among these rays, the subdivision-cost can be amortized over a "packet" of rays. When the subdivision is sufficiently close to the limit surface, the Ray Tracer can use standard tests to compute the intersection.

Numerical methods The third approach is a numerical method to approximate the intersection, known as *Newton's Iteration*, or *the Newton-Rhapson method* [PTVF97]. This method uses a very simple algorithm to iteratively find better approximations of the roots of non-linear functions. A nice property of the algorithm is, that in each subsequent iteration, the error of the approximation decreases quadratically, roughly meaning that in each iteration the number of correct digits are doubled. Additionally, the algorithm only requires the previous approximation, to compute the next approximation, thus keeping memory requirements low. The algorithm depends on the function, its derivative, and an initial guess. For this initial guess, it is very important to be close enough to the exact solution in order for the method to converge towards the correct root.

This method will be explained in more detail in Section 3.3.1. Obtaining suitable initial guesses for NURBS-surfaces will be discussed in Section 3.4.

3.3.1 Newton-Rhapson Method

The general idea is to start with the initial guess and compute the tangent line that approximates the function at that point Figure 3.2. The x-intercept of this tangent line will typically be closer to the root, than the initial guess. Now this process can be repeated to obtain closer approximations to the root of the function:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.1}$$

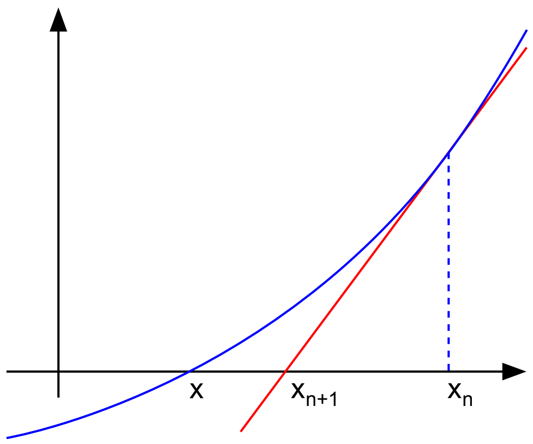


Figure 3.2: Newton's Iteration: each subsequent point is closer to the root.

However, we are dealing with a ray/NURBS-*surface* intersection, therefore the standard algorithm needs to be extended. The approach taken by [MCFS00] is discussed in here. The ray

$\vec{r} = \vec{o} + \lambda\vec{d}$, where $\vec{o}$ denotes the ray's origin and d the direction, is represented as the intersection of two orthogonal planes $\vec{p}_1 = (\vec{n}_1, d_1)$ and $\vec{p}_2 = (\vec{n}_2, d_2)$, where the $\vec{n}_i$ are orthogonal vectors of unit length, perpendicular to $\vec{d}$. The d_i, the planes distances to the world's origin, are given by $d_i = \vec{n}_i \cdot \vec{o}$, with $\vec{o}$ denoting the ray origin. The plane equations are setup by:

$$\vec{n}_1 = \begin{cases} \frac{(\vec{d}_y, -\vec{d}_x, 0)}{\sqrt{\vec{d}_x^2 + \vec{d}_y^2}} & \text{if } |\vec{d}_x| > |\vec{d}_y| \text{ and } |\vec{d}_x| > |\vec{d}_z| \\ \frac{(0, \vec{d}_z, -\vec{d}_y)}{\sqrt{\vec{d}_y^2 + \vec{d}_z^2}} & \text{otherwise,} \end{cases} \tag{3.2}$$

$$\vec{n}_2 = \vec{n}_1 \times \vec{d} \cdot |\vec{d}|^{-1} \tag{3.3}$$

$$d_1 = \vec{n}_1 \cdot \vec{o} \tag{3.4}$$

$$d_2 = \vec{n}_2 \cdot \vec{o}. \tag{3.5}$$

The first plane equation results in a plane with a normal lying in the xy-plane or the yz-plane, depending on the largest magnitude of the components of the direction of the ray Figure 3.3.

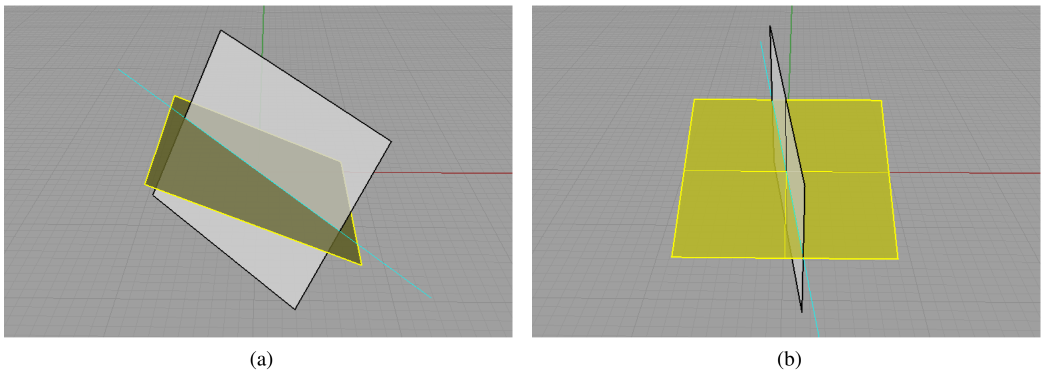


Figure 3.3: Plane representation for a ray for which $|\vec{d}_x| > |\vec{d}_y|$ and $|\vec{d}_x| > |\vec{d}_z|$ holds, results in a plane rotated about the z-axis and a plane perpendicular to it (a) and for a ray for which it does not hold, the resulting plane will be rotated about the x-axis (b). The cyan line represents the ray and the yellow plane is the first plane $\vec{p}_1 = (\vec{n}_1, d_1)$.

To find the intersection of a surface $\vec{S}(u,v)$ and a ray, we have to find the roots of:

$$\vec{R}(u,v) = \begin{pmatrix} \vec{n}_1 \cdot \vec{S}(u,v) + d_1 \\ \vec{n}_2 \cdot \vec{S}(u,v) + d_2 \end{pmatrix}. \tag{3.6}$$

Geometrically, this can be interpreted as the distance from the evaluated surface-point to the ray. $\vec{R}(u,v)$ becoming zero indicates that this distance becomes zero, hence an intersection point is found. The iteration will be continued until one of the following termination criteria is met:

1. If the distance to the real root falls below some user defined threshold ε, then an intersection point is found, i.e.

$$\left|\vec{R}(u_n, v_n)\right| < \varepsilon$$

   Since the obtained intersection point is an approximation, it will definitely not lie along the ray. Therefore, to obtain the ray parameter λ, the point is projected onto the ray:

$$\lambda = (\vec{S}(u,v) - \vec{o}) \cdot \vec{d}. \tag{3.7}$$

2. Whenever the iteration takes us further away from the root, the computation will be aborted, assuming divergence.
$$\left|\vec{R}(u_{n+1}, v_{n+1})\right| > \left|\vec{R}(u_n, v_n)\right|$$

3. A maximum number of iteration steps has been performed, also indicating divergence.

Whenever none of the criteria is met, the algorithm continues with a next iteration by updating the (u, v) values. However, since we are dealing with a bi-variate function, Equation 3.1 cannot be employed. Instead, the iteration step is extended to a two-dimensional problem, given by:

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - J^{-1}\vec{R}(u_n, v_n) \tag{3.8}$$

where J is the *Jacobian* matrix of $\vec{R}$ containing the partial derivatives:

$$J = \begin{pmatrix} \vec{n}_1 \cdot \vec{S}_u(u, v) & \vec{n}_1 \cdot \vec{S}_v(u, v) \\ \vec{n}_2 \cdot \vec{S}_u(u, v) & \vec{n}_2 \cdot \vec{S}_v(u, v) \end{pmatrix} \tag{3.9}$$

One problem which may occur, is that the Jacobian matrix turns out to be singular, making it impossible to compute the inverse. This may happen for example at non-regular areas of the surface (i.e. where the normals are zero). Therefore, in the case of a singular Jacobian, the (u, v) values are perturbed with some small random values. Algorithm 3 gives an overview on how to compute the intersection between a ray and a NURBS-surface, given an initial-guess value (u, v).

The core part of the algorithm, the so-called iteration, heavily depends on the evaluation of a surface-point together with its partial-derivatives. Several algorithms exist for efficient evaluation of surface-points and partial-derivatives.

3.4 Adaptive subdivision

The first step required for a direct NURBS Ray Tracing implementation is the creation of a *Bounding Volume Hierarchy* (Section 3.1.2). Normally, a leaf of a BVH bounds multiple primitives. Since the cost for the intersection test between a ray and a triangle[2] is relatively low compared to that of traversing the BVH, it is clever to group multiple primitives. However, when ray tracing NURBS-surfaces, this cost is very high, therefore, it is generally not a very good idea to group multiple NURBS-surfaces into a single BVH-leafnode. Instead of each BVH-leafnode bounding multiple surfaces, the surface is refined and for each sub-patch, a new child node is created bounding the convex hull of the sub-patch, as in [MCFS00]. Besides a reference to the corresponding surface-patch, these leaf-nodes also contain an *initial-guess* parameter value, for the Newton's Iteration root-finder (Section 3.3.1). This initial guess value is defined as the center of the parametric interval of the sub-patch's parametric domain. By increasing the number of leaf-nodes for a surface, the parametric interval becomes smaller, and a better initial guess value is obtained (Figure 3.4). Additionally, by increasing the number of leaf-nodes per surface, the number of intersection tests for rays missing the surface is reduced, because the bounding volume is more tight when using many small bounding volumes. In other words, rays will be culled away before the costly intersection test takes place. In order to construct a bounding volume hierarchy from the subdivision, the leaf-nodes are sorted along the axis with the greatest extent. The first half of the leaf-nodes becomes the first child node, and the second half the second child. Methods for constructing a subdivision will now be discussed.

[2]Or any other primitive for which a very efficient intersection test exists

Algorithm 3 The Newton Root-Finder

```
Require: $u$, $v$
Ensure: $\lambda$, $u$, $v$
 1: $error_{prev} \leftarrow \infty$
 2: $u_{initial} \leftarrow u$
 3: $v_{initial} \leftarrow v$
 4: for $i = 1$ to MAX_ITERATIONS do
 5:     $\vec{S} \leftarrow \text{EvaluateSurface}(u, v)$
 6:     $\vec{R} \leftarrow \begin{pmatrix} \vec{n}_1 \cdot \vec{S} + d_1 \\ \vec{n}_2 \cdot \vec{S} + d_2 \end{pmatrix}$
 7:     $error \leftarrow \left|\vec{R}_1\right| + \left|\vec{R}_2\right|$
 8:     if $error < \varepsilon$ then
 9:         return $\lambda$
10:     end if
11:     if $error > error_{prev}$ then
12:         return $\infty$
13:     end if
14:     $error_{prev} \leftarrow error$
15:     $J \leftarrow \text{ComputeJacobian}$
16:     if $\text{singular}(J)$ then
17:         $\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} u \\ v \end{pmatrix} + 0.1 \begin{pmatrix} (u_{initial} - u) \cdot \text{random}([0,1]) \\ (v_{initial} - v) \cdot \text{random}([0,1]) \end{pmatrix}$
18:     else
19:         $\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} u \\ v \end{pmatrix} - J^{-1}\vec{R}$
20:     end if
21: end for
22: return $\infty$
```

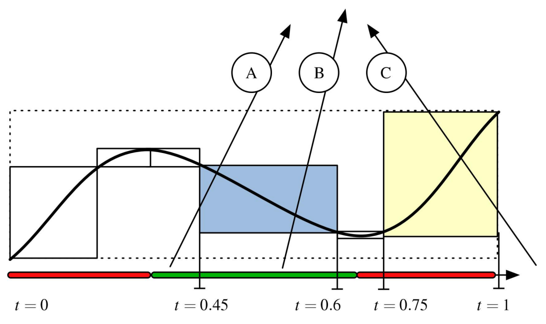


Figure 3.4: Example of a NURBS-curve with exemplary associated bounding volumes.

3.4.1 Recursive subdivision

One very simple method to subdivide a surface is to recursively split each surface-patch in the center of the parametric domain and the sides of the domain, resulting in four sub-patches. These new patches determine the knot-span used for evaluation.

Since there is no spatial relation between the control-points of the surface and its parametric domain, sampling in such a uniform way will not necessarily result in a uniform distribution of the sample points on the surface. For example, the subdivision could fail to represent highly curved areas of a surface if the sampling density is too low. A simple solution is to increase the sampling density, in order to refine such curved areas. However, this will unnecessarily increase the number of polygons in flat areas of the surface. A better solution is to adaptively determine the sampling density in different areas of the surface. This way, highly curved areas will automatically be sampled more dense than less curved areas.

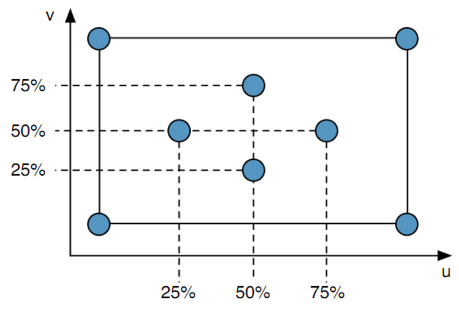
The uniform method is easily transformed into an adaptive version. Instead of using a fixed recursion depth, extra criteria can be included to determine whether or not a (sub-)patch needs more subdivision.

Normal based criteria

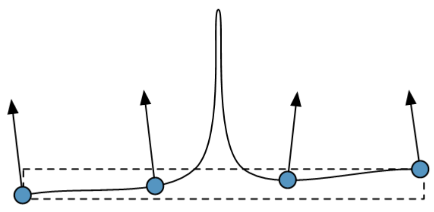
[Abe05] samples the normals $\vec{n}_i$ of the sub-patch at eight different locations (Figure 3.5a) and then uses the following criterion to stop the subdivision for that sub-patch [Abe05]:

$$\prod_{i=1}^{7} \vec{n}_i \cdot \vec{n}_{i+1} \approx 1.$$

The dot-product between two nearly-identical vectors is approximately equal to one[3]. In other words, the surface is considered flat if the normals are all approximately equal. However, this method is not always successful in capturing curved areas as Figure 3.5b indicates.



(a) The blue circles mark the positions where normals will be evaluated.



(b) The normals taken may satisfy the flatness criterion, but the peak in the middle of the spline is not recognizes properly.

Figure 3.5: Flatness criterion based on the normals of a sub-patch.

Straightness based criteria

A more thorough approach is taken by [Pet94], however their method operates directly on the control-point grid instead of the surface itself. A *straightness*-test is applied to every row and column of the control-point grid. The straightness-test return true if all points in the row (or column) are co-linear. However, this test alone is not sufficient. If the surface is twisted, the rows and columns could be straight, but the surface is nevertheless heavily curved. Therefore, a final check is made to determine if the corners are co-planar.

To split the sub-patch, a *refinement* step is applied using the *Oslo* algorithm [CLR80, BBB87]. This algorithm modifies the knot-vector of the non-straight direction by adding p (or q) new knots

[3]It is assumed that $|\vec{n}_i| = 1$.

at the parametric center of the sub-patch. Then, using this new knot-vector, the control-point grid is updated by adding control-points corresponding to these new knots. Using such a refinement operation, the shape of the surface remains unchanged. However, since p (or q) new knots are inserted, the new control-points corresponding to these knots touch the surface. Thus, two separate sub-patches are obtained along with their control-points (Figure 3.6).

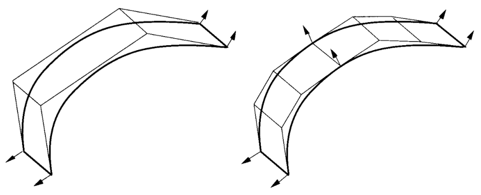
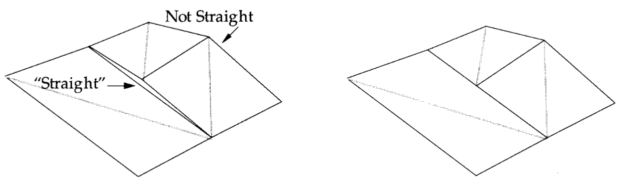


Figure 3.6: Splitting a sub-patch into two sub-patches obtaining two control-point grids.

During the subdivision, two adjacent sub-patches with different subdivision depths introduce cracks in the surface (Figure 3.7a) if the subdivision is used as a tessellation. In order to fix this, the surface points corresponding to the corners of the crack are projected onto the straight line (Figure 3.7b).



(a) Crack introduced by different subdivision depths. (b) Fixed crack by projecting the corners onto the straight line.

Figure 3.7: Procedure to fix a crack in the tessellation.

3.4.2 Direct subdivision

Yet another method for subdivision, particularly well suited for ray tracing, is described in [MCFS00], in which the refinement scheme is based on the curvature of each patch. Each patch is refined according to a heuristic approximating the curvature of that patch.

The heuristic operates on the *iso-curves*[4] and takes three measures into consideration: the maximum curvature of a curve-segment corresponding to a knot-span, the length of that curve-segment, and a bound on the deviation of the curve-segment from its linear approximation. The measure for the maximum curvature ensures that no multiple roots are found when finding an intersection between a ray and the surface (recall that this method is intended for ray tracing). The measure for the length of a curve-segment makes sure that the initial-guess values are close enough, especially for large patches. The first two measures result in the following heuristic value:

$$n_1 = C_1 \times \max_{[t_i,t_{i+1})}\{\text{curvature}(\vec{C}(t))\} \times \text{arclength}(\vec{C}(t))_{[t_i,t_{i+1})}.$$

[4]Iso-curves are the curves on a surface with a fixed u value or a fixed v value.

The third measure determines the number of segments that needs to be generated and is approximated by:

$$n_2 = C_2 \times \sqrt{\text{arclength}(\vec{C}(t))_{[t_i,t_{i+1})}}.$$

Combining the previous two heuristics, the heuristic value that determines the number of knots n to insert into the knot-span is determined by $n_1 \times n_2$:

$$n = C \times \max_{[t_i,t_{i+1})}\{\text{curvature}(\vec{C}(t))\} \times \text{arclength}(\vec{C}(t))^{3/2}_{[t_i,t_{i+1})}.$$

The C constant is a fineness constant and can be determined empirically.

Since the maximum curvature and the arc length are hard to compute, their values are heavily estimated, which usually results in a too fine refinement rather that too coarse. However, the computation is much faster compared to the exact functions. Furthermore, the constant C can be used to fine-tune the subdivision.

The final heuristic (after replacing many functions by their estimates) becomes:

$$n = C \times \frac{\max_{i-p+2\le j\le i}|\vec{A}_j| \times (t_{i+1}-t_i)^{3/2}}{(\frac{1}{p}\sum_{j=i-p+1}^{i}|\vec{V}_j|)^{1/2}}.$$

where

$$\vec{V}_j = p\frac{\vec{P}_j - \vec{P}_{j-1}}{t_{j+p}-t_j},$$
$$\vec{A}_j = (p-1)\frac{\vec{V}_j - \vec{V}_{j-1}}{t_{j+p-1}-t_j}.$$

The full derivation of the estimated maximum curvature and arc length functions can be found in [MCFS00].

The actual subdivision takes place in two subsequent steps. In the first step, the heuristic is applied to determine the number of knots that needs to be inserted into the non-empty knot-spans. The number of knots that needs to be inserted into each knot-span of the u direction is determined by applying the heuristic to each row of the control-point grid. Per knot-span, the maximum of all rows for this knot-span is used as the number of new knots. This is repeated for the v direction, but now for every column. After the numbers have been determined, the new knots are inserted uniformly distributed over the knot-span. As a final step, each patch is converted into a Bézier-patch by setting the multiplicity of the internal knots to p and q for the u and v parametric directions, respectively.

In the second step, the new control-point grid is computed from the new knot-vectors by using a method similar to knot-insertion (Section 2.5). While it is not important for this step to be fast (in the case of ray tracing it is only done once in a preprocessing phase), the actual implementation will not be discussed here an can be found in [MCFS00].

3.5 Evaluation Schemes

When evaluating a point on a NURBS-curve or surface, it is important to note, that not every basis-function needs to be computed. Since t lies in exactly one knot-span, only one zero-degree basis-function is non-zero. Consequently, only two basis-functions of degree one are non-zero. In general, only $p+1$ basis-functions of degree p are non-zero. In the following scheme, these *non-zero* basis-functions are emphasized:

Algorithm 4 EvaluateSurface

```
Require: $span_u$, $span_v$, $P_{ij}$, $p$, $q$,
    $u \in [u_{span_u}, u_{span_u+1})$, $v \in [v_{span_v}, v_{span_v+1})$ .
Ensure: $\vec{S}, \vec{S}_u, \vec{S}_v$
 1: $N_u[0..p], Nderiv_u[0..p] \leftarrow$ EvaluateBasisFunctionsAndDerivatives$_u$
 2: $N_v[0..q], Nderiv_v[0..q] \leftarrow$ EvaluateBasisFunctionsAndDerivatives$_v$
 3: $\vec{S}^h = (\vec{S}, S_w) \leftarrow \vec{0}$
 4: $\vec{S_u}^h = (\vec{S}_u, S_{u,w}) \leftarrow \vec{0}$
 5: $\vec{S_v}^h = (\vec{S}_v, S_{v,w}) \leftarrow \vec{0}$
 6: for $i = span_u - p$ to $span_u$ do
 7:   for $j = span_v - q$ to $span_v$ do
 8:     $\vec{S}^h \leftarrow \vec{S}^h + \vec{P}^h_{ij} N_u[i-(span_u-p)] N_v[j-(span_v-q)]$
 9:     $\vec{S_u}^h \leftarrow \vec{S_u}^h + \vec{P}^h_{ij} Nderiv_u[i-(span_u-p)] N_v[j-(span_v-q)]$
10:     $\vec{S_v}^h \leftarrow \vec{S_v}^h + \vec{P}^h_{ij} N_u[i-(span_u-p)] Nderiv_v[j-(span_v-q)]$
11:   end for
12: end for
13: $\vec{S} \leftarrow S_w^{-1} \vec{S}^h$
14: $\vec{S}_u \leftarrow (\vec{S}_u - \vec{S} S^{-1}_{u,w}) S_w^{-1}$
15: $\vec{S}_v \leftarrow (\vec{S}_v - \vec{S} S^{-1}_{v,w}) S_w^{-1}$
16: return $\vec{S}, \vec{S}_u, \vec{S}_v$
```

$$
\begin{array}{cccccccccc}
\cdots & \mathbf{N^p_{i-p}(t)} & \cdots & \mathbf{N^p_{i-3}(t)} & \mathbf{N^p_{i-2}(t)} & \mathbf{N^p_{i-1}(t)} & \mathbf{N^p_{i}(t)} & N^p_{i+1}(t) & N^p_{i+2}(t) & \cdots \\
 & & & & & & \vdots & & & \\
\cdots & N^2_{i-p}(t) & \cdots & N^2_{i-3}(t) & \mathbf{N^2_{i-2}(t)} & \mathbf{N^2_{i-1}(t)} & \mathbf{N^2_{i}(t)} & N^2_{i+1}(t) & N^2_{i+2}(t) & \cdots \\
\cdots & N^1_{i-p}(t) & \cdots & N^1_{i-3}(t) & N^1_{i-2}(t) & \mathbf{N^1_{i-1}(t)} & \mathbf{N^1_{i}(t)} & N^1_{i+1}(t) & N^1_{i+2}(t) & \cdots \\
\cdots & N^0_{i-p}(t) & \cdots & N^0_{i-3}(t) & N^0_{i-2}(t) & N^0_{i-1}(t) & \mathbf{N^0_{i}(t)} & N^0_{i+1}(t) & N^0_{i+2}(t) & \cdots
\end{array}
$$

For efficient evaluation, this is a very important fact: instead of multiplying and summing up all n control-points, actually only $p+1$ control-points are involved in the computation. Thus Equation 2.2 becomes:

$$\vec{C}^h(t)_{[t_a, t_{a+1})} = \sum_{i=a-p}^{a} \vec{P}^h_i N^p_i(t)$$

and for a surface, becomes:

$$\vec{S}^h(u,v)_{[u_a,u_{a+1}),[v_b,v_{b+1})} = \sum_{j=b-q}^{b} \left[\sum_{i=a-p}^{a} \vec{P}^h_{ij} N^p_i(u) \right] N^q_j(v)$$

Basis-function cache Algorithm 4 simultaneously evaluates a point on a NURBS-surface along with its partial derivatives, by first computing the basis-functions in both parametric directions and storing them in a basis-function *cache [AGM06]*. Then, the computed basis-functions are multiplied with the corresponding control-points and added to the result.

Directly evaluating the basis-functions using Equation 2.4 is not very efficient. At first, many higher-degree basis-functions depend on the same lower-degree basis-function, resulting in repeated

Algorithm 5 EvaluateBasisFunctionPowerBasisForm

```
Require: $t, p,$
  * $P \leftarrow \alpha_p$
  * $D \leftarrow 0$
for $i = p-1$ to 0
  * $D \leftarrow D * t + P$
  * $P \leftarrow P * t + \alpha_i$
endfor
return $(P, D)$
```

computations for these lower-degree basis-functions. Secondly, many computations of the basis-functions are actually unnecessary, since they depend on basis-functions whose active domain (where the basis-function is non-zero), does not contain the evaluation parameter value and are therefore zero for that value. Finally, a recursive implementation of this algorithm, when compiled, is not very efficient, since the basis-functions are evaluated many times and thus many function calls are issued, which cannot be in-lined by the compiler, resulting in poor performance.

3.5.1 Power-basis Form Evaluation

One possibility to avoid the recursion in Equation 2.4 is to convert the recursive basis-functions into *power-basis form* as in [Abe05].

The power-basis form of a basis-function is defined as:

$$N(t) = \sum_{i=0}^{p} \alpha_i t^i$$

Using Horner's scheme [PTVF97] the polynomial and its derivative can be evaluated very efficiently with linear time complexity:

$$\begin{aligned} N(t) &= \sum_{i=0}^{p} \alpha_i t^i \\ &= \alpha_0 t^0 + \alpha_1 t^1 + \cdots + \alpha_{p-1} t^{p-1} + \alpha_p t^p \\ &= \underbrace{t(\cdots t(t}_{\text{linear in } p}\, \alpha_p + \alpha_{p-1}) \cdots + \alpha_1) + \alpha_0 \end{aligned}$$

For each knot-span, a set of basis-functions in power-basis form is evaluated as well as their derivatives. The power-basis coefficients α_i are obtained in a preprocessing step by a polynomial expansion of Equation 2.4. For full details I refer to [Abe05].

Algorithm 5 evaluates the basis-function and its derivative simultaneously using only multiply-add[5] operations.

Although the power-basis form enables a very efficient evaluation of a basis-function and its derivative, it does require some preprocessing. Additionally, there is a lot of redundancy in the generated coefficients, since a polynomial is maintained for each knot-span for each basis-function, while these are derived from the knot-vector which uses much less memory. Finally, a rather bad property of polynomials in power-basis form is that the evaluation is numerically unstable if the coefficients vary greatly in magnitude [FR87].

[5] Some hardware architectures (e.g. GPUs) are able to perform a multiply operation followed by an add operation in a single combined *multiply-add* instruction.

3.5.2 Direct Evaluation

Although there are efficient iterative algorithms available for the original Cox-de Boor recurrence Equation 2.4, their complexity is still quadratic in the degree of the curve as opposed to the linear complexity of the power-basis form method. However, they are numerically stable, which is a very important property for the numerical root-finder (Section 3.3.1).

Observing the computation of all p-degree basis-functions (Equation 2.4) for $t \in [t_i, t_{i+1})$, it becomes clear there is a lot of redundancy in the computation of them:

$$N_{i-p}^{p}(t) = \left[\frac{t - t_{i-p}}{t_i - t_{i-p}}\right] N_{i-p}^{p-1}(t) + \left[\frac{t_{i+1} - t}{t_{i+1} - t_{i-p+1}}\right] N_{i-p+1}^{p-1}(t) \tag{3.10}$$

$$N_{i-p+1}^{p}(t) = \left[\frac{t - t_{i-p+1}}{t_{i+1} - t_{i-p+1}}\right] N_{i-p+1}^{p-1}(t) + \left[\frac{t_{i+2} - t}{t_{i+2} - t_{i-p+2}}\right] N_{i-p+2}^{p-1}(t) \tag{3.11}$$

$$\vdots$$

$$N_{i-1}^{p}(t) = \left[\frac{t - t_{i-1}}{t_{i+p-1} - t_{i-1}}\right] N_{i-1}^{p-1}(t) + \left[\frac{t_{i+p} - t}{t_{i+p} - t_i}\right] N_{i}^{p-1}(t) \tag{3.12}$$

$$N_{i}^{p}(t) = \left[\frac{t - t_{i}}{t_{i+p} - t_{i}}\right] N_{i}^{p-1}(t) + \left[\frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}}\right] N_{i+1}^{p-1}(t) \tag{3.13}$$

The last term of basis-function $N_j^p(t)$ and first term of basis-function $N_{j+1}^p(t)$ contain the same factor:

$$Q_j = \frac{N_{j+1}^{p-1}(t)}{t_{j+p+1} - t_{j+1}}$$

Note that for the first term of Equation 3.10 and the last term of Equation 3.13 this factor is zero, since $N_{i-p}^{p-1}(t) = N_{i+1}^{p-1}(t) = 0$ for $t \in [t_i, t_{i+1})$.
Now, let:

$$\alpha_j = t - t_{i+1-j}$$
$$\beta_j = t_{i+j} - t$$

Equations 3.10-3.13 are then:

$$N_{i-p}^{p}(t) = \alpha_{p+1}\left[\frac{N_{i-p}^{p-1}(t)}{\alpha_{p+1} + \beta_0}\right] + \beta_1 \left[\frac{N_{i-p+1}^{p-1}(t)}{\alpha_p + \beta_1}\right]$$

$$N_{i-p+1}^{p}(t) = \alpha_{p}\left[\frac{N_{i-p+1}^{p-1}(t)}{\alpha_{p} + \beta_1}\right] + \beta_2 \left[\frac{N_{i-p+2}^{p-1}(t)}{\alpha_{p-1} + \beta_2}\right]$$

$$\vdots$$

$$N_{i-1}^{p}(t) = \alpha_{2}\left[\frac{N_{i-1}^{p-1}(t)}{\alpha_{2} + \beta_{p-1}}\right] + \beta_p \left[\frac{N_{i}^{p-1}(t)}{\alpha_1 + \beta_p}\right]$$

$$N_{i}^{p}(t) = \alpha_{1}\left[\frac{N_{i}^{p-1}(t)}{\alpha_{1} + \beta_{p}}\right] + \beta_{p+1} \left[\frac{N_{i+1}^{p-1}(t)}{\alpha_0 + \beta_{p+1}}\right]$$

Algorithm 6 (named *"Inverted Triangle"*-method in [PT97]) simultaneously evaluates the basis-functions and their derivatives by reusing the previously computed values. Since terms resulting in a division by zero are not computed anymore, the algorithm requires no special treatment of such cases.

Algorithm 6 EvaluateBasisFunctionsDirect

```
Require: $i, t \in [t_i, t_{i+1}), p$.
Ensure: $N[0..p], D[0..p]$
1: $N[0] \leftarrow 1$
2: for $j = 1$ to $p$ do
3:     $\alpha_j \leftarrow t - t_{i+1-j}$, $\beta_j \leftarrow t_{i+j} - t$
4:     $R_N \leftarrow 0, R_D \leftarrow 0$
5:     for $k = 0$ to $j-1$ do
6:         $Q \leftarrow \frac{N[k]}{\alpha_{j-k}+\beta_{k+1}}$
7:         $N[k] \leftarrow R_N + \beta_{k+1} Q$
8:         $R_N \leftarrow \alpha_{j-k} Q$
9:         $D[k] \leftarrow p(R_D - Q)$
10:        $R_D \leftarrow Q$
11:    end for
12:    $N[j] \leftarrow R_N$
13:    $D[j] \leftarrow pR_D$
14: end for
```

3.5.3 Division-free Evaluation

The iterative method described in Section 3.5.2 does not require preprocessing, and the additional storage is even zero (excluding the knot-vector, which is always needed). However, this method contains division-operations, which are costly compared to basic operations such as multiply and addition. Another very fast, yet simple and memory-efficient, iterative approach is presented in [AGM06], for evaluating NURBS-curves and surfaces. Their method requires very little preprocessing, and the data resulting from it takes up only but a small amount of memory.

By carefully investigating the Cox de-Boor recurrence from Equation 2.4, one can see that a lot of constants are involved. By rewriting the equation, the constant part of the equation can be separated out:

$$\begin{aligned}
N_i^p(t) &= \left[\frac{t-t_i}{t_{i+p-1}-t_i}\right] N_i^{p-1}(t) + \left[\frac{t_{i+p}-t}{t_{i+p}-t_{i+1}}\right] N_{i+1}^{p-1}(t) \\
&= \left[\frac{1}{t_{i+p-1}-t_i}(t-t_i)\right] N_i^{p-1}(t) + \left[\frac{-1}{t_{i+p}-t_{i+1}}(t-t_{i+p})\right] N_{i+1}^{p-1}(t) \\
&= [a(t-t_i)]\, N_i^{p-1}(t) + [b(t-t_{i+p})]\, N_{i+1}^{p-1}(t) \\
&= [at+c]\, N_i^{p-1}(t) + [bt+d]\, N_{i+1}^{p-1}(t).
\end{aligned}$$

The resulting equation was rewritten in such a way, that evaluation of a basis-function requires only three multiply-add instructions and one multiply instruction, given that the depending basis-functions of lower degree are known:

$$\underbrace{\underbrace{[at+c]}_{\textit{multiply-add}}\; N_i^{p-1}(t) + \underbrace{\underbrace{[bt+d]}_{\textit{multiply-add}}\; N_{i+1}^{p-1}(t)}_{\textit{multiply}}}_{\textit{multiply-add}}$$

Where the constants can be precomputed in a preprocessing step:

$$a_i^p = \frac{1}{t_{i+p-1} - t_i} \qquad b_i^p = \frac{-1}{t_{i+p} - t_{i+1}}$$
$$c_i^p = -a_i^p t_i \qquad d_i^p = -b_i^p t_{i+p}$$

A slightly different approach is taken for computing all required basis-functions. Figure 3.8 shows the dependencies and evaluation order of the basis-functions. All basis-functions of degree higher than zero depend on two lower-degree basis-functions. Furthermore, the outer basis-functions depend on only one basis-function.

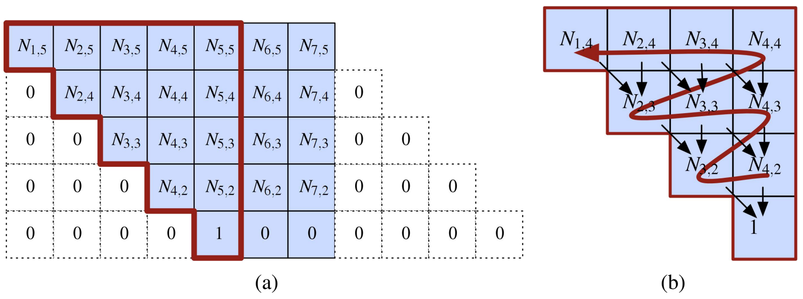


Figure 3.8: Evaluating the NURBS basis-functions.

Algorithm 7 simultaneously evaluates the basis-functions and their derivatives [AGM06].

Algorithm 7 EvaluateBasisFunctionsDivisionFree

```
Require: $i, t \in [t_i, t_{i+1}), p$.
Ensure: $N[0..p], D[0..p]$
1: $N[0] \leftarrow 1$
2: for $j = 1$ to $p$ do
3:    $D[j] \leftarrow p(a_i^j N[j-1])$
4:    $N[j] \leftarrow (a_i^j t + c_i^j) N[j-1]$
5:    for $k = j-1$ down to 1 do
6:       $D[k] \leftarrow p(a_{i-(j-k)}^j N[k] - b_{i-(j-k)}^j N[k+1])$
7:       $N[k] \leftarrow (a_{i-(j-k)}^j t + c_{i-(j-k)}^j) N[k]$
           $+(b_{i-(j-k)}^j t + d_{i-(j-k)}^j) N[k+1]$
8:    end for
9:    $D[0] \leftarrow p(-a_{i-j}^j N[j])$
10:   $N[0] \leftarrow (a_{i-j}^j t + c_{i-j}^j) N[j]$
11: end for
```

3.5.4 De Boor's Algorithm

De Boor's algorithm takes a very different approach in evaluating the points on a NURBS-curve or surface. Instead of first computing the basis-functions, and using them in a second stage by multiplying them with the control-points, it operates directly on the control-points.

Geometrically, it is identical to repeatedly inserting knots (Section 2.5) until the knot has a multiplicity p. The curve-point is then equal to the control-point (after refinement) interpolated by the curve at the knot. However, instead of computing each time all new control-points of the refinement, the algorithm limits itself to a subset of the refinement.

Algorithm 8 shows the *De Boor's Algorithm* for evaluating a NURBS-curve at t.

Algorithm 8 De Boor's Algorithm

```
Require: $t \in [t_i, t_{i+1})$, $p$.
Ensure: $\vec{C}(t)$
 1: if $t \neq t_i$ then
 2:     $s \leftarrow 0$
 3: else
 4:     $s \leftarrow$ multiplicity$(t_i)$
 5: end if
 6: $h \leftarrow p - s$
 7: for $j = 1$ to $h$ do
 8:     for $k = i - p + j$ to $k - s$ do
 9:         $\alpha_k^j \leftarrow \frac{t - t_k}{t_{k+p-j+1} - t_k}$
10:         $\vec{P}_k^j \leftarrow (1 - \alpha_k^j)\vec{P}_{k-1}^{j-1} + \alpha_k^j \vec{P}_k^{j-1}$
11:     end for
12: end for
13: return $\vec{P}_{i-s}^{p-s}$
```

To extend the algorithm to a surface-evaluation, observe the following (from Equation 2.8):

$$\vec{S}^h(u,v) = \sum_{j=0}^{m-1} \left[\sum_{i=0}^{n-1} \vec{P}_{ij}^h N_i^p(u) \right] N_j^q(v).$$

For a fixed j this becomes:

$$\vec{q}_j^h(u) = \sum_{i=0}^{n-1} \vec{P}_{ij}^h N_i^p(u).$$

Therefore, evaluation of $\vec{S}^h(u,v)$ can be done by first applying De Boor's Algorithm q times on the iso-v-curves, and then applying it one more time to the curve defined by:

$$\vec{S}^h(u,v) = \sum_{j=0}^{m-1} \vec{q}_j^h(u) N_i^p(v).$$

This results in a numerically very stable algorithm for computing NURBS-surface points. However, it does require the temporary storage of $2q$ surface-points.

3.6 Overview

In summary, NURBS-surfaces can be ray traced directly using a modified version of the basic ray tracing algorithm by including the following steps:

- In a preprocessing step:

    1. Subdivide the surface to obtain better initial-guess values for the root-finder, and to reduce the number of intersection tests (Section 3.4).

2. Construct a bounding volume hierarchy containing these initial-guess values from the subdivision from the previous step. After the bounding volume hierarchy is generated, the subdivision can be discarded (Section 3.1.2).

- During ray tracing:

1. Traverse the bounding volume hierarchy using a packet-based BVH traversal scheme (Section 3.2).
2. When reaching a leaf-node, use root-finding to obtain the intersection-point (Section 3.3.1).
3. From all obtained intersection-points, take the closest w.r.t. the ray, and use that intersection-point for further shading computations, and as a spawning point for secondary rays.

3.7 Discussion

The direct ray tracing of NURBS-surfaces poses an exciting challenge. Ray tracing itself is already a very expensive algorithm, combining it with NURBS-surface support will make it only more complex. However, the many advantages over standard rasterization and tessellation-based ray tracing are worthwhile to investigate an efficient implementation. At first, the reduced pre-processing time is a huge improvement, making interactive prototyping a lot easier. Secondly, as a consequence of using direct ray tracing of NURBS-surfaces, the required data, including preprocessed data, will be very compact, making it possible to ray trace very geometrically complex surfaces. Thirdly, exact representation makes it possible to zoom-in while preserving the model's smoothness: never will a linear approximation become visible (as opposed to the rasterization-based method and the tessellation-based ray tracing method). Finally, the benefits of ray tracing alone are the accurate optical phenomena, such as shadows, reflections and refractions.

In order to improve the performance of the ray tracer, efficient algorithms need to be employed for finding the intersections. The Newton-Rhapson method proves to be a very good candidate for this, due to its quadratic convergence rate and low memory requirements. By subdividing the surface in a preprocessing step, not only the surface is better approximated by the corresponding bounding volume hierarchy, but the smaller sub-patches also provide better initial-guess values.

For evaluation of the surface-points and their partial-derivatives, multiple algorithms are available. While the power-basis form provides a very simple method for evaluation, the memory-bandwidth required for it is rather high. Also, its stability decreases for increasing degrees. Whereas the direct form is more memory-friendly, as it does not rely on any preprocessed data. However, it performs divisions, which are very costly. The division-free method on the other hand is relatively memory-friendly and does not perform divisions at all, and is supposedly the fastest yet available according to the literature [AGM06]. The last method, the de Boor's algorithm is definitely not memory-friendly, as it requires a huge amount of registers. However, the method is very robust. Clearly, each method has its own advantages and disadvantages. However, due to the low memory-requirements of the direct and division-free methods, they are worthwhile for investigation on GPU.

Chapter 4

NURBS Ray Tracing on the GPU

Recently, graphical processing units (GPUs), originally only meant for rasterization-based graphics rendering, are being “misused” for other, general-purpose computations. With the increasing programmability of commodity GPUs, these chips are capable of performing more than the specific graphics computations for which they were originally designed. They are now capable coprocessors, and their high speed makes them useful for a variety of applications [OLG$^+$07]. As of now, NVIDIA GPUs are being used to accelerate computations for *computational fluid dynamics*, *finance*, *physics*, *life sciences*, *signal processing*, and many other non-graphics areas.

However, due to the limited memory of GPUs and streaming architecture, some extra challenges need to be tackled, when trying to exploit the raw processing power of modern GPUs for ray tracing. This chapter will discuss recent developments in GPU Ray Tracing, and even more recent, NURBS Ray Tracing on the GPU.

4.1 GPU Ray Tracing

Due to the parallel nature of the ray tracing algorithm (Section 1.2), researchers have long been trying to exploit the huge performance of GPUs. The first steps were taken with *The Ray Engine* [CHH02]. The ray tracer used the programmable shader pipeline of the GPU to compute all ray-primitive intersections. While the GPU was able to quickly compute the intersections, streaming the data to the GPU quickly became the bottleneck. This bottleneck in turn was avoided by moving all computations to the GPU [PBMH05]. However, this implementation required multiple rendering passes to ray trace a scene. One pass computed the eye rays, while a second pass traverses a uniform grid, a third pass computes the intersections, and the fourth pass shades the result. This process is shown schematically in Figure 4.1. Although the memory transfers were heavily decreased, they still were not able to outperform CPU-based ray tracers. Due to limited shader size and no branching support, many CPU-controlled rendering passes were necessary to traverse, intersect and shade the rays.

4.1.1 GPU Scene Traversal

kd-Tree Traversal

Also the implementation of kd-tree traversal has been researched. Since a stack is rather difficult to implement on a GPU, several methods have been devised to overcome this problem. [FS05] presented two methods for stackless traversal of a kd-tree on GPU, namely *kd-restart* and *kd-backtrack* [FS05]. Although their methods are better suited for the GPU, the high number of redundant traversal steps leads to relative low performance. By adding a short stack, [HSHH07] improved the kd-restart

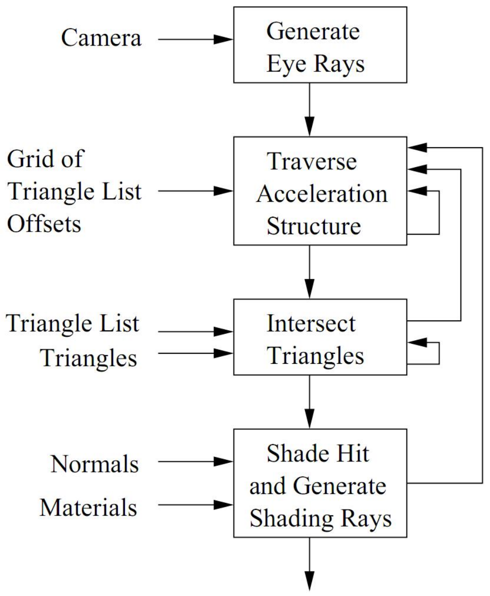

Camera
Generate Eye Rays
Grid of Triangle List Offsets
Traverse Acceleration Structure
Triangle List
Triangles
Intersect Triangles
Normals
Materials
Shade Hit and Generate Shading Rays

Figure 4.1: Multiple stages in the ray tracer of [PBMH05].

method [HSHH07]. They achieved a performance of 15.2M rays/s for the *Conference* scene (appr. 280k triangles). [PGSS07] presented a CUDA-based stackless packet traversal algorithm by using *ropes*, pointers to adjacent nodes [PGSS07]. Their traversal algorithm does not need to be restarted and is able to instantly continue the traversal starting from the corresponding node, resulting in a performance of 16.7M rays/s. However, their implementation requires six ropes for each node, increasing the memory requirements by a factor of 4, limiting the support to only medium-sized scenes. Additionally, even though optimized for the GPU architecture, it can still not utilize the full power of modern GPUs.

4.1.2 Stackless BVH Traversal

Although the kd-tree allows for fast scene traversal, the memory requirements are relatively high. Since the GPU has limited memory, the size of supported scenes for kd-trees is limited. However, BVHs generally require only 25% to 33% of memory compared to kd-trees, and compared to the stackless kd-tree traversal even 10% [PGSS07].

[TS05] presented a stackless traversal algorithm for the BVH which allows for efficient GPU implementations [TS05]. By storing the BVH tree in depth-first order, the tree is traversed in fixed-order. Therefore, no explicit reference needs to be kept for child-nodes. However, a ray could miss a node, and traversal needs to continue starting from another part of the tree. By including *skip-pointers* in the nodes, traversal can continue instantly from these other nodes (Figure 4.2). Although they outperformed both regular grids and the kd-restart and kd-backtrack [FS05] variants for kd-trees, their method is not as fast as the stackless kd-tree traversal [PGSS07].

4.1.3 Shared-Stack BVH Traversal

Although the BVH implementation of [TS05] does not depend on a stack, it uses a fixed-order traversal. The disadvantage of this is that many nodes, as well as primitives, will be tested for

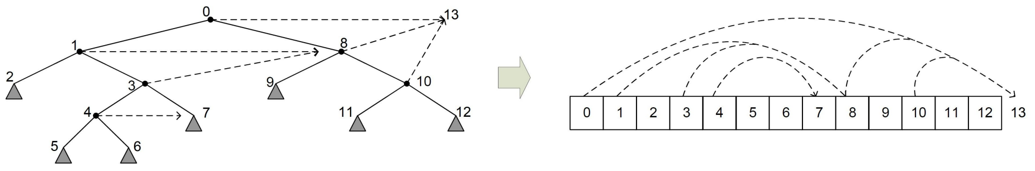


Figure 4.2: Stackless BVH traversal encoding example.

intersections even if they fall behind other primitives that will be traversed later. This leads to many redundant tests. [GPSS07] use an ordered, view dependent traversal, thus heavily improving performance for most scenes. However, since the traversal is ordered, a stack needs to be maintained. By tracing the rays in packets, the cost for this stack can be reduced, by amortizing it over the whole packet. By targeting their ray tracer at CUDA-enabled GPUs, several features can be exploited to implement this *shared stack*. The shared memory space of the GPU's multiprocessors is used to keep this stack. Access to this data area is as fast as accessing the local registers on the chip.

Their algorithm begins by traversing the hierarchy with the packet. Only one node is processed at a time. If the node is a leaf, the rays simply compute the nearest intersection with the geometry in it. If the processed node is not a leaf, the two children are loaded. The rays then intersect both children, to determine the traversal order based on the first intersected child. If the front-most child happens to lie behind an already intersected primitive, the child is considered as not being intersected. The next node is then collectively selected as the child-node which is front-most for the gross part of the rays. If any ray intersects the other child first, that child is "scheduled" by pushing it on the stack. If both children were missed, or if a leaf was processed, the next node is popped from the stack and its children are traversed. The algorithm terminates, if the stack is empty.

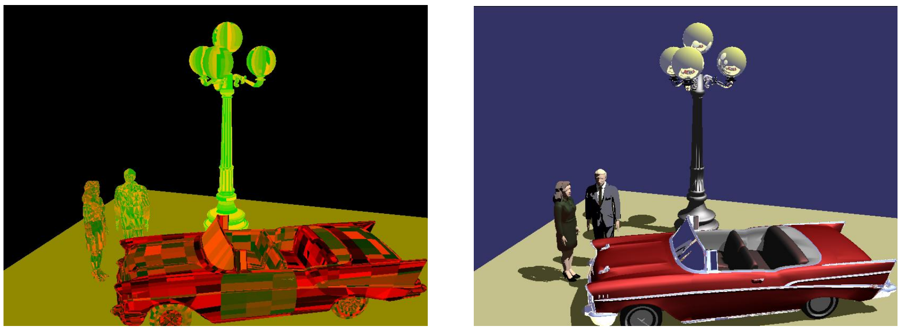
The results show that the BVH-based GPU ray tracer is competing with the stackless kd-tree ray tracer [PGSS07] with a performance of 16M rays/s for the Conference scene. Using a preprocessing step, a special ray/triangle intersection test can be performed, which results in a performance increase of 20%. Using this optimization, their ray tracer even reaches 19M rays/s. Although kd-trees have long been regarded as the preferred acceleration datastructure for ray tracing, their novel BVH traversal algorithm is easier to implement and uses less live registers, resulting in a higher utilization (63% compared to 33% for [PGSS07]). Using their implementation, they were even able to ray trace the 12.7 million triangle *POWER PLANT* scene at 1024×1024 image resolution with 3 fps, including shading and shadows. The BVH required only 230 MB, and therefore fits easily in GPU memory.

4.2 Hybrid Ray Tracing

Standard ray tracing uses an acceleration datastructure to reduce the complexity for a ray-scene intersection test from $O(n)$ to $O(\log n)$, where n is number of primitives. While this datastructure reduces the number of intersection tests dramatically, still a significant amount of time is spent for the traversal of this datastructure.

Another approach is *hybrid ray tracing*, where the GPU is used to rasterize the scene to obtain the intersections of the primary rays, after which this *ray casted* scene is further processed by the standard ray tracing algorithm (implemented on CPU or GPU), to include secondary effects such as reflections, refractions, shadows, etc. (Figure 4.3). Additionally, GPU shadow-maps can be used to avoid the intersection tests for the shadow-rays [BBDF05].

Interestingly, hybrid ray tracing is more beneficial to NURBS scenes than to triangle-based



(a) Rasterization phase of the hybrid algorithm. (b) Ray tracing phase of the hybrid algorithm.

Figure 4.3: Hybrid Ray Tracing phases.

scenes. Since the NURBS intersection test is extremely expensive compared to simple triangle intersections, every unnecessary intersection test avoided is worth much more compared to classical triangle tests. The following subsections discuss some hybrid techniques which accelerate the primary ray intersection stage of a NURBS ray tracer.

4.2.1 Extended Graphics Pipeline

[PSS+06] propose a conceptual extension of the standard triangle-based graphics pipeline by an additional intersection stage. They first generate a very rough approximation of the surface by subdividing the NURBS-surface patch. The convex hulls of this subdivision are then rendered using standard rasterization. In the fragment shader they include an intersection stage to compute the exact intersection values between the rays and the surface, after which the fragments are shaded [PSS+06]. Figure 4.4 shows a schematic overview of the extended graphics pipeline.

In order to reduce the number of ray/NURBS intersection tests, they employ some kind of *early ray termination*, similar to the early-z test. Their method requires the primitives to be sorted front-to-back, before being uploaded to the GPU. Before computing the ray/NURBS intersection, the depth value of the convex hull coming from the rasterizer is compared against the current value in the depth-buffer. If the convex hull lies behind the already computed pixel, the corresponding surface will also lie behind it, therefore the fragment/ray will be discarded and a costly intersection test can be skipped. Otherwise, the exact depth is obtained from the intersection, and the depth-buffer is updated accordingly.

Usually, the early-z test can be employed for planar primitives (such as triangles etc.), which heavily increases performance. This test compares the depth value coming from the rasterizer with the value in the depth-buffer, so that the entire fragment shader stage can be skipped if this fragment is behind another. However, the exact intersection for the NURBS-surface is determined during the fragment shader stage, consequently, the precise depth value can only be obtained using the fragment shader. Therefore, the fast early-z test cannot be employed in this case. However, it is implemented manually in the fragment shader to prevent the costly intersection tests. Test results show that this manual implementation does not decrease performance that much compared to a hardware early-z implementation [PSS+06]. Therefore, the benefit for the early ray termination is still very high.

To obtain the intersection between the ray and the surface, Newton's Iteration root finding method (Section 3.3.1) is used in combination with de Boor's method (Section 3.5.4). Three different techniques are used to obtain an initial guess for the (u,v)-values.

The first technique follows the approach of [MCFS00], by using multiple bounding volumes

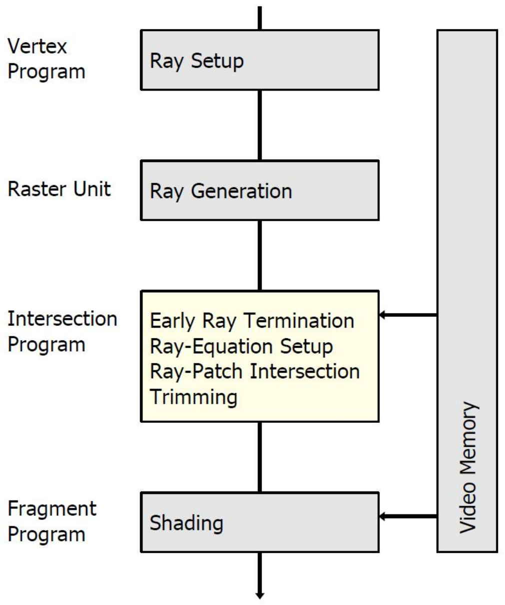


Figure 4.4: Overview of the algorithm stages and their relation to stages in the extended graphics pipeline.

per surface-patch. However, instead of using axis-aligned bounding boxes (AABBs), the bounding volume is the convex hull of a sub-patch itself and contains as initial-guess value the midpoint in the parameter range of the convex hull. An additional advantage compared to the AABBs, is that the smaller convex hulls approximate the surface better, so less rays will miss the surface.

For the second technique, called *uv-texturing*, the patch is further subdivided as in the previous method. But instead of using the midpoint as initial-guess, the parameter values correspond to the vertices of the convex hull, which are interpolated to obtain a closer initial guess. This method can be further optimized into a view-dependent method (technique three), by using the vertex shader to compute the intersection of the ray and the vertex to obtain exact (u, v)-values for the vertices, which again will be interpolated afterwards, resulting in an even closer initial guess. The test results show that the number of iterations is reduced when using the view-dependent variant. However, they do not provide a comparison with the view-independent variant with respect to speed.

Figure 4.5 and Figure 4.6 compare the number of required iterations for the intersection finder to converge. As can be seen, the uv-texturing method converges very fast, whereas the midpoint method requires several additional iterations. Figure 4.7 compares the view-independent method with the view-dependent method. It can be seen, that the view-dependent method requires less iterations in order to find the intersection.

Furthermore, using an adaptive subdivision, a coarser mesh can be generated globally, while locally, in very curved areas, the mesh can be subdivided a bit further. This will automatically determine the correct subdivision depth, which results in a maximum performance for the rasterizer. Additionally, since curved areas are subdivided further, less iterations are required, increasing the performance even more.

Their implementation is able to rasterize the bi-cubic NURBS-teapot consisting of 32 patches each uniformly subdivided 4×4 times (12698 triangles for the convex hulls), with a frame rate of 36 frames per second at a resolution of 1280×1024.

While they are able to quickly rasterize NURBS-surfaces, their method does not include secondary effects, such as shadows, reflections, refractions, etc. Furthermore, they are not able to fully exploit the coherence among rays, since the rays are processed independently in parallel using the fragment shaders. Using CUDA, this coherence could be exploited better, and a full-fledged ray tracer could be built. However, this is not the goal of the authors, since they proposed a conceptual extension to the standard graphics pipeline, to support the NURBS surface as a primitive.

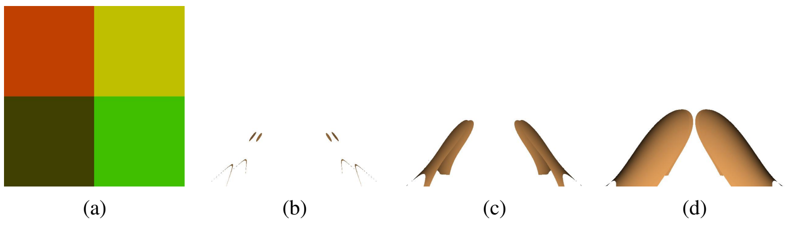


Figure 4.5: uv-texturing using midpoint of parameter range (uniform subdivision: 2×2). Figure 4.5a shows the corresponding uv-texture. Figure 4.5b to Figure 4.5d show the result for up to two, three, and four iterations, respectively.

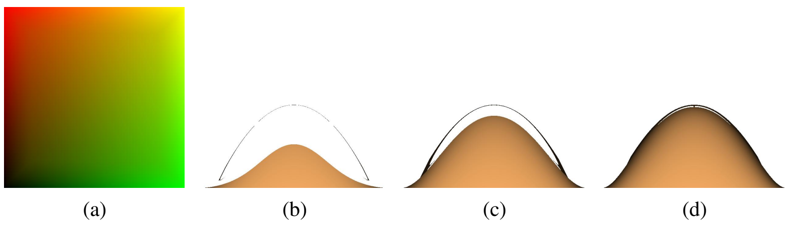


Figure 4.6: uv-texturing using control point mapping (uniform subdivision: 2×2). Figure 4.6a shows the corresponding uv-texture. Figure 4.6b to Figure 4.6d show the result for up to two, three, and four iterations, respectively.

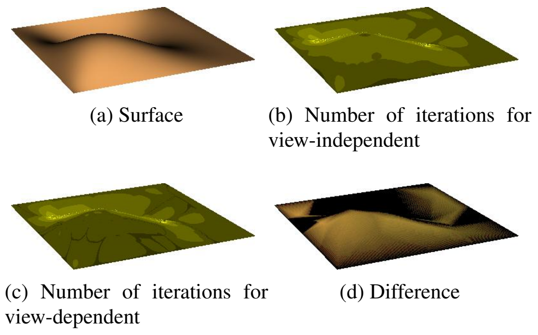

(a) Surface
(b) Number of iterations for view-independent
(c) Number of iterations for view-dependent
(d) Difference

Figure 4.7: Figure 4.7a, Figure 4.7b view-independent uv-texturing vs. Figure 4.7a, Figure 4.7c view-dependent uv-texturing (max. iterations: 10, subdivision: uniform, 4×4). Figure 4.7d is the difference between the initial values of both methods.

4.2.2 GPU/CPU Hybrid

As in the method of [PSS$^+$06] in Section 4.2.1, [ABS08] use a hybrid approach by rasterizing a coarse mesh using the GPU and then computing the exact intersections in a second stage. However, whereas [PSS$^+$06] use the convex hull of a subdivided NURBS-patch as input for the rasterization phase, the subdivision itself is used instead to obtain a more tight approximation of the surface. Also, the intersection stage is performed on the CPU as opposed to the GPU requiring costly GPU-CPU memory transfers. During the intersection stage, they employ the fast NURBS evaluation algorithm (Section 3.5.3), together with the Newton Iteration to obtain the exact intersection [AGM06].

Two variants are employed in their method. The first one, called *id-processing*, uses only the id of the triangle/surface for further processing. However, since no (u,v) values are available, an additional function call is required to obtain those values. The second method, called *uv-processing*, uses the vertices' (u,v) values. These values are interpolated in turn by the rasterizer to obtain an initial guess for the Newton Iteration. However, since an additional buffer is needed to hold the interpolated (u,v) values, more memory needs to be transferred from GPU-memory to CPU-memory. Comparing this variant to the view-dependent *uv-texturing*-approach [PSS$^+$06], there are several advantages. Since the vertices of the subdivided mesh lie exactly on the surface, the precise (u,v) values are already known, whereas [PSS$^+$06] require additional intersection tests using the vertex shader to obtain these values.

Although using the subdivision directly fits the NURBS-surface better than using the convex hull as in [PSS$^+$06], their method suffers from some artifacts. Rays that intersect the surface, but miss the tessellated silhouette, will not appear in the rasterized output, whereas standard ray tracing does include them (Figure 4.8a). In order to counter this artifact, they include a test in the intersection stage, to determine if a ray/pixel needs to traverse the entire scene, or it can depend on the hit reported by the rasterizer. Due to the mentioned artifact, every background pixel in the rasterization output therefore needs to be ray traced using a packet-based BVH ray tracer, in order to produce a correct result. While many background-pixels will actually be rays that do not intersect the surface, this check will incur some overhead. However, this overhead is relatively low, since no (or very few) NURBS intersection tests needs to be performed.

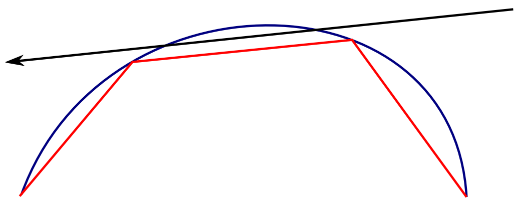
Another artifact that may occur is when the rasterized output contains a hit where the ray traced output does not, or hits another surface (Figure 4.8b). Therefore, an additional check, after the intersection test is used, to determine if the intersection test succeeded. If not, the standard ray tracing algorithm is used as well.

Although their artifact-handling fixes the majority of incorrect rasterized silhouettes, it is still incapable of correctly ray tracing (self-)intersecting surfaces (Figure 4.9a). Luckily, this artifact is avoided when tessellating not too coarsely. However, not mentioned by their paper, since the tessellation is an approximation, the surface at the silhouette can be missed by the rasterizer. Therefore, if another surface is behind the missed surface, additional artifacts will occur, resulting in a tessellated-looking silhouette (Figure 4.9b).

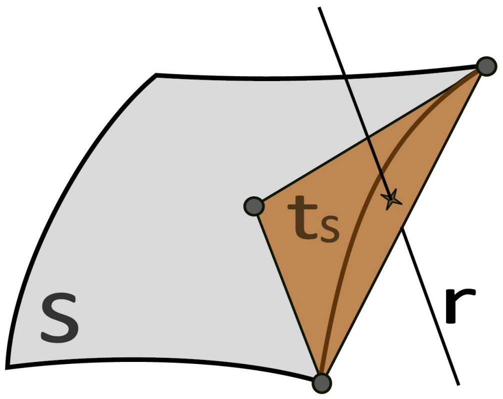
The test results show that using the hybrid approach, 20-70% of the costly NURBS-intersection tests can be avoided. Normally a ray would intersect multiple surfaces, but using the z-buffer the number of surfaces is reduced to only one (except for the artifact-handling). Interestingly, although uv-processing starts with a better initial guess, the cost for obtaining these guesses is very high since an additional data buffer needs to be transferred. Therefore, their id-processing variant outperforms the uv-processing variant.

Their implementation is able to ray cast the bi-cubic NURBS-teapot consisting of 32 patches subdivided over 12698 triangles, with a frame rate of 8.0 frames per second at a resolution of 512×512. Using a dual 3.0GHz quad-core processor, they reach 25.5 frames per second for uv-processing, and 41.7 frames per second for id-processing.

Although only primary rays are traced, this implementation is easily extended to a full-fledged



(a) A primary ray missing the tessellation, but hitting the surface.



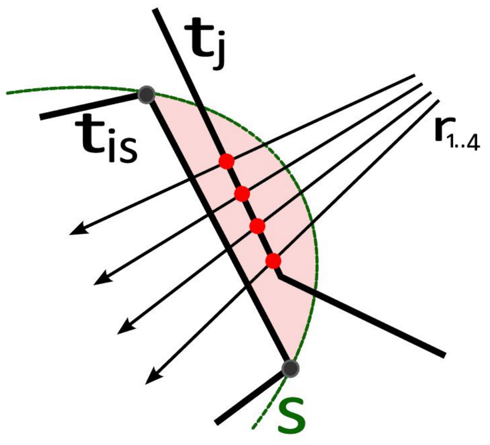
(b) A primary ray r hitting a triangle t_s from a tessellated surface while missing the original (parent) surface s.

Figure 4.8: Artifacts which are handled correctly.

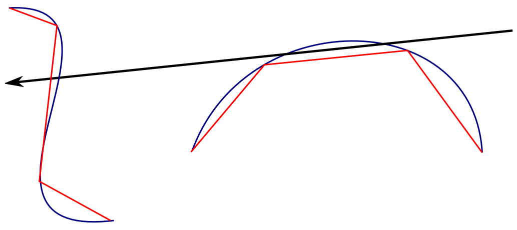
ray tracer, by simply shooting more rays in the intersection stage. However, these secondary rays will not be traced as efficiently as the primary rays.

4.2.3 Comparison

For the teapot scene, [PSS+06] reach 47M rays/s, whereas [ABS08] reach 11M rays/s (id-processing, using 8 threads). However, this comparison is not fair, since different hardware is used in both implementations as well as a different approach. Furthermore, the scenes are very different actually. The subdivision is performed differently, and another viewpoint is chosen for the camera. Nevertheless, the features can be compared. Whereas [PSS+06] rasterize the convex hull in order to produce an artifact-free result, [ABS08] rasterize the tessellation of the subdivision, introducing some possible artifacts. However, a convex hull requires more primitives to process, slowing down the rasterization stage, on the other hand, this is not the bottleneck. Using a tessellation also approximates the surface better, but more important, it provides more accurate initial-guess values. Using the convex hull, the best initial-guess values one can get are obtained by using view-dependent uv-texturing. This requires additional intersection tests during the vertex shader stage, slowing down the ray tracing process. The tessellated method implicitly uses view-dependent uv-texturing, since the vertices of the triangles lie exactly on the surface. The tessellation method also guarantees only one intersection test, except for some cases when an artifact is detected, where additional standard ray tracing is required. Using the convex hull, an intersection could be in front of the current front-most intersection according to the depth-buffer. Therefore, several intersection tests can be needed to obtain the correct intersected surface. However, this consequence can be minimized by first sorting the primitives front-to-back. Furthermore, the method by [PSS+06] is fully implemented using the programmable



(a) A packet of rays $r_{1..4}$ intersecting the wrong surface t_j. The altered shape t_{is} of the surface s causes objects in the shaded area to be drawn instead of culled and vice versa (if rays origin in the opposite direction)



(b) A primary ray hitting the front surface, but missing the tessellation. In the back the ray does hit the tessellation, however, this will result in a wrong hit.

Figure 4.9: Artifacts which cannot be handled correctly.

shader pipeline. Therefore, due to the parallel nature of the fragment processing units, coherence of initial values for the Newton Iteration between neighboring pixels/rays is difficult to exploit, only the caches can be used to exploit localness in memory fetches. The method of [ABS08], traces rays in packets, and therefore can profit from the coherence. Table 4.1 shows a comparison of the features for both hybrid methods.

4.3 Discussion

For ray tracing NURBS-surfaces, the GPU BVH ray tracer in Section 4.1.3 can be employed. It allows an efficient traversal of the bounding volume hierarchy (derived from the subdivision of the NURBS-surface) by including a shared-stack per ray-packet. Together with the Ray-Patch intersection routine (Section 3.3.1), and the evaluation schemes in Section 3.5, an efficient implementation should be possible.

Hybrid Ray Tracing is an interesting method for accelerating the primary intersection stage, which avoids the entire traversal stage. Although ray tracing will eventually outperform rasterization as the complexity increases, this is usually not the case for ray tracing NURBS-surfaces, as the

Feature	[PSS+06]	[ABS08]
Artifact-free	Yes	No (rare)
Initial Mesh	Convex Hull	Tessellation
Intersection Tests/Ray	≥ 1	1 (non-missed intersections)
Early ray termination	Yes (manually)	Yes
uv-Texturing	Yes	Yes
View-dependent	Yes (vertex shader)	Yes
GPU/CPU Intersection stage	GPU	CPU
coherence exploitation	Little (through caches)	Yes (through packets)

Table 4.1: Comparison of features for both hybrid methods.

rasterized mesh is only a rough approximation of the NURBS-surface, therefore, the complexity can be kept modest, resulting in a fast rasterization. But more important, since many triangles are being culled by the rasterization pipeline, through frustum culling, early-z culling, viewport culling, etc., the average number of intersection tests is reduced to only a few per ray. Most of the time requiring just only one intersection test, as most of the other triangles are culled away.

Two different approaches are employed for hybrid ray tracing. [PSS+06] implemented the entire ray tracer using vertex and fragment shaders, whereas [ABS08] used the CPU for computing the exact intersections. For the former method, the intersection tests take place during rasterization by using the fragment shader for computing the intersections. Since all fragments are processed independently by the GPU, little attention is paid to coherence among the rays, except for the caches. The main bottleneck in the latter is the expensive GPU-CPU memory transfer, especially for the *u/v processing* variant, since it requires the transfer of an additional buffer for the (u,v) values. Furthermore, the resulting (u,v) buffers use *RGBA* data, therefore adding some amount of overhead. However, recent GPUs support integer textures, which will heavily decrease the required bandwidth. As mentioned in their paper [ABS08], implementing the intersection stage on GPU, all memory transfers can be avoided. In combination with the packet-based GPU BVH traversal, a performance gain can be expected.

Finally, a second major difference between the two approaches lies in the rasterized mesh. Whereas [PSS+06] use the convex hull to ensure an artifact-free result, [ABS08] rasterize the refinement itself. Although some artifacts may occur by using the tessellation, the z-buffer guarantees that at most one intersection test is required for every pixel, except for surface boundaries, where a ray can miss the surface, as opposed to [PSS+06], where several intersection tests may be needed per pixel. Furthermore, almost all artifacts are correctly handled in the second intersection stage. The artifacts that can not be handled result from (self-)intersecting surfaces, which should not happen in reality. Additionally, since a tessellation is used, less primitives are generated, but more important, the tessellation results in a view-dependent *uv-texturing* approach for free, whereas [PSS+06] require additional intersection computations in the vertex shaders. Using view-dependent uv-texturing is currently the most efficient method, which converges very fast to the root.

Therefore, a NURBS ray tracing system should be possible to be fully implemented on GPU by using a hybrid approach for casting primary rays and falling back on GPU BVH ray tracing for missed intersections and secondary rays, such as shadows, reflections, refractions, etc.

Chapter 5

CUDA

With the introduction of the G80-architecture, NVIDIA has made a big step forward towards general-purpose computing using GPU hardware. The Compute Unified Device Architecture, or CUDA, is a general-purpose parallel computing hardware/software architecture. By choosing the well-known C-language as the basis for their architecture, any programmer familiar with this language, can immediately start developing parallel software. In addition, their programming language, C for CUDA, has some of the C++ features, including polymorphism, operator overloading, and function templates.

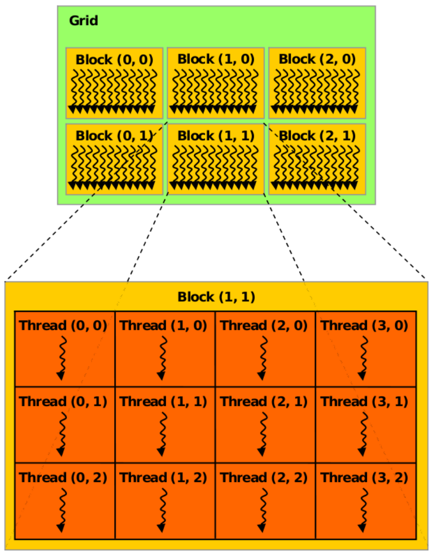
This chapter will briefly discuss the CUDA architecture, how it can be used to build highly parallel software, and which obstackles need to be faced.

5.1 Massive Multi-threading Architecture

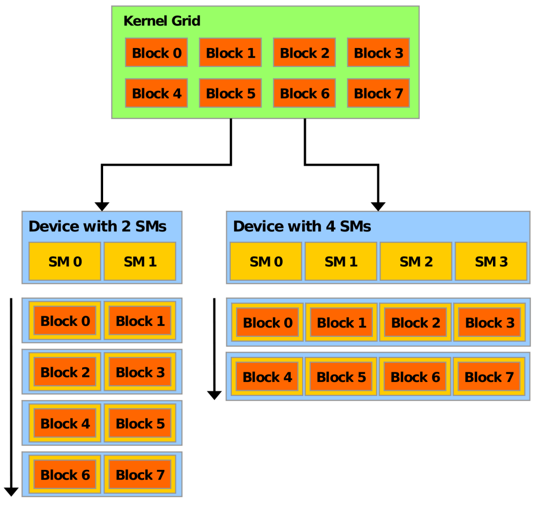
CUDA GPUs contain a number of multi-threaded *streaming multiprocessors* (SMs), each capable of executing SIMD-groups of threads concurrently. Each SM contains 8 cores, together with a scheduler which creates, manages, and executes concurrent threads in hardware with zero scheduling overhead.

In contrast to SIMD software for CPUs, in which 4 instructions are executed in parallel, CUDA applications launch a massive number of lightweight threads which are executed in parallel in a streaming fashion. All threads form a *grid*, which is divided into user-defined *blocks* (Figure 5.1a). Each block is assigned to a specific multiprocessor. Therefore, threads of the same block will always execute on the same multiprocessor. Groups of 32 threads, called a *warp*, are executed physically in parallel in SIMD-fashion. Additionally, each thread executes independently with its own instruction address and register state[1]. As long as all threads in a warp execute the same instruction, the warp achieves maximum *utilization*. Otherwise, it is serially executed in two (or more) parallel groups. Furthermore, to hide memory latencies, warps are time-sliced by a hardware scheduler with zero schedule-overhead. The thread blocks are enumerated and distributed to the available multiprocessors, which allows for a highly scalable system (Figure 5.1b). Finally, CUDA-GPUs supporting SLI, can be linked together to increase the performance even further. However, each GPU needs to be controlled using a separate thread, and data is not shared between GPUs, therefore, all data need to be uploaded twice. Nevertheless, by using SLI, the performance can be doubled in theory.

[1]NVIDIA refers to this as SIMT, for single-instruction, multiple-thread.



(a) Hierarchical overview of CUDA threads.



(b) A device with more multiprocessors will automatically execute a kernel grid in less time than a device with fewer multiprocessors.

Figure 5.1: Thread blocks and their mapping to the multiprocessors.

5.1.1 Predicated execution

Sometimes, a thread may decide not to execute a branch, while other threads belonging to the same warp do execute that branch. Such a warp will usually be split into as many thread-groups as required to enable a parallel execution of each group. However, when the branch contains only a small number of instructions, the compiler might decide to predicate the instructions. Threads which did not execute the branch will be deactivated, and thus skip the instruction.

5.2 Memory Hierarchy

There are five different types of memory available for use. The GPU's main memory, named *global-memory*, is very big, but the latency is very high. The global-memory is shared among all running threads, and even between kernel invocations. Each thread also has a private portion of this global-memory, called *local-memory*. Although it's optimized for efficient data transfers, it has the same latency as the global-memory, making it very slow in some situations. The *texture-memory*, is a special read-only portion of the global-memory which can be read using special *texture-fetch* functions. The advantage of using texture-memory, is that the texture-unit, which executes the fetches, caches its incoming data. Therefore, frequently accessed data is likely to appear in the cache, so a read from global-memory can be avoided. The latency to this cache is as fast as the access to a register. In addition, the texture-unit provides filtering functionality, to normalize values, interpolate values, etc. There is also a portion of global-memory, which is much alike the texture-memory: the constant-memory. It has some restrictions though, as being very small (only 64 kB), and has no filtering support. Finally, each multiprocessor has on-chip memory of 16 kB, called the shared-memory. As with the cache, the latency of this memory equals that of a register access. This memory is divided among the active blocks running on a multiprocessor. The threads in a block can all access the same shared data, however they cannot access the shared data of other blocks. The amount of shared-

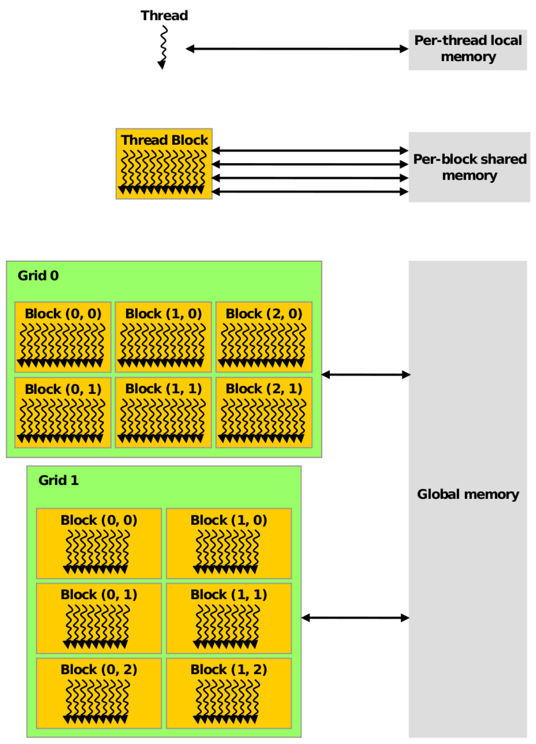


Figure 5.2: Memory Hierarchy.

memory a block is given, is determined by the total amount of shared-memory which is reserved by a kernel. Figure 5.2 illustrates the different types of memory.

5.2.1 Coalescing

Due to the wide memory bus, memory transactions always occur by fetching 32, 64, or 128 contiguous bytes simultaneously. Therefore, global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (either the first 16 threads, or the last 16) can be *coalesced* into a single memory transaction of 32, 64, or 128 bytes. In order to assist in coalescing, data is sometimes arranged as a structure of arrays (SOA), instead of an array of structures (AOS). In this way, the data for each member is clustered, and can be fetched using a single memory transaction.

5.2.2 Prefetching

The texture-unit also provides an implicit prefetching function. When fetching data from address X, the texture-unit will automatically prefetch the data on addresses X-1 and X+1. If the thread, or any other thread executed by the same multiprocessor, will read from a neighboring address, the data will already be available and can be fetched directly from the cache.

5.2.3 Bank conflicts

The data of the constant- and shared-memory are physically arranged into 16 memory-banks. As long as the threads in a half-warp do not read or write from the same memory-bank, there will be no conflicts. In case of a conflict, the memory-transactions will be performed serially for the number of distinct conflicts. However, a read from the same address (broadcast), will not result in a bank-conflict.

5.3 Launching a kernel

A kernel is defined in C for CUDA, which is an extension to the C-language. For example, the well-known *saxpy*-operation can be defined as follows:

Listing 5.1: Example CUDA-kernel: saxpy

```
__global__ void saxpy(float *s, float a, float *x, float *y)
{
    int id_x = (threadIdx.x + blockIdx.x * blockDim.x);
    int id_y = (threadIdx.y + blockIdx.y * blockDim.y);
    int id   = id_x + id_y * (blockDim.x * gridDim.x)

    s[idx] = a * x[idx] + y[idx];
}
```

This kernel first computes its unique identifier, based on its position in the grid. `threadIdx` is a vector containing the x and y location of the thread inside the block. `blockDim` gives the dimensions of every block. `blockIdx` represents the location of the block inside the grid. And `gridDim` gives the dimensions of the grid, in blocks.

After having computed the identifier, the `x` input element is read from global memory, multiplied with the `a`-factor, and added to the `y` input element, also from global memory. Finally, the result is written back to global memory. The code in Listing 5.2 launches a grid of 4×4 blocks, each containing 1024 threads.

Listing 5.2: Example of launching CUDA-threads

```
...memory allocations, and variables defined elsewhere
dim3 grid(4, 4);
dim3 block(32, 32);
saxpy<<<grid, block>>>(output_s, a, input_x, input_y);
```

5.4 Debugging

5.4.1 Emulation

The nvcc compiler driver provides an option to generate binary code which can be executed by the CPU, using the device emulation mode. Normally, it would not be possible to specify breakpoints when the application is being executed by the GPU. However, using this mode, CUDA-applications can be debugged, as if they were normal CPU-applications. Unfortunately, the emulation is not 100% accurate. Whereas the GPU can execute 32 threads physically in parallel, the emulation mode cannot. Therefore, additional synchronization points need to be set, in order to obtain the same results. Finally, the emulation mode runs very slowly. Nevertheless, to find bugs, the emulation mode is a very nice addition to the CUDA Toolkit.

Two other emulators which were available, are Barra [CDP] and GPUocelot [Dia09, DKK09]. Those emulators do not require a different compilation mode. Instead, they temporarily replace the CUDA-Runtime, in order to intercept all instructions. Their goal is to provide a 100% accurate , and much faster emulation. However, at the time of writing this thesis, the emulators were not yet capable of emulating applications designed for the CUDA 2.3 runtime. Nevertheless, an eye should be kept on these emulators.

5.4.2 Hardware breakpoints

When running Linux, breakpoints can be set to running CUDA-applications, using the cuda-gdb debugger (an extension to gdb). However, no X-Server may be running on the GPU which is currently being debugged, making it very difficult to visualize the output. Although it is possible to run the X-Server on another computer in the network, it is generally recommended to have a second GPU in the computer.

Unfortunately, the debugger did not work very well for the ray tracing system, probably because the generated code was very complex.

Part II

Development

Chapter 6

System overview

In this chapter, the *CUDA NURBS Ray Tracing System* (*CNRTS*) is presented: a GPU-based implementation of the recursive ray tracing algorithm capable of direct ray tracing of NURBS-surfaces. Besides primary-rays, the system is also able to cast shadow rays, reflection rays, and refraction rays up to any desired depth. Additionally, the system features an acceleration subsystem for speeding-up the casting of primary-rays. Finally, the system requires very little preprocessing, and the amount of used memory is kept to a minimum.

The system heavily depends on auxiliary datastructures. These datastructures are computed only once in a preprocessing step, and are then fed into the core of the system, which uses it for rendering. The next chapter will discuss the preprocessing phase. This chapter only provides a global overview of the system, the implementation details will be provided in Chapter 8.

Rendering Core The rendering of NURBS-surfaces is handled by the core of the CNRTS. It is composed of two subsystems, namely the *Primary-ray Accelerator* and the *Ray Tracer*. The first subsystem accelerates the *primary intersection stage* of the ray tracing system by limiting the number of ray/surface intersection tests to a single test[1]. Additionally, it accelerates the convergence of the root-finder by providing more accurate initial-guess values. *The Ray Tracer* subsystem then performs the classical recursive ray tracing algorithm, by tracing secondary-rays, including shadows, reflections and refractions. Figure 6.1 gives a high-level overview of the architecture of the core of the CNRTS. Both subsystems are executed entirely on the GPU with only a minimal amount of coordination by the CPU.

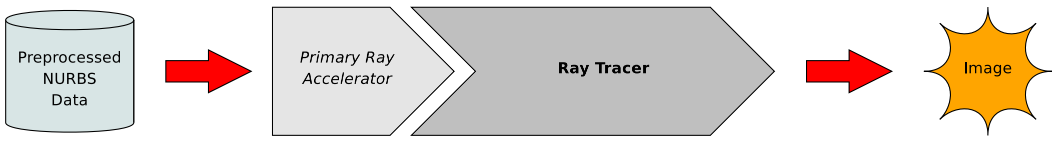


Figure 6.1: System overview of the rendering core of the CNRTS

6.1 Primary-ray Accelerator subsystem

A hybrid technique is employed to accelerate the tracing of primary-rays. The efficiency of the standard rasterization pipeline is exploited to limit the number of intersection tests and to obtain a

[1]As will be seen later, there are some cases in which this number can be slightly higher.

good starting point for the root-finder. Since the intention was to create an OS-independent system, OpenGL is chosen as the graphics API. However, the system is easily modified to use DirectX instead.

6.1.1 Rasterization

Based on the ideas by [ABS08, PSS⁺06, BBDF05], the Primary-ray Accelerator subsystem removes the necessity for the traversal of an acceleration data structure by first *rasterizing* a coarse tessellation of the NURBS-model. However, instead of just using the standard rendering pipeline (Section 1.1) output, a vertex-shader and fragment-shader are employed to generate a customized output.

The tessellation is obtained from a refinement of the NURBS-model using the method described in [MCFS00]. Before rasterization begins, two custom attributes are added to each vertex: a *patch-identifier* uniquely identifying the NURBS-surface and the span locations in the corresponding knot-vectors; and the *uv-parameter* value of the NURBS-surface corresponding to the face-corner. The tessellation is generated in a preprocessing step and is discussed in Section 7.1.

During rasterization, the rasterization-unit will interpolate the vertices' uv-parameter values providing a weighted average closely approximating the surface' uv-parameter corresponding to the pixel's true intersection-point[2]. Pixels containing a rasterized primitive, will provide a good starting point for finding the exact intersection-point: the patch-identifier identifies the NURBS-surface and spans[3], and the uv-parameter value provides a good initial guess-value for the root-finder to find the exact intersection-point. Pixels containing a background-value either do not contain a surface, or missed the surface due to too coarse tesselation. Figure 6.2 shows schematically the rasterization phase of the Primary-ray Accelerator.

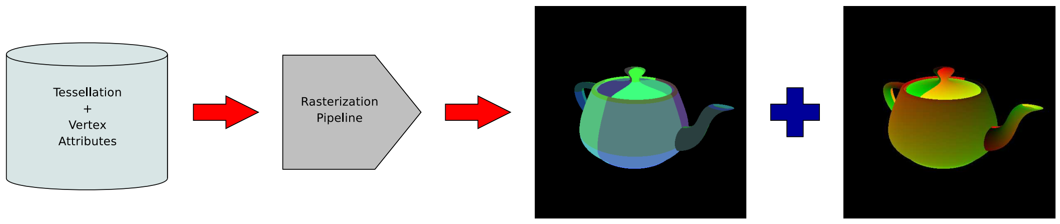


Figure 6.2: Schematic overview of the rasterization phase of the Primary-ray Accelerator. The output provides some cues for finding the exact intersection-point: the left rasterized image contains the patch-ids and the right rasterized image contains the uv-values encoded in red for u-parameter range and green for v-parameter range.

6.1.2 Traversal-less Ray Casting

As soon as the rasterization step completes, the actual ray tracing can begin. However, instead of immediately applying the entire recursive algorithm, only the primary rays are traced without spawning secondary rays, resulting in a *ray casted* image.

Normally, in order to find out which object is front-most w.r.t. the viewer, all the objects in the scene need to be traversed, or part of the scene when using an acceleration datastructure (see

[2]The patch-identifier however, does not require interpolation, since it's constant across the entire surface patch.

[3]There are some cases in which the rasterized output incorrectly identifies the surface. However, if the rootfinder happens to fail in finding an intersection-point, then the pixel can be "repaired" by the Ray Tracer, as will be seen next (Section 6.1.4 and Section 6.2). In the case the rootfinder reports an intersection, it cannot be repair, and the incorrect surface will appear in the result.

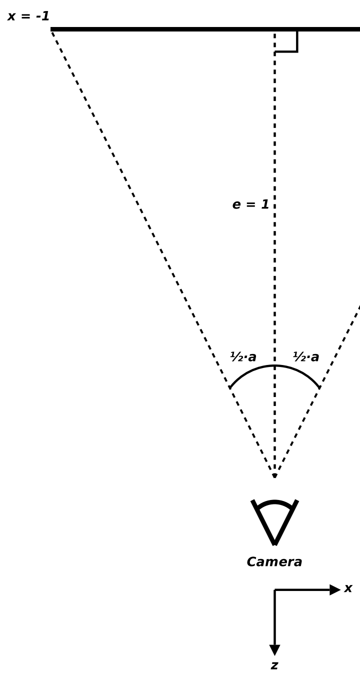


Figure 6.3: The ray's direction is derived from the viewing parameters. In here, *a* denotes the field-of-view. By setting the image-plane's distance *d* from the camera equal to 1, the position in the image plane is easily converted to world-coordinates.

Section 3.1). However, we actually already know nearly certain which object is front-most: the patch-identifier resulting from rasterization identifies the surface closest to the viewer along the ray being processed (see the left rasterized image in Figure 6.2). Thus, only one intersection test is required.

So all that's left to be done is computing the intersection based on the data output by the rasterizer. The shading computations are deferred to the ray tracer subsystem. The interpolated *uv*-parameters resulting from the rasterization are a perfect candidate for an *initial-guess value* for the root-finder (see the right rasterized image in Figure 6.2).

6.1.3 Primary-ray setup

In order to use the rasterization output as input for the ray caster, the pixel positions need to be mapped to primary-rays in such a way, that when the tessellation would be ray traced, the exact same output should be obtained.

Figure 6.3 shows how the camera is positioned w.r.t. the image-plane. The mapping is derived from the pixel position in the image plane, and the viewing parameters used for rasterization. Algorithm 9 shows the pseudo-algorithm to compute the ray from the pixel location and the viewing parameters.

Algorithm 9 Computation of a primary-ray

```
Require: pixel location $x$, $y$, image plane resolution $width$ and $height$, camera position $eye$, camera focal point $focus$, camera up-vector $up$, and field-of-view $a$.
Ensure: ray origin $O$, and ray direction $D$.
1: $O \leftarrow eye$
2: $D \leftarrow focus - eye$
3: $step_x \leftarrow \frac{\tan \frac{1}{2}a}{\frac{1}{2}width}$
4: $step_y \leftarrow \frac{\tan \frac{1}{2}a}{\frac{1}{2}height}$
5: $D \leftarrow D + (step_x \cdot (x - \frac{1}{2}width + \frac{1}{2})) \cdot (D \times up)$
6: $D \leftarrow D + (step_y \cdot (x - \frac{1}{2}height + \frac{1}{2})) \cdot up$
```

6.1.4 Artifacts

While the Primary-ray Accelerator enables fast ray casting for NURBS-surfaces, it does not guarantee that the resulting output will always be correct. There are three types of artifact which can appear, from which two of them can be fixed by the system.

The first type of artifact is related to the root-finder being skipped, due to a background-pixel in the rasterization output. There are two cases in which this will happen: either the ray does not intersect any surface-patch and vanishes into the background; or the ray does intersect a surface-patch, but due to a too coarse tessellation, it won't show up in the rasterization. While the former is correct, the latter produces an incorrect result, making the tessellation apparent in the result (Figure 6.4a). Since both cases produce a background-pixel, it is not possible to differentiate between them.

The second type of artifact which could occur happens when the root-finder diverges. Again, there are two cases in which this can happen: either the ray does not intersect any surface-patch, but the rasterization output reported a hit; or the ray intersects another surface-patch than the one reported in the rasterization output. Obviously the former is correct, but the latter will produce gaps appearing in the result as can be seen clearly in Figure 6.4b.

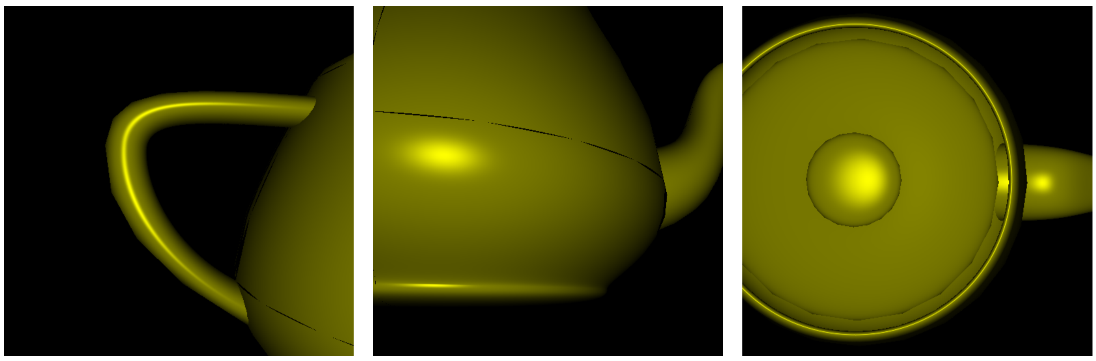
Both types of artifacts are easily handled by the system by forwarding background-pixels, and diverged pixels to the Ray Tracer subsystem in order to fix potential incorrect results. A disadvantage of this method is that correctly computed pixels will have to be computed twice.

A third possibility leading to an incorrect result is a too coarse tessellation which doesn't show up in the rasterization. Whereas this will be handled correctly for background-pixels, it will not be in the case of another surface lying behind the too coarse tessellated surface: this surface will be intersected by the ray, and thus an incorrect intersection-point will be returned. This appears clearly in the tesselation of the teapot (Figure 6.4c). While the former two types of artifacts are fixable by forwarding them to the Ray Tracing subsystem, this type of artifact is not. Since a valid intersection-point is returned by the root-finder, it is not possible to determine if it was the front-most. However, by using a not too coarse tessellation, these artifacts will occur rarely.

6.2 Ray Tracer subsystem

The heart of the rendering core is the *Ray Tracer* subsystem, which performs the full-blown recursive ray tracing algorithm. Apart from tracing secondary-rays, it also handles the artifacts described in Section 6.1.4 by retracing the primary-rays that either missed the tessellation (background-pixels) or diverged during the root-finding phase.

The *Ray Tracer* subsystem follows the methods described in [MCFS00]. A BVH is constructed from a refinement of the NURBS-model, in which each leaf-node represents a flat enough sub-patch



(a) Detectable corners appear due to a too coarse tessellation. (b) Gaps appear due to the tessellation. (c) Undetectable corners appear due to a too coarse tesselation.

Figure 6.4: Artifacts appear when using the Primary-ray Accelerator solely. In Figure 6.4a, background-pixels appear instead of the surface. By forwarding all background-pixels to the *Ray Tracer*, this artifact will disappear. Figure 6.4b shows some gaps, appearing in-between two surfaces. Figure 6.4c shows an artifact which cannot be detected using the algorithm, the only solution is to tessellate more fine.

of a NURBS-surface. The subdivision is generated in a preprocessing step which will be discussed in Section 7.1. The scene is traversed using a packet-based BVH-traversal scheme. The Newton-Rhapson method Section 3.3.1 is adopted to obtain the ray/NURBS-surface intersections.

The implementation details will follow in Chapter 8.

6.2.1 Kernel separation

The first problem arising in mapping the ray tracing algorithm to GPU is that CUDA does not have native support for recursive function calls. This problem could be overcome, by rewriting the algorithm to an iterative version and using the shared-memory of CUDA to implement a stack to push the spawned rays onto. While this may seem to be reasonable, it enforces us to write the entire algorithm as a single complex kernel. However, an increase in kernel complexity, generally leads to an increase in the number of used registers. This will reduce the occupancy of the kernel. And if no more registers are available, it will result in lots of slow local-memory transfers. Additionally, it is less likely the CUDA-compiler will do a good job in optimizing a complex kernel. Finally, most CUDA GPUs have a limit on the maximum execution time of a kernel. Therefore, merging all traversal steps, will increase the kernel's execution time, and eventually will result in program crashes.

Instead I have put the different ray tracing-steps into multiple CUDA-kernels. A single kernel for tracing primary/secondary rays, another kernel for tracing shadow-rays and yet another kernel for applying shading. The kernels propagate information to each other through the global-memory. A per-pixel ray-stack is used on which reflection and refraction rays are pushed to, and popped from when tracing secondary rays (Section 8.1.1). To store the intermediate intersection data required by the shader, a hitdata-buffer is used, containing the intersection-point, the surface-normal, the material-id, etc. (Section 8.1.2). A thin layer of CPU-code is used to coordinate the algorithm, by launching the different CUDA-kernels (Algorithm 6.1). Figure 6.5 schematically shows this separation.

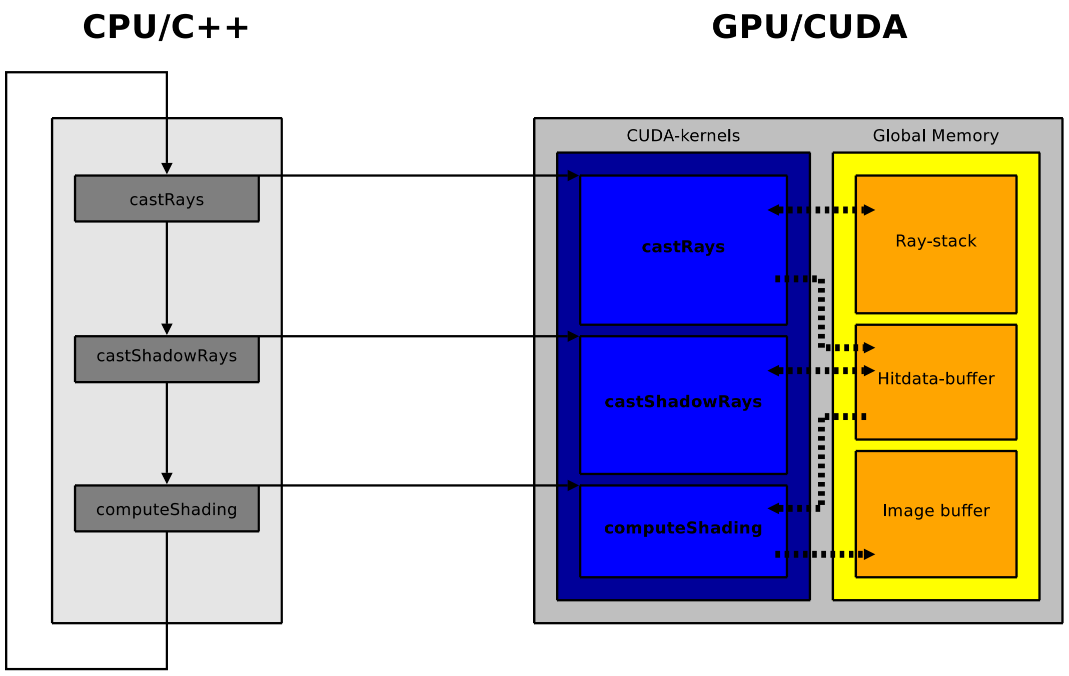


Figure 6.5: The ray tracing algorithm separated into different CUDA-kernels. The CPU part coordinates the algorithm by launching the kernels successively. The inter-kernel data propagation through global-memory is visualized by dashed lines, whereas the arrows indicate the flow/dependencies.

6.2.2 Tiling

Another problem that occurred during implementation was the amount of required memory for the temporary buffers, i.e. the ray-stack and hitdata-buffer. For a maximum recursion-depth of 3, at a resolution of 1024×1024, this results in 128 MB to be allocated for the ray-stack, and yet another 60 MB for the hitdata-buffer, totalling 188 MB (the derivation of these numbers can be found in Section 8.1.1 and Section 8.1.2). When comparing this number to other types of allocation (such as NURBS-data, BVH-data, image-buffers, etc.), it seems to be a rather high percentage of the total allocated memory. Although modern GPUs are capable of allocating such an amount of memory, these temporary buffers should not dominate the total amount of available memory, as it should actually be reserved for read-only data such as NURBS-data, BVH-data, textures, etc. What's even worse, is that the buffers could even prove a potential bottleneck: since the amount of memory required for the buffers is proportional to the image-resolution, increasing this resolution eventually will fail the allocation of them[4].

The solution to this problem is to divide the image-space in evenly-sized rectangular pixel-blocks, called *tiles*. Instead of applying the ray tracing algorithm onto the entire image, it is now applied successively to each tile. Each tile is given an attribute `tileIdx`[5], a two-dimensional index specifying the block position in image-space (analogous to the `blockIdx` and `threadIdx` parameters for position in grid and block, respectively). Based on the `tileIdx`, together with the `threadIdx`, the corresponding pixel location is computed, and accumulated with the color value from the shader.

[4]A full-hd resolution of 1920×1080 with a maximum recursion depth of 3 would require 372 MB, solely for the temporary buffers.

[5]In CUDA-terms, this would have been `gridIdx`.

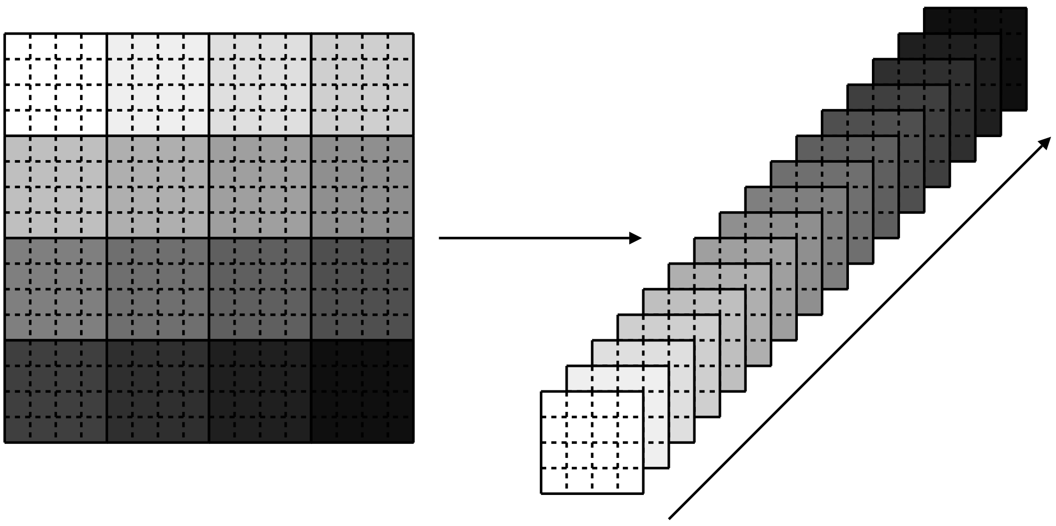


Figure 6.6: Separation of image-space into *tiles*: instead of applying the ray tracing algorithm onto the entire image, it is now applied successively to each tile. The tiles are visualized using different shades of grey, the pixels are separated by dashed lines.

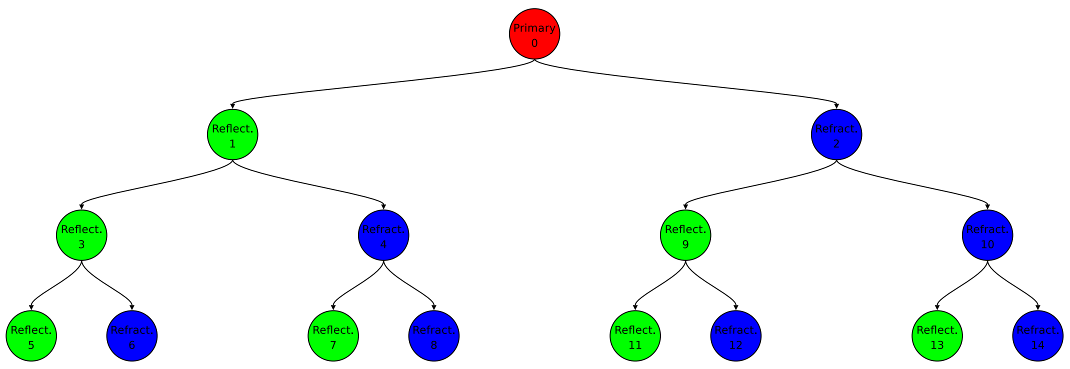
Figure 6.6 schematically shows this separation into tiles.

The main advantage of this method, is that it does not depend on the image-resolution anymore. Ray tracing an image with a maximum recursion-depth of 3 and a tile-size of 256×256 will always result in a temporary-buffer size of 11.75 MB, which is only a small fraction of the available GPU memory compared to the 188 MB when not using tiles.

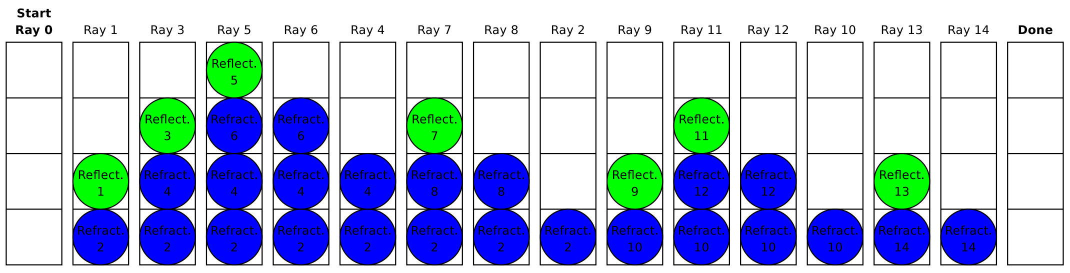
6.2.3 Indirect recursion

Figure 6.5 suggests an iterative rewrite of the recursive ray tracing algorithm. However, for the sake of simplicity, and since the CPU can't possibly become a bottleneck in this system, I've chosen to leave it recursive. In this way, the structure of the classical recursive ray tracing algorithm can be found literally in the CPU-part of the system.

Listing Algorithm 6.1 shows the CPU-code which coordinates the ray tracing algorithm. The `raytrace`-function is called and given a `tileIdx` which represents the current tile being processed. It then launches the `castRays` CUDA-kernel, which performs the primary or secondary ray-traversal step (including the root-finding and spawning of reflection- and refraction-rays). It then loops over all lights and performs the casting of shadow-rays, by launching the `castShadowRays` CUDA-kernel. Subsequently, it applies shading based on the intersection obtained from the `castRays`- and `castShadowRays`-kernels, by launching the `computeShading` CUDA-kernel. And finally, for the recursion-step, the reflection- and refraction-rays are traced by calling the `raytrace`-function. While recursively calling the raytrace-function, a stackpointer is provided from which the location of the pushed ray is derived. In order to mimic a depth-first traversal of the ray-hierarchy, in which the reflection-ray is spawned before the refraction-ray, the refraction-ray should be pushed *before* the reflection-ray. Therefore, in the first recursive call, the stackpointer is increased, to let it point to the top of the stack. Section 8.1.1 will provide more details about the ray-stack. Figure 6.7a and Figure 6.7b illustrate how the stack is populated during the execution of the algorithm.



(a) The ray-hierarchy corresponding to a maximum recursion-depth of 3. Each ray spawns two rays: a reflection-ray and a refraction-ray.



(b) The state of the stack while casting a ray corresponding to every node in the ray-hierarchy of Figure 6.7a.

Figure 6.7: Implementing recursion for CUDA using a stack.

Listing 6.1: CPU-code for the Ray Tracer subsystem

```
void raytrace(int2 tileIdx, int depth = 0, int stackpointer = 0)
{
    castRays<<<gridDim, blockDim>>>(tileIdx, stackpointer);

    for (int i = 0; i < light_count; i++)
    {
        castShadowRays<<<gridDim, blockDim>>>(tileIdx, i);
        computeShading<<<gridDim, blockDim>>>(tileIdx, i);
    }

    if (depth < maxRecursionDepth)
    {
        /* Trace reflection-rays */
        raytrace(tileIdx, depth + 1, stackpointer + 1);

        /* Trace refraction-rays */
        raytrace(tileIdx, depth + 1, stackpointer);
    }
}
```

6.2.4 Shading

As a final step, shading is applied by the *shader*, only if the *shadow-ray caster* informs us that the surface-point is illuminated by the light-source.

For the shading computation, a simplified Phong Illumination model is employed. For each pixel the following formula is used to compute the light intensity:

$$I_p = \sum_{\text{lights}} (k_a I_L + k_d (L \cdot N) I_L + k_s (R \cdot V)^{\alpha} I_L).$$

In this formula, I_p amounts for the light intensity *per intersection-point*, k_a the ambient reflection constant, k_d the diffuse reflection constant, k_s the specular reflection constant, and α is a *shininess* constant. Furthermore, I_L is the light-source's intensity.

This formula is computed for each spawned ray, and summed to obtain the final intensity for the pixel. Since this intensity can be greater than 1, the value is scaled by a global intensity scaling factor.

6.3 Summary

Figure 6.8 basically shows the path from a primary-ray up to the final pixel. A primary-ray is spawn by rasterizing the tesselation; a fragment is further processed by the primary-ray accelerator's root-finder; a hit may skip the ray tracer's traversal and root-finder and immediately continue by casting a shadow ray; misses and background pixels are traced conventially by the ray tracer;whereafter they are processed further by casting a shadow ray; shading is applied for non-blocked shadow-rays; and finally reflection and refraction rays are spawn and traced recursively.

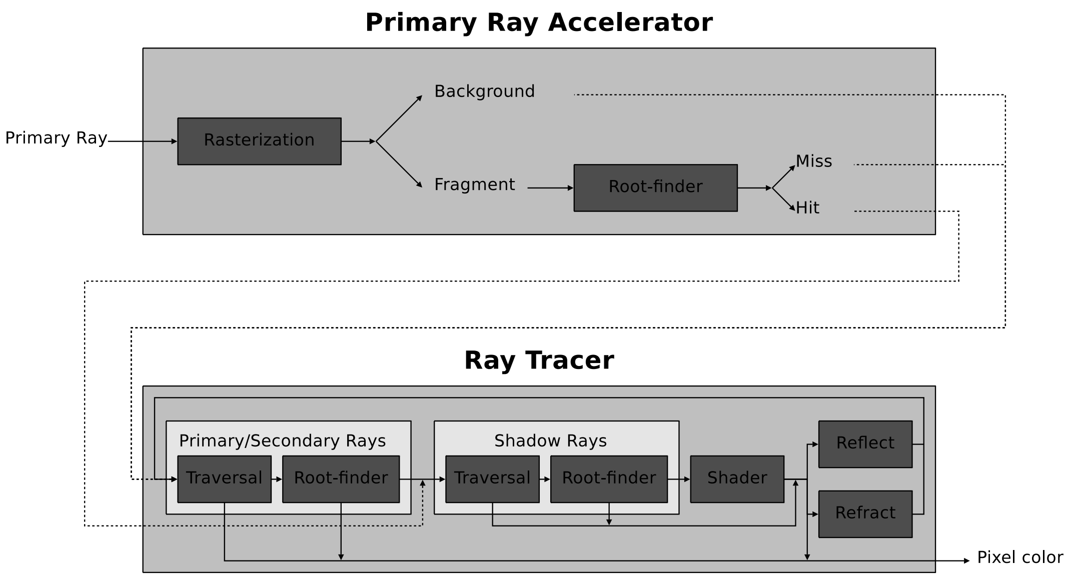


Figure 6.8: Lifetime of a *primary-ray* processed by the *Primary-ray Accelerator* and the *Ray Tracer*. A primary-ray is spawn by rasterizing the tesselation; a fragment is further processed by the primary-ray accelerator's root-finder; a hit may skip the ray tracer's traversal and root-finder and immediately continue by casting a shadow ray; misses and background pixels are traced conventionally by the ray tracer;whereafter they are processed further by casting a shadow ray; shading is applied for non-blocked shadow-rays; and finally reflection and refraction rays are spawn and traced recursively.

Chapter 7

Preprocessing

Using a separate program, a *Wavefront OBJ* file containing a NURBS-model is read, parsed, and converted to the required format. Afterwards, the NURBS-data is preprocessed to obtain the bounding volume hierarchy for the traversal, the tesselation for the rasterizer, and the Cox-de Boor items for the evaluation, which is then uploaded to the rendering core. This chapter describes the preprocessing phase of the program, and how the rendering core expects its data to be arranged.

7.1 Subdivision

Good performance, as well as an accurate result, is realized by a carefully subdivided NURBS-model. Both the *ray caster* and the *ray tracing* subsystems depend on it. To ensure a close enough linear approximation to the real surface, the NURBS-surface is first refined by inserting new knots into the knot-vectors. By adding new knots, new control-points can be computed, which represent *exactly* the same surface, but the convex-hull of the new refined surface also becomes closer to the true surface. Therefore, by adaptively determining how many knots to insert and where, a new refined mesh is obtained which can be used for tessellation and BVH generation.

The heuristic from [MCFS00] is employed for each u knot-span and corresponding column in the grid (or row, in case of the v knot-vector). The maximum value for this heuristic, applied to each row in the column, determines the number of knots to insert in the knot-span. This is repeated for each knot-span for both knot-vectors. As a final step the NURBS-surface is converted to Bézier-patches by setting the multiplicity of each internal knot in the new knot-vectors to the degree of the surface. The new control-grid is updated by computing the new control-points (Figure 7.2).

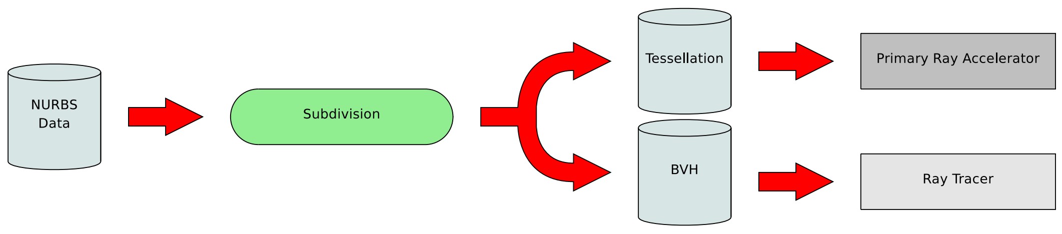


Figure 7.1: Overview of subdivision: the two subsystems both depend on the subdivision of the NURBS-model.

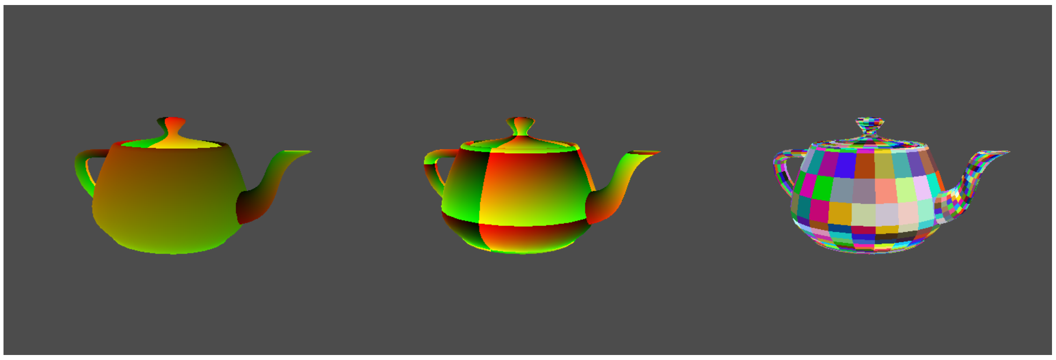


Figure 7.2: The NURBS-surface is converted into smaller Bézier-patches.

7.1.1 Tessellation

The ray caster subsystem depends on a tessellation in order to rasterize the NURBS-surface to obtain good starting values for the numerical root-finder. This tessellation is directly derived from the subdivision of the surface. The tessellation consist of a set of quadrilaterals each corresponding to a sub-patch. The corners of each quadrilateral map to the corners of the corresponding Bézier-patch. Each vertex contains a surface-identifier, a span-identifier, and an initial-guess uv-parameter.

7.1.2 BVH generation

The bounding volume hierarchy is easily constructed from the refined control-grid. From each sub-patch (corresponding to the new non-empty intervals in the refined knot-vectors), a bounding box is computed. Since the sub-patches are Bézier, the convex-hull is just the convex set of control-points corresponding to the sub-patch. The bounding box is now simply the minimum, respectively the maximum of these control-points (component-wise).

To generate the parent nodes higher up in the tree, the set of leaf-nodes are sorted along the axis of longest extent, and split in halve (object median). A new internal node is generated and is set as the parent of these two sets. This process continues recursively until a set of leaf-nodes contains only one leaf-node Figure 7.3.

7.1.3 Memory Layout

BVH Nodes

The BVH nodes are arranged depth-first, as an array of structures to allow efficient access using the parallel-read operation (Section 8.5.2). The array is stored in a texture bound to a single large piece of linear-memory. As can be seen in Listing 7.1, each node contains two child-pointers and two AABB for the corresponding children. To ensure the structure is correctly aligned to 16 bytes, two extra `floats` are padded.

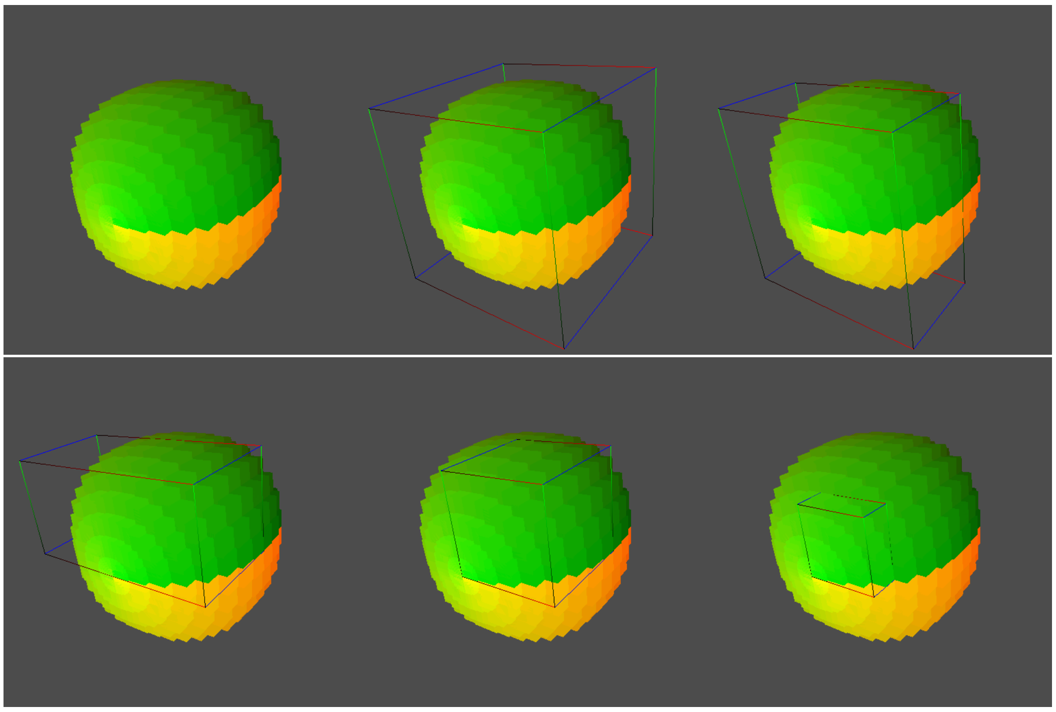


Figure 7.3: Bounding boxes of a BVH.

Listing 7.1: Definition of the `BVHNode`-struct

```
struct __align__(16) BVHNode
{
    int     child[2];
    float   aabb_left[3][2];
    float   aabb_right[3][2];

    float   padding[2];
};
```

Leaf-nodes

The leaf-nodes contain other data, than the internal BVH nodes. Therefore, a separate structure defines the leaf-nodes, which are actually just nurbs-patches. A nurbs-patch contains a reference to its parent surface, its corresponding knot-spans, and an initial-guess value. Additionally, an extra AABB is added to increase performance during traversal, as will be described in Section 8.5.2. Listing 7.2 shows the definition of the `NURBSPatch`-struct.

Algorithm 10 Cox-deBoor-items precomputation

```
Require: Knot-vector $knot$, $n$, $degree$
 1: $cdbitems \leftarrow []$
 2: for $d = 1$ to $degree$ do
 3:     for $i = n-1$ to $degree - d$ step $-1$ do
 4:         $cdbitem[0] \leftarrow \frac{1}{knot[i+d]-knot[i]}$
 5:         $cdbitem[1] \leftarrow \frac{-1}{knot[i+d+1]-knot[i+1]}$
 6:         $cdbitem[2] \leftarrow -cdbitem[0] \cdot knot[i]$
 7:         $cdbitem[3] \leftarrow -cdbitem[1] \cdot knot[i+d+1]$
 8:         $append(cdbitems, cdbitem)$
 9:     end for
10: end for
11: return $cdbitems$
```

Listing 7.2: Definition of the `NURBSPatch`-struct

```
struct __align__(16) NURBSPatch
{
    int       surface_id;
    int       span_id;
    float2    uv;

    float     aabb[3][2];
    float     padding[2];
}
```

7.2 Root-Finder Data

The root-finder depends heavily on the evaluation algorithm, which in turn requires the availability of the NURBS-surface data. Depending on the used evaluation scheme (Section 3.5), additional data may be required.

7.2.1 Basis-function data

Two evaluation schemes have been tested, namely the “inverted-triangle”-method, and the division-free method. While the former does not require any preprocessing, the latter depends on the precomputed *Cox-deBoor-items*. The computation of these items is fairly straightforward. For each surface, these items are computed row-wise. Algorithm 10 gives the pseudo-code for computing the items.

7.2.2 Memory layout

NURBS-surface data

The root-finder heavily depends on information describing the NURBS-surface, such as the degrees, control-points, knot-vectors, etc. The probability of two coherent rays hitting the same NURBS-surface, and therefore accessing the same data, is rather high. Therefore, the texture-memory is an ideal candidate for storing the NURBS-data. Frequently accessed surface-data will appear in the texture-cache, resulting in very fast fetch-times.

For packet-based traversals, the performance can be increased, by choosing a layout that minimizes the fetch-time. For this, the data is arranged as an array of structures. In this way, the data can be fetched very efficiently by a single instruction using a parallel-read.

For non-packet-based traversals, the AOS-approach is still beneficial over an SOA-approach, since reading a member will result in a pre-fetch of neighbouring members, which again will reduce the fetch-times. Furthermore, for the primary-ray accelerator, it could even be less efficient to store the surfaces as a SOA. At first, there is totally no spatial relation between the surfaces and the storage of them, and second, because the primary-ray accelerator only performs a single intersection test.

Because the data for the control-points and the knot-vectors are variable-sized, they cannot be included in the nurbs-surface data directly, and are therefore stored in two separate textures: one for the control-points, and another for the knot-vectors. These textures will be discussed in the next subsections.

Listing Listing 7.3 shows the definition of the structure describing the NURBS-surface.

Listing 7.3: Definition of the `NURBSSurface`-struct

```
struct __align__(16) NURBSSurface
{
    int n_u;
    int n_v;
    int degree_u;
    int degree_v;
    int offsetControlPoints;
    int offsetKnotVector_u;
    int offsetKnotVector_v;
    int material_id;
};
```

Control-points

CUDA provides a very optimized way for accessing texture data by exploiting the two-dimensional spatial coherence. However, in order to benefit from this efficiency, the control-points are required to be stored into a rectangular texture. Since the sizes of the control-point grids may differ, a lot of “empty” areas will appear in the texture.

Instead, the control-points of all surfaces are combined in a texture bound to a single large piece of linear-memory. The control-points are stored in row-major order. The rows of a surface are concatenated to the texture to form an indexible array. The index of the first control-point is stored into the `NURBSSurface`-struct and is used to find the corresponding control-points (ListingListing 7.3).

Knot-vectors

The knot-vector data is stored similarly to the control-point data. Using a single large texture both the knot-vectors are concatenated and appended to the texture. The starting index for each knot-vector is stored into the `NURBSSurface`-struct and is used to find the corresponding knot-vector (ListingListing 7.3).

Cox-deBoor items

The Cox-deBoor array is stored similarly to the control-point data. Using a single large texture both arrays are concatenated and appended to the texture. The starting index for each array is stored into the `NURBSSurface`-struct and is used to find the corresponding knot-vector (ListingListing 7.3).

Material data

Since there is no relationship between the *spatial position* of two surfaces having two distinct materials and the *memory addresses* of those materials, little advantageous can be expected by arranging the materials as a structure of arrays. Furthermore, while shading a pixel, every data-item of the material will be accessed successively. Therefore, the material-data should be clustered to allow prefetching (Section 5.2.2).

By creating a texture containing an array of material-structures, the data can be fetched most efficiently. Listing Listing 7.4 shows the definition of the structure describing a material.

Listing 7.4: Definition of the `Material`-struct

```
struct __align__(16) Material
{
    float3 diffuse;
    float3 specular;
    int    shininess;
    float  reflectance;
    float  transparency;
    float  refraction_index;
};
```

Chapter 8

Kernel details

This chapter will discuss all the implementation-specific details for the rendering core. First, all data structures will be described, which are used for inter-kernel communication. Then, the root-finder will be discussed, which will be used by the primary-ray accelerator subsystem as well as the ray tracer subsystem. Finally, the kernel-launch configurations are discussed, including the mapping of rays to threads.

8.1 Inter-kernel Communication

8.1.1 Ray-stack

To get around CUDA's absence of support for recursive function calls, a ray-stack is used. Instead of stepping recursively into the ray tracing function to trace reflection- and refraction-rays, the rays are pushed onto the ray-stack, and traced further during subsequent kernel-launches. Each kernel-launch starts by popping one secondary ray from the ray-stack, which is traced and possibly spawns two more rays.

The stack is implemented in global-memory as a structure of arrays (Section 5.2.1) containing stack-data for every thread. Each array maps to exactly one stack-level holding the data for each thread for that level. Figure 8.1 shows the memory layout of a stack for 4 threads using a stack-depth of 3. Reading from as well as writing to memory laid out like that, results in efficient coalesced transfers (Section 5.2.1).

Usually each thread maintains its own stack-pointer, since some rays will hit a surface and spawn rays, while others will not. Instead, each ray traverses the same path in the ray-hierarchy, and pushes two *dummy*-rays onto the stack whenever it does not spawn secondary rays. In this way, all threads can share the same stack-pointer. The dummy-ray can be identified by setting the x-value to $+\infty$. When popping a ray, an x-value of $+\infty$ indicates it's a dummy-ray and the thread should not participate in the traversal.

The definition of the structure can be found in listing Listing 8.1. The amount of memory in bytes required for allocating the ray-stack containing 8 `float`-arrays, is given by:

$$\|\text{RayStack}\| = width \cdot height \cdot (depth + 1) \cdot 32$$

where *width* and *height* represent the image-resolution, and *depth* equals the maximum recursion-depth. The $depth + 1$ factor equals the maximum stack-size, and is one larger than the maximum recursion depth, since the ray-hierarchy is traversed depth-first.

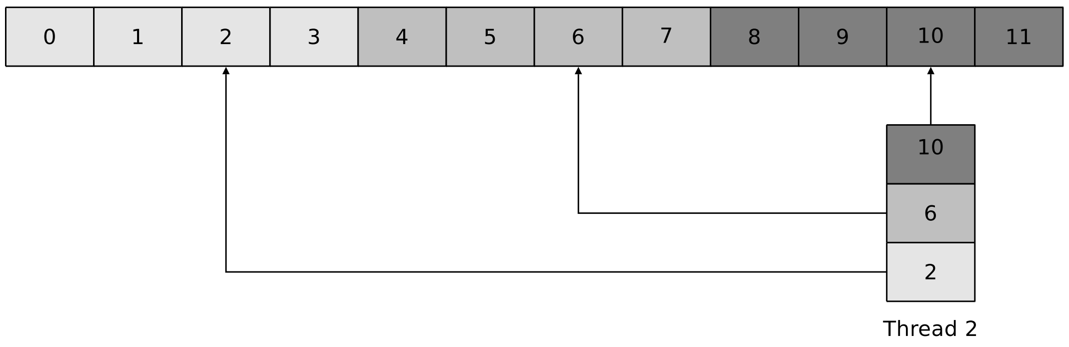


Figure 8.1: Memory layout of a stack inside global-memory. This stack is for 4 threads and has a maximum depth of 3. The exemplary "virtual" stack of thread 2 shows the mapping of threads and stack-levels to global-memory addresses.

Listing 8.1: Definition of the `RayStack`-SOA

```
struct RayStack
{
    struct
    {
        float *x;
        float *y;
        float *z;
    } o;

    struct
    {
        float *x;
        float *y;
        float *z;
    } d;

    float *opacity;
    float *refraction_index;
};

__constant__ RayStack rayStack;
```

8.1.2 Hitdata-buffer

The hitdata-buffer is used to propagate intermediate results from the ray caster to the shader. These include, among others, the coordinates of the intersection-point, the surface-normal corresponding to that intersection, the distance along the ray causing that intersection, etc.

The buffer is arranged in global-memory as a structure of arrays in order to hide the transfer latency (Section 5.2.1). The definition of the structure can be found in listing Listing 8.2. The amount of memory in bytes required for allocating the hitdata-buffer containing 13 `float`-arrays and 2 `int`-arrays, is given by:

$$||\text{HitDataBuffer}|| = \mathit{width} \times \mathit{height} \times 60$$

where *width* and *height* represent the image-resolution.

Listing 8.2: Definition of the `HitDataBuffer`-SOA

```
struct HitDataBuffer
{
    float *t;

    struct
    {
        int *surface_id;
        int *span_id;
    } patch_id;

    struct
    {
        float *x;
        float *y;
        float *z;
    } s;

    struct
    {
        float *x;
        float *y;
        float *z;
    } normal;

    struct
    {
        float *x;
        float *y;
        float *z;
    } d;

    float *u;
    float *v;
    float *opacity;
};

__constant__ HitDataBuffer hitDataBuffer;
```

8.1.3 Image-buffer

After shading is applied to each ray, its resulting color is accumulated to the corresponding pixel in the image-buffer. This buffer is not explicitly allocated by CUDA[1], but is acquired by mapping an OpenGL *Pixel Buffer Object* (PBO) into the address-space of CUDA. The buffer is arranged as successive quadruples containing `floats` for the red, green, blue, and alpha channels, respectively. When mapped, the buffer can be accessed as an ordinary `float4`-array.

After the rendering is finished, the buffer is unmapped, and displayed on screen. Listing Listing 8.3 shows the code required for initialization of the image-buffer and Listing 8.4 shows the code for rendering.

[1]However, when the goal of ray tracing is solely offscreen-rendering, i.e. in order to output a ray traced video, the buffer must be allocated by CUDA.

Listing 8.3: Initialization of the image-buffer.

```
/* Allocation of image-buffer */
size_t size = width * height * sizeof(float4);
glGenBuffers(1, &pboID);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pboID);
glBufferData(GL_PIXEL_PACK_BUFFER, size, 0, GL_DYNAMIC_DRAW);
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);

/* Registration to CUDA */
cudaGLRegisterBufferObject(pboID);
```

Listing 8.4: Usage of the image-buffer.

```
/* Mapping into address-space of CUDA */
cudaGLMapBufferObject((void **)&imageBuffer, pboID);

/* Rendering */
...

/* Unmapping */
cudaGLUnmapBufferObject(pboID);

/* Blitting */
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pboID);
glDrawPixels(width, height, GL_RGBA, GL_FLOAT, 0);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
```

8.2 Root-finding

The root-finder basically implements a slightly modified version of the Newton-Rhapson algorithm described in Section 3.3.1. Normally, when the *uv*-parameter leaves the surface-domain, the root-finder should report an intersection-miss. However, it appeared that sometimes for surfaces which actually do contain a valid intersection-point, the *uv*-increment becomes too large during the first iteration, which causes the parameter to leave the surface-domain too soon. The root-finder will stop searching too early, and reports a miss. To prevent this faulty early-termination, a minimum iteration-count is incorporated to prevent such events. This extra minimum is also applied to the criteria in which the error increases. The results will show (Section 9.2.2), that the quality of the output is increased, while the performance is not affected (if will actually increase a little).

Usually, a binary-search is performed to compute the knot-intervals from the *uv*-parameters. However, we already have this information: both the rasterization and the BVH provides us these data from the patch-identifier. From the span-id and the knot-vector, the patch-domain is derived which is used inside the root-finder. When the *uv*-parameter leaves the patch-domain, the span-id is updated according to the new location on the surface. If the *uv*-parameter leaves the surface-domain, the root-finder will exit (only after the minimum number of iterations has been reached).

8.3 Surface Evaluation

Algorithm 4 is used for evaluation of the surface-point and its partial-derivatives. An important optimization has been added however. Since the basis-functions (Section 3.5) are accessed very frequently, both during basis-function evaluation, as well as during the computation of the tensor-

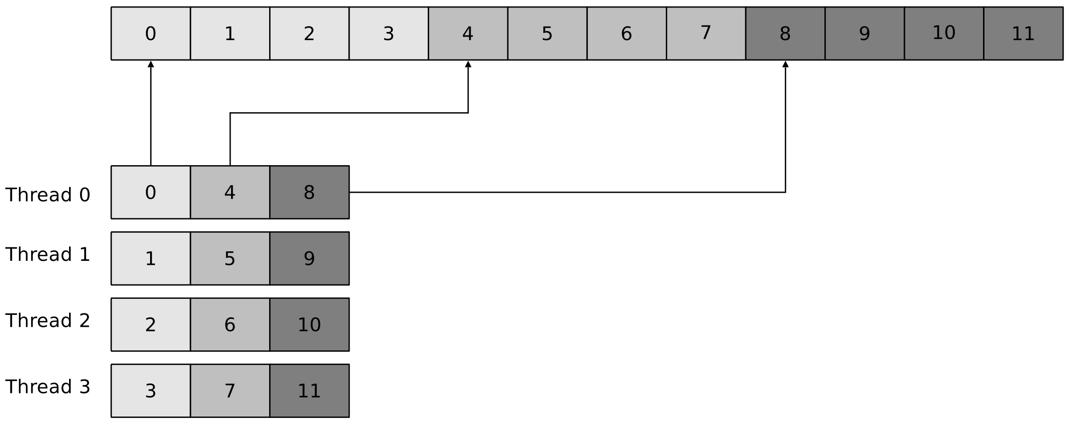


Figure 8.2: Arrangement of data for a cache inside shared-memory. This example uses an array of 3 elements for each thread, and a block size of 4 threads.

product, storing them in the local-memory will be rather slow. While local-memory may suggest that it's on-chip memory, it's actually not. Local-memory is located in the DRAM, and although the accesses are implicitly coalesced, they still have a very high latency. Therefore, a software-managed cache has been implemented which will now be described.

8.3.1 Software-managed Cache

In order to minimize these latencies, a cache is implemented in shared-memory (Section 5.2). For a block of threads, some shared-memory is reserved in which these arrays reside. To prevent bank-conflicts, each thread in a half-warp needs to access a different location. Therefore, the per-thread array-elements have a stride equal to the number of threads in the block, which is always a multiple of the width of a half-warp (Figure 8.2), in this way bank-conflict are avoided entirely. Listing 8.5 shows an example of a software-managed cache using the shared-memory.

Listing 8.5: Software-managed Cache

```
__shared__ float N[BLOCKSIZE * elements];
float element_0 = N[BLOCKSIZE * 0 + THREAD_IDX];
float element_1 = N[BLOCKSIZE * 1 + THREAD_IDX];
float element_2 = N[BLOCKSIZE * 2 + THREAD_IDX];
...
```

8.3.2 Basis-function evaluation

From the different evaluation schemes described in Section 3.5, only the direct and the division-free methods have been implemented. The power-basis method was not chosen, because it requires a lot of preprocessing, and does not result in a very stable evaluation method. The de Boor's algorithm requires too much registers and has also been skipped.

8.4 Primary-ray Accelerator subsystem

8.4.1 Rasterization

Usually, rasterization is directly output to the frame-buffer and displayed on screen. However, we need to obtain the resulting output, since it needs further processing. The OpenGL *Framebuffer Object* (FBO) extension, provides a highly efficient mechanism for redirecting the rasterization to a virtual frame-buffer. The FBO is a collection of attachable OpenGL buffer objects containing one or more render-buffers to which the rasterization is output, a depth-buffer, stencil buffer, auxiliary buffers, etc. Also, textures can be attached to the FBO to enable direct render-to-texture support. This system uses an FBO containing a render-buffer for the data and additionally a depth-buffer to enable proper depth ordering of the fragments.

Recall from Section 6.1.1, that each vertex gets some custom attributes. These attributes correspond to the surface-patch and the *uv*-parameter of the vertex. Since a single render-buffer is used to store the rasterization, the format of the custom attributes need to match, which is not the case (*uv*-parameters are floating-point and patch-identifier contains integer data). Therefore, the integer data is converted to floating-point data by performing a type cast on the binary data[2]. Due to numerical round-off errors, interpolating all attributes will result in an incorrect patch-identifier. Therefore, a custom GLSL-shader is employed which only interpolates the *uv*-parameters, but leaves the patch-identifier data untouched. Listing 8.6 and Listing 8.7 shows the shader programs which are used to obtain the correct result. The extra `flat` keyword denotes an attribute which will *not* be interpolated by the rasterization-unit, the resulting fragment will receive the value of an arbitrary vertex contained in the primitive.

Listing 8.6: Vertex shader

```
#extension EXT_gpu_shader4 : enable

attribute vec4 attr_data;

     varying vec2  uv;
flat varying float surface_id;
flat varying float span_id;

void main(void)
{
    gl_Position  = ftransform();
    uv           = attr_data.xy;
    surface_id   = attr_data.z;
    span_id      = attr_data.w;
}
```

[2]For example, the integer value `0x3f800000` would be converted to the floating-point value `1.0`.

Listing 8.7: Fragment shader

```
#extension EXT_gpu_shader4 : enable

     varying vec2  uv;
flat varying float surface_id;
flat varying float span_id;

void main(void)
{
    gl_FragColor = vec4(uv.x, uv.y, surface_id, span_id);
}
```

After rasterization has been completed, the render-buffer can be read by the ray caster. Unfortunately, CUDA currently does not support direct access to render-buffers. Therefore, an OpenGL *Pixel Buffer Object* (PBO) is used to copy the data from the render-buffer to the PBO. Although this requires another copy, the transfer is from GPU-memory to GPU-memory, which is very fast. After copying, the data is made available to CUDA by mapping the PBO into the address-space of CUDA. Instead of allocating a new buffer for this PBO, the image-buffer is used (Section 8.1.3).

8.4.2 Ray Casting

There are two ray casting kernels, one for the Primary-ray Accelerator subsystem, and another for the Ray Tracer subsystem. In this section, the former is discussed, which can be seen in Listing 8.8. It starts by reading the corresponding data-item from the rasterization buffer. If the surface-identifier value happens to be equal to zero, a background pixel has been identified, and no intersection needs to be found. A non-zero value however, indicates a surface-patch, consequently a ray should be cast for this pixel.

Since we already have a surface-patch candidate, we avoid the need for traversal of the scene. We can immediately try to find an intersection using the data from the rasterization.

Although the Ray Tracer subsystem should spawn secondary rays, it turns out to be more efficient to immediately spawn them. The reason is that if we would let the ray tracer spawn the rays, it would depend on the hitdata, which is read from global-memory. This will heavily stall the execution of the ray tracing kernel. As we already have all required data, i.e. the intersection-point, surface-normal, material-properties, etc., we can just as easy compute the reflection and refraction ray here. This will increase the performance in a later stage, since now only the intersection-distance has to be read to determine if the pixel can be skipped.

After having obtained an intersection, or if the ray was skipped since the rasterization reported a background-pixel, the `hitData`-record is written to global-memory through the `hitData`-buffer.

Listing 8.8: Ray Casting kernel (Primary-ray Accelerator)

```
template<bool spawnRays>
__global__ void castRay(int2 tileIdx, float4 *framebuffer)
{
    ::tileIdx = tileIdx;

    float4 data = framebuffer[OFFSET_SCREEN];

    HitData hitData;
    hitData.t = FLOAT_INFINITY;

    /* Skip background-pixels */
    if (__float_as_int(data.z) != 0)
    {
        Ray ray = fromPixel(X_SCREEN, Y_SCREEN);

        NURBSPatch nurbsPatch;
        nurbsPatch.uv = make_float2(data.x, data.y);
        nurbsPatch.surface_id = __float_as_int(data.z) - 1;
        nurbsPatch.span_id = __float_as_int(data.w);

        float2 uv_min;
        float2 uv_max;
        fetchPatchDomain(nurbsPatch, &uv_min, &uv_max);

        rootFinder(ray, nurbsPatch, uv_min, uv_max, hitData);

        if (spawnRays)
        {
            Ray reflectionRay = ray.reflect(&hitData);
            rayStack.push(&reflectionRay, stackpointer + 1);

            Ray refractionRay = ray.refract(&hitData);
            rayStack.push(&refractionRay, stackpointer + 0);
        }
    }

    __hitDataBuffer__.write(&hitData);
}
```

Parallel intersection test

Normally, when taking a SIMD-approach to ray-tracing, the scene is traced packet-based using a small packet-size (usually 2x2 rays). Often, four pixels in a block refer to the same patch which can then be intersected by at best four rays in parallel with only one call to the intersection routine. Occasionally, the rays intersect different patches. The intersection routine is then called for each distinct patch in the SIMD-packet.

However, due to CUDA's SIMD-width of 32 threads, the ray caster in this system uses a much larger packet, increasing the likeliness of the rays intersecting different patches. Calling the intersection routine for each distinct patch will result in many calls, in which only a small amount of rays will actually be testing the patch for an intersection. Hence, for a CUDA-based ray-tracer, employing such a method will decrease the multiprocessor's utilization heavily.

Because all dependent NURBS-data is located in texture-memory, such as the knot-vectors, control-grids, etc., it is possible for different rays to access different surface-patches. Therefore, the intersection test will be called only once for each thread, maximizing the utilization.

8.5 Ray Tracer subsystem

8.5.1 Ray Casting

The castRay-kernel implements the majority of the ray tracing algorithm. It starts by constructing a ray if it's casting a primary-ray, otherwise it simply pops a previously spawned reflection- or refraction-ray from the ray-stack (Section 8.1.1). It then traverses the scene using either the packet-based traversal scheme or the single-ray traversal scheme, which will be described in the next subsection. After traversal, the hitData-record is written to global-memory through the hitData-buffer. Finally, the reflection- and refraction-rays are spawned. Once again, regarding the stack-pointer, the reflection-rays are pushed "above" the refraction-rays.

In order to assist nvcc in optimizing as much as possible, the complexity of the code is reduced by using a templated function. Using templates, parts of the code can be enabled or disabled during compile-time. The primaryRay-flag indicates we are casting primary-rays, so we should not pop from the stack, but instead construct the ray in-place. One could argue this is less efficient, since it increases the complexity of the code. However, it turns out that popping of rays is quite expensive compared to constructing primary-rays, due to the global-memory latency. The same applies for the spawnRays-flag, which is used to enable or disable the reflection- and refraction-rays.

Listing Listing 8.12 shows the ray casting kernel for the ray tracing subsystem.

Listing 8.9: Ray Casting kernel (Ray Tracer)

```
template<bool primaryRay, bool spawnRays>
__global__ void castRay(int2 tileIdx, int stackpointer)
{
    ::tileIdx = tileIdx;

    HitData hitData;
    hitData.t = INF;

    Ray ray;
    if (primaryRay)
        ray = fromPixel(X_SCREEN, Y_SCREEN);
    else
        rayStack.pop(&ray, stackpointer);

    traverseRay(ray, hitData);

    hitDataBuffer.write(&hitData);

    if (spawnRays)
    {
        Ray reflectionRay = ray.reflect(&hitData);
        rayStack.push(&reflectionRay, stackpointer + 1);

        Ray refractionRay = ray.refract(&hitData);
        rayStack.push(&refractionRay, stackpointer + 0);
    }
}
```

8.5.2 Packet-based Traversal

For the packet-based traversal scheme, the method from [GPSS07] is used which is described in Section 4.1.3, with some additions specific for NURBS-ray tracing. It can be beneficial for all threads to exit as early as possible. Non-participating packets for example, may skip the entire traversal algorithm. This decision is made by performing a parallel-reduction prior to the traversal (Section 8.5.2).

Shared-memory stack

In a packet-based traversal each ray traverses the hierarchy in exactly the same order. Therefore, the stack required to push nodes onto, can be shared by the rays. This stack is implemented by pushing node addresses to an array in shared-memory. By reserving 24 integers for this array, scenes consisting of up to 16,777,216[3] sub-patches can be traversed, which should be enough.

Each thread maintains its own local stack-pointer. Popping elements from the stack is then simply done by decrementing the pointer and returning the value from the shared-memory array corresponding to this pointer. Pushing to the stack requires some synchronization however, since only one thread is allowed to write to the shared-memory if all threads want to write to the same location, which is the case. To prevent potential read-after-write hazards, a barrier is set before pushing the new value. For the same reason, after the data has been written, another barrier is set. Afterwards, the stack-pointer is incremented for each thread.

Listing 8.10 shows a code-fragment of how the shared-memory stack can be used to push a data-item onto the stack, and pop it from the stack afterwards.

Listing 8.10: Shared-memory stack

```
__shared__ int stack[STACK_SIZE];
int stack_pointer = 0;

/* Pushing an element to the shared-stack */
__syncthreads();
if (THREAD_IDX == 0) stack[stack_pointer] = data;
__syncthreads();
stack_pointer++;

/* Popping an element from the shared-stack */
data = stack[--stack_pointer];
```

View-dependent ordered traversal

As in [GPSS07], a view-dependent ordered traversal scheme is used, which increases performance for most scenes. A packet traverses the hierarchy, collectively deciding which child to traverse first. This decision is slightly altered however. *Inactive rays* (rays which do not intersect the current node) should not affect the decision-making of the algorithm, therefore the following modified formula is used to determine which child should be visited next:

$$i = (t_0 \neq \infty) + 2 \cdot (t_1 \neq \infty)$$

where t_0 is the intersection distance for the AABB of the left child, and t_1 is the intersection distance for the AABB of the right child. For each ray, this value is computed and the index is used to set the ith element in a shared-memory array to `true`, for which each four values are initialized with the value `false`. Consequently, the values of this array determine which children need a visit. If the fourth element is `true`, *or* the second and third element are *both* `true`, both children need a visit. Otherwise, the left child only needs a visit if the second element is `true`, the right child if the third is `true`, and no children if the first element is `true` (which denotes an inactive ray).

[3]Assuming a balanced tree.

The other decision is to determine which child needs to be visited first if the packet will visit both:

$$j = 2 \cdot (t_0 > t_1) - (t_0 \neq t_1)$$

j will result in -1 if for this ray the left child is closer than the right child, $+1$ if the right child is closer, and 0 if the ray is inactive. All these values are summed, and the resulting value decides which child will be visited next.

Parallel Reduction

A frequently used data-parallel primitive in parallel software is the *parallel reduction* (or parallel sum), in which an array of data-elements is “summed” using an associative operation (i.e. addition, multiplication, maximum, minimum, etc.). Usually, performing a reduction operation on an array of data requires n steps, for an array of size n. However, with the availability of a parallel multiprocessor, this operation can be performed very efficiently. Using a tree-based approach, in which each node (thread) computes the partial sum of its two children (array elements), an array of n elements can be summed in $\log_2 n$ steps.

Using this primitive, also functions such as *all* and *any* can be implemented, simply by using the *and* resp. *or* operators. Using these functions, a predicate can be tested for all threads to be true, or at least one thread in case of the *any* operation.

Listing 8.11 shows a code-fragment of how the parallel reduction is used to sum the 64 integer values in deciding which child should be visited first. Since CUDA-threads are executed in SIMD-groups of 32 threads, called *warps*, no synchronization is required during the reduction steps, as long as the size of the array does not exceed 64 elements.

Listing 8.11: Parallel-sum

```
__shared__ int data[64];
data[THREAD_IDX] = decision;
__syncthreads();

/* Reductions */
if (THREAD_IDX < 32) data[THREAD_IDX] += data[THREAD_IDX + 32];
if (THREAD_IDX < 16) data[THREAD_IDX] += data[THREAD_IDX + 16];
if (THREAD_IDX <  8) data[THREAD_IDX] += data[THREAD_IDX +  8];
if (THREAD_IDX <  4) data[THREAD_IDX] += data[THREAD_IDX +  4];
if (THREAD_IDX <  2) data[THREAD_IDX] += data[THREAD_IDX +  2];
if (THREAD_IDX <  1) data[THREAD_IDX] += data[THREAD_IDX +  1];
__syncthreads();

decision = data[0];
```

Parallel Memory Fetching

A highly optimized parallel algorithm to obtain data which is shared among all threads in the block, is the *parallel-read* algorithm. The algorithm is used to transfer a block of data from global-memory (or texture-memory) to shared-memory in an efficient way: by letting each subsequent thread read a subsequent data-element. Using this algorithm, a block of memory can be read using only one instruction as long as the number of data-elements does not exceed the number of threads in the block.

The efficiency lies in the coalescing of the memory-accesses: each thread accesses a subsequent address. Therefore, the memory controller can transfer the memory line-by-line exploiting the bandwidth of the memory-bus. Furthermore, a thread does not have to wait for the read to complete before reading the next data-element, since all data are read at once.

Packet-based Root-finding

The Primary-ray Accelerator subsystem fetches its data directly through the texture-unit, allowing the rays to simultaneously intersect different patches. The Ray Tracer subsystem on the other hand, traverses the scene packet-based. While processing a leaf-node, all rays in a packet always call the intersection routine using the same patch.

Since all rays access the same data, they can be fetched efficiently using parallel-read operations (Section 8.5.2) and transferred to shared-memory. Although the data-fetches are already cached by the texture-unit, using the parallel-transfer operations, the memory is accessed much more efficiently.

Extra Leaf-node Ray-AABB intersection test

If a packet visits a leaf-node because only a single ray intersects it (or a few rays), the other *"non-active"* rays in the packet pay the price for the packet-based traversal. Normally, this is not that big of a deal, since ray/primitive intersection tests are very cheap for basic primitives (i.e. triangles). However, now the other rays also have to participate in the root-finding. There is a possibility some rays converge or diverge more slowly than the rays that were actually intersecting the node. This will unnecessarily stall the traversal. Also, a ray could already have obtained an intersection which is more close than the intersection with the AABB of the to-be-tested leaf-node. In that case, the intersection test even becomes unnecessary.

By simply inserting an additional, repeated ray/AABB intersection test right before the root-finder, these situations can be avoided, reducing the total number of intersection tests and stalls.

Listing 8.12 shows the total packet-based traversal algorithm.

Listing 8.12: `Packet-based Traversal`-function

```
__device__ void traverseRayPacket(const Ray &ray, HitData &hitData)
{
    if (all<BLOCKSIZE>(!ray.participating()))
        return;

    SharedStack<MAX_STACK_DEPTH> stack;
    int nodeIndex = 0;

    for (;;)
    {
        if (IS_LEAF(nodeIndex))
        {
            __shared__ NURBSPatch nurbsPatch;

            fetchNURBSPatch(nurbsPatch, nodeIndex);
            fetchSharedEvaluationData(nurbsPatch);

            if (ray.participating() &&
                ray.intersectAABB(nurbsPatch.aabb, hitData.t) != INF)
                rootFinder(ray, nurbsPatch, hitData);

            if (stack.empty()) break;

            stack.pop(nodeIndex);
        }
        else
        {
            __shared__ BVHNode bvhNode;
            fetchBVHNode(bvhNode, nodeIndex);

            float t0 = INF;
            float t1 = INF;

            if (ray.participating())
            {
                t0 = ray.intersectAABB(bvhNode.aabb[0], hitData.t);
                t1 = ray.intersectAABB(bvhNode.aabb[1], hitData.t);
            }

            /* Set the flag for which child needs a visit (or both)
            M[0] == true --> No children intersected
            M[1] == true --> Only left child intersected
            M[2] == true --> Only right child intersected
            M[3] == true --> Both children intersected */
            __shared__ int M[4];
            if (THREAD_IDX < 4) M[THREAD_IDX] = 0;
            __syncthreads();

            M[(t0 != INF) + 2 * (t1 != INF)] = 1;
            __syncthreads();

            if (M[3] || M[1] && M[2])
            {
                /* -1 == prefer left,
                 *  1 == prefer right,
                 *  0 == no preference */
                bool go_left = sum<BLOCKSIZE>(2*(t0 > t1) - (t0 != t1)) < 0;

                stack.push(bvhNode.child[go_left]);
                nodeIndex = bvhNode.child[1 - go_left];
            }
            else if (M[1])
            {
                nodeIndex = bvhNode.child[0];
            }
            else if (M[2])
            {
                nodeIndex = bvhNode.child[1];
            }
            else
            {
                if (stack.empty()) break;
                stack.pop(nodeIndex);
            }
        }
    }
}
```

8.5.3 Single-ray Traversal

Although the packet-based scheme allows for an efficient traversal, it results in more leaf-nodes visited per ray. Therefore, when processing a leaf-node, the likeliness of a ray not intersecting that leaf-node is much higher than for single-ray traversal. As a consequence, the number of rays in a block actively participating in the root-finding, will be much lower than for single-ray traversal.

On the other hand, the packet-based traversal scheme will not lead to warp-divergence, since each ray in a packet follows the exact same traversal path. Single-ray traversal however, does lead to heavy warp-divergence. Some rays for example, may execute the root-finder, while others are traversing the BVH. A warp containing traversing and root-finding rays, cannot be executed simultaneously. Therefore, such warps will be executed in two subsequent steps: the first executing a traversal step, disabling the root-finding threads. The second step will execute the root-finder, and disables the other threads. However, since a root-finding step is very expensive, the inactive threads will have to wait very long, before they can continue, resulting in a very low utilization.

Another traversal scheme has been implemented, based on the quite recent optimized single-ray traversal scheme from [AL09]. Their method separates the root-finding from the traversal. Although warps still may diverge during traversal, they will join right after all rays have found a candidate leaf-node, so that all threads will participate in the root-finding, maximizing the utilization. Listing 8.13 shows the single-ray traversal algorithm.

Listing 8.13: `Single-ray Traversal`-function

```
__device__ void traverseSingleRay(const Ray &ray, HitData &hitData)
{
    if (!ray.participating())
        return;

    LocalStack<MAX_STACK_DEPTH> stack;
    stack.push(0);

    int nodeindex;
    while (true)
    {
        if (!traverseSingleRayNextNode(ray, hitData, nodeIndex, stack))
            return;

        NURBSPatch nurbsPatch;
        nurbsPatch.surface_id = fetchSurfaceID(nodeIndex);
        nurbsPatch.span_id = fetchSpanID(nodeIndex);
        nurbsPatch.uv = fetchUV(nodeIndex);

        float2 uv_min;
        float2 uv_max;
        fetchPatchDomain(nurbsPatch, &uv_min, &uv_max);

        fetchNURBSPatch(nurbsPatch, nodeIndex);

        if (ray.intersectAABB(nurbsPatch.aabb, hitData.t) != INF)
            rootFinder(ray, nurbsPatch, hitData);
    }
}

__device__ void traverseSingleRayNextNode(const Ray &ray, HitData &hitData, int &nodeIndex,
     LocalStack &stack)
{
    if (stack.empty()) return false;

    stack.pop(nodeIndex);

    for (;;)
    {
        if (IS_LEAF(nodeIndex)) return true;

        float t0 = ray.intersectAABB(bvhNode.aabb[0], hitData.t);
        float t1 = ray.intersectAABB(bvhNode.aabb[1], hitData.t);

        int M = (t0 != FLOAT_INFINITY) + 2 * (t1 != FLOAT_INFINITY);

        if (M == 3)
        {
            int go_left = t0 < t1;
            stack.push(fetchChildIndex(nodeIndex, go_left));
            nodeIndex = fetchChildIndex(nodeIndex, 1 - go_left);
        }
        else if (M == 1)
        {
            nodeIndex = fetchChildIndex(nodeIndex, 0);
        }
        else if (M == 2)
        {
            nodeIndex = fetchChildIndex(nodeIndex, 1);
        }
        else
        {
            if (stack.empty()) return false;
            stack.pop(nodeIndex);
        }
    }
}
```

8.5.4 Shadow-ray Casting

The `castShadowRay`-kernel is almost equal to the `castRay`-kernel, except for some simplifications. Listing 8.14 shows the CUDA-kernel .

Ray generation

Because the shadow-ray is simply a ray originating from the light-source's origin pointing to the intersection-point, the construction of such a ray is rather trivial. First, the t-parameter is read from the hitData-buffer to determine if we need to cast a shadow-ray. If it indicates a surface-point, the corresponding surface-point and normal is read from the hitData-buffer. In order to prevent self-intersection, the surface-point is slightly offset along the surface-normal, before the ray is constructed. If no shadow-ray needs to be cast, the ray is explicitly set to non-participating, to enable a packet-based traversal.

Traversal

Whereas in ray casting the front-most intersection is required, in shadow-ray casting any intersection will suffice. Consequently, the traversal can be aborted when the root-finder reports an intersection. Nevertheless, due to the packet-based traversal, a ray must continue traversing the scene until all rays have obtained an intersection, which is constantly checked using a parallel reduction.

Inter-kernel communication

Since the surface is unimportant, only a boolean indicating intersection or no intersection will suffice. Therefore, very little global-memory transfers are required to update the hitdata-buffer. Infact, only the t-parameter needs to be overwritten to indicate a non-illuminated surface-point. Because multiple light-sources are supported, setting this t-parameter to ∞ would break the system: when casting a shadow-ray for a second light-source, this parameter would indicate that there was no intersection at all. Therefore, the shadow-ray caster negates the t-parameter in case of an occluder. The shader can easily detect this value by allowing only positive values (and rejecting the value ∞). When processing a shadow-ray, the absolute value is taken from the t-parameter, ∞ values will be skipped, and any other values will result in casting a shadow-ray.

Finally, the shadow-ray caster is not recursive and hence does not require a stack to push rays onto, again reducing global-memory transfers.

Listing 8.14: Shadow-ray Casting kernel

```
__global__ void castShadowRay(int2 tileIdx, int stackpointer, int lightIndex)
{
    ::tileIdx = tileIdx;

    HitData hitData;
    hitDataBuffer.read_t(&hitData);

    if (all<BLOCKSIZE>(hitData.t == INF)) return;

    Ray shadowRay;

    if (hitData.t != FLOAT_INFINITY)
    {
        /* reset the t-value negated by a previous shadow-ray */
        hitData.t = fabs(hitData.t);

        hitDataBuffer.read_s(&hitData);
        hitDataBuffer.read_normal(&hitData);

        Light light(lightIndex);
        shadowRay.o = light.pos;

        hitData.s += hitData.normal * epsilonSelfIntersection;
        shadowRay.d = hitData.s - shadowRay.o;

        normalize(shadowRay.d);
        shadowRay.init();
        hitData.t = shadowRay.distance(hitData.s);
    }
    else
    {
        shadowRay.set_nonparticipating();
    }

    /* Traverse shadow-ray packet */
    traverseShadowRay(shadowRay, hitData);

    hitDataBuffer.write_t(&hitData);
}
```

8.5.5 Shading

The computeShading-kernel is fairly straightforward and is really just the lighting-model written as CUDA-code.

```
__global__ void computeShading(int2 tileIdx, int lightIndex, float4 *outputBuffer)
{
    ::tileIdx = tileIdx;

    float3 color = make_float3(0.0f, 0.0f, 0.0f);

    HitData hitData;
    hitDataBuffer.read_t(&hitData);

    if (hitData.t == INF) return;

    hitDataBuffer.read_hitData(&hitData);

    float3 L = environment.light[lightIndex].position - hitData.s;
    normalize(L);

    float3 R = math::reflect(L, hitData.normal);
    normalize(R);

    float LdotN = dot(L, hitData.normal);
    float RdotV = LdotN > 0.0f ? dot(R, hitData.d) : 0.0f;

    int material_id = MATERIAL_ID(hitData.patch_id.surface_id);

    color += hitData.opacity * sceneParameters.environment_ambient;
    color += hitData.opacity * MATERIAL_DIFFUSE(material_id) * max(0.0f, LdotN);
    color += hitData.opacity * MATERIAL_SPECULAR(material_id)
          * pow(max(0.0f, RdotV), MATERIAL_SHININESS(material_id));

    outputBuffer[SCREEN_OFFSET] += make_float4(color, 1.0f);
}
```

8.6 Kernel-launch configurations

It is not required for the different kernels to share the same block-size. Obviously, the packet-based ray casting kernels should match their packet-size, since they trace their scene packet-wise. The other kernels however, are not packet-based and therefore do not necessarily benefit from having the same block-size as the ray casters. Furthermore, due to the lesser complexity of the generated binary code for the Primary-ray Accelerator, the register usage is lower than the other ray casting kernels. Therefore, a larger block-size could possibly result in a performance increase. In the results-section of this thesis, different block-sizes are examined and a conclusion is drawn on what block-size is ideal for which kernel (Section 9.1.2).

8.6.1 Pixel/Thread mapping

Which pixel, and consequently which rays, will be assigned to which thread depends on the chosen block-sizes, as well as the size of the tiles. All pixels are distributed over several tiles and mapped to a *CUDA-grid*. Each tile, or grid, is divided into equal-sized groups of pixels which are mapped to a *CUDA-block*. Finally, each CUDA-block contains the threads executing the kernel. Figure 8.3 gives a schematic overview of the mapping for an example using an image-resolution of 16×16, a tile-size of 4×4, and a block-size of 2×2.

For the kernels the mapping from pixel to thread is not very interesting. What *is* interesting, is the *inverse-mapping*, which computes the pixel-position in image-space for a thread, based on the `threadIdx`, `blockIdx`, and `tileIdx`. Using the pixel-position, the primary-ray can be setup. Also,

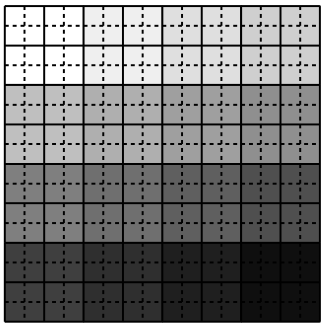


Figure 8.3: Mapping of pixels to threads. This example uses an image-resolution of 16×16, a tile-size of 4×4, and a block-size of 2×2. The tiles are visualized using a shade of grey, the blocks are separated by black lines, and the pixels are separated by dashed lines.

using the same indices, the offsets can be computed for data-buffer such as the image-buffer, the hitdata-buffer, the ray-stack and, as will be seen later, the datastructures in shared-memory.

Rather trivial, the positions and offsets are given by:

$$\begin{aligned}
tile_x &= threadIdx_x + blockIdx_x \cdot blockDim_x \\
tile_y &= threadIdx_y + blockIdx_y \cdot blockDim_y \\
image_x &= tile_x + tileIdx_x \cdot tileDim_x \\
image_y &= tile_y + tileIdx_y \cdot tileDim_y \\
block_{offset} &= threadIdx_x + threadIdx_y \cdot blockDim_x \\
tile_{offset} &= tile_x + tile_y \cdot tileDim_x \\
image_{offset} &= image_x + image_y \cdot imageDim_x
\end{aligned}$$

the *tileDim* and *imageDim* variables represent the tile-size and image-size, respectively.

Part III

Results

Chapter 9

Results

In this chapter, the runtime performance of the CUDA NURBS Ray Tracing System, described in previous chapters, will be examined. In Section 9.1, the procedure will be described how the system is tested. Section 9.2 will discuss the impact of several parameters on the quality of the resulting renderings. Section 9.3, will present the performance statistics, which is used to compare against other implementations. Then, Section 9.4 will investigate where the bottleneck is located, and where improvements might me possible. This chapter will conclude with a discussion about this system, comparing it with existing systems, and suggesting possible improvements.

9.1 Experimental Setup

In the tests, the two different traversal algorithms are being investigated, namely packet-based traversal and single-ray traversal (described in Chapter 8). For each algorithm, a scene is rendered without acceleration, using the standard ray tracer, and with acceleration, using the hybrid ray tracer. Finally, every kernel is compiled and tested with caching and without caching (described in Section 8.3.1). In total, 8 different variants will be tested: packet, packet_cache, single, single_cache, hybridpacket, hybridpacket_cache, hybridsingle, and hybridsingle_cache. For these variants, 6 kernels are produced: accelerator, accelerator_cache, packet, packet_cache, single, and single_cache. Additionally, each kernel will be further fine-tuned to determine a good blocksize/occupancy ratio (Section 9.1.2).

9.1.1 Testing Platforms

Two platforms have been used for obtaining the test-results. The first one is the slightly outdated NVIDIA GeForce 8800M GTX mobile GPU. The second one is the NVIDIA GeForce GTX 295, which has some nice improvements compared to the other. Not only is the per-multiprocessor occupancy higher due to the higher register count, but also more threads are processed physically concurrent, since the number of multiprocessors is 30 per GPU. Unfortunately, to exploit both GPUs, the system must support SLI (Section 5.1), which it doesn't. Table 9.1 summarizes the most important differences between the two platforms/architectures.

9.1.2 Fine-tuning

When hand-optimizing a kernel, it is important that three goals are kept in mind: first, divergence should be minimized by determining a good block size. Second, the occupancy should be maximized, in order to exploit the parallelism of the GPU. And third, memory latencies should be avoided by reducing the local-memory usage.

	8800M GTX	GTX 295
Release Date	19 November 2007	January 8, 2009
Compute Capability	1.1	1.3
CUDA Cores	96	480 (**240**/GPU)
Core Clock (MHz)	500	576 MHz
Memory Clock (MHz)	800	1998 (**999**/GPU)
Maximum Memory (MB)	512	1792 (**896**/GPU)
Memory Interface	256-bit	896-bit (**448-bit**/GPU)
Mem. Bandwidth (GB/s)	51.2	223.8 (**111.9**/GPU)
GFLOPS	360.0	1788.48 (**894.24**/GPU)
Operating System	Linux: Ubuntu 9.04	Windows Vista x64
CUDA Driver	2.3	2.3
CUDA Toolkit	2.3	2.3

Table 9.1: Testing platforms. The entries in bold denote the statistics per GPU. Since SLI is currently not supported, the performance is limited to only one GPU.

The block size also impacts occupancy. Although choosing a very small block size increases coherence, it will also decrease the occupancy. For kernels with a relatively large, fixed shared-memory use, the shared-memory will become the limiting factor. As the block size decreases, more block will be necessary to maximize the occupancy, however this will not be possible when the total amount of shared-memory required isn't available. Regarding the third goal, increasing the maximum number of registers a kernel may use, will lower the local-memory usage, since too much used registers will be spilled to the slow local-memory (which is located in global-memory). However, when using too many registers, the occupancy (goal one) will decrease.

Therefore, a balance must be found between the block size, and the maximum number of registers. Additionally, the dimension of the block will also influence the coherence among rays. However, it looks like square-shaped blocks do not imply the maximum coherence as preliminary tests have shown. Therefore, also the block dimension must be determined.

Determination of the best, most general options In order to determine which blockdim/maxregcount-combination is best, every reasonable combination is tried and the performance is measured for several scenes. Per scene, the ratio of the best combination and every other combination is computed to provide a normalized performance indication for all blockdim/maxregcount-combinations. Then, the averages are computed for each combination, which is then sorted. By choosing the blockdim/maxregcount-combination with the highest ratio, a kernel will be generated which deviates the least from the optimum blockdim/maxrregcount for an average scene.

Chosen options Table 9.2 gives the resulting blockdim/maxregcount-combinations used in the rest of this report. While the accelerator favors square-shaped block sizes, which is as expected since no real divergence can occur, the other kernels do not.

9.1.3 Test scenes

Several NURBS-scenes have been used to produce the results in this chapter. Table 9.3a lists some simple scenes containing only a few surfaces/patches. The degree varies from 1 to 5. Figure 9.1a shows the images corresponding to the scenes.

		8800M			GTX 295		
		dim	regs	occ	dim	regs	occ
Uncached	Accelerator	16×8	64	17%	32×8	64	25%
	Packet	2×32	32	33%	1×64	64	25%
	Single	1×64	40	25%	1×64	80	19%
Cached	Accelerator	$\begin{cases} 16 \times 8 & \text{if } d < 7 \\ 16 \times 4 & \text{otherwise} \end{cases}$	64	17%	8×8	92	13%
	Packet	2×32	64	17%	1×64	86	13%
	Single	2×32	64	17%	1×64	94	13%

Table 9.2: Chosen compiler-options. "dim" denotes the block dimension, "regs" denotes the maximum number of registers a kernel is allowed to use. "occ" denotes the occupancy of the kernel using these parameters, with a maximum degree of 3.

The scenes in Table 9.3b depict some well-known scenes often used for comparison. All scenes are of degree 3. Figure 9.1b shows the images corresponding to the scenes.

And finally, Table 9.3c shows some scenes with increasing number of patches. All scenes are of degree 3. Figure 9.1c shows the images corresponding to the scenes.

9.2 Image Quality

This section will discuss the impact of several parameters on the quality of the resulting renderings.

9.2.1 Hybrid Artifacts

This subsection discusses some scenes rendered using the hybrid technique, of which some were rendered correctly, and others were rendered with noticeable artifacts.

Figure 9.2 shows three renderings of the teapot scene. The left image shows the scene after it has been processed by the accelerator. Although there are some very tiny artifacts, they are hardly visible. Therefore, the teapot does not really require any further processing for the primary rays. Obviously, accelerator-artifacts are related to the fineness chosen for the scene, however, it does show that as long as no secondary rays and shadow rays are required, the accelerator alone is sufficient.

Although the teapot-scene can be rendered properly using the accelerator, other scenes cannot be rendered without noticeable artifacts. Figure 9.3a shows three renderings of the head-scene, in which the accelerator clearly fails to produce an artifact-free result. Apart from the "holes" in the model, which are caused by the linear approximation having a small offset from the surface, the model suffers from an incorrect tessellation. In this scene, not every control-point interpolates the surface, therefore, the tessellation will be far off from the surface at the domain boundaries, since the convex-hull is used for tessellation. Fortunately, these artifacts are all handled correctly by the ray tracer, which fixes these erroneous pixels.

Figure 9.3b, on the other hand, shows the same scene from a different view-point. In this view, a strange gap appears in the chin. Because the curvature of the surface around the chin is very high, the tessellation becomes too coarse to represent a smooth surface. To represent this part of the surface, a very high fineness would be required, which increases the size of the BVH as well. Ultimately, the artifact will become visible again when zooming in too close. Since the rasterization provided the accelerator the wrong surface, the ray tracer is not able to check if the accelerator converged correctly. Therefore, the hybrid result still contains this incorrect surface.

	cornellbox	head	ducky
Fineness	5	2	2
Surfaces	19	4	5
Patches	33	508	124
Control-points	158	969	602
Degree	1-2	2-5	2-5
BVHNodes	5776	10745	11246
BVH Depth	14	14	15
Quads	5777	10746	11247
Vertices	6452	16556	13977
Preprocessing time	0.13s	0.25s	0.25s
NURBS data	5.5 kB	21.0 kB	15.5 kB
BVH data	631.8 kB	1.2 MB	1.2 MB
Tessellation data	291.9 kB	685.3 kB	612.5 kB
Screen fill	100.00%	35.60%	27.30%

(a) Scenes containing several surfaces.

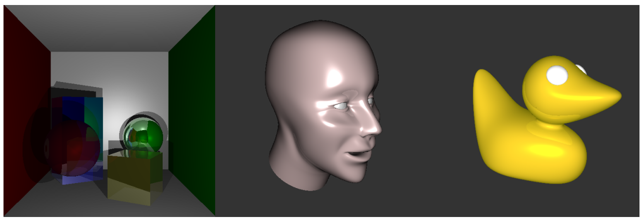
	teapot	killeroo-lowres	killeroo-highres	killeroo-closeup
Fineness	5.55	1.03	0.79	0.79
Surfaces	4	89	89	89
Patches	32	11532	46128	46128
Control-points	358	17181	56625	56625
Degree	3	3	3	3
BVHNodes	13089	104248	130952	130952
BVH Depth	15	18	18	18
Quads	13090	104249	130953	130953
Vertices	14444	184669	330891	330891
Preprocessing time	0.28s	2.76s	3.98s	3.98s
NURBS data	9.2 kB	372.1 kB	1.1 MB	1.1 MB
BVH data	1.4 MB	11.1 MB	14.0 MB	14.0 MB
Tessellation data	655.9 kB	7.23 MB	12.1 MB	12.1 MB
Screen fill	21.16%	14.10%	14.10%	92.30%

(b) Benchmark scenes for comparison with other systems.

	teapot-1^3	teapot-2^3	teapot-3^3	teapot-4^3	teapot-5^3
Fineness	5.55	5.55	5.55	5.55	5.55
Surfaces	4	32	108	256	500
Patches	32	256	864	2048	4000
Control-points	358	2864	9666	22912	44750
Degree	3	3	3	3	3
BVHNodes	13089	104719	353429	837759	1636249
BVH Depth	15	18	20	21	22
Quads	13090	104720	353430	837760	1636250
Vertices	14444	115552	389988	924416	1805500
Preprocessing time	0.28s	2.52s	6.52s	16.15s	32.28s
NURBS data	9.2 kB	72.9 kB	245.8 kB	582.6 kB	1.1 MB
BVH data	1.4 MB	11.2 MB	37.8 MB	89.5 MB	174.8 MB
Tessellation data	655.9 kB	5.1 MB	17.3 MB	41.0 MB	80.1 MB
Screen fill	28.50%	28.70%	28.20%	28.10%	28.60%

(c) Scenes containing many surfaces.

Table 9.3: Test scenes used for the results.


(a) Scenes containing several surfaces. From left to right: cornellbox, head, ducky.

(b) Benchmark scenes for comparison with other systems. From left to right: teapot, killeroo-lowres, killeroo-highres, killeroo-closeup.

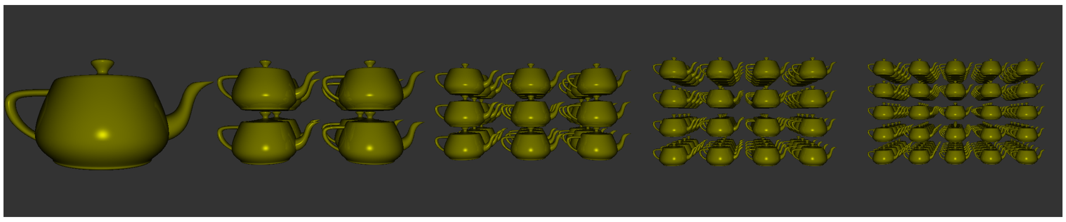

(c) Scenes containing many surfaces. From left to right: teapot-1x1x1, teapot-2x2x2, teapot-3x3x3, teapot-4x4x4, teapot-5x5x5.

Figure 9.1: Test scenes used.

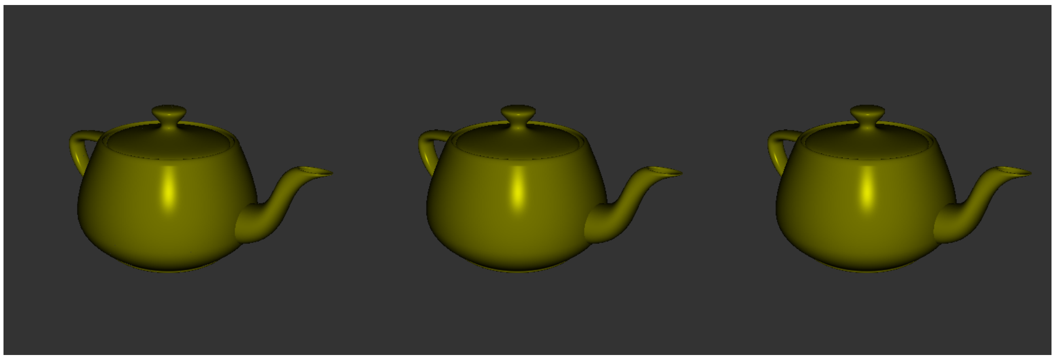
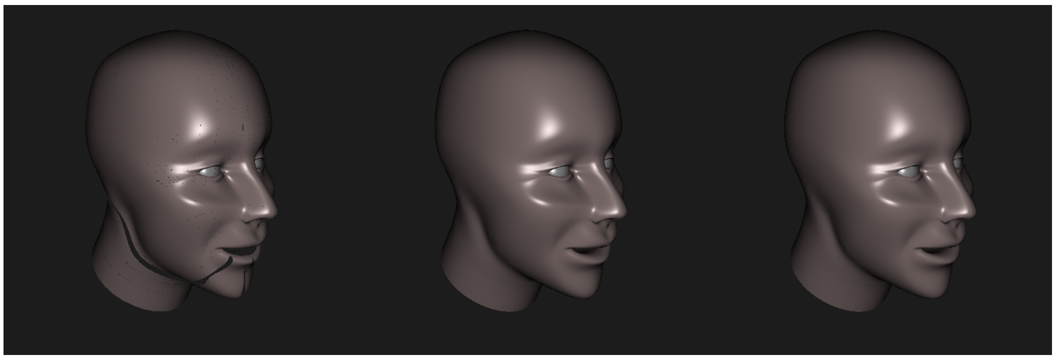
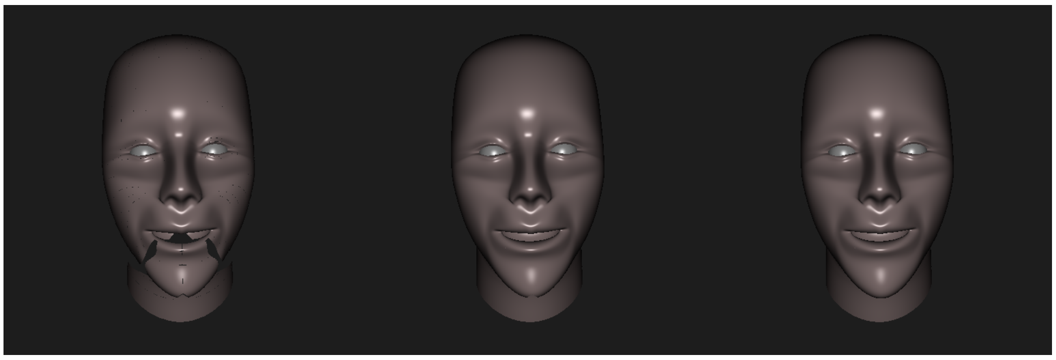


Figure 9.2: teapot-scene: accelerator (left), full hybrid (middle), standard ray traced (right)



(a) Fixable artifact in the head-scene: accelerator-only (left), hybrid (middle), ray traced reference (right).



(b) Non-fixable artifact in the head-scene: accelerator-only (left), hybrid (middle), ray traced reference (right)

Figure 9.3: head-scene

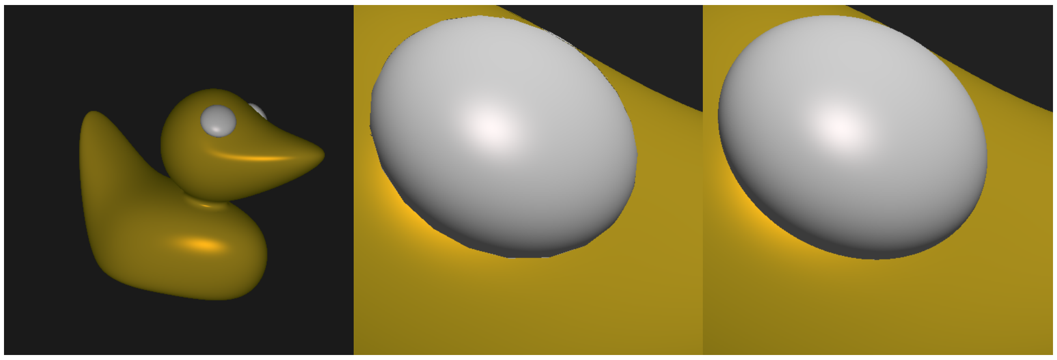


Figure 9.4: This image shows a close-up of the ducky-scene: hybrid (left), and normally ray traced (right). It seems that the hybrid method can sometimes introduce artifacts, when zooming in too close.

min. #iterations	Frame rate (FPS)	Quality of result
3	2.2	Correct
2	2.5	Few holes
1	2.6	Small holes
0	1.9	Long Gaps

Table 9.4: Different epsilon values.

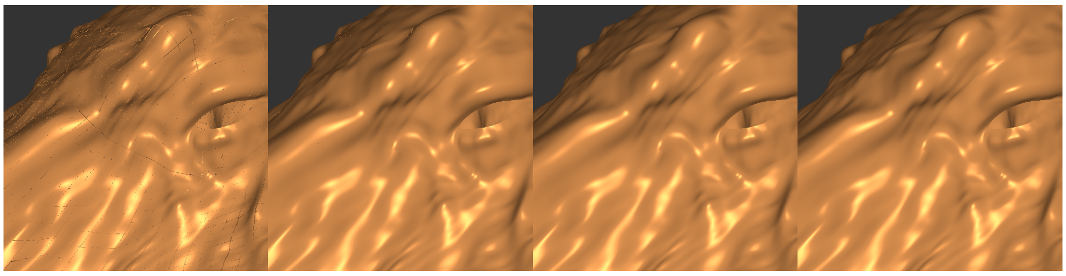


Figure 9.5: Killeroo-closeup-scene rendering with a increasing *miniterations*-value. The images correspond to the *miniterations*-values in Table 9.4.

Zooming in on the eye in Figure 9.4 will reveal the tessellation, which is not fixable using hybrid ray tracing. Nevertheless, when viewed from some distance, the hybrid result appears correct.

9.2.2 Min-iterations switch

Sometimes, the root-finder may incorrectly return a miss, for example if the iteration jumps out of the patch's domain. However, sometimes this already happens during the first iteration. Usually, it's only the first iteration that tends to have a somewhat larger uv-increment, therefore, the system includes an extra parameter, which pushes the uv-parameter back into the patch's domain as long as the minimum number of iterations has not yet been reached.

Although less iterations should be preferred, it seems to increase performance a bit when specifying a minimum of one iteration.

9.3 Performance

In this section the performance of the ray tracing system will be examined. The different variants will be investigated to determine which variant performs the best on average. Finally, the system will be compared against existing NURBS ray tracing systems.

9.3.1 Primary Rays

As can be seen clearly from Table 9.5 and Table 9.6, as well as Table 9.7, the single-ray variant almost always outperforms the packet-based variant, using both architectures. What's interesting, is that the uncached version of the single-ray variant is not much slower than its cached brother, which is only about 14-17% faster on average (Figure 9.6). On the GTX 295 architecture however, it is actually faster without the cache (for standard ray tracing). On the other hand, the packet-based variant does benefit from the cache, increasing the average performance with 30-53% for 8800M and 27-37% for GTX 295.

Scene	Standard (ms)		Hybrid (ms)		
	Packet	Single	Accelerator	Packet/Total	Single/Total
cornellbox	163.89 **27.90**	129.13 **21.06**	0.91+16.75 **0.84+4.71**	22.36/40.02 **4.64/10.20**	28.28/45.94 **5.05/10.61**
cached	116.26 **20.84**	117.44 **23.73**	0.91+14.21 **0.84+3.94**	16.42/31.54 **4.11/8.90**	20.34/35.46 **4.99/9.77**
ducky	449.43 **89.44**	288.38 **47.42**	0.95+31.25 **0.87+6.86**	265.66/297.86 **50.19/57.92**	231.65/263.86 **44.98/52.71**
cached	260.20 **71.88**	240.77 **49.76**	0.95+26.28 **0.87+6.29**	169.63/196.87 **30.93/38.09**	223.94/251.18 **37.93/45.09**
head	517.43 **112.22**	406.29 **67.52**	2.11+44.17 **0.89+8.89**	345.66/391.94 **81.26/91.05**	323.85/370.13 **62.77/72.56**
cached	291.03 **82.94**	309.22 **63.71**	2.11+36.96 **0.89+8.05**	224.68/263.76 **49.88/58.82**	261.91/300.98 **49.91/58.85**

Table 9.5: Timings in milliseconds for the primary rays of some small scenes per variant. The hybrid section shows two durations: the first one denotes the duration for tracing rays which were missed during acceleration. The second duration denotes the total duration for hybrid ray tracing (acceleration+ray tracing). The accelerator is separated into rasterization+acceleration. The bold items are the timings from the GTX 295 test-system, the other timings are from the 8800M test-system.

This suggests that the majority of execution-time is spent somewhere else, where the use of cache is not possible. In Section 9.4 the variants will be investigated more thoroughly.

9.3.2 Secondary Rays

Observing the diagrams in Figure 9.7, Figure 9.8, and Figure 9.9, it becomes clear that for secondary rays, the single-ray variant is again superior to the packet-based variant. While the diagrams for shadow-rays show that the use of cache is even more beneficial in the case of the 8800M test-system, all other diagrams suggest there is little advantage in using the cached variant above the uncached. This is generally a good sign, as future architectures may allow more threads to run concurrently. In order to maximize the occupancy, the shared-memory use should be kept to a minimum. In this case, no cache would be used, hence this cannot become a limiting factor.

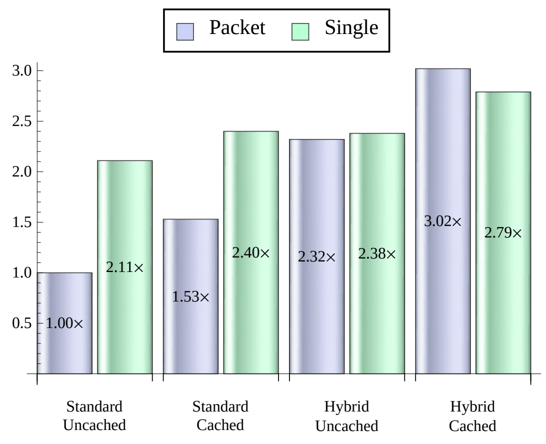
9.3.3 Scene size

A first look at the results of the massive teapot scenes (Table 9.7) reveals a logarithmic relation between the number of BVH-nodes in a scene and the time to render (Figure 9.10 and Figure 9.11). Although this is characteristic for ray tracing systems in general, the results of the killeroo-models (Table 9.6) seem to suggest that the performance of the ray tracing system is not directly related to the number of BVH-nodes. The high-resolution model, which contains a lot more control-points and BVH-nodes, does not perform worse than the low-resolution model; sometimes it even outperforms the low-resolution model. This might be explained by the fact that the increase in the number of BVH-nodes does not create a new level in the BVH. However, it does create smaller leaf-nodes, decreasing the number of intersection-tests returning a miss.

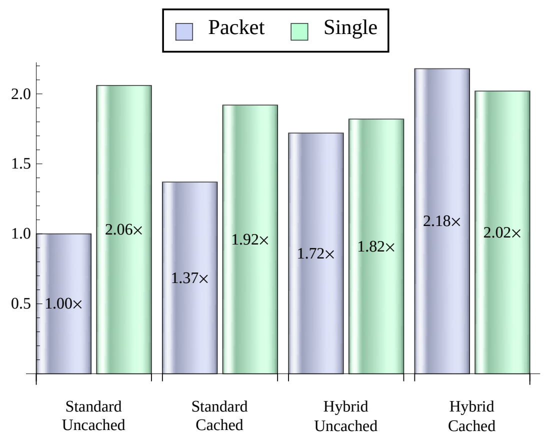
When comparing the durations of the accelerator for the low-resolution and the high-resolution killeroo-model scenes, as well as the massive teapot scenes, it becomes clear that the number of

Scene	Standard (ms)		Hybrid (ms)		
	Packet	Single	Accelerator	Packet/Total	Single/Total
teapot	231.57	142.96	0.86+19.25	121.29/141.40	129.17/149.28
	60.53	**34.76**	**0.85+5.28**	**32.88/39.01**	**30.43/36.55**
cached	140.89	117.75	0.86+14.83	92.25/107.93	106.99/122.67
	40.31	**35.78**	**0.85+4.85**	**21.49/27.19**	**25.03/30.73**
killeroo-lowres	727.55	245.16	4.68+22.19	206.60/233.46	198.97/225.84
	120.16	**41.41**	**1.82+5.96**	**41.26/49.04**	**35.12/42.90**
cached	451.56	206.45	4.68+17.20	165.47/187.35	169.29/191.17
	81.26	**45.37**	**1.82+6.10**	**28.23/36.15**	**31.29/39.20**
killeroo-highres	809.36	229.85	7.09+23.21	231.71/262.00	194.05/224.35
	120.60	**41.58**	**4.90+6.13**	**41.56/52.58**	**35.43/46.45**
cached	499.88	191.16	7.09+17.85	169.37/194.30	156.62/181.55
	85.00	**42.85**	**4.90+5.83**	**30.44/41.16**	**30.62/41.35**
killeroo-closeup	324.34	326.97	9.47+58.37	168.00/235.84	190.09/257.93
	61.39	**44.88**	**4.91+11.01**	**32.77/48.69**	**30.11/46.03**
cached	227.17	253.88	9.47+43.43	139.63/192.53	165.26/218.16
	45.74	**47.69**	**4.91+8.30**	**28.59/41.80**	**27.33/40.53**

Table 9.6: Timings in milliseconds for the primary rays of the benchmark-scenes per variant. The hybrid section shows two durations: the first one denotes the duration for tracing rays which were missed during acceleration. The second duration denotes the total duration for hybrid ray tracing (acceleration+ray tracing). The accelerator is separated into rasterization+acceleration. The bold items are the timings from the GTX 295 test-system, the other timings are from the 8800M test-system.

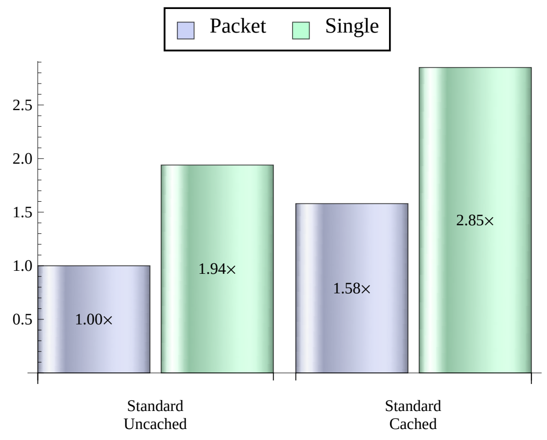


(a) 8800M

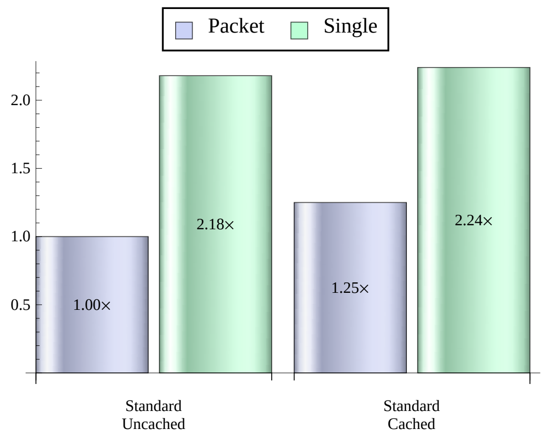


(b) GTX 295

Figure 9.6: Average speedup of primary rays w.r.t. reference implementation (packet-based, no caching, no acceleration).

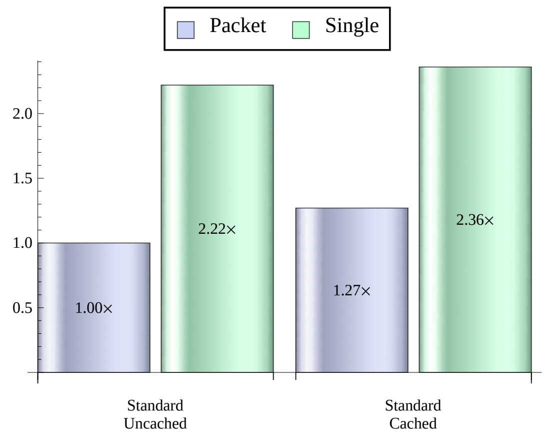


(a) 8800M

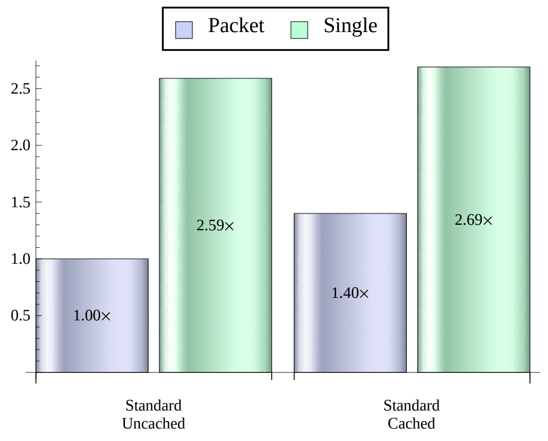


(b) GTX 295

Figure 9.7: Average speedup of shadow rays (level 0) w.r.t. reference implementation (packet-based, no caching).

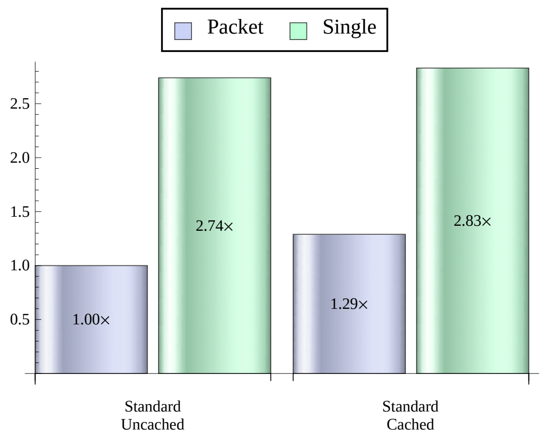


(a) 8800M

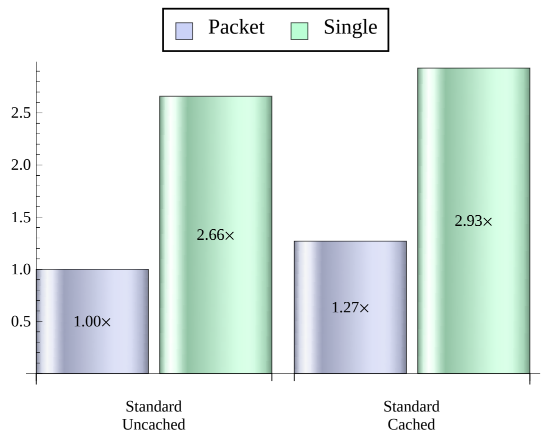


(b) GTX 295

Figure 9.8: Average speedup of reflection rays (level 1) w.r.t. reference implementation (packet-based, no caching).



(a) 8800M

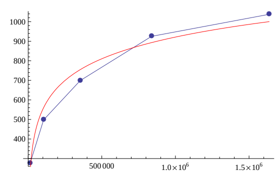


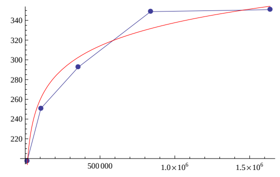
(b) GTX 295

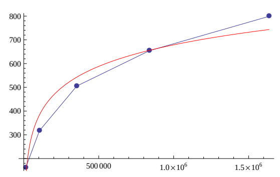
Figure 9.9: Average speedup of refraction rays (level 1) w.r.t. reference implementation (packet-based, no caching).

Scene	Standard (ms)		Hybrid (ms)		
	Packet	Single	Accelerator	Packet/Total	Single/Total
teapot-1x1x1	276.02	197.52	0.89+17.77	143.13/161.79	165.68/184.34
	54.23	**30.98**	**0.87+5.52**	**27.00/33.39**	**28.63/35.01**
cached	161.62	148.36	0.89+13.36	81.49/95.74	116.45/130.70
	36.42	**32.59**	**0.87+5.55**	**19.13/25.54**	**24.85/31.26**
teapot-2x2x2	497.88	250.74	4.66+26.31	209.74/240.71	225.97/256.94
	82.14	**41.74**	**1.67+5.70**	**38.21/45.59**	**37.91/45.28**
cached	317.40	220.85	4.66+19.74	150.71/175.12	192.33/216.74
	56.35	**44.96**	**1.67+4.92**	**29.83/36.42**	**36.27/42.86**
teapot-3x3x3	698.79	292.41	12.46+25.25	279.60/317.31	263.72/301.42
	100.92	**46.78**	**9.41+5.48**	**41.31/56.19**	**41.76/56.64**
cached	505.26	277.16	12.46+19.11	193.85/225.43	229.09/260.67
	80.88	**52.37**	**9.41+4.83**	**36.93/51.16**	**40.69/54.93**
teapot-4x4x4	925.00	348.83	28.17+26.53	343.61/398.30	287.33/342.02
	128.80	**58.36**	**39.68+6.35**	**51.11/97.13**	**53.39/99.42**
cached	654.11	345.03	28.17+20.89	276.56/325.61	268.93/317.99
	98.98	**62.14**	**39.68+6.97**	**45.36/92.01**	**43.01/89.66**
teapot-5x5x5	1037.69	350.81	43.94+27.05	298.20/369.19	287.21/358.20
	161.27	**57.39**	**89.45+8.27**	**56.73/154.45**	**47.17/144.89**
cached	798.94	348.87	43.94+20.95	270.19/335.08	274.00/338.89
	125.80	**68.86**	**89.45+8.54**	**59.35/157.34**	**48.88/146.87**

Table 9.7: Timings in milliseconds for the primary rays of the large scenes per variant. The hybrid section shows two durations: the first one denotes the duration for tracing rays which were missed during acceleration. The second duration denotes the total duration for hybrid ray tracing (acceleration+ray tracing). The accelerator is separated into rasterization+acceleration. The bold items are the timings from the GTX 295 test-system, the other timings are from the 8800M test-system.


(a) packet, nocache


(b) single, nocache


(c) packet, cached

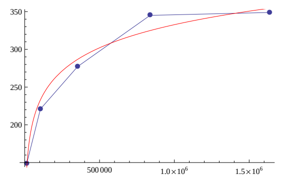
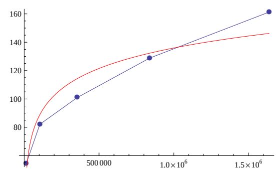
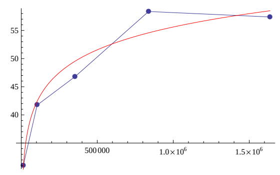
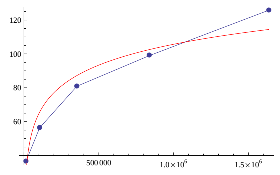

(d) single, cached

Figure 9.10: Durations of the variants using the 8800M test-system, as a function of the number of patches. The red line shows the curve corresponding to a logarithmic function fitted to the data.


(a) packet, nocache


(b) single, nocache


(c) packet, cached

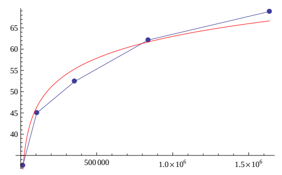

(d) single, cached

Figure 9.11: Durations of the variants using the GTX 295 test-system, as a function of the number of patches. The red line shows the curve corresponding to a logarithmic function fitted to the data.

	Teapot	Killeroo-highres	Head
Direct evaluation	4.9	4.5	3.0
Division-free evaluation	4.7	4.3	2.8

Table 9.8: Performance in FPS for different evaluation methods. The results were obtained using the cached single-ray ray tracer on the 8800M test system.

control-points in a scene or the number of patches does not influence the performance at all, which is actually quite obvious, since each pixel contains up to one patch to find a root. The rasterization however, is influenced by the number of patches, since it's linear in the number of sub-patches. Therefore, the performance is independent on the number of control-points as long as the rasterization is fast enough. An interesting detail, is that starting with the teapot-3x3x3 scene, the GTX 295-test-system using the standard *uncached* single-ray variant begins to outperform every other hybrid variant. Starting with the teapot-5x5x5 scene, it even outperforms the rasterization phase, making the hybrid approach obsolete. However, I doubt the maximum rasterization throughput has really been reached, since that part was not fully optimized yet, especially since the 8800M should perform worse.

9.3.4 Surface-Evaluation Scheme

As it turned out, the direct evaluation method in slightly faster than the division-free evaluation method, which is clearly visible in Table 9.8.

	Standard			Hybrid
Scene	P	P+S	P+S+1	P
cornellbox	7.5/**34.6**	3.0/**16.7**	1.8/**10.1**	26.0/**59.8**
head	3.1/**13.2**	2.0/**8.7**	0.7/**3.2**	4.8/**15.0**
ducky	4.0/**18.3**	2.6/**11.6**	1.0/**4.9**	6.5/**22.0**
teapot	5.4/**24.5**	3.5/**17.3**	1.3/**6.5**	8.7/**30.1**
killeroo-lowres	4.6/**20.5**	3.0/**13.6**	1.0/**4.9**	6.6/**23.0**
killeroo-highres	4.9/**20.5**	2.2/**13.4**	0.5/**5.0**	6.2/**20.7**
killeroo-closeup	3.7/**19.2**	2.9/**12.4**	1.0/**2.9**	5.9/**20.4**
teapot-1x1x1	5.5/**28.2**	4.2/**20.5**	1.6/**8.0**	9.0/**33.3**
teapot-2x2x2	4.3/**21.4**	1.8/**14.7**	0.6/**4.8**	6.6/**24.1**
teapot-3x3x3	3.5/**18.7**	1.7/**12.2**	0.6/**3.8**	5.1/**17.2**
teapot-4x4x4	2.8/**15.4**	3.0/**10.0**	1.8/**3.2**	3.9/**10.2**
teapot-5x5x5	2.8/**15.5**	2.9/**10.1**	1.7/**3.2**	2.2/**6.1**

Table 9.9: Overview of the performance of the ray tracing system. The numbers represent the performance in frames per second. P denotes the primary rays, S denotes the shadow rays (level 0), and 1 denotes the level-1 secondary rays (reflection+refraction+shadow). The bold entries denote the frame rates for the GTX 295 test-platform.

9.3.5 Preferred variant

Based on the observations in the previous subsections, Table 9.9 gives an overview of the performance data of the system using the overall-best variants. In here, the single-ray implementation using the cache has been selected as the preferred variant for standard ray tracing on the 8800M test platform, and the uncached version on the GTX 295 platform. For hybrid ray casting, the cached version of the packet-based implementation has been selected, for both test platforms.

9.3.6 Comparison

Table 9.10 and Table 9.11 give a comparison with two other existing CPU-based direct NURBS ray tracing systems, described in [AGM06, ABS08]. Both systems implement the division-free evaluation method described in Section 3.5.3, and employ a packet-based scheme for scene-traversal. In addition, the system described in [ABS08] implements the hybrid method described in Section 4.2.2 and has a variant which uses 8 cores in parallel. Both systems use SIMD to execute 4 rays in parallel, per core. The performance of the 8800M test platform seems to outperform the single-core variants of the systems. Although the frame rate of the teapot-scene of the system described in [AGM06] is much higher than in [ABS08], the frame rate for the high-resolution killeroo scene is the same. Furthermore, the impact of the number of patches, as well as the screenfill is much higher for the CPU-based systems, than for the GPU-based system (CNRTS), suggesting the system is better suited for massive models.

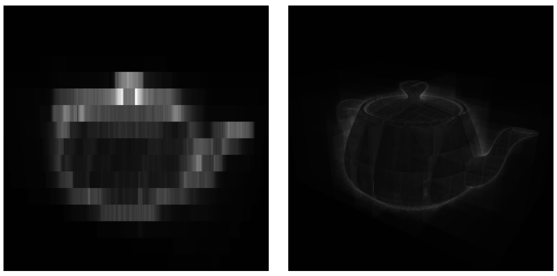
The GTX 295 platform performs much better, with a speedup of 4-5×. However, the CPU-based ray tracing system is still much faster when using 8 cores, although the frame rate is not much higher for the killeroo-scene. For hybrid ray tracing, the ID-processing variant of the CPU-based ray tracing system is much faster than the GPU-based system, while the uv-processing variant is comparable.

	CNRTS	[AGM06]	[ABS08] (Standard)	
Scene		1 core	1 core	8 cores
teapot	5.0/**24.5**	7.2	5.2	34.2
killeroo-lowres	4.6/**20.5**	3.8	-	-
killeroo-highres	4.9/**20.5**	3.3	3.3	22.4
killeroo-closeup	3.7/**19.2**	1.5	-	-

Table 9.10: Primary-ray frame rate comparison of several existing systems for *standard* NURBS ray tracing. The bold entries denote the frame rates of the GTX 295 test platform. The selected variants, are the single-ray cached variant for the 8800M platform, and the single-ray uncached variant for the GTX 295 platform.

	CNRTS	[ABS08] (Hybrid)		
		1 core	8 cores	
Scene			*ID*-processing	*u*/*v*-processing
teapot	9.0/**30.1**	8.0	41.7	25.5
killeroo-lowres	6.6/**23.0**	-	-	-
killeroo-highres	6.2/**20.7**	5.9	30.8	22.1
killeroo-closeup	5.9/**20.4**	-	-	-

Table 9.11: Primary-ray frame rate comparison of several existing systems for *hybrid* NURBS ray tracing. The bold entries denote the frame rates of the GTX 295 test platform. The selected variant, for both platforms, is the packet-based cached variant.



(a) Packet-based (b) Single-ray

Figure 9.12: Number of BVH-nodes visited.

9.4 Analysis

In this section, the variants are compared and analyzed to identify areas with room for improvement.

9.4.1 Traversal

The total number of BVH-nodes visited by each ray during traversal, is different for each variant. Figure 9.12 gives a visualization which compares the number of nodes visited per ray for the teapot-scene. It's clear that each ray in the same packet visits the same number of nodes. Also, Figure 9.12b indicates that the maximum number of visited nodes using the single-ray variant is much smaller than the packet-based variant.

	Single	Packet
cornellbox	14.6M (181.2k)	19.6M (181.3k)
head	4.2M (1.5M)	9.9M (2.7M)
ducky	2.9M (943.7k)	8.5M (1.9M)
teapot	2.6M (1.1M)	7.8M (2.1M)
killeroo-lowres	2.2M (932.2k)	14.1M (2.6M)
killeroo-highres	2.2M (952.0k)	15.3M (2.7M)
killeroo-closeup	8.3M (602.1k)	12.9M (902.8k)
teapot-1x1x1	2.9M (1.2M)	6.8M (2.3M)
teapot-2x2x2	3.6M (1.4M)	14.1M (3.4M)
teapot-3x3x3	4.6M (1.8M)	25.7M (4.3M)
teapot-4x4x4	4.3M (1.5M)	34.6M (5.1M)
teapot-5x5x5	5.9M (2.3M)	51.0M (6.3M)

Table 9.12: Total number of visited BVH nodes for every scene. The numbers between curly brackets denote the number of visited nodes when using hybrid ray tracing.

Table 9.12 lists the cumulative number of visited nodes for each scene, comparing the single-ray variant and the packet-based variant, as well as their hybrid versions. It is evident, that for every scene, the packet-based variant visits a lot more nodes than the single-ray variant. Note that zooming in on a model, reduces the number of visits for the packet-based variant (compare killeroo-highres and killeroo-closeup). This is of course caused by the fact that zooming in on the model, will enlarge the bounding-boxes from the packet's perspective. Consequently, less surface-area of the bounding-box will be intersected by the packet's volume, resulting in fewer intersected nodes. Although the screenfill is higher, it turns out to be more efficient for packet-based traversal (observe the smaller execution time in Table 9.6). The single-ray variant on the other hand, requires much more node visits for the close-up scene.

Also observe, that for increasing BVHs (teapot-scenes), the number of visited nodes for the packet-based variant increases rapidly w.r.t. the single-ray variant. This is easily explained by the fact that the increase in number of nodes in the hierarchy, combined with the same screenfill used (zoomed to fit the screen), will shrink the size of the bounding-boxes, again from the perspective of the rays. Therefore, the opposite is expected of zooming in on the scene, namely, that the number of visited nodes increases. This is confirmed by the table.

9.4.2 Root-finding

Table 9.13 shows statistics about the root-finder. Although the number of visited nodes is much larger for the packet-based variant, than the single-ray variant, the total number of intersection tests and iterations is roughly the same. This is caused by the extra Ray-AABB intersection test performed just before the root-finder stage. However, the execution times per iteration actually differ a lot. Although both variants use *caching* to keep the computed basis-functions, the packet-based variant performs much better.

First, during a root-finding call, every ray in the packet, when participating in the root-finding, will test against the same surface-patch. Therefore, the data-coherence will be perfect: each thread requires the exact same data. Moreover, since the data is first transferred to the shared-memory, which acts again as a software-managed cache, the memory-latencies will be completely hidden.

The single-ray variant will fetch its data through the texture-unit. Although this will be cached as well, it proves to be more efficient to put it into shared-memory. Furthermore, since the rays may

	Intersection tests	Iterations		Newton-iteration	
	Cumulative	Cumulative	Average	Cumulative	Average
Packet	123,042	313,993	2.6	$9.0 \cdot 10^9$	28,589
Single	119,408	304,833	2.6	$22.6 \cdot 10^9$	74,210
Accelerator	55,382	92,201	1.7	$1.1 \cdot 10^9$	12,328
Packet (Hybrid)	16,790 (72,172)	44,049	2.6	$1.3 \cdot 10^9$	29,749
Single (Hybrid)	16,745 (72,127)	43,923	2.6	$3.0 \cdot 10^9$	68,060

Table 9.13: Root-finder statistics for the teapot-scene (cached). The Newton-iteration numbers denote durations in GPU clock-cycles.

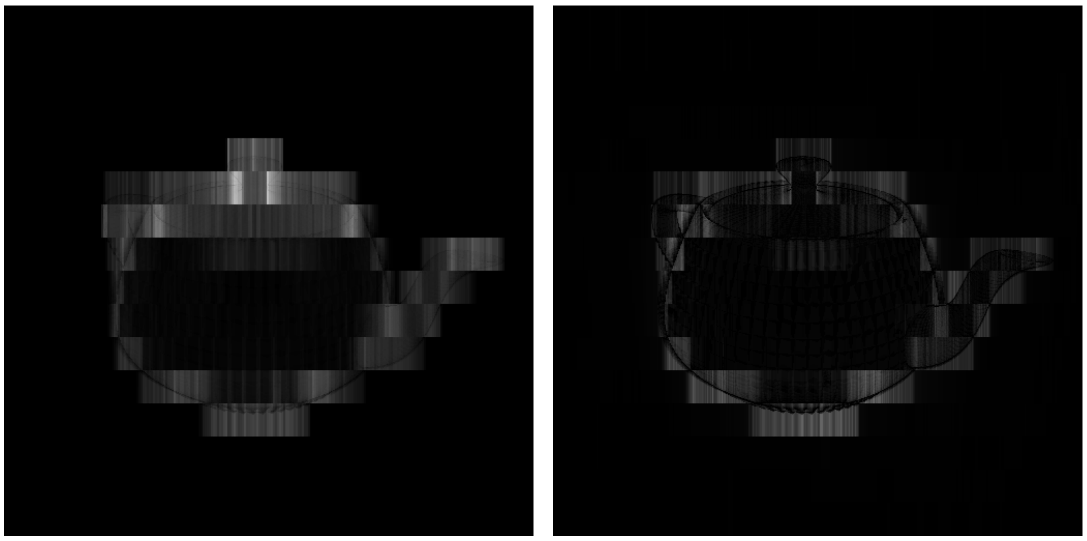
fetch different surface-patches, the memory-coherence will be much less than the packet-based variant. Also, because the data may be scattered in the texture, fetches will result in uncoalesced accesses the first time, which is very inefficient. The packet-based variant even has one extra advantage, in that the data will be fetched coalesced by all threads cooperatively.

Unfortunately, for the packet-based variant, the single-ray variant will eventually win for other reasons, which will be investigated in the next subsection.

9.4.3 Utilization

Apart from root-finding being a very expensive computation, a poor utilization of the multiprocessors will make it perform even worse. Ideally, all cores of a multiprocessor should be executing, to reach the full potential. However, this is not always the case. During packet-based traversal, some rays may intersect a leaf-node, and others do not, in which case only a subset of the packet will be participating in the root-finding. The other cores are shut-down for a moment, until every ray's root-finder-call has finished executing, after which all rays in the packet continue their execution. Generally, the utilization refers to that of a warp. However, since we are traversing packet-wise, the utilization now covers all warps in the packet. In the worst case, if only one ray is active, the entire packet will be blocked until that ray is done. Unfortunately, this is the price we have to pay for the advantage of having a fast shared-stack and a very efficient node/patch fetch implementation. Figure 9.13a shows a visualization of the total idle-time per pixel for the teapot-scene rendered using the packet-based variant (using caching). As can be seen, a large portion of the execution-time the threads are idle. This is the major reason for the packet-based traversal for being rather slow, regarding the brute horsepower of the GPU.

The single-ray variant on the other hand, does not suffer from these restrictions, since the traversal happens separately. As a consequence, during the root-finding phase, all rays will be active (apart from the rays which did not intersect a leaf-node), maximizing the multiprocessor's utilization. However, some rays may finish earlier (either successful or not) than some other rays in the same warp, which causes the already finished rays will have to wait for the entire warp to finish. Therefore, also the single-ray variant suffers from some under-utilization. Although this under-utilization is restricted to only one warp and will not affect other warps. Finally, rays in the single-ray variant may have already traversed the entire BVH, while other threads in the same warp are still busy. Note, that this will never occur using the packet-based variant, where idle-time is caused by a minority of rays intersecting a leaf-node. Figure 9.13b gives a visualization of this idle-time. Finally, Figure 9.14 shows a histogram of the average utilization during root-finding. The image clearly shows that most of the time, only a very small subset of rays is active, which causes the high idle-time in the other image.


(a) Packet-based (Cached) (b) Single-ray (Cached)

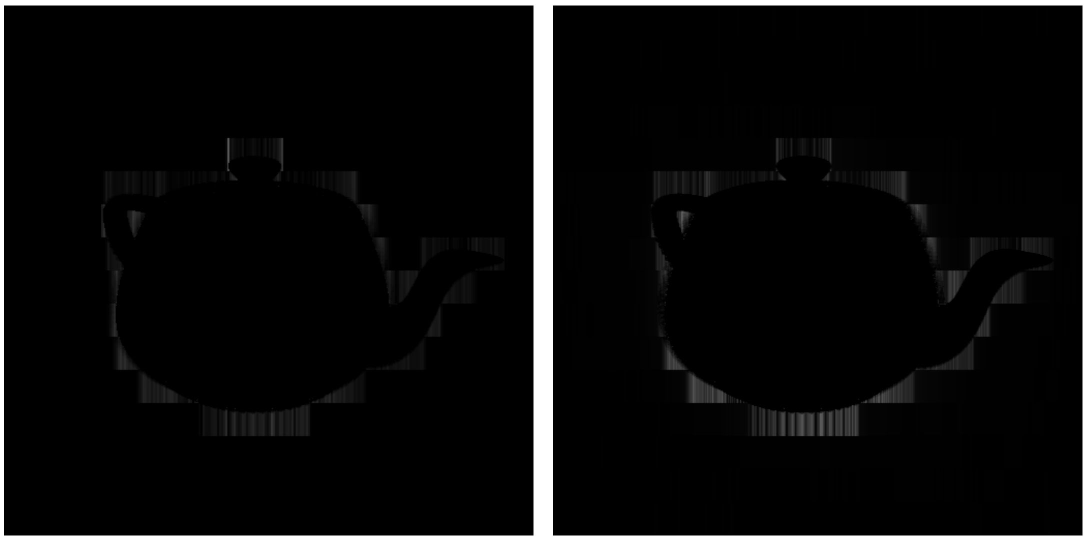

(c) Packet-based (Hybrid+Cached) (d) Single-ray (Hybrid+Cached)

Figure 9.13: Amount of time spent idle of total durations for each thread.

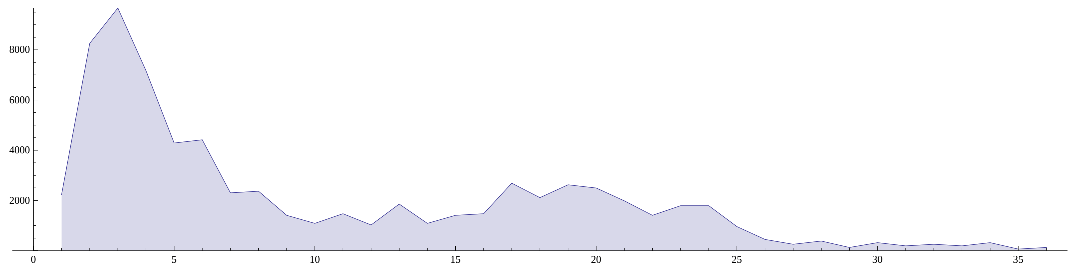


Figure 9.14: Frequencies of the average number of rays participating in the root-finding.

9.4.4 Low-level Profiling

An easy way to obtain statistics about running CUDA-kernels, is by enabling the CUDA-profiler. Running CUDA-applications will then automatically generate a log-file containing low-level information for each kernel-launch. Among these measurements are: running time for a kernel, occupancy, number of divergent branching warps, etc. By using the CUDA-profiler, some insight may be gained about kernels, and why some perform less than expected.

Branching warps

Table 9.14 lists the collected information about main kernel for each variant based on the rendering of the teapot-scene. A high difference in the number of warp-branches is revealed, which is caused by the difference in traversal of each variant. When traversing the BVH packet-based, the set of visited nodes is the union of the set of nodes which would be visited when traversing each ray separately. Therefore, much more nodes will be visited in the packet-based variant than the single-ray variant, depending on the coherence of the rays in the packet. The coherence is likely to drop, due to the much larger packets (64 rays). The possibility that rays will hit different BVH-nodes will therefore be much larger than when using smaller packets such as 2x2 packets which is usually the case ([AGM06, ABS08]). Because each extra visited node requires an extra traversal step, extra code will be executed, resulting in more branches. The difference in the number of instructions executed confirms this.

The difference in the number of branching warps between the single-ray variants is explained by the fact that their block-sizes differ. Therefore, threads will execute differently depending on their corresponding rays.

Diverging warps

During packet-based traversal, most branches never will be divergent, because the entire packet decides the order of traversal, causing the entire warp to follow the same execution path. However, there will always be branches which will cause divergence. Some rays for example, may take longer to find an intersection while root-finding, causing the warp to diverge, leading to warp serialization (which is confirmed by the table). Additionally, rays may skip the root-finder entirely, if it misses the sub-patch's bounding-box.

The single-ray variant on the other hand, does not share a common traversal path, and therefore may diverge during traversal. Since the traversal is separated from the root-finder, the divergence is kept to a minimum. Nevertheless, it is still larger than the packet-based variant.

	Nocache		Cache	
	Packet	Single	Packet	Single
GPUTime (ms)	308	216	190	168
occupancy (%)	33	25	17	17
branching warps	1.189.939	664.956	1.189.939	527.565
divergent warps	36.775	45.486	36.775	35.816
instructions +/-	9.6M	6.9M	9.4M	5.4M
warp serialize	780k	21k	820k	19k
local load	1.591.277	1.431.112	4.574	67.343
local store	4.253.364	5.066.636	40.152	320.800

Table 9.14: CUDA profiling data for the CNRTS-kernels (based on the teapot-scene).

Warp serialization

Whereas the number of divergent warps of the single-ray variant (uncached) is much higher than for packet-based, the number of warp serializes is a lot smaller ($37\times$). Warp serialization is caused by bank-conflicts to either the shared-memory or the constant memory. Since the number of visited nodes for packet-based is larger, this number will be larger.

Local-memory accesses

The effect of using cache is very evident in the table. The number of local loads/stores is heavily reduced for the cached variants, which is obvious since the caches avoid local memory transfers.

9.5 Discussion

Standard Ray Tracing Although nowadays packet-based ray tracing is considered to be the preferred way of ray tracing on CPU-architectures (and actually also on GPU-architectures), the single-ray implementation seems to perform much better on GPU. At first, the rather large packet-size, enforced by the 32-wide SIMD-architecture, causes the packets to be less coherent compared to CPU-implementations. On the CPU, these packets are usually much smaller, since the SIMD-width is 4. As a consequence, the overall efficiency will be much lower on GPU, as less rays will participate in the root-finding.

While a naive single-ray implementation will suffer heavily from ray divergence, the implementation used in here tries to avoid these under-utilization problems as much as possible. By separating the traversal from the root-finder, the divergence only affects the traversal stage. Under-utilization occurs only if some rays belonging to a warp have not yet returned, either successful or unsuccessful. As soon as all rays return, the warp enters the root-finding stage, in which each ray will try to find an intersection (except those for which no leaf-node was found during traversal). Therefore, the utilization during root-finding, which is the most expensive operation in the system, in much higher than the packet-based implementation.

Hybrid Ray Tracing Although the accelerator phase is very fast, the additional ray tracing phase necessary to fix the small number of artifacts, takes up the majority of the total hybrid rendering time (Table 9.5, Table 9.6). Therefore, the benefit of using a GPU/CPU-hybrid is rather low and less than expected. Looking at the processing time required to fix the relatively small amount of pixels, it seems that the system does not scale very well over the number of rays. For highly complex scenes, hybrid ray tracing becomes actually slower than standard ray tracing (Table 9.7).

Furthermore, not always the same quality is guaranteed as standard ray tracing. Some artifacts may appear, due to wrong tessellations, which cannot be handled by the ray tracer (Figure 9.3b).

However, because the total number of pixels that require a repair is generally very low, the accelerator could act as a very fast "low-detail" method to navigate through the scene.

Caching While the use of a cache does improve the performance of the 8800M architecture, it is less obvious for the GTX 295 architecture. For the primary rays, using standard single-ray tracing, the use of cache even decreases performance a little (Figure 9.6). However, for shadow- and secondary-rays, the speedup becomes more apparent (Figure 9.7, Figure 9.8, and Figure 9.9).

Although the difference is not that big for the GTX 295 architecture, it shows there is some improvement when using a cache. Furthermore, since the cache is implemented using the shared-memory, the occupancy is lowered, suggesting that the performance increase would be much bigger when the same occupancy could be maintained.

The upcoming Fermi architecture has an hierarchical caching system built-in. This would make the software-managed cache obsolete. Nevertheless, for "older" architectures, the cache proves to be worthwhile.

Basis-function evaluation Although [AGM06] presented their method as the best performing evaluation scheme on SIMD-architectures, using CUDA their division-free basis-function evaluation method perform slightly worse than the direct evaluation-method described in [PT97]. Since a lot of preprocessed data needs to be fetched to evaluate a set of basis-functions, the advantage of not having to compute a division diminishes, whereas the direct evaluation-method only requires the knot-vector data. Therefore, the advantage of using the division-free method above the direct evaluation-method is very small.

Preprocessing Although the focus of this thesis was not on efficient ways to generate BVHs, the timings in Table 9.3 show that the preprocessing time is usually under a second up to a few seconds for moderately complex scenes, which is not very high. Nevertheless, this might be improved even further by generating the BVH on the GPU also.

Supported scenes Table 9.3 shows that the system is capable of ray tracing scenes containing 4000 patches. Obviously, more complex scenes are supported which is only limited by the available GPU-memory.

The maximum degree the system can handle is limited by the available shared-memory. When using the uncached variant, virtually any degree is supported. Of course, the intersection-time will increase also.

Time-complexity While standard rasterization is linear in the number of objects, ray tracing isn't. Table 9.7, together with Figure 9.10 and Figure 9.11 clearly show a logarithmic increase in rendering time, in which the single-ray implementations appear to converge to a constant rendering time.

The linear nature of the rasterization algorithm becomes apparent in the hybrid ray tracing results in Table 9.7. It can be seen that the rasterization time increases much faster than the ray trace time. However, eventually, the rasterization time will be too high, making it unusable for interactive ray tracing.

Chapter 10

Conclusions and Future Work

Ray Tracing is by no doubt a very expensive algorithm. For regular planar primitives, such as triangles, quadrilaterals, polygons, etc. the difficulty generally lies in the traversal of the acceleration datastructure, since computing an intersection between a ray and a primitive is computationally cheap to implement. However, when ray tracing NURBS-surfaces, the bottleneck has shifted to the intersection computation, which is very expensive to compute. Furthermore, due to the incoherent nature of the ray tracing algorithm, it is very difficult to provide the GPU with homogeneous workloads since rays may take longer to compute.

In this thesis we have presented a system capable of ray tracing NURBS-surfaces with shadows, reflections, and refractions up to any depth, and fully implemented on the GPU using CUDA. By separating the traversal from the root-finder, a single-ray implementation was able to outperform the generally-preferred packet-based implementation.

Contrary to the common believe that GPUs are not suitable for ray tracing, due to their streaming SIMD-architecture, the results suggest the crossing-point is being reached between CPUs versus GPUs, in favor of GPUs. Although the system currently does not yet outperform the 8-core CPU-implementation from [ABS08], it does leave the 1-core implementation far behind. And the long idle-times suggest that there is plenty of room for improvement.

Although using a hybrid approach increases performance, it does not perform as expected. Looking at the processing time required to fix the relatively small amount of pixels, it seems that the system does not scale very well over the number of rays. Because artifacts may remain, and rasterization times can become larger than standard ray tracing, it is doubtful that hybrid ray tracing will remain beneficial in the future. It could act however as a very fast “low-detail” method to navigate through the scene.

10.1 Future Work

Trimming Although the system is able to ray trace virtually any NURBS-model, it does not yet have support for *trimming curves*, which is commonly used in industries to ease the design of complex models. Without trimming curves, surfaces with holes are very difficult to model. The surface will have to be wrapped around the hole, resulting in many patches. The availability of support for trimming curves would therefore be a great addition to the system.

SLI Currently, the ray tracing system is limited to use only a single GPU. Therefore, the 295GTX’s full potential is far from being reached. By adding support for multiple GPUs in an SLI configuration, the ray tracing performance can be doubled theoretically. However, since the GPUs do not share their memory-spaces, all data needs to be uploaded twice.

Persistent Threads More recently, [AL09] have taken a total different approach towards ray tracing on GPUs, by using “persistent threads”. Instead of mapping each thread to a screen-pixel, they let the threads fetch unprocessed rays from a ray-pool. If a thread finishes processing its ray, it fetches the next ray from the pool. In this way, all threads will be kept busy, heavily increasing the utilization.

Another benefit from this approach is, that now only rays will be processed that were spawned previously. Currently, all pixels will be traced, but terminated rays will exit earlier. The hybrid approach could benefit from this as well, as there are usually few rays which need to be repaired.

Fermi Last but not least, the upcoming new architecture by NVIDIA looks very promising. Apart from the brute horsepower, it contains some nice new features, very useful for ray tracing. The built-in cache hierarchy, will automatically increase performance for much processes in the ray tracer. The possibility to run different kernels concurrently sounds very interesting, and opens new ways to improve today’s state of the art.

Bibliography

[Abe05] O. Abert. Interactive Ray Tracing of NURBS Surfaces by Using SIMD Instructions and the GPU in Parallel. Master's thesis, University of Koblenz-Landau, 2005.

[ABS08] O. Abert, M. Bröcker, and R. Spring. Accelerating Rendering of NURBS Surfaces by Using Hybrid Ray Tracing. 2008.

[AGM06] O. Abert, M. Geimer, and S. Muller. Direct and Fast Ray Tracing of NURBS Surfaces. In *Proc. IEEE Symposium on Interactive Ray Tracing 2006*, pages 161–168, 18–20 Sept. 2006.

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM.

[BBB87] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An introduction to splines for use in computer graphics and geometric modeling*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[BBDF05] S. Beck, AC Bernstein, D. Danch, and B. Fröhlich. CPU-GPU Hybrid Real Time Ray Tracing Framework. 2005.

[BBLW07] C. Benthin, S. Boulos, D. Lacewell, and I. Wald. Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces. Technical report, Scientific Computing and Imaging Institute, 2007.

[BM99] W. Boehm and A. Müller. On de Casteljau's algorithm. *Computer Aided Geometric Design*, 16(7):587–605, 1999.

[CDP] Sylvain Collange, David Defour, and David Parello. Barra, a Parallel Functional GPGPU Simulator.

[CHH02] N.A. Carr, J.D. Hall, and J.C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2002.

[CLR80] Elaine Cohen, Tom Lyche, and Richard Riesenfeld. Discrete b-splines and subdivision techniques in computer-aided geometric design and computer graphics. *Computer Graphics and Image Processing*, 14:87–111, 1980.

[Cox72] M.G. Cox. The numerical evaluation of b-splines. *IMA Journal of Applied Mathematics*, 10:16, 1972.

[dB72] C. de Boor. On calculating with B-splines. *J. Approx. Theory*, 6(1):50–62, 1972.

[Dia09] G. Diamos. The Design and Implementation Ocelot's Dynamic Binary Translator from PTX to Multi-Core x86. 2009.

[DKK09] G. Diamos, A. Kerr, and M. Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. 2009.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.

[FS05] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22. ACM New York, NY, USA, 2005.

[GPSS07] J. Gunther, S. Popov, H. P. Seidel, and P. Slusallek. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proc. IEEE Symposium on Interactive Ray Tracing RT '07*, pages 113–118, 10–12 Sept. 2007.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7:14–20, 1987.

[Hav01] V. Havran. *Heuristic Ray Shooting Algorithms. Czech Technical University*. PhD thesis, Ph. D. dissertation, 2001, 2001.

[HSHH07] D.R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive kd tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174. ACM Press New York, NY, USA, 2007.

[Kaj82] J.T. Kajiya. Ray tracing parametric patches. In *Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 245–254. ACM New York, NY, USA, 1982.

[LYTM06] C. Lauterbach, S.E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45. Citeseer, 2006.

[MB90] J.D. MacDonald and K.S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[MCFS00] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *JOURNAL OF GRAPHICS TOOLS*, 5(1):27–52, 2000.

[OLG$^+$07] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Blackwell Synergy, 2007.

[PBMH05] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2005.

[Pet94] John W. Peterson. Tesselation of nurb surfaces. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 286–320. Academic Press, 1994.

[PGSS07] S. Popov, J. Gunther, H.P. Seidel, and P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Blackwell Synergy, 2007.

[Pro05] Jana Procházková. Derivative of b-spline function, 2005.

[PSS$^+$06] H. F. Pabst, J. P. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray Casting of Trimmed NURBS Surfaces on the GPU. In *Proc. IEEE Symposium on Interactive Ray Tracing 2006*, pages 151–160, 18–20 Sept. 2006.

[PT97] L.A. Piegl and W. Tiller. *The Nurbs Book.* Springer, 1997.

[PTVF97] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* CAMBRIDGE UNIVERSITY PRESS, 1997.

[TS05] N. Thrane and LO Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, 2005.

[WBB08] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets-Efficient SIMD single-ray traversal using multi-branching BVHs. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 49–57, 2008.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[WH80] T. Whitted and N.J. Holmdel. An Improved Illumination Model for Shaded Display. *Communications*, 1980.

[WSBW01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. In *Computer Graphics Forum*, volume 20, pages 153–165. Blackwell Synergy, 2001.