Martin Sattlecker

Reyes Rendering of Renderman Scenes on the GPU

Masterarbeit

zur Erlangung des akademischen Grades
Diplom-Ingenieur

Masterstudium Informatik

eingereicht an der
Technischen Universität Graz

Betreuer
Prof. Dr. Dieter Schmalstieg
Dr. Markus Steinberger

Institut für Maschinelles Sehen und Darstellen

Fakultät für Informatik und Biomedizinische Technik

Graz, Mai 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, an dere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

Abstract

In recent years graphics processing units have become more and more powerful. They are now capable of executing arbitrary code in a massively parallel fashion. The Reyes rendering pipeline is a commonly used method of rendering higher order surfaces in offline renderers. It is possible to execute the Reyes pipeline in parallel. In this work we show a Reyes renderer that is capable of rendering simple scenes with interactive to real time frame rates. Our implementation runs on the graphics card and uses a persistent Megakernel which performs the entire rendering process in one kernel call. The scenes for our renderer are given as Renderman scenes. Our renderer supports materials and surface displacement through the Renderman shading language. We show a compiler that translates these shaders into CUDA code. The shaders are compiled and loaded at runtime. We support texture access from the shaders. Our renderer supports bicubic Bezier patches and Catmull-Clark subdivision surfaces as input geometry. We show an algorithm for subdividing Catmull-Clark subdivision surfaces on the graphics card. This algorithm creates patches from the faces of a subdivision mesh which are then processed in parallel. It takes advantage of the fact that faces of a subdivision mesh with a regular neighborhood can be converted to Bezier patches. We show that the inclusion of this conversion can increase the performance of the rendering pipeline dramatically.

Kurzfassung

In den letzten Jahren hat sich die Rechenleistung von Grafikkarten stark gesteigert. Sie sind jetzt in der Lage beliebige Berechnungen auf der großen Anzahl an parallelen Rechenkernen auszuführen. Die Reyes Rendering Pipeline ist eine beliebte Methode für offline Renderer um glatte Oberflächen zu rendern. Es ist möglich die Reyes Pipeline parallel auszuführen. In dieser Arbeit zeigen wir einen Reyes Renderer der einfache Szenen mit interaktiven bis Echtzeit Frameraten rendern kann. Unsere Implementierung wird auf der Grafikkarte ausgeführt und benutzt einen persistenten Megakernel. Dieser führt die gesamte Renderpipeline in einem Kernelaufruf aus. Die Szenen für unseren Renderer sind als Renderman Szenen gegeben. Unser Renderer unterstützt Materialien und Oberflächen-Displacement Funktionen die als Renderman Shader gegeben sind. Wir zeigen einen Compiler, der diese Shader in CUDA Code übersetzt. Die Shader werden zur Laufzeit des Programms compiliert und geladen. Wir unterstützen Texturen in den Shadern. Unser Renderer unterstützt bikubische Bezier Flächen und Catmull-Clark Subdivision Surfaces als geometrische Primitiven. Wir zeigen einen Algorithmus, der Catmull-Clark Subdivision Surfaces auf der Grafikkarte unterteilt und rendered. Dieser Algorithmus erstellt einen Patch für jede Face in einem Catmull-Clark Subdivision Surface. Diese Patches werden dann auf der Grafikkarte parallel verarbeitet. Der Algorithmus wandelt Faces mit einer regulären Umgebung in Bezier Flächen um. Wir zeigen, dass diese Umwandlung die Performance unseres Renderers stark verbessert.

Contents

List of Figures

1. Introduction

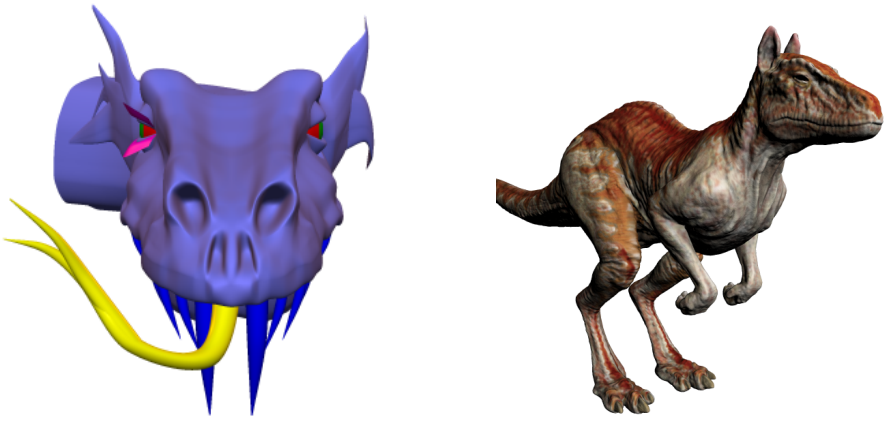
In the recent years graphics processing units (GPU) have evolved from accelerators of fixed function rendering pipelines into programmable massively parallel processors. In addition to graphics toolkits like OpenGL it is now also possible to execute programs written in languages like CUDA or OpenCL. Today's GPUs have thousands of processing cores that are capable of performing arbitrary computations. The performance of GPUs for parallelized algorithms is several times higher than that of a CPU.

The Reyes rendering pipeline was developed by Cook et al. [CCC87] in 1987 at Pixar. Reyes is an acronym for Render Everything You Ever Saw and was designed to create photo-realistic results. It is widely used in offline renderers such as Pixar's Renderman [Pixa]. The Reyes algorithm enables the rendering of curved surfaces such a Bezier patches or Catmull-Clark subdivision surfaces. It also supports displacement mapping.

In addition to the Reyes rendering pipeline Pixar also introduced the Renderman interface (RI) specification [Pix05]. It defines a protocol to describe 3-dimensional scenes which can be rendered with a Reyes renderer. The specification also contains a way to define the lighting of a scene and the materials of the geometry in the scene. This is accomplished by the introduction of the Renderman shading language (RSL). So in addition to the scene geometry a RI scene also contains light sources and materials that are defined by RSL functions.

The structure of the Reyes pipeline makes it possible to run the algorithm in parallel. However implementing the pipeline on the GPU is challenging as it contains a recursive loop. We present an implementation of the Reyes rendering pipeline on the GPU, that is able to render scenes according to the RI specification with materials and surface displacement given as RSL shaders. With our implementation it is possible to view and walk through

Renderman scenes interactively. In Figure 1.1 we present renderings of two Renderman scenes that where rendered with our implementation of the Reyes rendering pipeline.



(a) Dragonhead (b) Killeroo

Figure 1.1.: Rendering produced by our implementation of the Reyes pipeline.

The remainder of this work is structured as follows: In section 2 we discuss the related work. In section 3 we present the geometric primitives our implementation is able to render. In section 4 we give an overview of the Reyes pipeline. In section 5 we introduce the Renderman scene description and the Renderman shading language. In section 6 we describe our implementation of the Reyes pipeline. In section 7 we present the results of our renderer and compare the performance of different settings This is followed by the conclusion in section 8.

2. Related Work

In 1987 Cook et al. [CCC87] introduced the Reyes image rendering architecture, designed for photo-realistic results. It is based on the idea of adaptive surface subdivision. The algorithm was designed to be run in parallel. In the Reyes pipeline each geometric primitive is recursively subdivided by the so called bound and split loop. When a subdivided primitive is small enough it is diced into a grid of micropolygons. A micropolygon is a quad polygon with a size that is not greater than 1 pixel. These micropolygons are then shaded and sampled to produce the output image. We will describe the algorithm in more detail in section 4. In combination with the introduction of general purpose computation on the graphics card this gave rise to several GPU based Reyes implementations.

Patney and Owens [PO08] were the first to move the entire Reyes pipeline to the GPU. The most challenging part to parallelize in the Reyes pipeline is the bound and split loop that subdivides the input geometry. This loop recursively subdivides the primitives until they are small enough for dicing. Patney and Owens approach to implementing this recursive loop on the GPU was to use a breadth first approach. Their implementation uses 3 kernel launches per subdivision level to split the geometry. Followed by one kernel launch for the creation of the micropolygons. These micropolygons are then rendered using OpenGL They support bicubic Bezier patches as input primitives. This approach leads to a high number of kernel launches and to a large overhead for CPU - GPU synchronisation.

Patney et al. [PEO09] showed an approach similar to Patney and Owens [PO08]. Their algorithm supports Catmull-Clark subdivision surfaces. The algorithm is based on a breadth first approach and performs an view dependent subdivision of the input primitives. They produce a crack free mesh using special templates between two faces with a different subdivision level.

The first algorithm that implemented the entire Reyes pipeline on the GPU was RenderAnts [Zho+09]. The bound and split loop is similar to the one of Patney and Ownes [PO08]. The following dice, shading, and sample stages are also implemented on the GPU. Between those stages RenderAnts introduces a set of scheduling stages. In these stages the currently drawn region is subdivided, so that the memory requirements for rendering the parts are smaller than the device memory. The subdivided parts of the screen are then processed one after another. Their implementation uses 8 different kernels for the Reyes pipeline. Most of them are launched multiple times due to the division in the scheduling stages and the recursive bound and split loop. This leads to an overhead from the high amount of kernel launches that are needed to render a scene. RenderAnts is the only other GPU Reyes implementation that is capable of rendering Renderman scenes. It also supports materials using the Renderman shading language.

In 2010 Tzeng et al. showed the first implementation of the Reyes pipeline using a persistent kernel [TPO10]. In their implementation the persistent kernel is only used to compute the recursive bound and split loop. The other stages of the Reyes pipeline were performed by 4 different kernels. For their persistent kernel they use a distributed queuing approach with task donation. Tzeng et al. were the first to achieve interactive frame rates for the Reyes pipeline on a single GPU.

Nießner et al. [Nie+12] presented a method for fast rendering of Catmull-Clark subdivision surfaces on the GPU. Their approach relies on the tessellation shader of the DirectX pipeline. They perform an adaptive subdivision of the input mesh. This means, that they convert faces with no extraordinary vertices to bicubic B-Spline patches after each subdivision of the mesh. These patches are then rendered using the DirectX tessellation shader. Only faces with extraordinary vertices are then further subdivided using the subdivision rules. This approach leads to fewer faces to be rendered while still producing the correct limit surface. The subdivision steps are precomputed. They build a table with information needed for the subdivision steps at each level. This table only needs to be recomputed if the topology of the mesh changes. Animations of the mesh can be performed without recomputing it. This makes this approach especially suitable for animated scenes.

Over the years there have also been several approaches to render Catmull-

Clark subdivision surfaces by approximating them with parametric patches. Loop et al. [LS08] have shown a method to approximate them with bicubic Bezier patches. In general, approximating methods do not reproduce the subdivision surface perfectly. In the case of regular faces, however, the method by Loop et al. [LS08] produces the same surface as the initial subdivision mesh. We take advantage of this fact as described in section 3.2.2.

In 2014 Steinberger et al. introduced Whippletree [Ste+14], an approach to schedule dynamic, irregular workloads on the GPU. Whippletree makes it possible to execute complex algorithms like the Reyes rendering pipeline entirely on the GPU. It uses a persistent Megakernel approach with queues. In Whippletree different parts of a algorithm can be implemented as procedures. A procedure is a function that takes a work item from a queue and performs computations on it. It is also possible to spawn the execution of other procedures from within a procedure by inserting work items into a queue. This makes it possible to execute the Reyes pipeline, including the recursive bound and split loop, in one kernel launch. In the Whippletree paper Steinberger et al. show an implementation of Reyes rendering. We [SS15] extended the implementation by a proper rasterization algorithm, motion blur, depth of field and displacement mapping. Our implementation of the Reyes pipeline builds on top of this work.

3. Geometric Primitives

Our implementation of the Reyes pipeline supports two types of primitives: Bezier patches and Catmull-Clark subdivision surfaces. In this chapter we will give a short overview of these primitives.

3.1. Bezier Patches

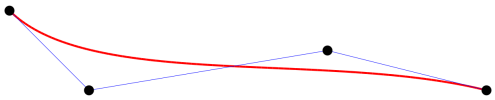
Bezier patches are parametric surfaces. They are based on Bezier curves. We will first describe Bezier curves.

3.1.1. Bezier Curves

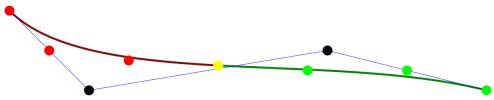
Bezier curves are parametric curves that are heavily used in computer graphics. They are defined by a set of control points. A Bezier curve has to have a degree. The degree determines how many control points influence the position of the curve at a certain parametric value. The number of points that influence the position is $degree + 1$. E.g. the points on a linear Bezier curve are influenced by two control points. Our implementation only considers Bezier patches that are based on Bezier curves of degree three. Those curves are called cubic Bezier curves. For these, the position of the curve is determined by four control points.

The end control points of a Bezier curve lie directly on the curve, whereas the mid control points only influence the direction of the curve. Bezier curves produce smooth curves that connect the first and last control point. Two Bezier curves of degree d can be joined with C_{d-1} continuity. This means that cubic Bezier curves can be joined with C_2 continuity which leads

to a smooth appearance. The curve positions are all within the convex hull of the control points. An example of a cubic Bezier curve can be seen in figure 3.1(a).



(a) Initial Curve



(b) Curve Divided by DeCasteljau Algorithm

Figure 3.1.: An example cubic Bezier curve

DeCasteljaus Algorithm The DeCasteljau algorithm makes it possible to split a given Bezier curve into two shorter Bezier curve of the same degree. The combination of the two curves exactly resembles the original curve. The split can be performed at an arbitrary parametric value. The DeCasteljau algorithm makes it possible to evaluate Bezier curves in a recursive manner. The algorithm can also be used to evaluate the position of the Bezier curve at an arbitrary parametric value. In Figure 3.1(b) a Bezier curve that was split at the parameter value 0.5 is shown. The combination two new curves, shown in green and red, resemble the initial curve seen in Figure 3.1(a).

3.1.2. Bezier Surfaces

Bezier surfaces are parametric surfaces that use Bezier curves as a basis. They are defined by a grid of control points. This control grid can be seen as a number of Bezier curves in u direction, or as a number of Bezier curves in v direction. The position of the surface at a certain parametric value u_t, v_t is computed by first evaluating the surrounding $degree + 1$ Bezier curves around the point in u direction at parameter value u_t. This results in a Bezier curve in v direction which is then evaluated at v_t to get the surface point.

In Figure 3.2 is an example of the surface position evaluation of a bicubic Bezier patch. The surface is evaluated at parameters $u = 0.5$ and $v = 0.5$. The black dots denote the control points of the Bezier patch. These control points can be seen as four curves in u direction. From evaluating these curves at the u parameter value of 0.5 a curve in v direction is created. This curve is shown in red. The surface position is then computed by evaluating this curve at the v parameter value 0.5.

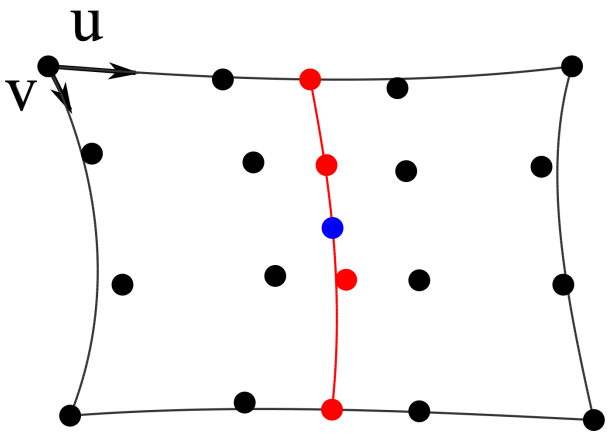


Figure 3.2.: Bezier Patch: Evaluation of the Surface position at parameter value (0.5, 0.5)

Our implementation only supports bicubic Bezier patches. They are based on cubic Bezier curves which are defined using 4 control points. This means, that each patch is defined by a grid of 4×4 control points. The four control

points in the corner of the patch lie directly on the resulting surface. The other points influence the surface position, but don't lie on the surface itself.

DeCasteljaus Algorithm Bezier patches, like curves, can be split using the DeCasteljau algorithm. To split a patch into two patches along the v parameter value of 0.5 the DeCasteljau algorithm is used on the Bezier curves in v direction. Every curve is split into two curves at parameter value 0.5. The combination of the first halves of each curve form the first new patch. The other halves form the other smaller patch. The combination of the two new patches creates the same surface as the initial patch. To split a patch along the u direction the same algorithm is applied in the u direction.

Normal computation The normals can be computed from the tangents in u and v directions. To compute the tangent at a specific parametric value u_t, v_t, along the u direction, we first have to compute the cubic Bezier curve that runs along the parametric value v_t in u direction. This is done with the same algorithm that is used for computing the surface position. The normal can be computed from the resulting cubic Bezier curve using the DeCasteljau algorithm for quadratic Bezier curves [She].

$$tangent(u_t) = deCasteljau(P2 - P1, P3 - P2, P4 - P3, u_t) \tag{3.1}$$

Where $P1$ to $P4$ are the control points of the computed bicubic Bezier curve. The same algorithm can be used to compute the tangent in v direction. From the tangent, the normal is computed using the cross product.

$$normal = tangent_v \times tangent_u \tag{3.2}$$

3.2. Catmull-Clark Subdivision Surfaces

Catmull-Clark subdivision surfaces are another kind of smooth surfaces. They were introduced by Catmull et. al [CC78] in 1978 and are widely

used in computer graphics. A Catmull-Clark subdivision surface is given as a control polygon mesh with arbitrary topology. The actual surface is computed by repeatedly subdividing the input mesh. This subdivision creates a smooth surface which is C_2 continuous on the whole mesh except for extraordinary vertices where it is C_1 continuous. Extraordinary vertices are control points with a valence that is not four. We also call those irregular vertices. Vertices with valence four are called regular vertices.

The ability to generate smooth surfaces from polygon meshes with arbitrary topology presents an advantage over parametric surfaces such as Bezier surfaces. Small details can be added to a model without creating large amounts of points in the rest of the model. This makes modelling of subdivision surfaces easier and generates a more compact representation of a model.

3.2.1. Subdivision Rules

The actual surface of a Catmull-Clark subdivision surface is computed by recursively subdividing the faces of the input mesh into smaller faces. In this section we will describe these subdivision rules.

The first part of the subdivision is to add new points to the subdivided mesh. The second part of the subdivision moves the original vertices of the mesh. A complete subdivision step consists of the following three steps:

1. **Insert face points:** For each face a new point is added to the mesh. The position of the point is the average of all points of the face.

$$F_i = \frac{1}{m} \sum_{j=1}^{m} V_j \tag{3.3}$$

   Where m is the number of points in the face and V_j are the vertices of the face.
2. **Insert Vertex Points:** For each edge connecting two vertices a new point is inserted. The position of the new point is computed:

$$E_i = \frac{1}{4} \cdot (V_0 + V_1 + F_1 + F_2) \tag{3.4}$$

V_0, V_1 are the endpoints of the edge, and F_1, F_2 are the face points of the adjacent faces.

3. **Move Vertex Points:** Each vertex point is moved to a new position:

$$V_i = \frac{F}{n} + \frac{2E}{n} + \frac{V(n-3)}{n} \tag{3.5}$$

Where F is the average of all adjacent face points, E is the average of all adjacent edge points and V is the old vertex point.

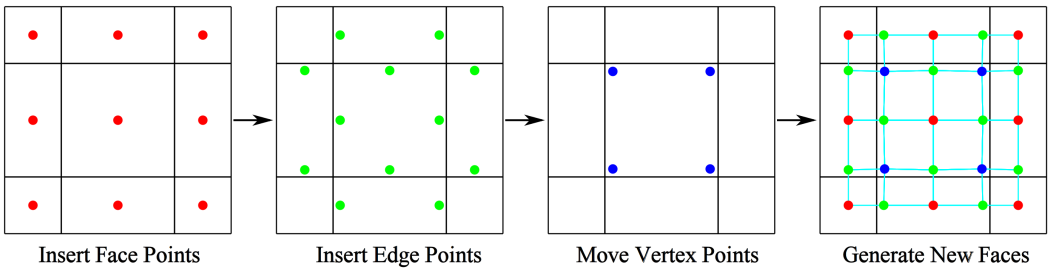


Figure 3.3.: Catmull-Clark subdivision step with a mesh of 9 faces. Note that the illustration only shows the subdivided faces that contain one of the 4 inner vertices.

The faces of the subdivided mesh are then assembled from these points: A face with m vertices is split into m new faces. Each of the new faces contains a face point, a vertex point, an edge point and another vertex point. The new faces always contain four vertices. This means, that all newly generated faces are quads. Therefor a subdivision mesh only consists of quads after one subdivision. The valence of the face point is equal to the number of vertices in this face. The valence of the edge points is four. The valence of the vertex points is equal to the valence of the vertex point before the subdivision The subdivision steps are visualised in 3.3. The example shows a face that is surrounded by eight other faces. All vertices of the face have valence four and the face is a quad. Therefore, the resulting points all have valence four.

Limit Rules

The vertices of a subdivision mesh describe the surface of a subdivision surface, but they do not lie on the surface itself. The points however approach

the surface when the number of subdivisions of the mesh goes to infinity. Therefore we call it the limit surface of a subdivision mesh. It is possible to compute the limit of the subdivision for a vertex with a modified version of the Move Vertex rule:

$$V_i^{limit} = \frac{F}{n(n+5)} + \frac{4E}{n(n+5)} + \frac{nV}{n(n+5)} \tag{3.6}$$

This rule is obtained by computing the limit of repeatably applying the Move vertex rule as shown by Halstead et al. [HKD93].

Normal Rules

The normal of a subdivision surface can be computed from the cross product of two tangents. The tangents can be computed as follows:

$$t_x = \sum_{i=1}^{n} A_n \cos\left(\frac{2\pi i}{n}\right) E_i + \sum_{i=1}^{n} \left(\cos\left(\frac{2\pi i}{n}\right) + \cos\left(\frac{2\pi(i+1)}{n}\right)\right) F_i \tag{3.7}$$

$$t_y = \sum_{i=1}^{n} A_n \sin\left(\frac{2\pi i}{n}\right) E_i + \sum_{i=1}^{n} \left(\sin\left(\frac{2\pi i}{n}\right) + \sin\left(\frac{2\pi(i+1)}{n}\right)\right) F_i \tag{3.8}$$

where $A_n = 1 + \cos\left(\frac{2\pi}{n}\right) + \cos\left(\frac{\pi}{n}\right)\sqrt{18 + 2\cos\left(\frac{2\pi}{n}\right)}$

Where n is the valence of the vertex. E_i are the vertices that are connected to the vertex with an edge. F_i are the vertices that connect the edge vertices E_i and E_{i+1}. They are called E and F because after a subdivision the normal can be computed from the newly created edge, face and vertex points. E_i are taken from the edge points and F_I from the face points. Note that this formula only works for quad meshes. Other meshes must be subdivided at least once to obtain a quad mesh.

The normal can then be computed by the cross product:

$$normal = t_x \times t_y \tag{3.9}$$

The rules where derived by Halstead et. al [HKD93].

Border Rules

The aforementioned rules are only valid for vertices that are surrounded by faces. They do not work for vertices that lie on the border of a subdivision mesh. In these cases special rules apply:

Subdivision Rules During subdivision there are modified rules for the edge points and the vertex points. The computation of the face points is the same as in the non border rules.

1. **Insert Edge Points** The edge point is computed from the two endpoints $V_{0/1}$:
$$E_i = \frac{1}{2}(V_0 + V_1) \tag{3.10}$$
2. **Move Vertex Points** The new vertex point is only influenced by the points on the border:
$$V_i = \frac{1}{8}(6V + V_0 + V_k) \tag{3.11}$$
Where V is the vertex point and $V_{0/k}$ are the two adjacent points on the border.

Limit Rules The limit rule is also modified and is similar to the move vertex rule for borders:

$$V_i^{limit} = \frac{1}{6}(4V + V_0 + V_k) \tag{3.12}$$

The limit rule, and the following normal rules, were derived by Biermann et. al [BLZoo].

Normals Rules Like in the non-border subdivision rules, the normal is computed from two tangents. The tangent t_x can easily be computed:

$$t_x = V_k - V_0 \tag{3.13}$$

There are two formulas for the computation of t_y. One for the case if the vertex has valence two and another general rule for valence greater than two.

1. **Valence 2**
$$t_y = -4 * V + V_0 + V_1 + 2F_i \tag{3.14}$$
Where F_i is the face point adjacent to the vertex
2. **General**
$$t_y = \alpha V + \beta_0 V_0 + \sum_{i=1}^{k-1} \beta_i V_i + \sum_{i=0}^{k-1} \gamma_i f_i + \beta_k V_k \tag{3.15}$$
Where α, β_i, γ_i are computed as follows:
$$R_k = \frac{1 + cos(\frac{\pi}{k}}{ksin(\frac{\pi}{k})(3 + cos(\frac{\pi}{k}))} \tag{3.16}$$
$$\alpha = 4R_k(-1 + cos(\frac{\pi}{k})) \tag{3.17}$$
$$\beta_0 = \beta_k = -R_k(1 + 2cos(\frac{\pi}{k})) \tag{3.18}$$
$$\beta_i = \frac{4sin(i\frac{\pi}{k})}{k(3 + cos(\frac{\pi}{k}))} \tag{3.19}$$
$$\gamma_i = \frac{sin(i\frac{\pi}{k}) + sin((i+1)\frac{\pi}{k})}{k(3 + cos(\frac{\pi}{k})} \tag{3.20}$$
Like in the standard normal computation rule v_i and f_i denote the vertices near the vertices the normals are computed for. An illustration of these points can be seen in figure 3.4. It shows a vertex with valence 5 that lies on the border of a subdivision surface.

The normal is then computed from the cross product of the two tangents.

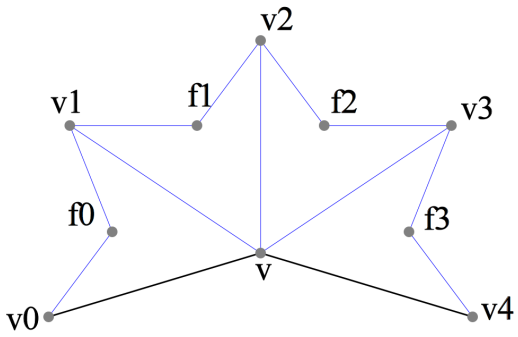


Figure 3.4.: Example of a subdivision surface near a border.

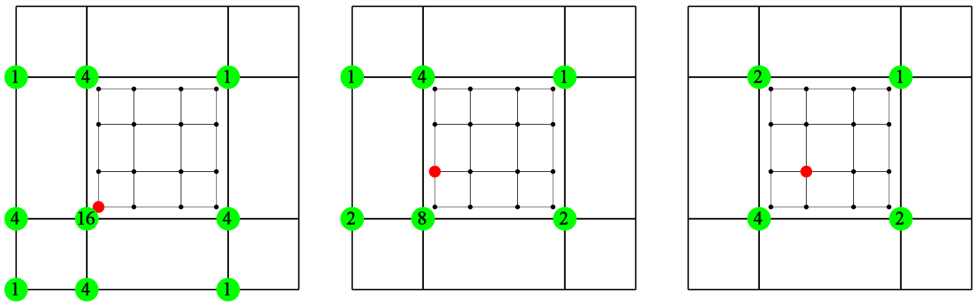
3.2.2. Approximation using Bezier Patches

Loop et al. [LS08] showed that Catmull-Clark subdivision surfaces can be approximated using Bezier patches. In the general case the resulting surface positions and normals are not the same as the original surface. Therefor it cannot be used to accurately render subdivision surfaces. There is one case however when the approximation using Bezier patches produces the same surface as the Catmull-Clark subdivision mesh. This is the case when a face is a quad and all four vertices of the face have valence four and no border. We call those faces regular faces, and we use this property to render such faces faster.

To convert such a regular face to a Bezier patch, the neighboring vertices and faces are considered. As the face itself is a quad face, and it is bordered by 8 other quad faces, there are 16 points that need to be considered. In figure 3.5 an example of such a face can be seen.

Computation of the Bezier patch

The Bezier control points are computed using masks on the subdivision surface control points. The masks can be seen in figure 3.5. The red point denotes the output point that is influenced by the mask points. The mask points are highlighted green. The number inside the points gives the weight



(a) Corner Points (b) Outer Points (c) Inner Points

Figure 3.5.: Masks for the computation of Bezier patches from a regular Catmull-Clark subdivision surface face

of the point. In the computation the weights are normalized to get a sum of 1. There are three different masks for different points in the output patch:

1. **Corner points** are the points at the corner of the output patch. The four output points are computed by rotating the mask seen in Figure 3.5(a).
2. **Outer points** are the points on the edges of the output patch. There are eight outer points. They are computed by mirroring and rotating the mask seen in Figure 3.5(b).
3. **Inner points** are the four remaining points inside the output patch. They again are computed by rotating the mask seen in Figure 3.5(c).

These rules produce 16 points in a grid of 4×4. These 16 points describe a Bezier patch as seen in section 3.1. This Bezier patch produces the same surface as the face of the Catmull-Clark subdivision surface that was converted.

4. The Reyes Rendering Architecture

4.1. Pipeline Stages

The Reyes (Render Everything You Ever Saw) image rendering architecture was developed by Cook et al. in 1987 [CCC87] as a method to render photo-realistic scenes with limited computing power and memory. Today it is widely used in offline renderers like e.g. Pixar's Renderman. Reyes renders parametric surfaces using adaptive subdivision. A model or mesh can, e.g., be given as a subdivision surface model or as a collection of Bezier patches. As a direct rasterization of such patches is not feasible, Reyes recursively subdivides these patches until they cover roughly a subpixel or less. Then, these patches are split into a grid of approximating quads which can be rasterized easily. The Reyes rendering pipeline is divided into five stages. These stages are not simply executed one after another, but include a loop for subdivision, which makes Reyes a challenging problem with unpredictable memory and computing requirements. The pipeline stages are visualized in Figure 4.1 and listed in the following paragraphs. Figure 4.2 shows an example of the split and dice stages of the Reyes pipeline using a Bezier patch.

Before a geometric primitive is rendered using the Reyes pipeline we create patches from the primitive which are then processed by the pipeline. For Bezier patches this is straightforward: Every patch is processed by the pipeline. In the case of Catmull-Clark subdivision surfaces a patch is created for every face in the mesh.

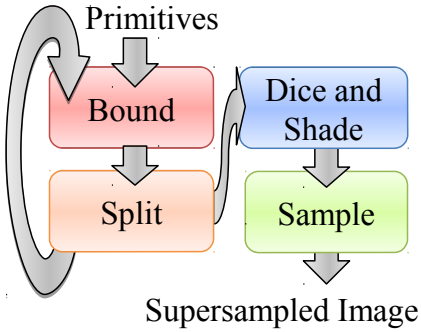


Figure 4.1.: The Reyes Pipeline Stages

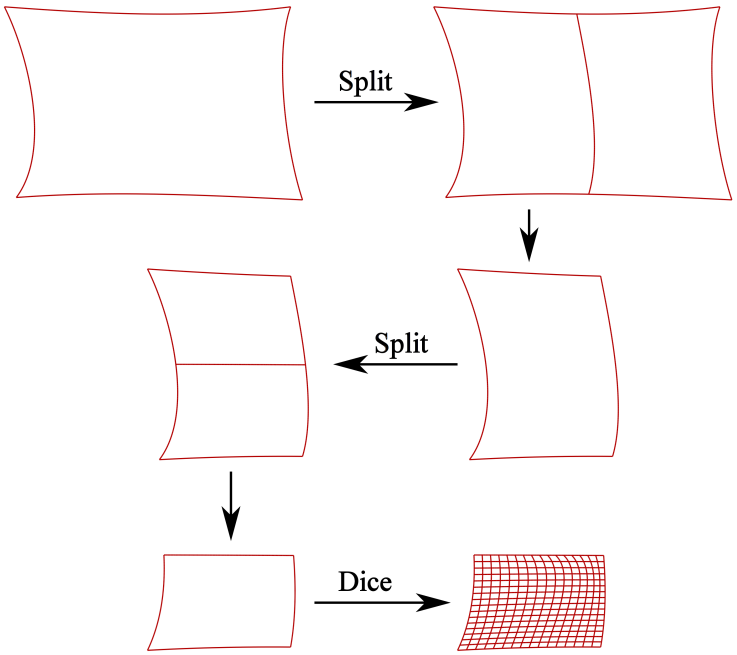


Figure 4.2.: Example pass of the Reyes pipeline. The figure shows the split and dice of a Bezier patch. The example starts with the input patch in the top left. It is split two times and then diced into a grid of micropolygons.

Bound The bound stage decides what happens with a given patch. Every patch is first processed by the bound stage. It clips the input primitive against the viewing frustum. If a primitive is not culled, a bound formula is applied. It decides, if a geometric primitive can be forwarded to the dicing stage. The formula is based on the screen space size of the primitive. If the input primitive is not small enough for dicing it is forwarded to the split stage.

Split Splits the input primitive into several smaller primitives. In the case of a Bezier patch it can be subdivided either along the u direction, or along the v direction. The split is performed using the DeCasteljau algorithm described in section 3.1.1. This results in two smaller Bezier patches. Faces of a Catmull-Clark subdivision surfaces are split into four smaller faces by applying the subdivision rules as seen in section 3.2.1. After the split, the new geometric primitives are again checked by the bound stage.

Dice A given primitive is diced into a grid of micropolygons. Dicing means, that a given geometric primitive is divided into a grid of points. These grid points form the micropolygons that are sampled and shown in the output image. A micropolygon is a quadrilateral polygon that is approximately the size of one subpixel in the output image. The points in this grid are then forwarded to the shade stage. After shading the micropolygons are assembled and forwarded to the sample stage.

Shade Each geometric primitive can have an associated displacement shader. If a displacement shader is present, the grid points in a given micropolygon grid are displaced in the shade stage. Then every grid point is shaded with a surface shader, resulting in a color for every point.

Sample In the sample stage a given micropolygon is sampled. In our implementation the sampling stage is implemented as a micropolygon rasterizer. It loops over all subpixels in the axis aligned bounding box of a micropolygon. If a subpixel lies inside the micropolygon, the rasterizer

interpolates the colors of micropolygon corner points and writes them to the output image.

Composite The composite stage combines the subpixels in every pixel of the output buffer to generate the final output image. In our implementation the composite stage computes an average of all subpixels in a pixel and displays them in an OpenGL viewport.

5. Renderman Scene description

Our implementation supports Renderman scenes using the Renderman Interface (RI) Specification. This specification describes a way to define a three dimensional scene with lighting and materials. In this section we give a short overview over the scene description. We will only cover the high level concepts and some important details. For further information see the Renderman Interface specification [Pix05] [Pixb].

There are two ways to define a Renderman scene: There is a C interface Here the graphics state is manipulated by function calls. There is also the Renderman Interface Bytestream (RIB) Protocol. This is an ASCII scene description, which consists of statements that are equivalent to the functions of the C interface. Our implementation only support RIB files for scene description.

5.1. The Renderman Interface

A Renderman scene given as a RIB file is a collection of statements. These statements define a graphics state for rendering the scene. The graphics state is a hierarchy of different sections. In the following we call these sections nodes as they can be seen as nodes of a hierarchy tree. They contain global settings called options, setting for the following geometric primitives called attributes and the geometric primitives themselves. In the following sections we will explain the most important hierarchies, options and attributes.

A statement in a RIB file is defined by the statement name followed by the statement parameters. The name and the parameters are given as ASCII strings. Here you can see a example of such a statement:

```
Translate 18.3095 -43.5387 46.8071
```

This statement defines a translation of the coordinate system in `x`, `y` and `z` direction.

5.1.1. Hierarchies

A node in the hierarchy of a RI scene is defined by a `Begin` and an `End` statement. E.g. a transformation hierarchy block is defined as follows:

```
TransformBegin
<Statements>
TransformEnd
```

Our implementation supports the following hierarchy sections:

- **Frame:** Defines a frame that is rendered. A RI scene can consist of an arbitrary number of frames. Our implementation only supports scenes with a single frame. When entering a Frame hierarchy all options are saved and restored at the end of the frame.
- **World:** All geometry of a frame must be defined inside a world block. The only exception are `Object` blocks which are not supported by our implementation. The rendering options cannot be changed inside the world node. The world block also defines the `world` coordinate system.
- **Attribute:** Saves the current attributes and restores them after the end of the block.
- **Transform:** Saves the current transformation and restores it after the end of the block.

These hierarchy sections also save the current attributes and restore them after the end of the block. The World section of a frame can contain multiple Attribute and transform sections. These nodes can also be nested arbitrarily. To get the parameters of a geometric primitive the attribute nodes from the primitive up to the root node are searched for attributes. If a attribute is defined multiple defines the last occurrence in the hierarchy is used.

Transformations

The transformation from a geometric primitive in the scene to the camera is defined by transformation statements and the transformation hierarchy nodes. The transformation hierarchy nodes only save and restore the current transformations. The transformations themselves are defined by statements inside the hierarchies. These statements are listed in section 5.1.2 and section 5.1.3.

There are three important named coordinate systems for a given geometric primitive. The `camera`, `object` and `shader` coordinate systems. The `camera` coordinate system is defined by the `Projection` statement and it defines the camera the scene is rendered with. The `object` coordinate system defines the coordinate system of the current geometric primitive. The `shader` coordinate system is the coordinate system in which the surface and displacement shader are defined. All parameters of the shaders are defined in this coordinate system if not stated otherwise. For all scenes we have rendered the `object` and `shader` coordinate systems are the same.

The transformation matrix for the transformation from a given coordinate system to another one is computed by multiplying the transformation matrices of the transformation statements in the current transformation section up to the target coordinate system. If the origin coordinate system is higher in the hierarchy than the target system the inverse matrix from the target to the origin system is used.

5.1.2. Options

In this section we will discuss the most important program options in a RI scene. Options are global settings in the Renderman scene. This is not a complete list of all supported options. For more informations see the RI specification.

- **Projection:** Defines the camera projection matrix. The RI specification defines `orthographic` and `projective` cameras, but only `projective` cameras are supported in our implementation. The `projective` camera takes the field of view as an additional argument. This statement also

defines the `camera` coordinate system in which the shading operations are performed.
- **Format:** Defines the resolution of the output image. In our implementation the resolution is preset to the given value, but can be changed manually by resizing the render output window.
- **Clipping:** Defines the near and far z-plane of the camera.
- **Shader Search Path:** The shaders of a scene are given as shader source code files as described in section 5.2. This option adds folders to the path in which the application searches for shader files.

5.1.3. Attributes

In the following we describe the most important attributes in a RI scene. Attributes are local settings that apply to subsequent geometric primitives. This again is not a complete list of all supported attributes.

- **Transformations:** The following statements push transformation onto the current transformation hierarchy. As described in section 5.1.1 the transformation matrices constructed from these statements define the coordinate systems in the scene.
  - **Scale:** Scales the model in `x,y,z`.
  - **Translate:** Moves the model in `x,y,z` direction.
  - **Rotate:** Rotates the model around a given rotation axis.
  - **Transform:** Transformation matrices can also be directly given as an array of 16 float values.
- **Materials:** The material of a primitive is given by the surface and displacement shaders that are defined in the attribute hierarchy of the primitive. The material names must correspond to a shader of the right shader type that can be found in the defined shader path. Additionally there is a number of standard shaders that are predefined and can be used without additional shader files.
  The parameters of the shaders are passed as name, value pairs. They can also be defined as statements in the RI scene. E.g. the `Color` statement defines the surface color. Point, Normal and Vector parameter are transformed from the current coordinate system to the `camera`

coordinate system when they are passed to the shader. This is done because the shading computations are performed in `camera` space whereas the parameters given in the RI file are always given in the current coordinate system. Our implementation supports two types of shaders. More information about the shaders can be found in section 5.2 and in the RI specification.

  - **Surface:** Defines the surface shader of a primitive. The shader is called before the rasterization of a micropolygon grid. It defines the surface color of the rendered geometry.
  - **Displacement:** Defines a displacement shader that is executed before the surface shading is performed. The displacement shader can move the grid points in the micropolygon grids to add small details to a model. It is not mandatory to give a displacement shader for a geometric primitive. If no displacement shader is given the displacement step of the Reyes pipeline is skipped.

- **LightSource:** Light sources are also defined as shader instances. Their parameters are defined the same ways as for the material shaders. Note that if the light has a position or direction associated with it, it is transformed to the `camera` space when the shader instance is created.

5.1.4. Geometric Primitives

A geometric primitive is given as a statement in RIB file. The parameters of a mesh are given as geometry attributes inside the statement. Those are name, value pairs, where the name specifies the attribute and the values are given as an array of values. The geometry attribute `P` defines the control point positions of a primitive and must always be present. The texture coordinates can be given as geometry attributes with the names `s` and `t` or as a single geometry attribute `st`. The RI specification also states that all shader parameters can be given as geometry attribute values, but our implementation does not support this at the moment.

The RI specification supports a number of different geometric primitives. In our implementation we only support Bezier patches and Catmull-Clark subdivision surfaces:

- **Patch:** Defines a patch. The RI supports bilinear and bicubic patches with an arbitrary base. Our implementation however only supports bicubic patches with a Bezier basis function. For more information on Bezier patches see section 3.1. The basis function is defined by the `Basis` statement which must occur before a patch is defined. The geometry of a bicubic patch is given by 16 control vertices. The texture coordinates can be given as an array of 4 points that correspond with the 4 corners of the patch.
- **SubdivsionMesh:** Defines a Catmull-Clark subdivision mesh as described in section 3.2. The mesh structure is defined by a series of arrays: The first array defines the number of vertices in each face in the mesh. One integer number is given for every face in the mesh. Then for each of this faces and each vertex in a face the point index is given. This index defines the index in the geometry attribute for the current vertex. After that a list of tags defines special attributes for certain faces, vertices or edges. E.g. a face can be tagged as a hole which means that it is not rendered.
  After the tags the geometry attributes are stated. The number of positions is defined by the maximum index in the index array. Apart from the position texture coordinates can be given. We support two different ways of defining the texture coordinates:
  - **Varying** means that every vertex has one texture coordinate. In this case, the number of texture coordinates must be the same as the number of positions.
  - **Facevarying** means that in every face, every vertex is assigned a texture coordinate. This means, that a vertex that is present in more than one face can have a different texture coordinate in each of these faces. In this case, the number of texture coordinates must be the sum over the number of vertices of all faces.

5.2. The Renderman Shading Language

As already state, materials in RI scenes are specified as shaders. Those shaders must be written in the Renderman Shading Language (RSL). RSL is

a C like language with the addition of several shading specific constructs and types. The grammar of the RSL is given in appendix A.

5.2.1. Language description

RSL shaders are given as shader source files with the extension `.sl`. A RSL can contain an arbitrary number of shader and function definitions. Our implementation supports three types of shaders: Surface, Displacement, and Light shaders (See section 5.2.2). The RI specification also defines Image and Volume shaders, but our implementation does not support them and they will not be discussed here.

A shader is defined by the keyword for the shader type followed by the name of the shader, the parameters, and the statements in curly brackets. The definitions looks very similar to a function definition in C.

A function definition is very similar to a shader definition, but instead of the shader keyword the return type is given. Functions can be called from within the shader body.

Here you can see an example of a light shader definition:

```
light pointlight(
  float intensity = 1;
  color lightcolor = 1;
  point from = point "shader" (0,0,0); )
{
  illuminate(from)
    Cl = intensity * lightcolor / L.L;
}
```

Note that apart from the shader type there are several differences to C. We will not discuss all of these differences, but in the following we will give an overview of the additional variable types, light loops, transformations and the built-in functions of the RSL.

Variable Types

In the following is a list of the variable types in the RSL. Note that the RSL does not support integer variables. If a integer constant is given it is converted into a floating point number.

- **float** 32-bit floating point number.
- **Point** A point in space. If not stated otherwise a point is given in the camera coordinate system. A point is described by three float numbers.
- **Normal** A face normal. Like a point it is also described by three float numbers.
- **Vector** A vector in space. Also defined by three float numbers.
- **Color** The standard color system in RSL is RGB, but colors can be transformed into other color spaces like HSV or HSl. RGB colors are described by three normalized floats.
- **String** The RSL supports strings and string operations. We do support the string variable type, but due to limitations on the GPU our implementation does not support string operations.
- **Matrix** A 4×4 matrix. Can be used as a transformation matrix.

Light Loops

Light loops are a special language construct used for lighting the scene. There are three types of light loops which must either be defined in a light or surface shader:

- **Illuminance:** Must be defined in a surface shader. This loop defines an iteration over all active light sources with an Illuminate or Solar loop. Within each iteration the light shader function of the current light source is called. Inside the Illuminance loop two additional variables are defined: `L` is the direction to the light source and `Cl` is the light color.
- **Illuminate:** Must be defined in a light shader. It is called in the iterations of the Illuminance light loop. It defines the casting of light from a specified point in space. The additional variable `L` defines the vector from the light source to the shaded surface point. The length of `L` is the distance between the light source and the surface.

- **Solar:** Is similar to the Illuminate statement, but does not specify a light origin. A light direction must however be given. The light is cast from infinity in this direction.

All three types of light loops can specify a cone. In the case of light shaders this cone specifies the direction in which light rays are cast into the world. This is useful e.g. for spotlights or directional solar light sources. In case of surface shaders it specifies from which directions the surface receives light. Often this is set to the normal vector $\pm 90°$, to only allow illuminating the front of the surface.

If a light shader does not specify a illuminate or solar statement it is classified as an ambient light source. In addition to the light loops there are three Built-in functions that loop over all light sources. They are described in the section built-in Function 5.2.1.

Transformations

Another special language construct are transformations between color spaces and coordinate systems. For example a transformation of the point `(0,0,0)` to the coordinate system `shader` looks like this.

```
point var = point "shader" (0,0,0)
```

This automatically multiplies the given value with the right transformation matrix. In this example the point transformation matrix from `camera` space to `shader` space would be used. The `camera` coordinate system is the standard coordinate system in shaders.

Color transformations work the same way for variables of the type `color`. The standard color space is `rgb`. Our implementation supports transformations to and from `hsl` and `hsv`.

Built-in Functions

In this section we will list some of the most important built-in functions in RSL. This is not a comprehensive list, and we do not include self explanatory functions like `min` or `max`. For a complete list of functions see the RI

specification [Pix05]. Note that our implementation does not support all built-in functions described in the specification.

- **transform, ntransform, ctransform:** transform points, normals, and colors to the given space. This performs the same transformations described in section 5.2.1. With these functions it is however also possible to specify the coordinate system from which the variable is transformed.
- **calculatenormal:** calculates the approximate normal at the given position. This function takes the position of the surface and can be used to recompute the normal after displacement mapping. The normal is approximated using the neighboring positions in the micropolygon grid.
- **ambient:** loops over all ambient light sources and adds up the light colors.
- **diffuse:** is equivalent to the light loop:

  ```
  illuminance(P, N, PI/2)
    C += Cl + normalize(L).N;
  ```

  It loops over all active light sources and computes the diffuse part of the lighting.
- **specular:** is equivalent to the light loop:

  ```
  illuminance(P, N, PI/2)
    C += Cl + specularbrdf(normalize(L), N, V, roughness);
  ```

  It loops over all active light sources and computes the specular part of the lighting.
- **texture:** is a texture lookup function. The texture name is the texture file name. The texture coordinates are given as normalized floats.

Preprocessor

The RSL supports a C-like preprocessor. It supports all C preprocessor statements. We did not implement all of them. A list of the supported preprocessor statements can be seen in section 6.4.1.

5.2.2. Shader types

Our implementation of the Reyes pipeline supports three shader types. They are described in the following:

Surface Shaders

Surface shaders compute the surface color of a geometric primitive. According to the RI specification, the surface shader also computes the opacity of the surface, but as we do not support transparent objects, the opacity is ignored. The surface shader function is called for every grid in the micropolygon grid. It takes the surface parameters as input, most importantly the position and the normal of the surface, but also the texture coordinates s and t. The surface shader is responsible for the lighting of the scene. This is done through the light loop which call the light shader functions of every active light source.

Displacement Shaders

Displacement shaders, like surface shaders are called for every grid point in the micropolygon grid. They are called before the surface shader and can displace the surface of a geometric primitive. Typically this displacement is along the normal of the surface and according to a displacement texture. The output of a displacement shader is the new position and the new normal of the surface. The normal of the surface needs to be recomputed inside the displacement shader after the surface is displaced. It is nor mandatory for a surface to have a displacement shader. If no displacement shader is given, the normal and position of the surface are not modified.

Light Shaders

Light shaders describe the light sources in the scene. Every light source is associated with a light shader function. Like described in the surface shader section, these functions are called in the light loops of the surface

shaders. The light shaders compute the light direction and light color that is accessible inside the light loops of the surface shaders.

6. Implementation

We implemented the Reyes renderer using C++, CUDA and Whippletree [Ste+14]. In this chapter we will first give a short overview of our program. Then we will discuss the subdivision on the GPU for Bezier patches and Catmull-Clark subdivision surfaces. Finally we will describe the integration of the RSL shading language into our program.

Program Overview In the following we will give a short overview of the different steps in the execution of our program.

1. **Parse the RIB File:** The RIB file is parsed using Boost Spirit [Guz]. This results in a hierarchy of RI statements.
2. **Compile the Shaders:** The shader source files given by the RIB file are compiled, written to a CUDA file, and then added to the Reyes pipeline. See section 6.4.
3. **Build the Scene:** The parsed RIB statements are processed to generate the scene data. This generates the geometry, the transformation hierarchy, the attribute hierarchy, the light sources and the render options. Shader instances for the rendering of the geometric primitives are created during this step.
4. **Load the Reyes Pipeline:** We decided to put the rendering pipeline and all other GPU related things into a separate library that is loaded at runtime. In this step, the RSL shader functions are added to the Reyes pipeline. The pipeline is compiled into a `dll`. This `dll` is then loaded by the program. It contains all methods to add the geometry to the scene and render it.
5. **Load the Geometry:** The scene geometry is prepared for rendering and then sent to the GPU. While doing this the shader parameters are also sent to the GPU.

6. **Render the Scene:** The scene is rendered. Unlike other Reyes renderers the result is not immediately written to the output image. The scene can be viewed in an OpenGL viewport. The camera can be moved around the scene.

6.1. Reyes Pipeline

As already mentioned, the Reyes pipeline is built using Whippletree [Ste+14]. The Reyes pipeline stages are implemented as Whippletree procedures. We implemented pipelines for rendering Bezier patches (section 6.2) and Catmull-Clark subdivision surfaces (section 6.3). Those two pipelines are combined into one Whippletree technique. Thus it is possible to render Catmull-Clark subdivision surfaces and Bezier patches at the same time. This is also important for the rendering of regular subdivision patches which uses some procedures from the Bezier patch pipeline.

The whole pipeline is outsourced to a library that is compiled and loaded at application startup. We decided to do this to support the loading of RSL shaders at runtime. The pipeline library is responsible for preparing the geometry for GPU subdivision, sending the geometry to the GPU, preparing and loading the shaders onto the GPU and rendering the scene.

Micropolygon Rasterization

In the rasterization step of the pipeline we perform a simple bounding box based rasterization: First the axis aligned bounding box of the micropolygon is computed. Then for every pixel in this bounding box an inside test is performed. If the pixel lies inside the bounding box the colors of the micropolygon vertices are interpolated using barycentric coordinates. Then a z-test is performed. Finally, the pixel is written to the output image. The last two steps are performed using a compare and swap loop because the graphics card we used does not supports 64-bit atomic minimum or maximum instructions.

For the rasterization we use one thread per micropolygon. This approach is faster than using a stamp based method considering that a typical micropolygon has a bounding box of 1 pixel as shown by Fatahalian et al. [Fat+09]

Holes The fact that the Reyes algorithm can produce different subdivision levels for neighboring patches can lead to holes in the output image. These are caused by the limitations of numerical precision of floating point numbers and the fact that different levels of subdivision use different points on the surface for the rasterization. E.g. if a patch is subdivided once more than a neighboring patch, this patch uses twice the amount of points for the micropolygons. As the geometric primitives we use are curved, the diced points do not necessarily lie on the edge of one of the micropolygons of the other patch. This effect is amplified when displacement mapping is used because in addition to using different points on the surface this points might also be displaced differently.

These holes are generally much smaller than a single subpixel. We therefor decided to slightly enlarge the micropolygons to close the holes.

6.2. Bezier Patches

Models made of Bezier patches usually consist of a number of separate Bezier patches. These patches need little preparation before they are sent to the GPU. The control points can be sent to the GPU without any preprocessing. The shading information, the texture coordinates and the u/v coordinates are added to the control points. The shading information consists of a shader id and the three parameter texture offsets for the displacement and surface shader (see section 6.4.3). The following Whippletree procedures implement the Reyes rendering pipeline for Bezier patches:

Bound This procedure is executed for every input geometric primitive. It clips patches that are completely outside the view frustum. It also checks if a patch needs to be split. If the size of the screen space bounding box is

below a certain threshold in x and y direction, the patch is forwarded to dicing. The threshold for dicing is determined by the biggest dice dimension. To choose the right dice procedure the size of the bounding box is again tested against the thresholds of the dice procedures with the different dice dimensions. If the bounding box is larger than the dice threshold, the patch is forwarded to the split procedure in U or V direction. This procedure uses 16 threads per input patch. Each thread is responsible for one control point.

Split U/V and Bound There are two split procedures. One for the split in U direction and one for the split in V direction. In this procedure, a given patch is split in halves using the DeCasteljau algorithm. Then, the same checks as in the Bound procedure are execute for each of the two new patches. They are then either clipped, forwarded to dicing, or forwarded to Split U/V. This procedure uses 4 threads per input patch. Each thread is responsible for one row/column of control points.

Dice and Shade There are three dice and shade procedures in our program. They differ in the number of micropolygons that are produced in the dicing stage. We call this number the dice dimension. The three procedures have a dice dimension of 15×15, 7×7, and 3×3. This results in a grid of points of 16×16, 8×8, and 4×4 respectively. Each of this grid points is first displaced using the current displacement shader. Then each grid point is shaded using the surface shader. Finally each micropolygon is rasterized.

We decided to implement three procedures with different dice dimensions to decrease the oversampling, and thus the rendering performance, for small patches.

The grid points are computed using the DeCasteljau algorithm. First, we compute a number of Bezier curves by subdividing the patch in U direction. The number of curves is given by the dice dimension. Then these curves are subdivided in V direction to get the positions of the grid points. These procedures use one thread per grid point. This results in 256, 64 and 16 threads for each of the dice dimensions. The rasterization is done using one

thread per micropolygon. They use three floating point numbers as shared memory per grid point for the grid point positions.

6.3. Subdivsion Surfaces

Due to their less regular topology Catmull-Clark subdivision surfaces are harder to render. We have split the rendering of subdivision surfaces into two stages. First, the surfaces are subdivided once on the CPU. Then a patch is created for every face in the subdivided mesh. Those patches are then sent to the GPU for rendering.

6.3.1. Subdivision on the CPU

Our implementation of Catmull-Clark subdivision surface on the GPU is only capable of subdividing quad faces. Catmull-Clark subdivision surface however can have arbitrary topology. Which means, that faces with more or less vertices can occur in a mesh. To render such meshes, each subdivision surface is first subdivided on the CPU once. This produces a mesh of quad faces. During this subdivision faces with more or less vertices are divided into a corresponding number of quad faces.

The subdivision is performed according to the rules seen in section 3.2. For the computation of the points the algorithms often need access to vertices nearby. E.g. the face point computation needs all the vertices in a given face. For this a half edge data structure is used:

The Half Edge Data Structure

The half edge data structure can be used to describe a mesh with arbitrary topology. A half edge can be seen as a pointer from one mesh vertex to another, with information of the surrounding mesh. It consists of a origin Vertex, a twin half edge, a next half edge, a previous half edge and a face:

- **Vertex:** The origin vertex of the half edge.

- **Next:** This is another half edge. The origin vertex points at the end point of the current half edge.
- **Previous:** This is also another half edge. The next pointer of this half edge points to the current half edge.
- **Twin:** Points to the half edge, that has the vertex of the face-next point as origin and the current origin vertex as end point. The twin is only present when the half edge does not lay on a mesh border.
- **Face:** Points to the face of the current half edge. The face contains additional information about the mesh. In our implementation each face contains its texture coordinates and a flag that defines if the face is a hole. If the face is specified as a hole in the RI scene it is not rendered.

With these pointers it is possible to navigate around the surrounding mesh. There are two important operations that are needed for subdividing a mesh:

- **Rotate Around Face:** To rotate a half edge around a mesh, the next or previous pointers of a half edge are considered.
- **Rotate Around Vertex:** There are two ways to rotate around a vertex: Forward, and backward. To rotate forward around the vertex we take the twin, and then the next pointer of the twin. To rotate backward, we take the previous pointer, and then the twin of the previous pointer. Rotation around the vertex is only possible if there is no border in the rotation direction.

In addition to the half edges we also store the vertices and faces. These are referenced by the half edges, but can also be accessed separately. The vertices contain the position of the mesh control points. The faces contain the texture coordinates and a pointer to one of the half edges in the corresponding face. Which half edge in the face is referenced does not matter as rotation around the face is cheap.

The Subdivision

Before the subdivision, the half edge data structure for a given mesh is built. Then the subdivision according to the rules of Catmull-Clark subdivision

surfaces is performed. The precise rules can be seen in section 3.2.1.

First, for every face a face point is inserted, by iterating over all faces. Then for every edge an edge point is inserted, by iterating over all half edges. These points are stored in maps to access them in the move vertex step of the subdivision. After the vertices are moved, the new faces are constructed from the previously computed points. For every face in the old mesh a number of faces is constructed. The number of new faces in a face is determined by the number of vertices in the old face. The texture coordinates are computed before the construction of the new faces.

Texture Coordinates As already stated in section 5.1.4, texture coordinates can be given in two different ways: varying and facevarying. The only difference in handling these two types is in the construction of the initial mesh structure.

The texture coordinates are stored in the faces of the subdivision mesh. There is one coordinate for every vertex in the face. During subdivision the coordinates are interpolated linearly along the edges and the faces. This means, that the texture coordinate of a face point is the mean of all vertex texture coordinates. The coordinate of a edge point is the mean of the end point texture coordinates in the corresponding face. The coordinates on the vertex points do not change during subdivision.

6.3.2. Subdivision on the GPU

After the first subdivision on the CPU the subdivision meshes are guaranteed to only have faces with four vertices. We now can generate the patches for the subdivision on the GPU.

Patch Creation

We create one patch for every face in a given subdivision mesh, except when a given face is classified as a hole. A patch contains the face and all vertices that are needed to subdivide this face. This means, that all vertices of all

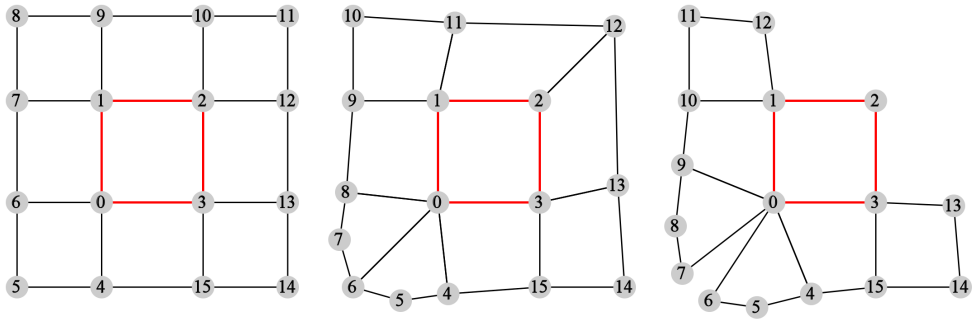
neighboring faces are part of the patch. We call the face that is subdivided the inner face, and the neighboring faces outer faces. In addition the patch needs the texture coordinates and the information about the shaders. This includes the shader id and the parameter texture offsets for the displacement and surface shaders.

The vertices of the faces in the patch are stored in an array in the patch. The first four entries are the positions of the inner face vertices in clockwise order. Followed by the vertices of the outer faces that are not part of the inner faces. The order of these vertices is determined by the position around the inner patch. They are arranged clockwise around the inner face. The first vertex after the inner vertices is part of a corner face in the first corner of the patch. Examples of the point order can be seen in figure 6.1. In these examples the red face in the middle is the inner face. This is the face that is rendered by subdividing this patch. The number on the vertices denote the index of the point in the patch.

The subdivision of a patch is performed by four threads. Every thread is responsible for subdividing the faces adjacent to the corner vertex. The inner face point is computed by the first thread. The edge points of the inner face are computed by the four threads. Each thread is responsible for the edge that starts at the respective corner point and ends at the next corner point in clockwise direction. The face point of the face that contains this edge is also computed by the same thread. We call those faces edge faces. The other outer faces are called corner faces.

We decided to create three types of patches for performance reasons. These three types can handle different kinds of topology around the face. They are called regular, multi-face and border patches.

Aside from the points each patch also contains two arrays of integer for the number of corner faces per corner and the border position at each corner. We will explain the meaning of those number in the following paragraphs. These arrays are present in all patch types even tough they are only used in multi-face and border patches. We decided to do this because it makes it easier to cast the type between patch types. This is needed in the check procedure and the split procedures. These procedures insert a patch into the queue for the right patch type after checking or subdividing the input patch.



(a) Regular Patch (b) Multi-Face Patch (c) Border Patch

Figure 6.1.: Example patches from Catmull-Clark subdivision surfaces.

The patches also contain four texture coordinates and four uv coordinates. One of each for every corner of the inner face. They also contain information about the displacement and surfaces shaders: The shader id and the parameter offset textures for each parameter type.

Regular Patch A regular patch consists of a inner face surrounded by eight outer faces. Each of the vertices in the inner face has valence four. An example of a regular patch can be seen in figure 6.1(a). Regular patches always have 16 control points. The subdivision of such a patch is straightforward: The indices of the needed points for the face points, edge points and vertex points in each corner can be computed easily. The subdivision needs very little branching and no loops. After the new points are computed a new face is created in every corner. All the newly created patches are regular patches. Again the assembly of the new patches is straightforward. Thus the subdivision for these kinds of patches is the fastest.

Additionally to the fast subdivision these patches can be converted into Bezier patches as seen in section 3.2.2. This makes it possible to use the Bezier patch dice procedure to dice these kind of patches. This procedure is capable of dicing a patch into a larger number of micropolygons. Therefor less subdivision steps are necessary to produce micropolygons that are the size of one subpixel.

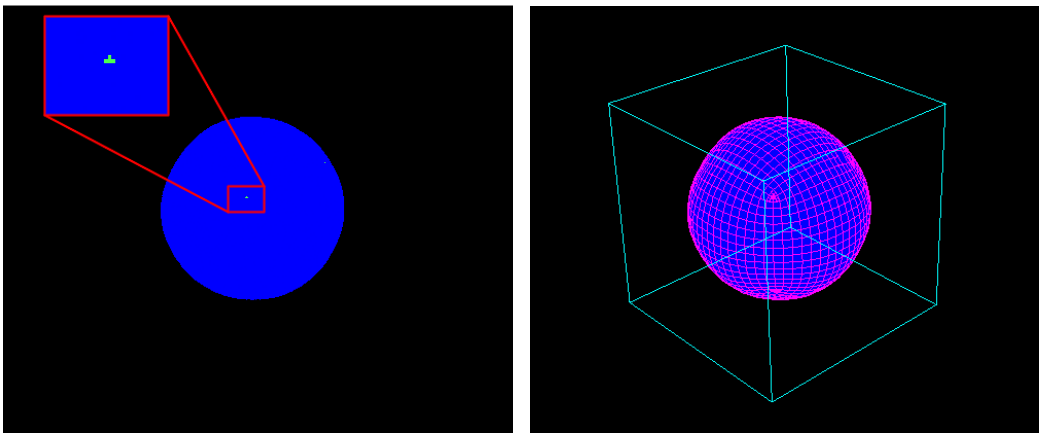
Multi-Face Patch In a multi-face patch each vertex of the inner face can have an arbitrary number of neighboring vertices, with some limitations (See limitations section below). An example of a multi-face patch can be seen in figure 6.1(b). This example shows a patch with a valence five vertex in the lower left corner and a valence three vertex in the top right corner. Note that every regular patch is also a multi-face patch and can therefor be subdivided with the multi-face procedure.

The subdivision of such a patch is more complicated than in the regular case: Before the subdivision the offsets for the points in each corner have to be computed. The patch contains information for how many corner faces each vertex of the inner face has. From the number of faces per corner point we can compute the number of points at the corner, the number of face points, and the number of edge points. From these we can compute the index offsets for the points of the outer faces and the indices of the inserted points in each corner. With these offsets the outer faces in each corner can be assembled. The computation of the corner face and edge point needs a loop which makes the computation slower than the regular case.

The subdivision again produces one new patch for every corner. The assembly of the new patches is again more complicated than in the regular case. Each new face may have more than one corner face.This is however only possible for the corner where the vertex point of the initial patch has multiple faces. This means, that after one subdivision on the GPU, or two subdivisions in total, each multi-face patch can have at most one irregular corner. The patches that are created at the regular corners of a patch are regular patches. Therefor they can be inserted into the queues for the regular patches. This speeds up the rendering because the regular split procedure is faster and regular patches can be diced into more micropolygons.

For an example mesh with multi-face patches see figure 6.2. The figure on the left shows which dice procedures where used for which output pixels. Here the irregular vertex near the camera only produces three pixels that were diced with the multi-face dice procedure. The surrounding of the irregular vertex was converted to regular patches after subdivision. The figure on the right shows that the geometry around the irregular vertex is subdivided into much smaller patches before dicing. This means that more subdivision steps must be performed. Notice that the patches get smaller as

the irregular vertex is approached. This is because the regular patches are produced at every subdivision level. The first subdivision levels produce the bigger patches that are then immediately diced. The patches get smaller after each subdivision.



(a) Visualization of the Dice Procedures (b) Patches before dicing and Inital Mesh

Figure 6.2.: Visualisation of dicing of multi-face patches. The rendering shows the subdivision of a cube mesh. The blue pixels were diced using the Bezier dice procedure. The green pixels were diced using the multi-face dice procedure. The red rectangle in the left figure is shown magnified.

In the dicing stage the patch is subdivided once more. For the vertex points, the limit surface and the normal rules are used as seen in section 3.2.1. After the subdivision these rules are also applied to the edge points and the face point of the inner face. These points all have valence four. Therefor the computation is straightforward. This produces a micropolygon grid of 2×2 micropolygons. From here one the dicing is similar to the one of a Bezier patch. The grid points are displaced with the displacement shader. Then they are shaded using the surface shader. Finally each micropolygon is rasterized.

Border Patch In a border patch each vertex of the inner face can be adjacent to a border of the subdivision surface. Like a multi-face patch the

vertex points can have an arbitrary number of neighboring vertices, with the same limitations. An example of a border patch can be seen in Figure 6.1(c). This example shows a patch with a border at the lower left vertex and a border that goes from the upper left vertex to the lower right vertex. Note that every multi-face patch is also a border patch and can therefor be subdivided with the border procedures.

The subdivision is more complicated than the multi-face patch: The index computation is similar to the multi-face patch. In addition to the information about the number of corner faces per vertex these patches also need information about the borders. This information is given as an integer value and can have the following values:

- **-1:** This means, that there is no border at the vertex.
- **A positive number n:** This means, that there is a border at the vertex. The border is between the corner face n-1 and the corner face n of the corner.
- **0:** This means, that there is a border at the vertex. The border is between the edge face of the next corner counterclockwise in the face and the first corner face of the corner.
- **-2:** This means, that there is a border at the vertex. The corner is at the edge face between the corner and the next corner clockwise in the face.

This must be considered in the index computation. The vertices and edges that are on the border of a mesh need different subdivision rules as seen in section 3.2.1. This increases the divergence in case of a border.

The subdivision again produces four new patches. One in each corner. The assembly of the patches is similar to the multi-face patches. If a vertex of the initial patch is regular, the subdivision produces a regular patch in this corner. As in the multi-face case, these patches are inserted into the queues of the regular patches. The assembly of the new patches needs to consider the borders between the corner faces and the possibly missing edge faces. It is therefor slower than the multi-face case.

The dicing of border patches is similar to the multi-face patches. If there is a border adjacent to a vertex or edge however, the subdivision rules for borders have to be used. This again increases the divergence when a border

is present. After the 2×2 micropolygon grid is created, the dicing is the same as in the multi-face patches.

Procedures

In this section we will describe the Whippletree procedures that are used in rendering Catmull-Clark subdivision surfaces on the GPU.

Bound This procedure is executed for every input geometric primitive. It is the same for all three types of patches. The patch type is checked to insert the primitive into the right queue. The procedure is similar to the bound procedure for Bezier patches. It clips patches that are outside the view frustum. It checks if a patch needs to be split. For this, the screen space size of the inner face is considered. If the size of the axis aligned bounding box of the face is below a certain threshold, the patch is forwarded to dicing. Due to the different dice procedures for regular and multi-face/border patches this threshold is different for the different patch types. If the face is too big, the patch is forwarded to the split procedure of the current patch type.

This procedure uses 4 threads. Each thread is responsible for one corner of the inner face. It uses a single 32 bit integer as shared memory. This memory is used for the culling against the view frustum.

Split Regular/Multi-Face/Border This procedure splits a given patch into four smaller patches. The split algorithms are described in the previous section. After the subdivision the newly created patches are checked again with the same formula as in the bound procedure. They are then either clipped, forwarded to the according dicing stage or split again. The multi-face and border procedures check the topology of the new patches and insert the patches into the queues for the regular procedure if possible. The border procedure also checks if a subdivided patch can be inserted into the queues of the multi-face procedures.

The split procedures use 4 threads. Each thread is responsible for computing the face, edge and vertex points around a corner. It is also responsible for

computing the offsets and the assembly and insertion into a queue of the four subdivided patches.

Dice Regular This procedure performs the conversion of a regular subdivision surface patch to a Bezier patch and then uses the Bezier patch dice function for the dicing stage. The algorithm for the conversion to a Bezier patch can be seen in section 3.2.2. The procedure first converts the input patch to a Bezier patch. Then this new primitive is inserted into the according Bezier dice procedure. There is one procedure for every Bezier dice procedure (dice dimensions of 15×15, 7×7, and 3×3).

This procedure also uses 4 threads. Each thread is responsible for computing the four Bezier control points near the corner of the input patch. The procedures does not use shared memory.

Dice Multi-Face/Border This procedure dices the input patch into a grid of 2×2 micropolygons. The computation of the grid points is described in the previous section. After the dicing each grid point is displaced using the displacement shader. Then each grid point is shaded with the surface shader. Finally each micropolygon is rasterized.

The procedure uses 4 threads for the subdivision into the 4 micropolyons. It then uses 9 threads, one for each grid point, to displace and shade the grid points. Due to limitations in the Whippletree framework this procedure is execute using 16 threads.

Limitations

Our implementation of Catmull-Clark subdivision surface on the GPU has several limitations.

- **Valence 2 Vertices:** We do not currently support vertices with valence two. It would be possible to include the support of valence two vertices into our patch structure, but the performance would suffer, because they introduce several special cases.

- **Number of Vertices per Patch:** Catmull-Clark subdivision surface support vertices with arbitrary high valence. To support this we would need subdivision patches which support an arbitrary number of points in the patch. This would mean that each patch may use a different amount of memory. Whippletree does not support this, and if it was supported the performance would probably suffer. Therefor we limited the number of points in a patch to 24. This means if a patch only has one irregular vertex with no border, this vertex can have a maximum valence of 8.

6.4. RSL Shader Compiler

The RSL shaders are integrated into our pipeline using a transcompiler to CUDA. The compiler first performs a preprocessor pass, then the Parser creates an abstract syntax tree (AST) from the resulting output. The CUDA code is created from the AST. Finally the generated code is compiled by the CUDA compiler to get the executable shader functions.

6.4.1. Preprocessor

The preprocessor supports the following statements:

```
#define identifier definition
#define identifier(arg0, ..., argn) definition
#ifdef identifier
#ifndef identifier
#endif
#include<filename>
```

The statements have the same semantics as the equivalent preprocessor statements in C. Note, that our implementation does not support all preprocessor statements defined in the RI specification.

6.4.2. Parser

The RSL parser was created using Boost Spirit [Guz]. The grammar for RSL can be seen in appendix A. The parser creates an AST that represents all statements in the shader. The AST is not processed before it is given to the code generator.

6.4.3. CUDA Code generation

The AST is handed to the CUDA code generator. The code generator tries to produce CUDA code that looks as similar to the original RSL code as possible. However, as CUDA and RSL are different languages this is not always possible. E.g. RSL has built-in light loops, that are converted into `for` loops over all lights. Additionally, the code must be callable from the Reyes pipeline without knowledge of the shader parameters. Therefor the shader parameters are loaded inside the shader function.

Types

The transcompiler does not have a type checker. It assumes, that the code only contains valid type conversion and the CUDA compiler takes care of the conversions.

In the following I will show how the different types are handled. We use a mathematics library which provides the `float3` and `float4x4` data types.

- **Strings:** Our implementation only supports string constants as string operations on the GPU would be difficult to implement and probably very slow. All string constants are converted into an unsigned 32 bit integer using the standard C++ hash function `std::hash`. The constants in the RSL source code are hashed at compile time. The ones in the RIB files are converted when the shader instances are created.
- **Vector Types:** All vector types (Color, Point, Vector, Normals) are represented as a `float3` variable. This means that conversions between these types are possible and will not produce a warning from the compiler.

- **Transformation Matrices:** Are represented as a `float4x4` variable.

Shader Parameter

It is not feasible to give the shader parameters directly as parameters to the CUDA functions because the number of parameters can be arbitrarily high. This means, that for every shader the Reyes pipeline would need to be recompiled, which takes a considerable amount of time with Whippletree. Additionally, the GPU memory consumption and transactions would grow as the parameters would need to be present in every subdivided patch.

We therefor decided to put the parameters values in texture memory. The values are distributed over three texture:

- **Float Texture Values:** Contains the parameters of type float, string and matrix. The string texture values are converted to an unsigned 32 bit value. The bits of the integer are then placed in the floating point texture and interpreted as an integer after loading. It only contains the matrices that are defined as parameter of the shader and not as transformations.
- **Transformation Texture Values:** Contains the matrices for the transformation between the different coordinate spaces. The compiler requests all the needed transformations. The first transformation always is the point transformation from `shader` to `camera` space. The second transformations is the normal transformation from `shader` to `camera` space.
- **Float3 Texture Values:** Contains the values of all the vector types.

To access the values, each shader instance contains an offset for each of the textures. The offset points to the first parameter value of the respective type. The order of the values in the texture is the same as the one in the shader definition. The position of the values in respect to the offset is determined in the shader compiler.

Light Loops

Solar and Illuminate The light loop statements in the light shaders are converted into `if` statements. The `if` condition checks, if the light is inside the cone of the calling illuminance loop and if the surface position of the calling surface shader is inside the cone of the light source. The checks are only performed if the mentioned cones are specified. Inside the if statement the light direction is computed.

For every light loop, the whole light shader is executed.

Illuminance The illuminance statement loops over all active light sources. It is converted to a for loop that loops over all light sources in the scene instead. The number of light sources, the types of lights sources, and its parameters are again stored in textures:

- **Light Shader ID Texture:** Contains the shader ids of the different light sources. The id determines which light source is associated with which light shader. The first value of this texture contains the number of lights in the scene. Note that our implementation always treats all the lights as activated. The RI specification mentions an active light list, which we do not support.
- **Light Shader Offset Texture:** Contains the offset for the parameter values of each light source. For each light source there are three offset values (float, transformation, float3). The values are stored in the same three textures as the values for the displacement and surface shaders.

So, for a given `Illuinance` loop, the number of lights in the scene is loaded from the texture. Then for every entry in the light list, the shader id and the parameters offsets are loaded from the offset texture. Then the shader function with the given id is executed. Finally, the statements defined inside the illuminance loop are executed. Inside this loop the incoming light direction `L` and the light color `Cl` are defined and set by the executed light shaders.

Textures

Our implementation supports textures that are given as `tiff` files. The textures are loaded after the RI scene is assembled from the RIB file, but before

the Reyes pipeline is executed. In our current implementation the texture files must be defined in the RIB file. Texture names that are embedded in the RSL code are not recognized.

For every loaded texture, a CUDA texture object is created. These are placed in a CUDA source file which is then included in the Reyes pipeline. The file is then either included in the generated shader file (See section 6.4.4) or a separate file is created, which is then linked to the Reyes pipeline. The generated file also includes functions for accessing the textures by filename as specified in the RI specification. These functions take texture coordinates as normalized floating point coordinates. Like all string values, the texture filenames are hashed at compile time.

Built-in Functions

The RSL defines a set of built-in functions that the shaders may use. As already mentioned in section 5.2.1 our implementation only supports a subset of the functions found in the specification. The functions are implemented in CUDA. Some of the functions need access to standard shader parameters and some special parameters that were added by our implementation. E.g. for the `calculateNormal` function we need access to a shared memory array to approximate the normal using the neighboring positions.

Multiple Shader Files

The shader path of a RI file normally contains more than one shader source file. Each of these files is compiled to CUDA. The resulting code is then written into one generated shader file. It is possible, that the different shader source files contain shaders and functions with the same name. To support this, we put all functions and shaders of a single source file into a unique namespace.

Limitations

Our implementation of RSL has several limitations. They exist partly because of performance concerns on the GPU and partly because of our incomplete implementation.

- **String operations** As already mentioned in the type section, our implementation computes a hash of every string in the shader. This makes it impossible to support string operations like `concat`. The only string operations that are supported are assignment and comparison.
- **Return Type Polymorphism:** The RSL allows for functions to be overloaded based on the return type. E.g. the `noise` function can return either a single `float` value or one of the `float3` type values. Our implementation does not have a type checker which could infer the most suitable return type at compile time, and CUDA does not allow return type polymorphism. Therefor we do not support overloading on return types. There are several functions in the built-in library that define multiple versions with different return types. For these functions we only implemented one return type.
- **Calls to Some Built-in Functions from other Functions:** As already mentioned, some of the built-in functions need access to the shader parameters. It is not possible to call these functions from within other functions because these parameters are only available directly in the shader. Built-in function that do not need any of these parameters can be called from anywhere.
- **Active Light List** As already mentioned in the light source section, we do not support the active light list. The RI specification states that defined lights can be activated and deactivated using the `Illuminate` statement. In our implementation all lights are assumed to be active for the entire scene.

6.4.4. Integration into the Reyes Pipeline

The Execute Shader Function The compilation of the RSL shaders is performed on program startup. After the generation of the CUDA code for all shader source files, we generate the functions that calls the shader.

The Reyes pipeline cannot call the shader functions directly as they are only known after the shaders were compiled. Therefor a function that calls the right shader function is created by the CUDA code generator. The function takes the shader id, the parameter texture offsets, and the standard shader parameters and calls the shader with the corresponding id. A separate function for displacement and surface shaders is created. There is no function for light shaders as they are only called from within the surface shader functions.

Integration into the Pipeline After the code generation, the generated CUDA code is written into a CUDA file. The same is done for the texture access code. This files are part of the Reyes pipeline project. When the shaders and the texture information are written to their respective files, the Reyes pipeline project is compiled. The shader and texture functions are linked to the already compiled Reyes pipeline. The inclusion of the functions into the pipeline CUDA file would result in a better performance at runtime, but the compile time using Whippletree makes this impractical.

The resulting `dll` is then loaded by the program.

7. Results

In this chapter we will discuss the results of our implementation of the Reyes rendering pipeline. First, we will list the scenes that we used for evaluation and show renderings of these scenes. Then we will compare the renderings of the hole closing algorithm with different parameters. Finally we will discuss the performance impact of multiple parameters of our implementation.

All renderings were performed on a PC with an Intel i5-4590 CPU @3.3GHz, 8GB RAM and a Nvidia GeForce GTX680 GPU with 4GB VRAM. They were run on Windows 8.1 using CUDA 7.5 and Microsoft Visual Studio 2013.

7.1. Rendering Results

7.1.1. The Scenes

We evaluated our implementation using multiple scenes in RIB format. The scenes are described in the following. All renderings were performed using 16 times supersampling. For the hole closing algorithm a factor of 0.1 was used. For comparison the Dragonhead and the Killeroo scene were also rendered with Pixar's Renderman [Pixa] version 20. In table 7.1 we show information about the geometric primitives in the scenes and how many patches were created to render the scene.

	#Subdivision Meshes	#Subdivision Faces	#Patches
Cube	1	96	384
Killeroo	466	9218	11576
Dragonhead	na	na	1236
Killeroo 2	11	4308	15048
Teapot	1	448	1792

Table 7.1.: Informations about the geometric primitives in the scenes. The values show the number of Catmull-Clark subdivision surface in the RIB file, the number of faces in these meshes and the number of patches that were sent to the GPU. The Dragonhead scene does need have any subdivision meshes as it only contains Bezier patches.

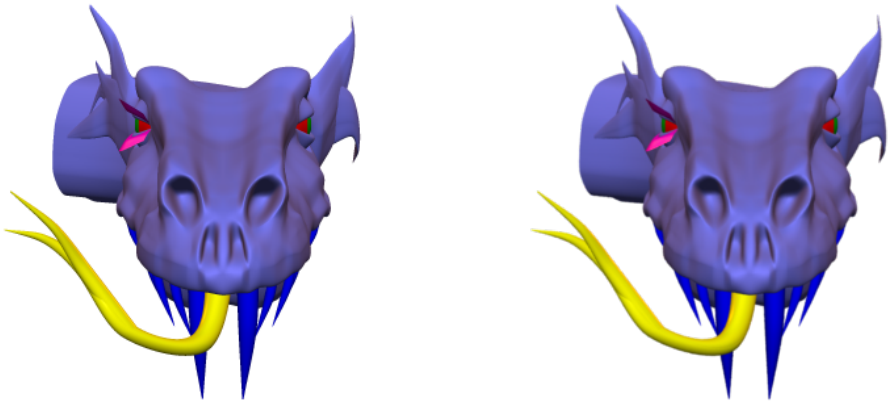
Dragonhead

The Dragonhead model [1] is the only model we evaluated that consists of Bezier patches. It is modelled with 1236 Bezier patches. For the surface a plastic shader is used, with different colors for different parts of the model. The model does not use a displacement shaders. Renderings of the Dragonhead can be seen in figure 7.1.

Cube

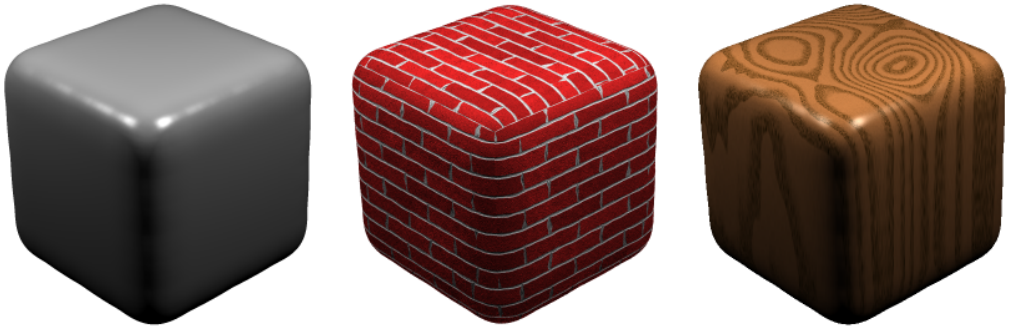
The Cube model was exported from Blender using the plugin RIBMosaic [Gab]. The model is a single Catmull-Clark subdivision mesh. The control mesh is a cube with 16 quad polygons on each side, which produces the appearance of a Cube with smooth edges and corners. In Figure 7.2 renderings of the Cube model with three different materials can be seen: The standard plastic shader, a procedural brick shader and a procedural wood shader. The two procedural shaders were taken from the shader library of the RIBMosaic plugin. The bricks in the brick shader are computed from the texture coordinates of the model. The wood texture is computed from the surface positions. They can be found in the surface shader files `brick1.sl` and `wood2.sl`.

[1] Dragonhead: http://ricpp.sourceforge.net/samples.html



(a) Our Implementation (b) Pixar's Renderman

Figure 7.1.: Dragonhead model rendered with our implementation and Pixar's Renderman.



(a) Plastic (b) Brick (c) Wood

Figure 7.2.: Renderings of the Cube model with different shaders.

Killeroo

The Killeroo model [2] consists of Catmull-Clark subdivision surfaces. It is composed of 466 subdivision surfaces. Each of those only consists of few faces surrounded by hole faces. In figure 7.3 a rendering of the Killeroo model can be seen. The model uses a displacement shader with a displacement texture. The surface shader also has a texture and is otherwise similar to the standard plastic shader.

In figure 7.4 an illustration of the dicing of the Killeroo model is shown. In figure 7.4(a) the borders of the patches before dicing them into micropolygons are shown. The two images below show an illustration of the dice procedures that were used to compute the micropolygons for each pixel in the output image. A blue pixel means, that it was diced using the dice procedure for regular patches, i.e. it was converted into a Bezier patch before dicing. A green pixel means that it was diced using the dice procedure for multi-face patches, and for a red pixel the dice procedure for border patches was used. The rendering of these illustrations were performed using no supersampling for a better visualisation. The two illustration of the dice procedures were produced with two different settings of the Reyes pipeline. The illustration in Figure 7.4(b) was rendered by always using the most efficient type of patch type after each split operation. The image is mostly composed of pixels that were diced with the regular dice procedure as after each split of a multi-face or border patch 3 of the 4 created patches are regular patches. In Figure 7.4(c) the patches that were produced by each split operation used the same patch type as the initial patch. In this image the different patch types of the initial patches around the non-regular vertices can clearly be seen.

The renderings using the real surface shaders of the model produce the same result with each method. The performance of the method shown in Figure 7.4(b) however is significantly better as discussed in section 7.2.3.

[2] Killeroo: http://ricpp.sourceforge.net/samples.html

(a) Our Implementation (b) Pixar's Renderman

Figure 7.3.: The Killeroo model rendered with our implementation and Pixar's Renderman.

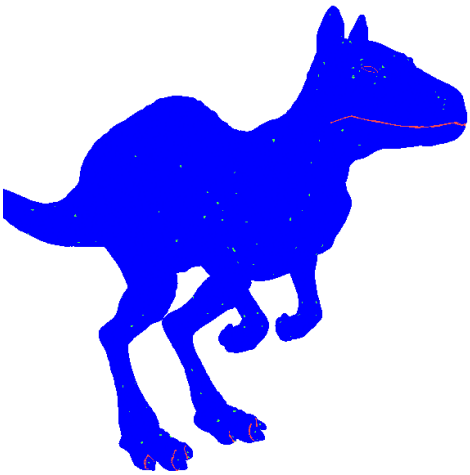
Killeroo 2

The Killeroo 2 model [3] is similar in appearance to the Killeroo model. This model however is composed of 11 larger subdivision surfaces instead of the high number of small surfaces in the first Killeroo model. The two meshes also differ in the size of the model and the topology of the meshes. The first Killeroo model for example has several borders patches around the mouth, the eyes and the toes whereas the Killeroo 2 model has no border patches. Additionally the renderings are performed from a different viewpoint and are rendered in a different resolution. Therefor we do not compare the performance or appearance of the two Killeroo models.

In Figure 7.5 renderings of the Killeroo 2 model can be seen. The left renderings shows the model with a plastic shader whereas the right one was rendered using a wood shader. The wood shader is the same as in the Cube

[3] Killeroo 2: http://www.headus.com.au//samples2012/killeroo-mk2-subd-2x1/download.html

(a) Patches before Dicing

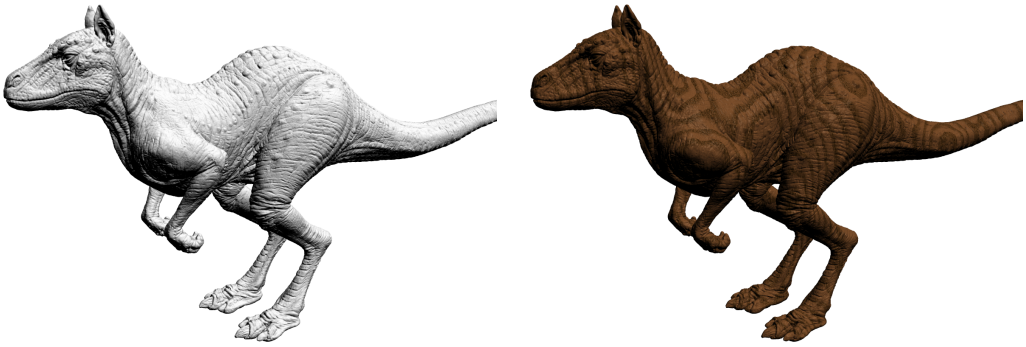


(b) Dice Procedures: Optimal after Split

(c) Dice Procedures: Same after Split

Figure 7.4.: Illustration of the patches before dicing and the different subdivision surface dice procedures. Subfigure (a) shows an illustration of the patches before dicing. Subfigure (b) shows the dice procedures when regular patches are used after a split operation if possible. Subfigure (c) shows the dice procedures when the patch types remain the same after a split operation. The pixel colors depict the patch type: Blue: Regular, Green: Multi-Face, Red: Border.

model. In both renderings we use a displacement shader that displaces the surfaces according to a displacement texture.



(a) Plastic (b) Wood

Figure 7.5.: The Killeroo 2 model. The two renderings show the model with two different shaders.

Teapot

The Teapot model [4] is a model of the Utah teapot using Catmull-Clark subdivision surfaces. It consists of one subdivision mesh. In Figure 7.6 you can see a rendering of the Teapot model. The model was rendered using the standard plastic shader.

7.1.2. Holes

As described in section 6.1 we implemented an algorithm to close the holes that appear in the rendering through different subdivision levels an floating point precision. In figure 7.7 example outputs of these hole closing algorithm can be seen. The images show a rendering of the Killeroo 2 model with different enhancement factors. The image was rendered with no supersampling and a green background to better show the holes and

[4] Teapot: http://ptex.us/samples.html

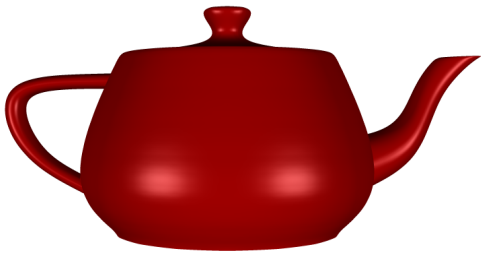


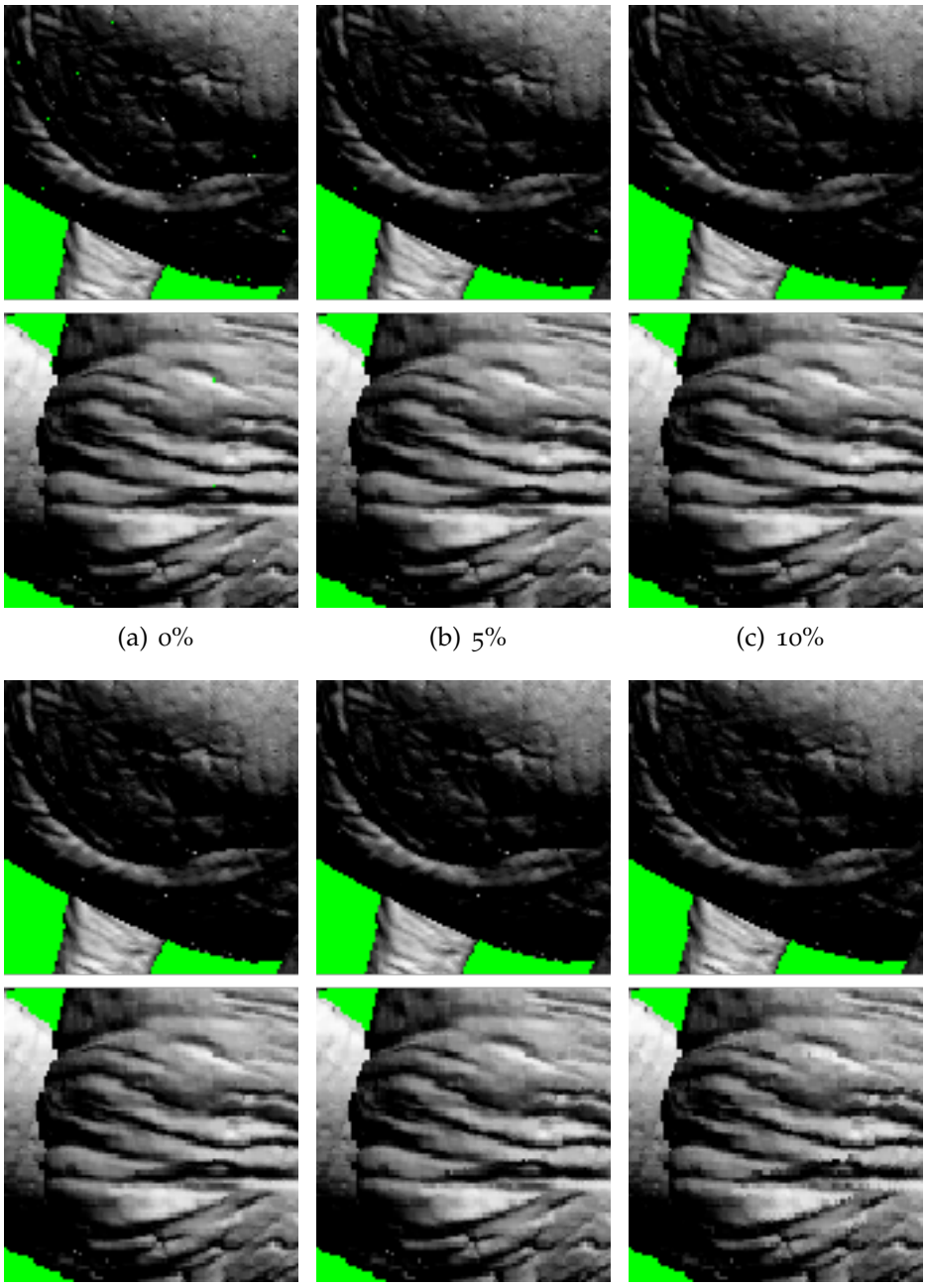
Figure 7.6.: The Teapot model.

artifacts. Note that some holes appear green and some grey. The green holes show the background behind the models whereas the grey ones show the arm of the model. We chose to show the belly of the model because most holes can be seen in this region. The knee on the other hand shows little holes, but in this region the artifacts created by our algorithm are more visible.

The lower enhancement factors show little artifacts but a higher number of holes in the rendering. The higher the factor gets the less holes can be seen. Even when a low factor of 5% is used a significant part of the holes disappear. At a factor of 100% no holes are seen in the rendering, there are however visible artifacts. These artifacts can be seen best in regions with high contrast, e.g. at the knee of the model. They appear because the resulting colors of micropolygons are written to surrounding pixels. A few of these artifacts start to appear even when the enhancement factor is only 5%. They get noticeable at around 20%. We decided to use an enhancement factor of 10% as a trade off between artifacts and holes.

7.2. Performance comparison

In this section we will discuss several options and their influence on the performance of our implementation. All performance comparisons where performed using the fastest configuration if not mentioned otherwise. In this configuration the supersampling is set to 1, the generated shader and



(a) 0% (b) 5% (c) 10%

(d) 20% (e) 50% (f) 100%

Figure 7.7.: Comparison of the hole closing algorithm with different factors. The images show parts of the Killeroo 2 model. The upper ones are near the belly. The lower ones show the knee of the model. The captions denote the enhancement factor of the hole closing algorithm.

texture code is included into the Reyes pipeline, subdivision surfaces always use the best patch type and three dice procedures were used.

7.2.1. CUDA Linking Performance

	Cube Brick		Killeroo		Dragonhead	
Time	Run (ms)	Comp (s)	Run (ms)	Comp (s)	Run (ms)	Comp (s)
Linking (R)	40	18.19	101	15.94	35	15.10
Linking (G)	38	18.16	95	15.89	31	15.15
Linking (GT)	38	14.35	95	12.23	31	11.47
No Linking	33	52.48	56	51.64	21	49.30
No Shader	9	0.45	44	0.45	11	0.45

	Killeroo 2		Killeroo 2 Wood		Teapot	
Time	Run (ms)	Comp (s)	Run (ms)	Comp (s)	Run (ms)	Comp (s)
Linking (R)	193	16.38	288	18.81	94	15.14
Linking (G)	187	16.40	274	18.47	84	15.06
Linking (GT)	189	12.70	271	14.72	84	11.49
No Linking	124	49.96	242	53.13	54	49.22
No Shader	79	0.45	79	0.45	35	0.46

Table 7.2.: Runtime for different CUDA linking configurations. The runtime per frame (Run) is given in milliseconds. The time the Reyes pipeline took to compile the shaders and link them (Comp) is given in seconds. The letters in the parentheses define where the parameter textures and the shader textures are located.

When including the compiled RSL shaders into the Reyes pipeline we have a few choices how we include the generated code into the library. We can either place the code into a separate compilation unit (Linking) or into the Reyes pipeline compilation unit. In addition to the shader code we also have options on where to place the parameter textures and the shader textures.

In CUDA linking different compilation units can have a big influence on the performance of the generated program. This is mainly because the compiler

can optimize the program better if all code is known in the compile step. Especially the shader calling functions can utilize these optimization because the RI specification specifies a high number of standard shader parameters. Most shaders do not use all of these parameters so they can be removed by the compiler if this fact is known at compile time.

The performance results of these different placement can be seen in table 7.2. In the following we will discuss the different configurations:

1. **Linking (R):** The Reyes pipeline, the shader code and the shader texture code are placed in three different compilation units. The parameter textures are placed with the Reyes pipeline. This configuration has the worst runtime. That is because parameter textures are mainly used by the shader code. The Reyes pipeline only loads the transformation matrices from these textures.
2. **Linking (G):** This configuration is similar to the first one. The only difference is that the parameter textures are placed with the shader code. It is 5-10% faster than the first one. The compile time is practically the same.
3. **Linking (GT):** This configuration is similar to the second one, but in addition to the parameter textures, the shader textures are also placed with the shader code. The runtime of models without textures is exactly the same as in the second configuration. This is because the shader functions are not used. The Killeroo also has the same runtime as before. The Killeroo 2 with the wood shader is slightly faster than before, but the Killeroo 2 with the plastic shader is slightly slower. This is likely to different optimization steps of the CUDA compiler when all the code is known versus the linking of the textures. In this configuration the compile time is lower than in the previous cases because the linker needs to link one fewer file. The compile time will however be worse when only the texture files have changed.
4. **No Linking:** This configuration places everything in they Reyes pipeline compilation unit. It is about 15-40% faster than the previous ones. That is because the compiler can perform more optimizations when all the code is present. The optimizations have a higher impact when the shader is simple. E.g. the Dragonhead with its plastic shader has a 33% decrease in render time, whereas the cube with the more complicated brick shader has only a 15% decrease. This is because the shader call

itself needs a larger percentage of the overall render time when the shader is faster, but also because the simple shaders use less of the standard shader parameters which can be optimized away.

5. **No Shader:** We also performed a test with no RSL shaders activated. Instead a simple phong shader with one light source was used. This function is hardcoded into the Reyes pipeline so no linking is performed. The compile time is fast because the Reyes pipeline does not need to be recompiled when no shaders change. The performance of all the scenes is significantly faster. The speedup of the scenes with the more complicated shaders is higher than the ones with the simpler shaders as expected. However the performance of the scenes with the simpler shaders also improves drastically. This is mainly because the parameters of the shaders need to be loaded from texture memory and because of the light loops. In each light loop the parameters of the light shaders also need to be loaded from texture memory.

As it can be seen from the data, the placement of the shader code can have a large impact on the applications performance. E.g. the performance difference for the Killeroo model is approximately 45% between the worst and best runtime. However using the option to put the shader code into the Reyes pipeline compilation unit increases the compile time. If the shader code does not change over successive runs the no linking option is preferable. Otherwise the option to put the parameter textures, the shader textures and the shader code in one additional compilation unit is preferable.

It can also be seen that the usage of RSL shaders has a large impact on the performance. While in the case of the simple shaders in the Killero model the difference in performance is small, but still noticeable, the performance impact of more complicated shaders is much larger. E.g. for the Cube with the brick shader there is a factor of 3.6 in the render time.

The compile time is influenced by the complexity of the used shader, but the difference is much smaller than the runtime difference. The difference is less than 4 seconds between the slowest and fastest shader compilation. The compile time in table 7.2 only includes the compile time of the CUDA compiler. The compilation of the RSL code to CUDA code is not included. This however does not make much difference in the results as the longest compile time (brick shader) is only 0.11 seconds.

7.2.2. Usage of Different Subdivision Surface Patch Types

Used Split Procedures	All	Init All	Only Border
Cube Brick	33	39	200
Killeroo	56	119	OOM
Killeroo 2	124	264	OOM
Killeroo 2 Wood	241	402	OOM
Teapot	54	114	OOM

Table 7.3.: The performance for the usage of the different subdivision surface patch types. All: regular and multi-face patches are used initially and after a split. Init All: regular and multi-face patches are used initially. The patch type remains the same after a split. Only Border: Only border patches are used. OOM means that a queue ran out of memory.

As mentioned in section 6.3.2 our implementation distinguishes between three different subdivision surface patches. Regular patches that consist of four vertices with valence 4. Multi-Face patches that can handle vertices with different valences, and border patches. These can handle patches near borders of the mesh. Regular patches are the fastest as the subdivision is the least complicated and they can be converted to Bezier patches before dicing. This makes it possible to use larger dice dimensions and therefor less split operations are necessary.

In table 7.3 a performance comparison between different usages of these patch types can be seen. We use three different configurations:

1. **All:** Regular and multi-face patches are used initially and also after each split operation if possible.
2. **Init All:** Regular and multi-face patches are used initially, but after each split operation the patch type of the input patch is used.
3. **Only Border:** Only border patches are used, initially and after each split operation.

In all scenes the performance is best when the most suitable patch type is always selected. The performance cost for only using the most suitable patch type initially is largely dependent on the number of non-regular vertices in the model. The cube with only 8 non-regular vertices only loses about 15% performance whereas the Killeroo 2 model takes more than twice the

time to render. When only border patches are used initially and after a split, the performance drops significantly. The Cube model takes about 6 times as long to render as in the optimal case. We could not compare the other models as the queues filled up too fast, ran out of memory, and the application crashed.

In figure 7.4 the used dice procedures for the All and Init All configuration are shown for the Killeroo model. It can be seen, that in the All configuration the regular dice procedures are used almost exclusively. The other dice procedures are only used for few pixel around the non-regular vertices.

7.2.3. Dice Procedures

In section 6.2 we mentioned that we use three dice procedures for Bezier patches with different dice dimensions. These dice dimensions are also used for the dicing of regular subdivision surface patches as seen in section 6.3.2. The dice dimensions of these procedures are 15×15, 7×7, and 3×3. In table 7.4 we compare the performance of this configuration against using only one dice procedure with dice dimension 15×15.

The usage of multiple dice procedures increases the performance for most scenes. That is because these scenes are subdivision surfaces with a number of non-regular vertices. When subdividing around this vertices the split procedures emit smaller and smaller regular patches with each subdivision level. These patches can then be diced using less threads when smaller dice dimensions are available. Thus, the performance increases.

The increase of performance is the most dramatic for the Killeroo scene. For this scene the rendering performance almost doubles. That is because the Killeroo model contains the most non-regular vertices, including a number of border patches which are the most complicated and slowest to compute patches. In the Cube scene the performance gets worse when using multiple dice dimensions. This scene only has 8 non-regular vertices and therefor the smaller dice dimensions are hardly used. The performance increase likely comes from a better optimization of the CUDA compiler when less code needs to be compiled, or from a lower number of used registers.

For the Dragonhead scene the performance gets about 10% worse when using multiple dice dimensions. This scene only consists of Bezier patches which do not have non-regular vertices. Therefor, even when multiple dice procedures are available they are not used much. Again, the performance increase likely stems from better compiler optimizations.

	One Dice Procedures (ms)	Multiple Dice Procedures (ms)
Cube Brick	29	33
Killeroo	98	56
Dragonhead	19	21
Killeroo 2	182	124
Killeroo 2 Wood	331	242
Teapot	70	54

Table 7.4.: Performance comparison for the number of dice procedures. We compare the usage of one dice procedure with dice dimension 15×15 against the usage of three dice procedures with dice dimension 3×3, 7×7, and 15×15.

7.2.4. Supersampling

In table 7.5 we compare the performance of different supersampling factors. The performance difference between the supersampling factors is roughly equivalent to the difference between the supersampling factors. I.e. the performance difference between a supersampling factor of 4 and a supersampling factor of 1 is approximately 4. The performance difference between no supersampling and 4 times supersampling varies between 3.4 for the Dragonhead model and 4.1 for the Cube model. This variation can be explained by the fixed time overhead each frame and the fact that for a bigger supersampling factor the model needs to be subdivided into smaller patches before dicing.

7.2.5. Micropolygon Rasterization

In table 7.6 a statistic of the micropolygons of the different scenes is shown. The table contains the number of rasterized micropolygons while rendering

Supersampling	1 (ms)	4 (ms)	16 (ms)
Cube Brick	33	135	545
Killeroo	56	191	724
Dragonhead	21	71	277
Killeroo 2	124	446	1725
Killeroo 2 Wood	241	924	3637
Teapot	54	203	779

Table 7.5.: The performance for different supersampling factors.

the scenes and information about the size of the micropolygons. The four right columns show the percentage of the micropolygons in a scene with a specific bounding box size. The size of the bounding box is given as the number of pixels inside the bounding box.

It can be seen, that for most scenes the proportion of micropolygons with no pixels is about two thirds of all micropolygons. In the Teapot scene it is even higher with 75%. This means, that most micropolygons do not contribute to the output image. This fact has a large negative impact on performance as a large number of unnecessary split, dice and shading operations are performed.

The easiest way to increase performance would be to increase the size of the dice threshold. However this would also increase the number of micropolygons that are bigger than one pixel. Our implementation already produces up to 1 percent micropolygons with a 2 pixel sized bounding box. However, the size of a micropolygon should not be bigger than one pixel. Therefor an increase of the dice threshold is not possible without breaking this constraint.

	Number of Micropolygons	Pixels in BB (%)			
		0	1	2	4
Cube Brick	621,223	65.4	33.9	0.7	0.0
Killeroo	2,468,833	66.8	32.4	0.8	0.0
Dragonhead	769,507	65.0	34.2	0.9	0.0
Killeroo 2	4,799,538	64.6	34.6	0.8	0.0
Teapot	2,131,872	75.5	24.2	0.2	0.0

Table 7.6.: Statistics of the micropolygon sizes in the different scenes.

8. Conclusion and Future Work

8.1. Conclusion

We have shown an implementation of the Reyes rendering pipeline that is able to render simple scenes at interactive to real time frame rates on a GPU. The renderer supports Bezier patches and Catmull-Clark subdivision surfaces as input primitives. Scenes can be given as Renderman scenes. Our implementation supports materials and surface displacement through the Renderman Shading Language.

We have shown a method to render Catmull-Clark subdivision surfaces on the GPU. In this method, a patch is created for every face in a given subdivision mesh. These patches are then sent to the GPU and rendered according to the Reyes pipeline. If possible these patches are converted into Bezier patches before the dicing step of the Reyes pipeline to increase performance.

We have shown a compiler that translates the shaders into CUDA code that can be included in our rendering pipeline. With this shader compiler we are able to support complex materials such as a procedural wood or brick shaders. The inclusion of shaders into the pipeline has a moderate to high impact on the performance of the implementation, depending on the complexity of the shader. We noticed, that linking the shader code to a precompiled Reyes pipeline decreases the runtime performance dramatically compared to including the shader code in the same translation unit. The compile time however increases drastically when including the shader code in the Reyes pipeline translation unit.

8.2. Future Work

Our implementation currently uses a bound formula that is based on the axis aligned bounding box of a geometric primitive. We use a similar formula for both Bezier patches and Catmull-Clark subdivision faces. This approach produces a large number of micropolygons that are smaller than one sub-pixel. A better check formula would probably increase the performance of our implementation dramatically.

The algorithm to close the holes, which are produced during rendering, is not accurate and produces artifacts. An algorithm that ensures that no holes are produced between patches should be implemented.

We currently only support basic Catmull-Clark subdivision surfaces. Support for creases and semi-sharp creases could be added to this primitive type. Our implementation also does not support vertices with valence two. Support for these could also be added.

We currently subdivide each subdivision mesh once before sending the patches to the GPU. This is done to ensure that the mesh only consist of quad faces. This first subdivision step could be moved to the GPU.

Our implementation currently only supports Renderman scenes with one frame. Support for rendering multiple frames could be added. In Renderman scenes shadow maps are usually generated using multiple frames. This means, that with support for multiple frames in a scene we could also support shadow maps.

Appendix

Appendix A.

Renderman Shading Language Grammar

⟨*procedures*⟩ ::= (⟨*shaderdefinition*⟩
| ⟨*functiondefinition*⟩
| ⟨*preprocessorstatement*⟩)*

⟨*shaderdefinition*⟩ ::= ⟨*shadertype*⟩ ⟨*identifier*⟩ '(' [⟨*formals*⟩] ')' '{' ⟨*statements*⟩ '}'

⟨*shadertype*⟩ ::= 'light' | 'surface' | 'displacement'

⟨*type*⟩ ::= 'float' | 'string' | 'color' | 'point'
| 'vector' | 'normal' | 'matrix' | 'void'

⟨*functiondefinition*⟩ ::= [⟨*type*⟩] ⟨*identifier*⟩ '(' [⟨*formals*⟩] ')' '{' ⟨*statements*⟩ '}'

⟨*formals*⟩ ::= ⟨*formalvariabledef*⟩ (';' ⟨*formalvariabledef*⟩)* [';']

⟨*formalvariabledef*⟩ ::= ⟨*outputspec*⟩ ⟨*typespec*⟩ ⟨*defexpressions*⟩

⟨*variables*⟩ ::= ⟨*variabledefintions*⟩ (';' ⟨*variabledefintions*⟩)*

⟨*variabledefintions*⟩ ::= ⟨*externspec*⟩ ⟨*typespec*⟩ ⟨*defexpressions*⟩

⟨*typespec*⟩ ::= ['varying' | 'uniform'] ⟨*type*⟩

⟨*defexpressions*⟩ ::= ⟨*defexpression*⟩ (',' ⟨*defexpression*⟩)*

⟨*defexpression*⟩ ::= ⟨*identifier*⟩ [⟨*definit*⟩]

⟨*definit*⟩ ::= '=' ⟨*expression*⟩

⟨*outputspec*⟩ ::= ['output'];

⟨*externspec*⟩ ::= ['extern'];

⟨*statements*⟩ ::= statement+

⟨*statement*⟩ ::= ⟨*variabledefintions*⟩ ';'
| ⟨*compoundstatement*⟩
| ⟨*assignexpression*⟩ ';'
| ⟨*procedurecall*⟩ ';'
| ⟨*returnstatement*⟩
| ⟨*loopmodstmt*⟩ ';'
| ⟨*ifstatement*⟩
| ⟨*loopstatement*⟩
| ⟨*preprocessorstatement*⟩
| ⟨*functiondefinition*⟩

⟨*loopstatement*⟩ ::= ⟨*loopcontrol*⟩ ⟨*statement*⟩

⟨*ifstatement*⟩ ::= 'if' ⟨*relation*⟩ ⟨*statement*⟩ ['else' ⟨*statement*⟩]

⟨*returnstatement*⟩ ::= 'return' ⟨*expression*⟩ ';'

⟨*compoundstatement*⟩ ::= '{' ⟨*statements*⟩ '}'

⟨*loopcontrol*⟩ ::= ⟨*whileloopcontrol*⟩ | ⟨*forloopcontrol*⟩ | ⟨*lightloopcontrol*⟩

⟨*whileloopcontrol*⟩ ::= 'while' ⟨*relation*⟩

⟨*forloopcontrol*⟩ ::= 'for' '(' ⟨*expression*⟩ ';' ⟨*relation*⟩ ';' ⟨*expression*⟩ ')'

⟨*lightloopcontrol*⟩ ::= ⟨*lightlooptype*⟩ '(' [expressionlist] ')'

⟨*loopmodstmt*⟩ ::= ('break' | 'continue') [⟨*int*⟩]

⟨*expressionlist*⟩ ::= ⟨*expression*⟩ (',' ⟨*expression*⟩)* ⟨*expression*⟩ = ⟨*logopor*⟩

⟨*logopor*⟩ ::= ⟨*logopand*⟩ ['||' ⟨*logopor*⟩]

⟨*logopand*⟩ ::= ⟨*relopeq*⟩ ['&&' ⟨*logopand*⟩]

⟨*relopeq*⟩ ::= ⟨*relopueq*⟩ [('==' | '!=') ⟨*relopeq*⟩]

⟨*relopueq*⟩ ::= ⟨*binopadd*⟩ [('>' | '>=' | '<' | '<=') ⟨*relopueq*⟩]

⟨*binopadd*⟩ ::= ⟨*binopcross*⟩ [('+' | '-') ⟨*binopadd*⟩]

⟨*binopcross*⟩ ::= ⟨*binopfactor*⟩ ['^' ⟨*binopcross*⟩]

⟨*binopfactor*⟩ ::= ⟨*binopdiv*⟩ ['*' ⟨*binopfactor*⟩]

⟨*binopdiv*⟩ ::= ⟨*binopdot*⟩ ['/' ⟨*binopdiv*⟩]

⟨*binopdot*⟩ ::= ⟨*expressiontop*⟩ ['.' ⟨*binopdot*⟩]

⟨*expressiontop*⟩ ::= '(' ⟨*expression*⟩ ')' | ⟨*unaryexpression*⟩ | ⟨*primary*⟩

⟨*unaryexpression*⟩ ::= ('-' | '!') ⟨*expression*⟩

⟨*typecastexpression*⟩ ::= ⟨*type*⟩ [⟨*spacetype*⟩] ⟨*expression*⟩

⟨*primary*⟩ ::= ⟨*constant*⟩
| ⟨*texture*⟩
| ⟨*procedurecall*⟩
| ⟨*typecastexpression*⟩
| ⟨*assignexpression*⟩
| ⟨*triple*⟩
| ⟨*sixteentuple*⟩
| ⟨*arrayindexedvar*⟩

⟨*constant*⟩ ::= ⟨*float*⟩ | ⟨*string*⟩

⟨*arrayindexedvar*⟩ ::= ⟨*identifier*⟩ [⟨*arrayindex*⟩]

⟨*arrayindex*⟩ ::= '[' ⟨*expression*⟩ ']'

⟨*triple*⟩ ::= '(' ⟨*expression*⟩ ',' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'

⟨*sixteentuple*⟩ ::= '(' ⟨*expression*⟩ 15 * (',' ⟨*expression*⟩) ')'

⟨*spacetype*⟩ ::= ⟨*quotedstring*⟩

⟨*relation*⟩ ::= ⟨*expression*⟩

⟨*assignexpression*⟩ ::= ⟨*identifier*⟩ [⟨*arrayindex*⟩] ('=' | '+=' | '-=' | '*=' | '/=')
⟨*expression*⟩

⟨*procedurecall*⟩ ::= ⟨*identifier*⟩ '(' [⟨*procarguments*⟩] ')'

⟨*procarguments*⟩ ::= ⟨*expression*⟩ (',' ⟨*expression*⟩)*

⟨*texture*⟩ ::= 'texture' '(' ⟨*texturefilename*⟩ [⟨*channel*⟩] [⟨*textureargumentss*⟩] ')'

⟨*texturefilename*⟩ ::= ⟨*expression*⟩

⟨*channel*⟩ ::= '[' ⟨*expression*⟩ ']'

⟨*textureargumentss*⟩ ::= ',' ⟨*expression*⟩ (',' ⟨*expression*⟩)*

Bibliography

[BLZ00] Henning Biermann, Adi Levin, and Denis Zorin. "Piecewise smooth subdivision surfaces with normal control." In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 2000, pp. 113–120 (cit. on p. 13).

[CC78] Edwin Catmull and James Clark. "Recursively generated B-spline surfaces on arbitrary topological meshes." In: *Computer-aided design* 10.6 (1978), pp. 350–355 (cit. on p. 9).

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. "The Reyes Image Rendering Architecture." In: *ACM SIGGRAPH*. 1987, pp. 95–102 (cit. on pp. 1, 3, 17).

[Fat+09] Kayvon Fatahalian et al. "Data-parallel rasterization of micropolygons with defocus and motion blur." In: *High Performance Graphics 2009*. 2009, pp. 59–68 (cit. on p. 35).

[Gab] Eric Back Gabriel Nützi. *RIBMosaic Exporter for Blender*. URL: https://github.com/gabyx/RIBMosaic (visited on 04/25/2016) (cit. on p. 55).

[Guz] Joel de Guzman. *Boost Spirit Parser Library*. URL: http://boost-spirit.com/home/ (visited on 03/22/2016) (cit. on pp. 33, 48).

[HKD93] Mark Halstead, Michael Kass, and Tony DeRose. "Efficient, fair interpolation using Catmull-Clark surfaces." In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM. 1993, pp. 35–44 (cit. on p. 12).

[LS08] Charles Loop and Scott Schaefer. "Approximating Catmull-Clark subdivision surfaces with bicubic patches." In: *ACM Transactions on Graphics (TOG)* 27.1 (2008), p. 8 (cit. on pp. 5, 15).

[Nie+12] Matthias Nießner et al. "Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces." In: *ACM Transactions on Graphics (TOG)* 31.1 (2012), pp. 6–6 (cit. on p. 4).

[PEO09] Anjul Patney, Mohamed S Ebeida, and John D Owens. "Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces." In: *Proceedings of the conference on high performance graphics 2009*. ACM. 2009, pp. 99–108 (cit. on p. 3).

[Pixa] Pixar. *Pixar's Renderman*. URL: renderman.pixar.com/view/renderman (visited on 04/25/2016) (cit. on pp. 1, 54).

[Pixb] Pixar. *The Renderman Manual*. URL: https://renderman.pixar.com/resources/current/RenderMan/home.html (visited on 03/22/2016) (cit. on p. 21).

[Pix05] Pixar. *The Renderman Interface Specification*. 2005. URL: https://renderman.pixar.com/view/rispec/ (visited on 03/22/2016) (cit. on pp. 1, 21, 30).

[PO08] Anjul Patney and John D. Owens. "Real-time Reyes-style adaptive surface subdivision." In: *ACM Trans. Graph.* 27.5 (2008), pp. 143–143 (cit. on pp. 3, 4).

[She] C.K. Shene. *Lecture notes for Introduction to Computing with Geometry at Michigan Technological University*. URL: http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-der.html (visited on 04/25/2016) (cit. on p. 9).

[SS15] Martin Sattlecker and Markus Steinberger. "Reyes rendering on the GPU." In: *Proceedings of the 31st Spring Conference on Computer Graphics*. ACM. 2015, pp. 31–38 (cit. on p. 5).

[Ste+14] Markus Steinberger et al. "Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU." In: *ACM Trans. Graph.* 33.6-6 (Nov. 2014), 228:1–228:11. ISSN: 0730-0301. DOI: 10.1145/2661229.2661250. URL: http://doi.acm.org/10.1145/2661229.2661250 (cit. on pp. 5, 33, 34).

[TPO10] Stanley Tzeng, Anjul Patney, and John D Owens. "Task management for irregular-parallel workloads on the GPU." In: *High Performance Graphics*. 2010, pp. 29–37 (cit. on p. 4).

[Zho+09] Kun Zhou et al. “RenderAnts: Interactive Reyes Rendering on GPUs.” In: *ACM SIGGRAPH Asia*. 2009, 155:1–155:11 (cit. on p. 4).