FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Feature-Adaptive Catmull-Clark Subdivision on the GPU

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Christian Kößler
Matrikelnummer 0928004

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass.(FWF) Dr.techn. Dipl.-Mediensys.wiss. Przemyslaw Musialski

Wien, 07.10.2013

(Unterschrift Christian Kößler) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Feature-Adaptive Catmull-Clark Subdivision on the GPU

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Christian Kößler
Registration Number 0928004

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass.(FWF) Dr.techn. Dipl.-Mediensys.wiss. Przemyslaw Musialski

Vienna, 07.10.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Christian Kößler
Johannagasse 21, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Christian Kößler)

Abstract

Catmull-Clark subdivision is a powerful standard modeling technique and has already been used extensively in CGI for motion pictures or computer games. An artist creates a coarse polygon mesh, that is computationally converted into a high-quality smooth surface. Due to the recursive nature of the subdivision algorithm and the large number of polygons, that are generated during the mesh-refinement, it is not well suited for realtime-environments. There exist several approaches to generate a Catmull-Clark subdivision surface which use current GPU technologies to overcome these issues.

In this thesis, we use Compute Shaders for subdivision and a Cubic Bezier Patch which takes advantage of the tessellation pipeline to get an optimized algorithm without the loss of visual quality. With the presented method, the support of (Semi)Sharp Creases, which are an important feature to achieve a more realistic look, is also given. The practical part of this thesis is integrated into the Helix 3D Toolkit SharpDX Framework.

Kurzfassung

Catmull-Clark subdivision ist eine leistungsfähige Standard-Modellierungstechnik und wurde bereits ausgiebig für CGI in Filmen oder Computerspielen verwendet. Ein 3D-Modellierer erstellt ein grobes Modell, das automatisch in eine hochwertige glatte Oberfläche umgewandelt wird. Durch die rekursive Art des Algorithmus und der großen Anzahl an generierten Polygonen, die während der Erstellung erzeugt werden, ist diese Verfeinerungstechnik für Echtzeitumgebungen jedoch nicht optimal. Es gibt mehrere Ansätze, die aktuelle GPU-Technologien verwenden, um Catmull-Clark subdivision für den Echtzeiteinsatz zu optimieren.

In dieser Arbeit verwenden wir Compute Shaders zur regulären Unterteilung, und generieren einen Cubic Bezier-Patch, der die Fähigkeiten der modernen Tessellation-Pipeline ausnutzt, um einen optimierten Algorithmus ohne visuellen Qualitätsverlust zu erhalten. Mit dem vorgestellten Verfahren werden außerdem (Semi)Sharp-Creases unterstützt, die einen realistischeren Look erzeugen. Der praktische Teil dieser Arbeit wird in das 3D Toolkit SharpDX Framework integriert.

Contents

CHAPTER 1

Introduction

1.1 General Information

The creation and representation of models is an essential part in the field of computer graphics, since rendering and animation are directly affected by the decision, how a model is represented. Depending on the demands of the tasks that have to be accomplished, there exist several methods how an object can be represented. There are raw representations, where the individual elements are not connected to one another. An example is the point cloud, which is a set of points, or a triangle soup, which is a set of triangles, that are not connected and have no adjacency informations. Another group are solid representations, like voxels (volumetric elements), which represent values on regular grid in three dimensional space, or a csg (constructive solid geometry), where complex surfaces can be created out of simple objects, that are combined with boolean operations. A widely used option to represent an object is a polygonal mesh, which consists of vertices, edges and faces. A vertex is a position in 3d space with additional attributes such as color or normal vector. An edge connects two vertices and a face is a closed set of edges. A face usually consists of three (triangle face), or four vertices (quad face), but can also contain more vertices.
A common way for artists to generate high-quality 3d models, is the creation of a coarse polygon mesh with a low vertex count, which is also called a hull. The hull is smoothed automatically and the resulting polygon mesh with a high polygon count is used for rendering. If the model has to be changed, it is only necessary to change the vertices of the hull, and the resulting model can be recreated without further manual work. This opens also the possibillity for animation, because only the vertices of the hull have to be animated. The automatic refinement of a mesh can be accomplished with subdivision surfaces.
A subdivision algorithm takes the coarse mesh, and refines it to a mesh, which is smoother than the original one. This refinement step is repeated recursively to produce a sequence of finer shapes that finally lead to a smooth surface. This resulting surface is called a limit surface. The algorithm that decides how the mesh is refined, is specified by subdivision rules. There exist different subdivision schemes, each of them defining their own set of rules. There exist

approximating schemes like Catmull-Clark subdivision [Catmull and Clark, 1978], Doo–Sabin scheme [Doo, 1978] or the algorithm of Loop [Loop, 1987] . Those schemes generate vertices that only approximate the vertices of the base mesh, whereas the interpolating schemes touch the initial vertices of the base mesh. Examples for interpolating schemes are Butterfly [Dyn et al., 1990] or Kobbelt [Kobbelt, 2000].
The Catmull-Clark scheme operates on quad meshes. The initial mesh does not necessarily consist of quad faces only. Triangles and other surfaces of arbitrary topological type are subdivided to quads, so after the first iteration, the mesh consists of quads only. The Catmull-Clark scheme specifies rules for Face-, Edge- and Vertexpoints:

- each face point is the average of its face points
- each edge point is the average of the endpoints of the edge and the neighbouring face points
- each vertex point is a weighted average of the old vertex point, the old adjacent edgepoints and the new adjacent facepoints

A detailed view of the rules with the correct weights is given in Chapter 3. Figure 1.1 shows the Catmull-Clark scheme applied to a cube.

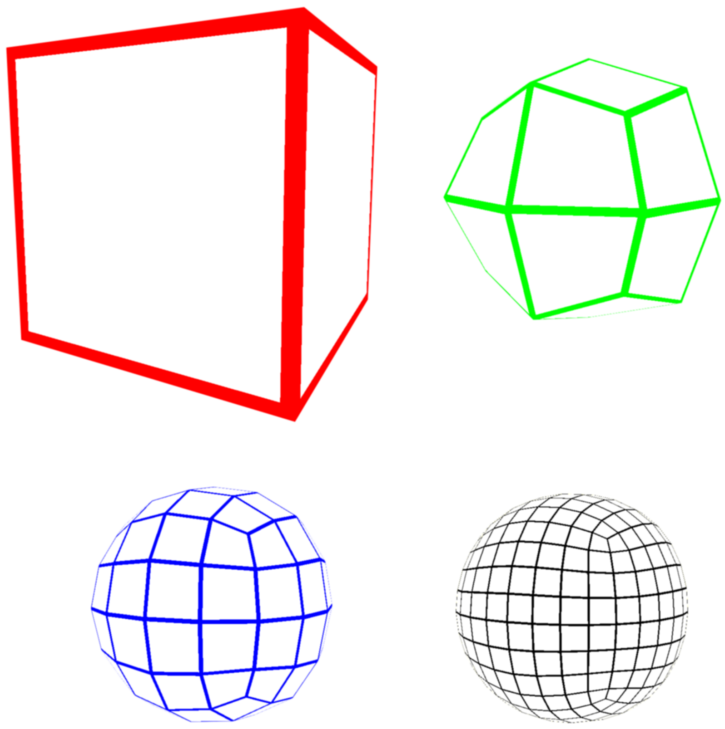


Figure 1.1: The coarse base mesh (red) and the first 3 Catmull-Clark subdivision iterations of a cube.

1.2 Problem Statement

Regular Subdivision, as it is defined by Catmull and Clark [Catmull and Clark, 1978], implies that every face is subdivided at every iteration step. The number of faces increases by the factor of 4 for every new subdivision level. Only with few iterations, the resulting globally refined mesh consists of a much larger amount of polygons than the base mesh.
For a realtime-application that uses the GPU for rendering, there exist some significant drawbacks. One point is, that modern GPUs work best, if as many as possible tasks can be parallelized. The iterative structure of the algorithm is contrary to that paradigm, because a new subdivision step can only be started, if the last step has completed. Second, there is no possibility to utilize effective level-of-detail algorithms to reduce the huge amount of polygons which must be processed by the graphics pipeline. An often wanted feature of artists is also the possibility to create semi-sharp creases. That gives a designer more options in the model-creation-process, and leads to more realistic objects, but (semi)-sharp creases are originally not part of the Catmull-Clark scheme. Although it is possible to model semi-sharp creases with traditional methods (e.g. bevel edges), that creates additional geometry in the base mesh, which makes further editing of the mesh more complicated and ruins the idea of a simple base mesh.

1.3 Goals

The goal of this work was to use and combine current GPU techniques like Tessellation and Compute Shaders to achieve a significantly improved Catmull-Clark subdivision, both in performance increase and in memory reduction compared to the original scheme. In addition, the goal was to handle special modelling features like infinitely sharp and semi-sharp creases. It should be possible to tag edges of the base mesh with a sharpness value, that influences the subdivision algorithm. A seamless transition between different subdivison levels, not only for vertex positions, but for attributes like normals and texture coordinates, was also a necessary criteria, since features like normal- and displacement mapping were also a target that we wanted to achieve.
There exist several approaches to solve the problems stated above, they are discussed in Chapter 2. In this work, one of those methods was implemented and Chapter 4 dicusses the issues and problems.

CHAPTER 2

Related Work

2.1 Early Approaches

Bolz and Schroeder [Bolz and Schröder, 2004] implemented a method for surfaces built through linear combination of basis functions. They precomputed tessellations and used these to evaluate the surface at runtime entirely on the GPU. Musialski [Musialski et al., 2007] presented a method to compute multiresolution displaced subdivision surfaces on the GPU that performs in real-time. They used the recursive nature of subdivision surfaces to add geometric detail on different resolutions. A general overview of efficient substitutes for subdivision surfaces on the GPU was given by Tianyun Ni and Ignacio Castano [Ni et al., 2009]

2.2 Approximation with Bicubic Patches

Loop and Schaefer [Loop and Schaefer, 2008] use Bicubic Patches to approximate the Catmull-Clark surface. Each quadrilateral polygon in a Catmull-Clark control mesh corresponds to a single bicubic patch except for quadrilaterals that contain one or more extraordinary vertices (a vertex not touched by exactly four quadrilaterals). They use for their approximation a collection of bicubic patches (one for each face of a quad-mesh). These patches are smooth everywhere except along edges leading to an extraordinary vertex where they are only C^0 , therefore shading discontinuities may result. To overcome this issue, they additionally create independent tangent patches that produce a continuous normal field to get the appearance of a smooth surface. No restrictions on number or valence of extraordinary vertices per patch are given. Figure 2.1 shows the control point labeling of geometry patches and control vectors of the tangent patches.

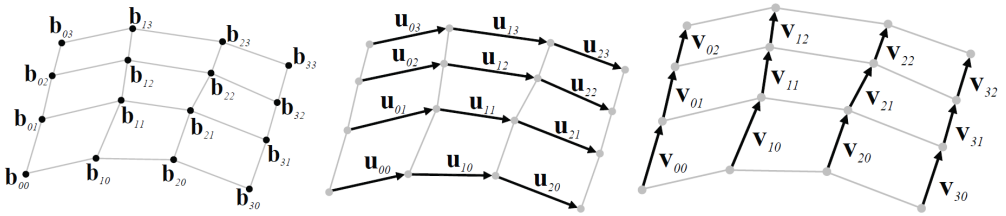


Figure 2.1: control point labeling of geometry patches and control vectors of the tangent patches (from [Loop and Schaefer, 2008]).

Adjacent-neighbour-information is used to generate the 4 corner, 8 edge and 4 interior points of the geometry patch. Three sets of fixed-weight masks (see [Loop and Schaefer, 2008]) are used. The weights imply, that each edge point lies at the midpoint of two interior points belonging to adjacent faces and each corner point lies at the centroid of the 4 interior points that surround that vertex. The 4 corner controlpoints interpolate the limit position of the Catmull-Clark surface. The tangent patches have 12 control vectors and generate a continuous normal field. The results of this method are shown in Figure 2.2

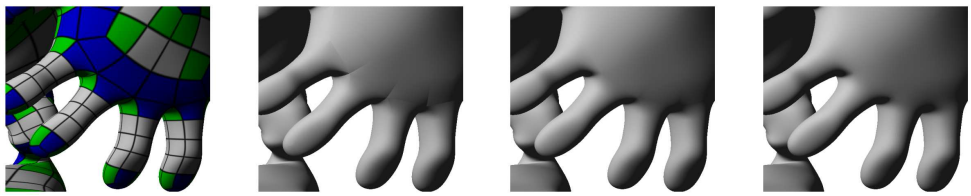


Figure 2.2: From left to right: Catmull-Clark patch structure, Geometry patch approximation, Geometry/Tangent patch approximation and Catmull-Clark limit mesh (from [Loop and Schaefer, 2008]).

2.3 Real-Time Creased Approximate Subdivision Surfaces

The work of Kovacs [Kovacs et al., 2009] is a direct extension of the work of Loop and Schaefer [Loop and Schaefer, 2008], with the addition of surfaces with sharp creases and corners. Especially the issue is addressed, that vertices of the control mesh, that are tagged as corners need to be interpolated. Shading artifacts that result from using incorrect tangents at corner vertices, are shown and a scheme for tangents that leads to better visual quality is presented. Figure 2.3 shows the tangent problem for corners.

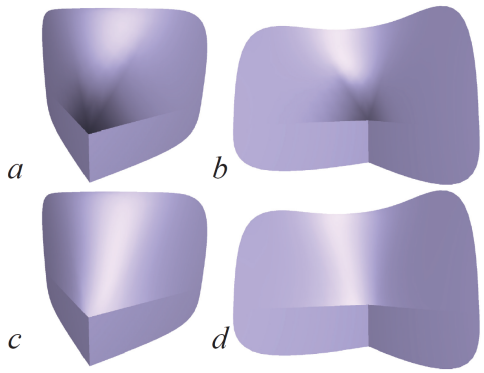


Figure 2.3: Comparison of different tangent definitions for a corner: a,b: Edge tangents are computed as linear combinations of two crease tangents (modified Catmull-Clark). c,d: The scheme of Kovacs [Kovacs et al., 2009] used for tangent control vectors for the same control meshes.

The results of this method are shown in Figure 2.4

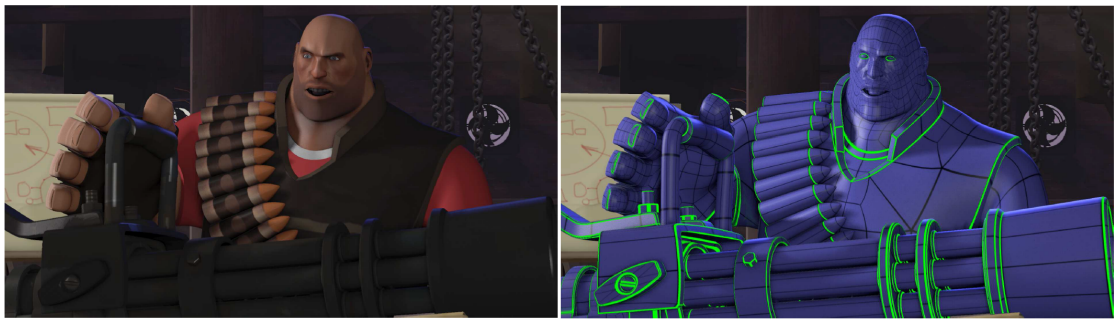


Figure 2.4: Character from the game Team Fortress 2 modeled as a Catmull-Clark subdivision surface and rendered with the technique of [Kovacs et al., 2009]. The character and his weapon contain sharp features which require crease support to render correctly. In the second image, the black lines indicate patch edges with tagged crease edges highlighted in green.

2.4 Direct Evaluation

The method of Stam [Stam, 1998] directly evaluates subdivision surfaces at arbitrary parameter values. It requires a projection of control points into the eigen space, and for the evaluation 2n+8 eigenbasis functions must be evaluated (n=valence). Furthermore, the method requires that extraordinary patches contain only one extraordinary vertex. If there are patches with more than one extraordinary vertex, the mesh must be subdivided once beforehand.

2.5 Gregory Patches

Gregory patches are a modified form of traditional tensor product and triangular polynomial patches. This modification was introduced by Gregory [Gregory, 1974] to overcome the difficult problem of ensuring that a collection of patches, sharing a vertex and pairwise sharing edges, meet with tangent plane continuity [Loop et al., 2009] This method approximates the subdivision surface in the regular case exactly, replacing irregular patches (patches with one or more extraordinary vertex) with a single rational patch that joins with G^1 continuity to the surrounding patches. The technique is not limited to quad patches, but is general enough to work with mixed quad/triangle surfaces. It requires 20 control points (quad patches), respectively 15 control points (triangle patches) for evaluation, so it is a reduction of the number of controlpoints compared to the Loop and Schafer ACC patches [Loop and Schaefer, 2008]. Figure 2.5 shows the control point scheme of a gregory patch.

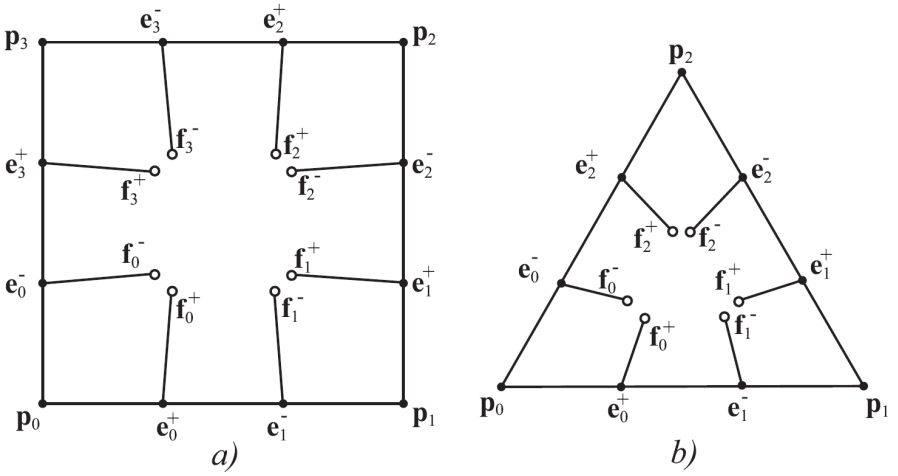


Figure 2.5: The control point label scheme for a) a tensor product, and b) a triangular Gregory patch (from [Loop et al., 2009]).

Problems can occur, since Gregory patches possess singularities (zero denominator) at patch corners, and the surface itself is rational with complicated tangent functions. Results can be seen at Figure 2.6.

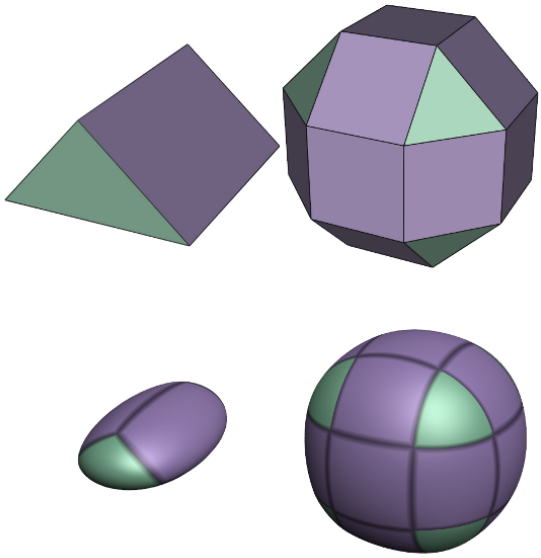


Figure 2.6: From top to bottom: the input mesh, the patch structure (irregular quad patch in purple, triangular patch in green) from [Loop et al., 2009].

2.6 Feature Adaptive Subdivision

In the work of Niessner [Niessner et al., 2012] the input mesh is analyzed and the following "features" are identified:

- regions around extraordinary vertices (vertices having other than four neighbors)
- regions around semi- or infinitely sharp creases
- regions affected by hierarchical detail

The mesh is subdivided only in the vicinity of these features. At each stage of local subdivision, new bicubic patches are generated that are directly rendered using hardware tessellation. Since subdivision happens only locally, the time and memory requirements are significantly less than the naive approach of globally subdividing the entire base mesh each step [Niessner et al., 2012]. Figure 2.7 shows the adaptive subdivision scheme near feature regions. The method uses a scheme, which DeRose [DeRose et al., 1998] has developed to support semi-sharp creases. Results of this method with different crease values for edges can be seen at Figure 2.8.

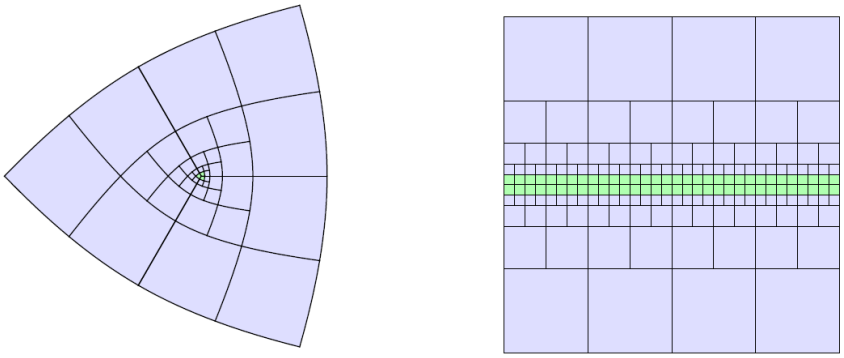
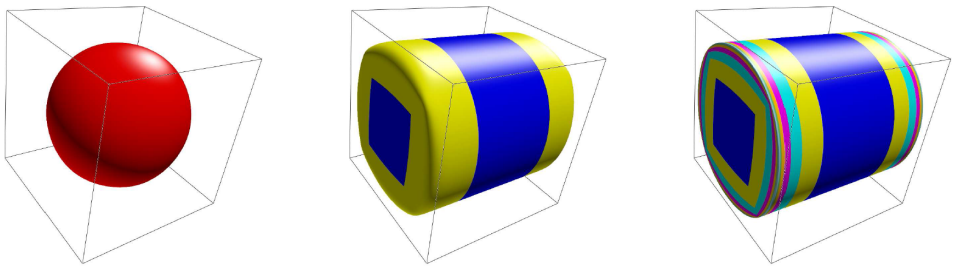


Figure 2.7: The arrangement of bicubic patches (blue) around an extraordinary vertex (left), and near an infinitely sharp crease right). Patches next to the respective feature (green) are irregular (From [Niessner et al., 2012]).



(a) (b) (c)

Figure 2.8: All images are derived from a cube used as the base mesh. (a) no edges tagged; (b) front and backface edges have a sharpness of 3; (c) front and backface edges have a fractional sharpness of 7.8. Each color represents a distinct level of subdivision (from [Niessner et al., 2012]).

CHAPTER 3

Method

As a basis for our work we use the adaptive subdivision scheme from Niessner [Niessner et al., 2012], which uses the subdivision rules and the extended rules for semi-sharp creases as defined in the work of DeRose [DeRose et al., 1998]. In the work of Niessner there is no accurate description about the construction of their bezier-patch, so we use the definition for the geometry-bezier-patch from the work of Loop and Schaefer [Loop and Schaefer, 2008], as well as their masks for controlpoint creation. Our advantage to the method of Niessner [Niessner et al., 2012] is an alternative way to solve the issues about adjacent patches with different subdivision levels. Our method avoids the split of those patches into triangles and thereby the necessity for additional bezier-patch definitions (details at Chapter 3.4).

3.1 Subdivision Rules

The regular subdivision rules, as they are specified at the work from DeRose [DeRose et al., 1998]:

- Face Rule: f^{i+1} is the centroid of the vertices surrounding the face.
- Edge Rule: $e_j^{i+1} = \frac{1}{4}(v^i + e_j^i + f_{j-1}^{i+1} + f_j^{i+1})$
- Vertex Rule: $v^{i+1} = \frac{n-2}{n}v^i + \frac{1}{n^2}\sum e_j^i + \frac{1}{n^2}\sum f_j^{i+1}$

Figure 3.1 shows the labeling scheme of the points.

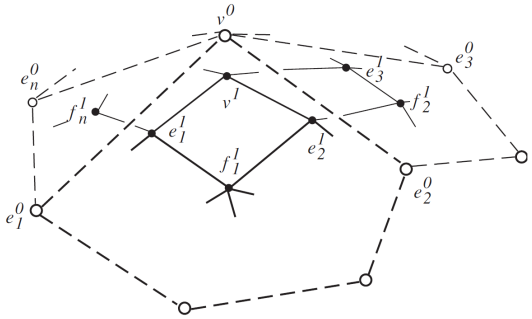


Figure 3.1: Labeling of vertices of the base mesh around the vertex v^0 of valence n (from [DeRose et al., 1998]).

3.2 Semi-Sharp Crease Rules

To add the possibility of creases, the subdivision rules are supplemented with the sharpness-rules of DeRose [DeRose et al., 1998]: Subdividing a sharp edge creates two child edges, each of which are tagged with the sharpness value of the parent minus one. A vertex v_j containing exactly two crease edges e_j and e_k is considered to be a crease vertex. Crease Rules:

- Edge Rule: $e_j^{i+1} = \frac{1}{2}(v^i + e_j^i)$
- Vertex Rule: $v_j^{i+1} = \frac{1}{8}(e_j^i + 6v^i + e_k^i)$

If a vertex is adjacent to three or more sharp edges or located on a corner, its successor is derived by the corner rule:

- Corner Rule: $v^{i+1} = v^i$

The sharpness rules are combined with the regular Catmull-Clark subdivision rules and adapted by Niessner [Niessner et al., 2012] to deal with fractional smoothness (e.s is the sharpness of the edge):

- e with $e.s = 0 \rightarrow$ smooth rule
- e with $e.s \geq 1 \rightarrow$ crease rule
- e with $0 \leq e.s \leq 1 \rightarrow (1 - e.s) * e_{smooth} + e.s * e_{crease}$

Vertex sharpness v.s is the average of all incident edges with e.s > 0, k is the count of those edges.

- v with $k < 2 \rightarrow$ smooth rule
- v with $k > 2$ and $v.s \geq 1 \rightarrow$ corner rule

- v with $k > 2$ and $0 \leq v.s \leq 1 \rightarrow (1 - v.s) * v_{smooth} + v.s * v_{corner}$
- v with $k = 2$ and $v.s \geq 1 \rightarrow$ crease rule
- v with $k = 2$ and $0 \leq v.s \leq 1 \rightarrow (1 - v.s) * v_{smooth} + v.s * v_{crease}$

3.3 Adaptive Subdivision

At the preprocessing stage we analyze the mesh on a per-face basis. Depending on the valence and the sharpness of the vertices (derived by the sharpness of the edges) of the face, it is tagged either as regular (all valences equal 4, and no vertices with sharpness > 1), or as irregular. Since our method is not limited to quad-only meshes, but also supports triangles, all triangle faces are also tagged as irregular. There exists one exception to the sharpness rule, which becomes effective, when the last subdivision level is already reached, and there is still a sharp edge present. In this exceptional case, if a face was tagged as irregular only because of the sharpness criteria, that face is still tagged as regular, and the bezier-patch-construction algorithm has to handle it with a special treatment, a more detailed explanation is given at Chapter 3.7.As we currently do not support hierarchical edits, like in the work of Niessner [Niessner et al., 2012], these constraints are sufficient.

The irregular faces get split according to the subdivision scheme. Figure 3.2 shows the split for each supported face type.

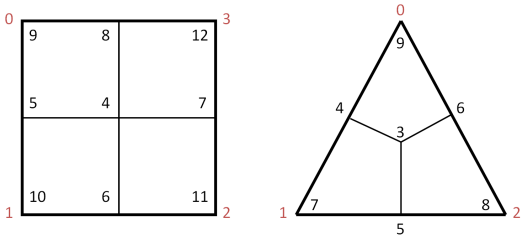


Figure 3.2: Old and new indices after the split of an irregular face (old indices are tagged red).

On the resulting reindexed mesh the analyzing, tagging and splitting is repeated, until the maximum subdivision level is reached. Figure 3.3 shows a resulting mesh near extraordinary vertices.

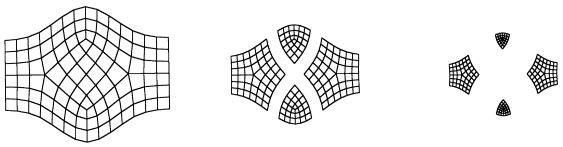


Figure 3.3: Adaptive subdivision scheme applied on a grid with four extraordinary vertices. Subdivision is only performed in areas next to extraordinary vertices (from [Niessner et al., 2012]).

3.4 Different Subdivision Levels

By the use of adaptive subdivision, adjacent patches correspond not always to the same subdivision level, their levels can also differ by one. A problem with watertightness of the resulting mesh can arise from that fact. On patch borders, on the one side exists only one edge, and at the other side exist two edges with a vertex inbetween (see Figure 3.5 (Left)). Niessner [Niessner et al., 2012] solves this problem with Transition Patches, that connect the different subdivision levels as illustrated in Figure 3.4.

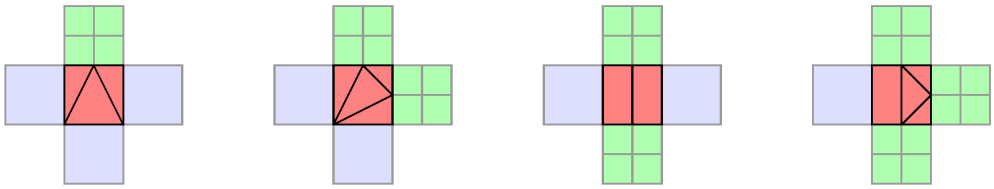


Figure 3.4: There are five possible constellations for TP. While TP are colored red, the current level of subdivision is colored blue and the next level is green. The domain split of a TP into several subpatches allows full tessellation control on all edges, since shared edges always have the same length (from [Niessner et al., 2012]).

This approach reintroduces triangles into the topology, which we have specially sorted out with the quadrangulation at the first subdivision step, because our bezier-patch-algorithm operates on quads. The core idea of our solution is to take advantage of the fact, that the use of tessellation offers the possibility to change the edgefactors of the affected edges. If such an edge is identified, the tessellation factor at the appropriate edge of the patch with the lower subdivision level is set twice as high, compared to the other patch. The missing vertex at the patch with the lower subdivision level is automatically created by the tessellator unit of the GPU. At Figure 3.5 our solution is outlined:

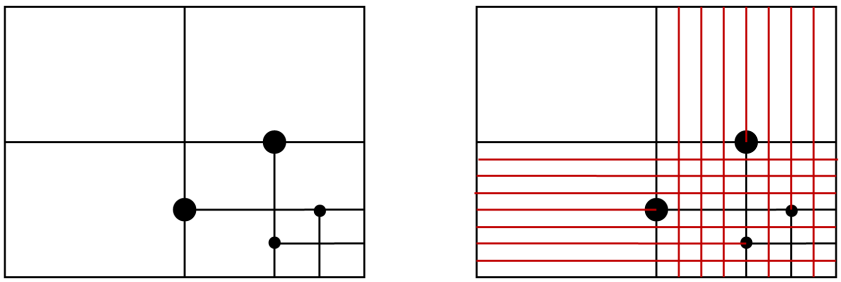


Figure 3.5: The left side shows the problems with different edge counts, at the right side this is compensated by the tessellation factors (tagged red).

The calculations of the vertices that are shared between two neighbouring patches is the same for all of them, whether it exists already in the geometry, or is created by the tessellation pipeline.

With this approach, no zero-sized patches, or any other kind of stitching faces are needed. An explanation on the implementation of the solution is given at Chapter 4.6.

3.5 The Cubic Bezier-Patch

The chosen method combines the regular subdivision algorithm with the use of bicubic bezier-patches, wherever it is able to generate the Catmull-Clark limit surface. Those patches were identified as Regular Patches by the mesh-analyzing algorithm. As a basis for our patch-construction we use the geometric patch definition from Loop and Schaefer [Loop and Schaefer, 2008].

The bicubic patch needs 16 control points for a quad face (Figure 3.6).

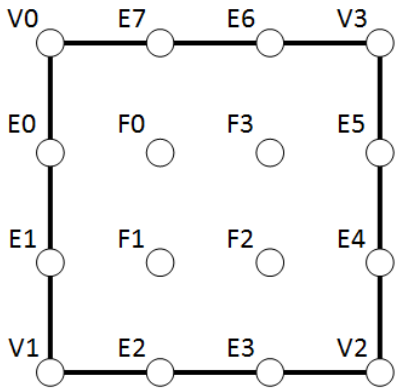


Figure 3.6: The 16 control points of the bicubic quad-patch.

As weighting factors for the controlpoints we use the masks, as defined for the geometric patch from Loop and Schaefer [Loop and Schaefer, 2008]. Three different mask types exist to get the 4 face points, the 4 vertex points, and 8 edge points of the control point mesh. The masks and the weighting factors are shown at Figure 3.7. The weighting factors have to be normalized.

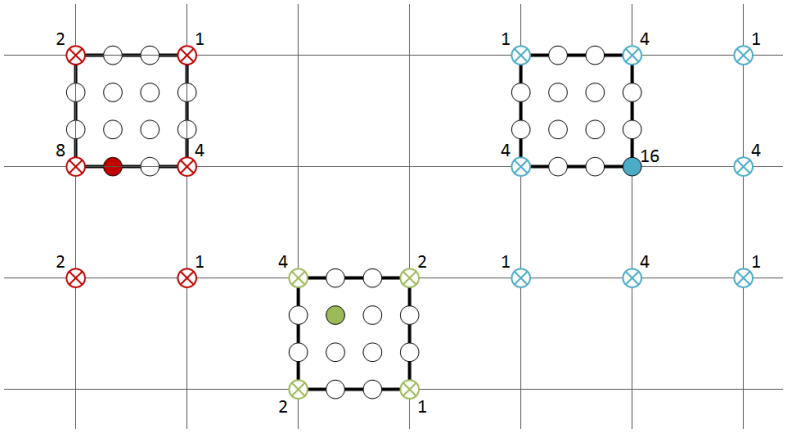


Figure 3.7: Adjacent vertices used for edge points (red), face points (green) and vertex points (blue).

3.6 Virtual Faces

As described in the last Section, to create a correct cubic bezier-patch, proper adjacent information is needed. With the stitching of different subdivision levels, the problem arises, that some needed vertices of the adjacent information are not available (Figure 3.8, left side)

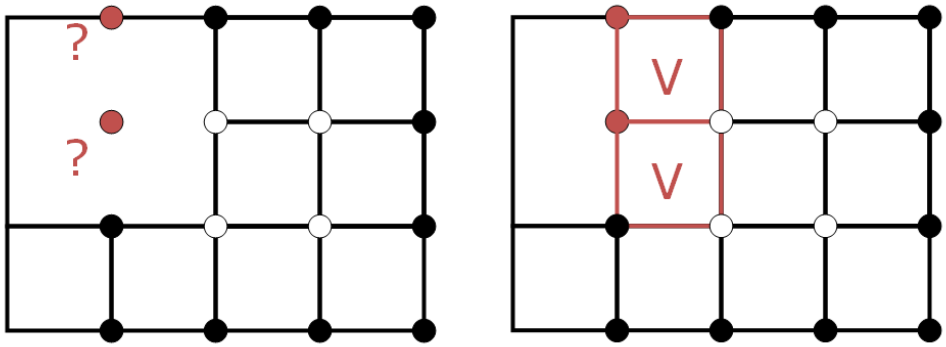


Figure 3.8: Left: Issue of missing adjacent information, right: solution with Virtual Faces.

To overcome this problem, we introduce a new type of patch, The Virtual Face. This is a supporting patch, which is only needed as a source of adjacent information for the regular patches. It is not rendered, and therefor does not strain the tessellation pipeline. It is also not necessary to subdivide the whole patch, as in regular subdivision, but only at those edges, where a patch with a higher subdivision level is adjacent, as shown in Figure 3.8, right side.

Figure 3.9 shows the virtual faces in our application.

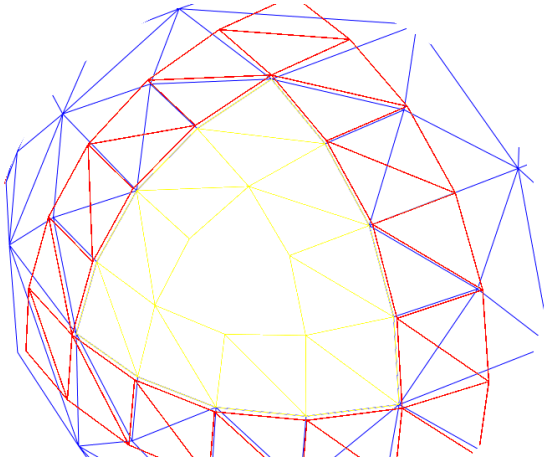


Figure 3.9: Blue: The patches with level-1, Yellow: The patches with the current level, Red: The supporting virtual faces.

3.7 Special Remark about Semi-Sharp Creases

The sharpness rule from Chapter 3.3 says, a patch is irregular if the face has at least one vertex that is part of a an edge with a sharpness value > 1. A more simple approach would be, just to check the four edges of the patch, and if at least one of them is sharp, to mark the patch as irregular. Figure 3.10 (Left) shows, if that approach is chosen, issues can arise, regarding the rule, which says, subdivision levels at adjacent patches may only differ by 1. Violating this constraint would introduce new problems, for instance the issue with the calculation of correct tessellation factors at shared patch borders.

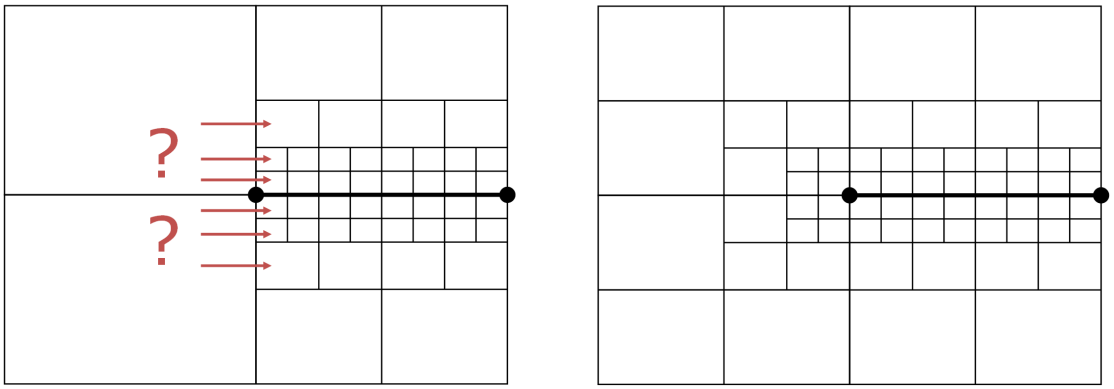


Figure 3.10: Left: The problem of adjacent patches, that have a difference of subdivlevels > 1, (the sharp edge is the thicker line) , Right: the solution, an appropriate subdivision around the vertex.

In this scenario, the solution is that sharpness is not determined per edge. Instead, every vertex of the face is checked, whether it is part of a sharp edge or not. If at least one such vertex is found, the patch is declared irregular. This information is not only used for the decision, if a face should be further subdivided or not, it is also stored at the meta-information section of the patch-adjacent table (see Chapter 4.4), because the bezier patch has also to deal with sharpness. There is some computational overhead, as it needs only to be done only once at startup (with the condition, that sharpness does not change at runtime), the overhead is negligible. The resulting subdivison scheme is shown in Figure 3.10 (Right) and the implementation in our project can be seen in Figure 3.11.

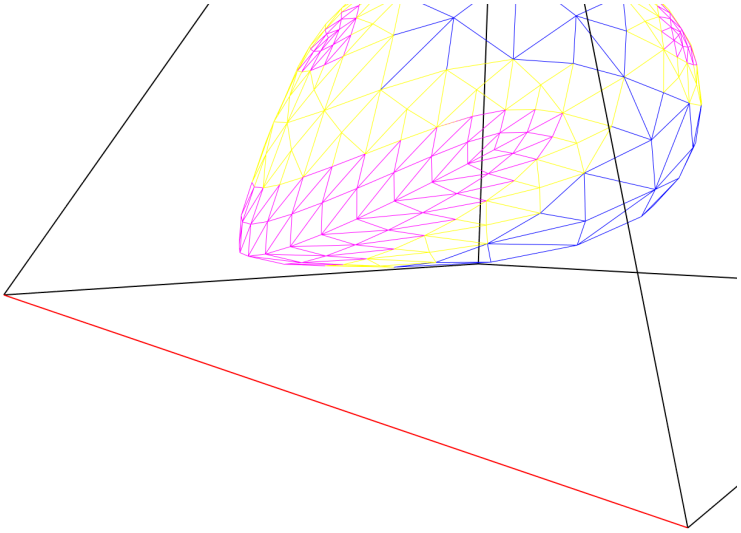


Figure 3.11: The solution implemented in our project (the sharp edge is drawn red at the hull of the object).

How the patch is constructed using the tessellation pipeline and how the necessary adjacent information is provided to the GPU is explained at Chapter 4.4.

CHAPTER 4

Implementation

4.1 Libraries, Frameworks and Technology

SharpDX

Sharp DX [SharpDX, 2013] is an open source C#/DirectX API, which allows the user to write DirectX applications in managed .NET code without any significant reduced performance, compared to natively built DirectX applications.

Helix 3D Toolkit SharpDX

Helix 3D Toolkit [Helix 3D Toolkit, 2013] is a collection of custom controls and helper classes for WPF. Helix 3D Toolkit SharpDX is a version of the toolkit that is based on SharpDX. The practical part of this thesis is integrated into the Helix 3D Toolkit SharpDX Framework.

Used technologies

Following technologies are used:

- Microsoft Effects Framework [Microsoft, 2013b]
- HLSL Shaders with ShaderModel 5.0 [Microsoft, 2013a]
- DirectX Compute Shaders (DirectCompute) [Microsoft, 2013c]
- DirectX Tessellation Pipeline [Microsoft, 2013d]
- Model View ViewModel [Microsoft, 2013f] (for the Demo Application)

4.2 Data-Parallel Subdivision

As in Chapter 3 described, Catmull-Clark subdivision is an iterative process, where in every iteration step the mesh is further subdivided. Because of the nature of the subdivision rules, it is necessary that neighbourhood information is available. For an algorithm which runs on the CPU, this is no problem, but on the GPU, this is not inherently given. So, since we set the constraint that the mesh topology is not changing during runtime, we have to prepare tables with index information only once at the startup of the program. For every subdivision level, one such table is built based on the table of the last subdivision level. The mesh shown in Figure 4.1 is generated with the table shown at Figure 4.2 according to Niessner [Niessner et al., 2012].

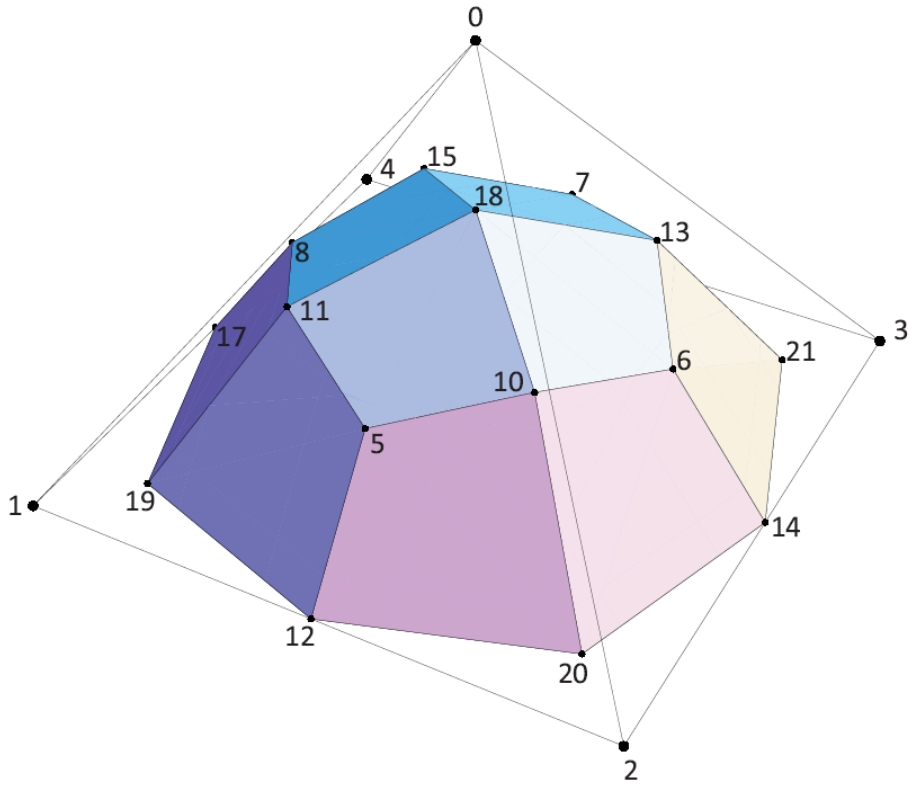


Figure 4.1: Pyramid mesh after 1 subdivision level, edges at the base are tagged as crease (from [Niessner et al., 2012]).

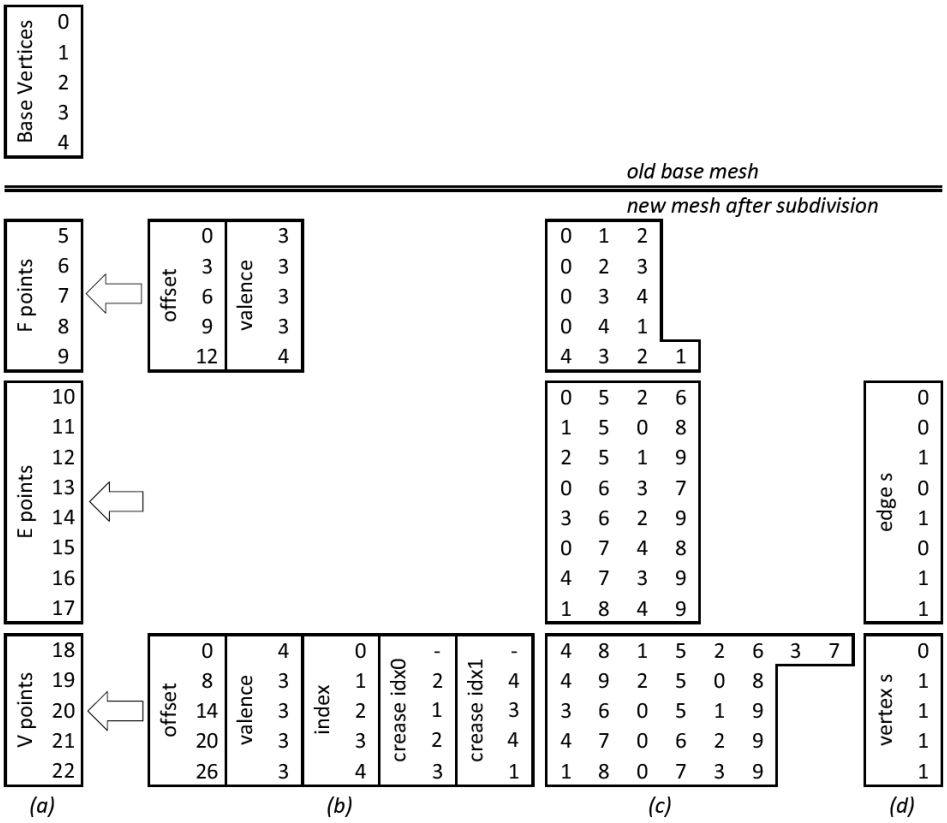


Figure 4.2: Subdivision tables for the pyramid of Figure 4.1: (a) is the vertexbuffer, (b) contains topology information, (c) are indices which point into the vertex buffer and (d) provides the edge and vertex sharpness (from [Niessner et al., 2012]).

4.3 Compute Shaders

Since the project is realized with DirectX11, the use of Compute Shaders (DirectCompute) is possible. To perform the subdivision, three different compute shader kernels are used, one for each vertex type (Face-, Edge- and Vertex-Kernel). Since the Edge-Kernel uses calculation results from the Face-Kernel, the different kernels are not running in parallel, only within one shader stage the vertices are calculated in parallel. For performance reasons, all kernels operate at the same Vertex Buffer, which has to support simultaneous read and write operations. A possible way to achieve this, is to use a

```
RWBuffer<float> vertexBufferRW;
```

in the shader.
In this case, the requirement for the simultaneous read/write process is, that only scalar values are allowed. For convenience some structs and methods are defined:

```
struct Vertex
{
    float4 p;
    float4 c;
    float2 t;
    float3 n;
    float3 t1;
    float3 t2;
};

struct VertexRaw
{
    float values[NUM_ELEMENTS];
};

Vertex readVertex(int index)
{
    VertexRaw v;
    [unroll] for (int i=0;i<NUM_ELEMENTS;i++)
      { v.values[i] = vertexBufferRW[index * NUM_ELEMENTS + i]; }
    return (Vertex)v;
}

void writeVertex(int index, Vertex v)
{
    VertexRaw vRaw = (VertexRaw)v;
    [unroll] for (int i=0;i<NUM_ELEMENTS;i++)
      { vertexBufferRW[index * NUM_ELEMENTS + i]=vRaw.values[i]; }
}
```

For the framework side, it is important to set all the correct bindflags. The UnorderedAccess-flag is set to support the simultaneous read/write-ability, which is needed at the Compute-Shader stage. The ShaderResource-flag is necessary, since we use the VertexBuffer at the bezier-patch stage to provide adjacency informations in a read-only manner. The VertexBuffer-flag is needed, because the buffer is also used as usual vertex buffer for rendering.

```
BufferDescription vbDesc = new BufferDescription();
vbDesc.BindFlags = BindFlags.ShaderResource
                             | BindFlags.UnorderedAccess
                             | BindFlags.VertexBuffer;
vbDesc.Usage = ResourceUsage.Default;
vbDesc.CpuAccessFlags = CpuAccessFlags.None;
vbDesc.OptionFlags = ResourceOptionFlags.None;
vbDesc.SizeInBytes = DefaultVertex.SizeInBytes * numberOfVertices;
```

As a counterpart to the RWBuffer in the compute shader, the UnorderedAccessView has to be initialized properly:

```
this.vbUAVDesc = new UnorderedAccessViewDescription();
this.vbUAVDesc.Format = Format.R32_Float;
this.vbUAVDesc.Dimension = UnorderedAccessViewDimension.Buffer;
this.vbUAVDesc.Buffer.FirstElement = 0;
this.vbUAVDesc.Buffer.ElementCount =
     numberOfVertices *
     DefaultVertex.SizeInBytes / 4; //Number of Elements of 1 Vertex
```

Compute Shaders can be run on many threads in parallel, within a thread group. A desired number of thread groups can be invoked by the Dispatch-Command. The number of threads in a thread group is defined by the numthreads-attribute in the shader. Figure 4.3 explains the scheme:

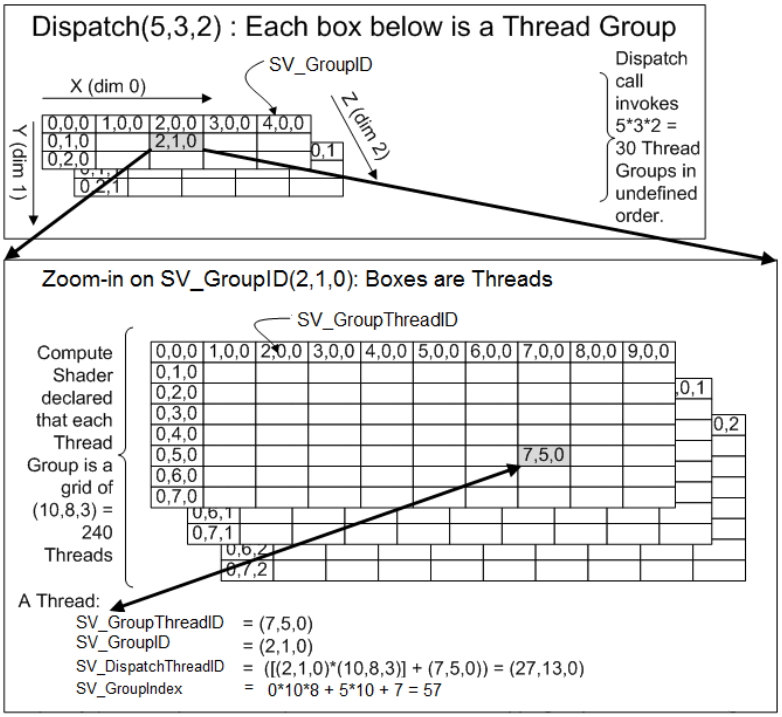


Figure 4.3: relationship between dispatch and numthreads (from [Microsoft, 2013e]).

In our application, the correct number of threads has to be invoked for the three subdivision kernels at each subdivision level:

```
int facedispatch = ((t.CountFaceVertices - 1) / numthreads) + 1;
int edgedispatch = ((t.CountEdgeVertices - 1) / numthreads) + 1;
int vertexdispatch = ((t.CountVertexVertices - 1) / numthreads) + 1;
```

```
this.faceKernelPass.Apply(this.device.ImmediateContext);
this.device.ImmediateContext.Dispatch(facedispatch, 1, 1);

this.edgeKernelPass.Apply(this.device.ImmediateContext);
this.device.ImmediateContext.Dispatch(edgedispatch, 1, 1);

this.vertexKernelPass.Apply(this.device.ImmediateContext);
this.device.ImmediateContext.Dispatch(vertexdispatch, 1, 1);
```

At the shader, we use the SVDispatchThreadID to calculate the correct offset for the respective entry in the subdivision table, for instance to get vertexid and valence:

```
void FaceKernelMain (uint3 gtid : SV_GroupThreadID,
                     uint3 dtid : SV_DispatchThreadID)
{
   int index = dtid.x;
   int vertexid = subdivTable[index * STRIDE + 0];
   int valence = subdivTable[index * STRIDE + 2];
}
```

For the use of effects and rendering features like normal- uv- and displacementmapping, the following attributes of a vertex get subdivided: Position, Normal, Texture, Tangent, Bitangent, Color

4.4 Creation of the Bicubic Bezier Patch

To get the necessary adjacent information to the GPU, which is not inherently given at the tessellation stage, we use adjacent tables (similar to the subdivision tables from the previous Chapter), that provide meta information about the current patch and the neighbouring patches, as well as the indices of the neighbouring vertices that are used to generate the control points. Figures 4.4 and 4.5 demonstrate an example entry for a specified face. The use of the adjacent meta information fields is explained in the Chapter 4.6.

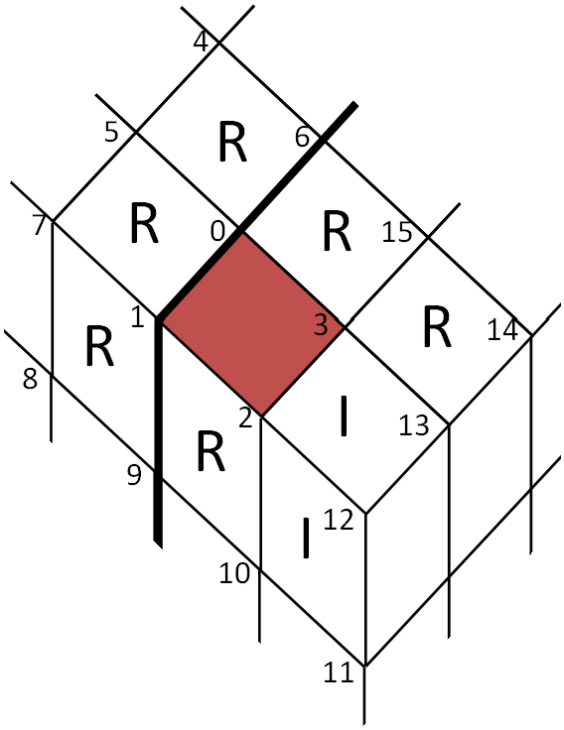


Figure 4.4: Adjacent faces for a regular patch (red), the thicker line represents a sharp edge, (R=Regular, I=Irregular).

Patch Type	Vertex sharpnesses				Direct Adjacent Face Types				Cross Adjacent Face Types				Adjacent Vertices											
Regular	1	1	0	0	R	R	I	R	R	R	I	R	6	4	5	7	8	9	10	11	12	13	14	15

Figure 4.5: Adjacent table for the patch illustrated at Figure 4.4:the example entry is generated for the red patch with indices [0 1 2 3].

The adjacent information is gathered by the following algorithm (iterated for all 4 vertices)(Figure 4.6):

1. Get all edges of a vertex
2. Sort out all edges, which are incident
3. Get the face, which contains all vertices of the remaining edges
4. Write all vertices (except the source vertex) of the found face to the adjacent table (in the right order)
5. Write the meta information of the found face to the adjacent table

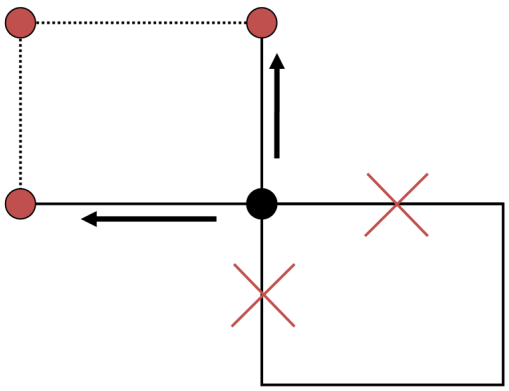


Figure 4.6: Getting the necessary adjacent information (The red vertices).

That algorithm delivers all adjacent vertices and the meta-information of the cross-adjacent faces. The direct-adjacent faces are derived easily with the help of the vertices of the incident edges.

4.5 Rendering of the Bicubic Bezier-Patch

The rendering of the bicubic bezier-patch is implemented with the tessellation technique of DirectX. The control points of the patch are calculated at the hull shader stage of the tessellation pipeline.

The hull shader stage

The hull shader stage consists of two separate passes, one per incoming patch vertex and one patch-constant-function, where the tessellation factors are calculated. To get a seamless transition between different patches, not only the vertex positions have to be calculated, but also normals, texture coordinates, and tangents have to be taken into consideration. Since the Patch-Constant-Function has a patch-constant-output-limit of 32, it would be impossible to stream all the necessary control point information from there to the domain shader stage. We adapt the existing scheme, as it is shown in Figure 3.6). We use the per vertex-part of the hull shader and define ownership-rules for each of the 16 controlpoints, so that every patch vertex has the ownership for 4 output control points. The ownership-scheme is shown in Figure 4.7

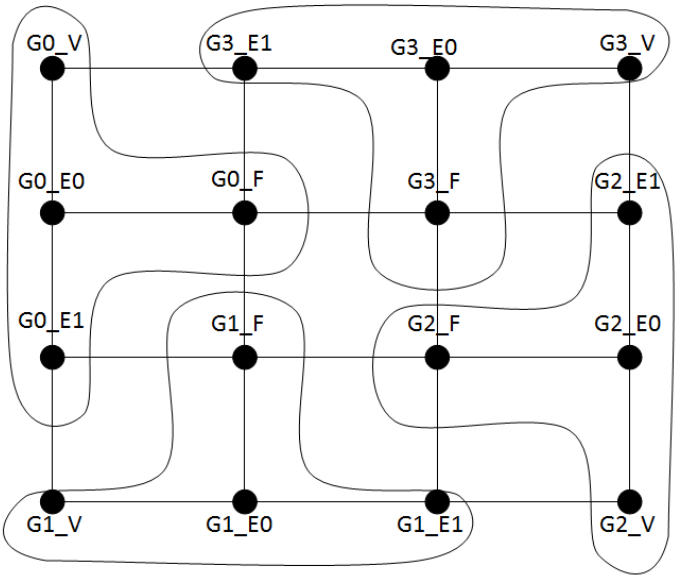


Figure 4.7: Ownership structure for the 16 control points, 4 different groups, 4 controlpoints per group.

A regular patch always creates controlpoints, that generate the smooth limit surface, with one exception, namely if one or more edges is infinitely sharp. In our case, semi-sharpness is completely handled by the rules defined in the subdivision kernels, because the rule of DeRose [DeRose et al., 1998] guarantees a sharpness of zero after enough subdivision levels. If that is not the case, and an edge proves to be still sharp at the last subdivision level, it is a real sharp edge. The bezier-patch has to consider this special case, otherwise, the sharp edge would be smoothed out. The information in the adjacent table (figure 4.5) is used to make the necessary changes in the control point calculation. The affected vertices are held on the position they had before, and are not weighted with their adjacent information like the vertices that lie on smooth edges, this secures the correct preservation of hard edges.

The use of RWBuffers in Shader Model 5.0 is only possible in compute- and pixelshaders. The vertex buffer at this stage is used as render buffer, and as source for the adjacent vertices. Since at this stage (and the following render stages) the vertex buffer is used as readonly-resource, we can bind it as

```
Buffer<float> vertexBuffer;
```

At the framework side the Vertex Buffer must be bound as a ShaderResourceView:

```
ShaderResourceViewDescription srvDesc =
      new ShaderResourceViewDescription();
srvDesc.Format = Format.R32_Float;
```

```
srvDesc.Dimension = ShaderResourceViewDimension.Buffer;
srvDesc.Buffer.FirstElement = 0;
srvDesc.Buffer.ElementCount =
     numberOfVertices *
     DefaultVertex.SizeInBytes / 4;
```

Another issue is a correct definition of the hull shader output format, which meets our criteria. In HLSL, arrays of structs are not allowed, and also the use of the same struct more than once inside of another struct is not possible, because all of the semantics have to be unique. This leads to the following definition of our output structure:

```
struct HSOutput
{
    HSInput v;
    HSEdge0 e0;
    HSEdge1 e1;
    HSFace f;
};

struct HSEdge0
{
    float3 p     : E0_POSITION;
    float2 t     : E0_TEXCOORD0;
    float3 n     : E0_TEXCOORD1;
    float3 t1    : E0_TEXCOORD2;
    float3 t2    : E0_TEXCOORD3;
};

struct HSEdge1
{
    float3 p     : E1_POSITION;
    float2 t     : E1_TEXCOORD0;
    float3 n     : E1_TEXCOORD1;
    float3 t1    : E1_TEXCOORD2;
    float3 t2    : E1_TEXCOORD3;
};

struct HSFace
{
    float3 p     : F_POSITION;
    float2 t     : F_TEXCOORD0;
    float3 n     : F_TEXCOORD1;
    float3 t1    : F_TEXCOORD2;
    float3 t2    : F_TEXCOORD3;
```

```
};
```

The domain shader stage

The fixed-function tessellator stage generates new vertices, according to the tessellation factors of the patch-constant function of the hull shader stage. At the domain shader stage the cubic bezier-patch is evaluated for every generated vertex. To get the correct weighting, we use bernstein-basis-functions, which operate at the given input patch u/v-coordinates (example in hlsl code):

```
float4 BernsteinBasisCubic(float u)
{
    float i = 1.0f - u;

    return float4( i * i * i,
                   3 * u * i * i,
                   3 * u * u * i,
                   u * u * u );
}

float4 basisU = BernsteinBasisCubic( uv.x );
float4 basisV = BernsteinBasisCubic( uv.y );
```

For a seamless transition between different patches, the evaluation contains the following vertex attributes: Position, Normal, Texture, Tangent, Bitangent. If the patch is an Irregular patch (stored in the meta information of the adjacent table), those vertex attributes are interpolated linearly.

At this stage, a displacement can be applied to the position, if a displacement map is available. The displacement map is sampled at the calculated texture coordinate, and the resulting value is added to the vertex position along the normalized normal vector. These displacements are also seamless across the border of two adjacent patches. The resulting values are forwarded to the chosen pixel shader.

4.6 Holes

According to Niessner [Niessner et al., 2012], "the arrangement of bicubic patches created by adaptive subdivision ensures that adjacent patches correspond either to the same subdivision level, or their subdivision levels differ by one." At patch borders with different subdivision levels, there are holes in the mesh introduced, and the watertightness is not given anymore, see Figure 4.8 (Left).

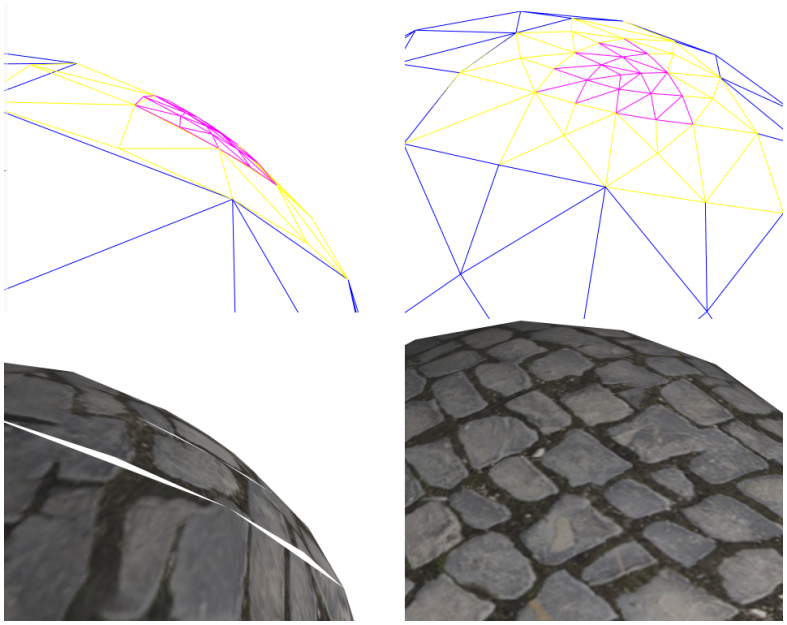


Figure 4.8: Left: Different subdivision levels introduce holes, Right: holes are avoided with adapted tessellation factors.

As explained in Chapter 3.4 we use the adjustment of tessellation factors at those edges, where the levels differ by 1. The information in the adjacent table (figure 4.5) is used to determine, at which edges a subdivision level change is happening. If such an edge is identified, the tessellation factor at the appropriate edge of the patch with the lower subdivision level is doubled in comparison to the other patch, to ensure the same tessellation factors at all shared edges (Figure 4.8 (Right)).

Our project also provides the option to perform adaptive tessellation based on the edge size in screen-space. This technique does not work at the transition borders, because the edge sizes in screen-space are calculated, before the tessellator compensates the edge-factor difference. As a result, the calculated edge sizes are always different at those borders, the tessellation factors would not match. That would reintroduce holes in the geometry. So, only for this special case at those borders a distance-to-the-camera approach is used, the edgefactors are calculated with the reciprocal value of the distance. Since the factors at the other edges and the interior-factors are derived from the screen-space approach, there is an equally tessellated mesh produced at most

parts anyway. Figure 4.9 shows the adaptive tessellation at transition borders.

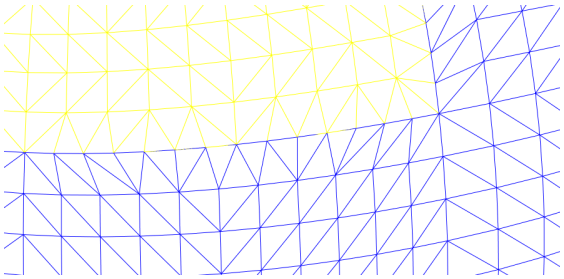


Figure 4.9: Adaptive tessellation at different subdivision levels (lower level is blue, higher level yellow).

CHAPTER 5

Results

We used our arc model to test the increase-rate of the number of faces, compared to the regular subdivision scheme. No edges are sharpened. From level 2, the increase of the number of faces is constant. Already from level 4, a decrease in the number of faces of 94 % compared to the regular subdivision scheme can be obtained. In Figure 5.1 the used arc model is shown with enabled tessellation, Figure 5.2 shows the resulting numbers.

Next, the sharpness values of all edges were increased. At value 2, there is a saving of 91 %, compared to the regular subdivision scheme, at value 4 the saving is still at 47 %, although there exist features up to subdivision level 4. The difference shows, that edges with value 2 can be handled at an earlier subdivision level by the bezier-patch algorithm. Figures 5.4 and 5.5 show the resulting values.

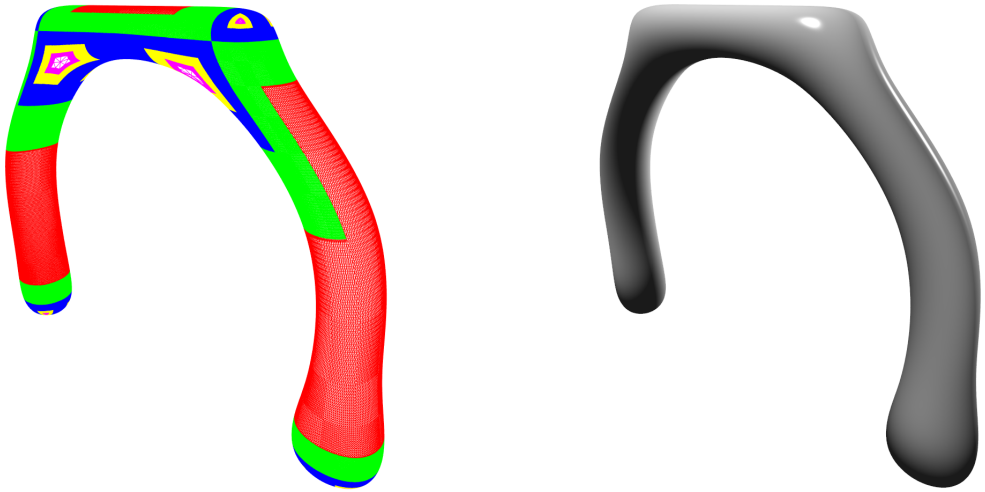


Figure 5.1: The arc model, at the left side the patches are seen (different colors indicate the different subdivision levels), at the right side the shaded model (all edges are smooth).

Level	Faces Regular	Faces Irregular	Faces Total	Compared to Reg. Subdiv	Face # Reduction
0	19	27	46	46	0.00 %
1	71	56	127	184	30.98 %
2	239	56	295	736	59.92 %
3	407	56	463	2944	84.27 %
4	575	56	631	11776	94.64 %

Figure 5.2: results for model: arc; sharpness: 0.

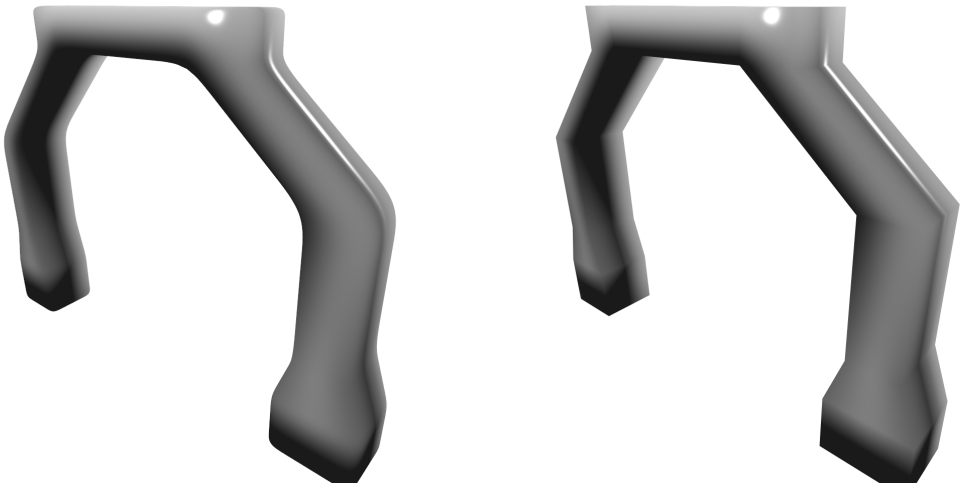


Figure 5.3: The arc model, at the left side, all edges are tagged with sharpness value 2, at the right side with a value of 4.

Level	Faces Regular	Faces Irregular	Faces Total	Compared to Reg. Subdiv	Face # Reduction
0	0	46	46	46	0.00 %
1	0	184	184	184	0.00 %
2	680	56	736	736	0.00 %
3	848	56	904	2944	69.29 %
4	1016	56	1072	11776	90.89 %

Figure 5.4: model: arc; sharpness: 2.

Level	Faces Regular	Faces Irregular	Faces Total	Compared to Reg. Subdiv	Face # Reduction
0	0	46	46	46	0.00 %
1	0	184	184	184	0.00 %
2	184	552	736	736	0.00 %
3	1104	1288	2392	2944	18.75 %
4	6200	56	6256	11776	46.87 %

Figure 5.5: model: arc; sharpness: 4.

Next, the cube model with different sharpness tags (equal at all edges) is presented.

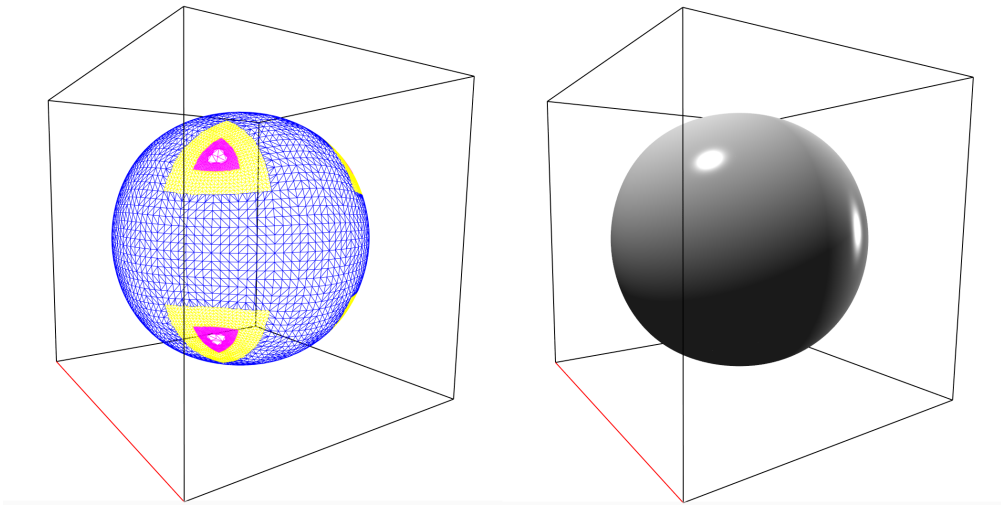


Figure 5.6: The cube model, all edges with sharpness 0.

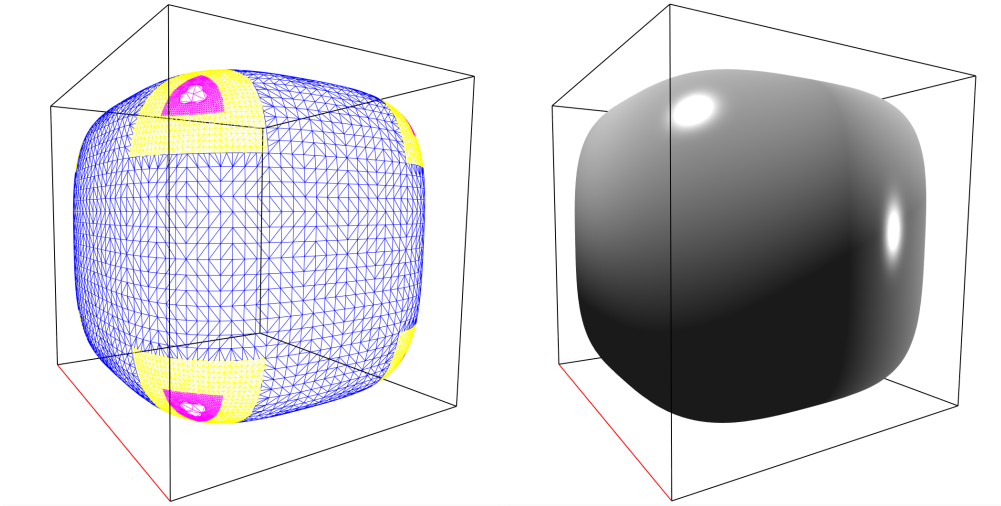


Figure 5.7: The cube model, all edges with sharpness 1.

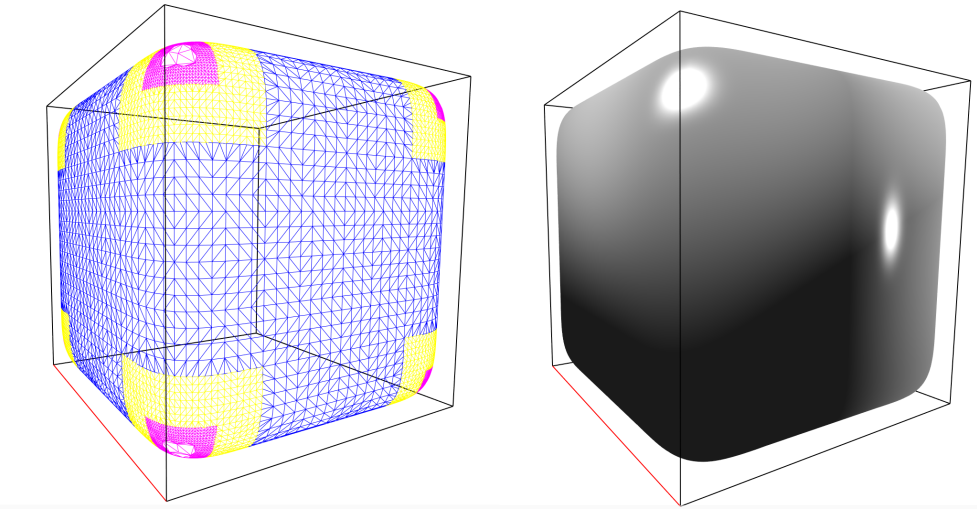


Figure 5.8: The cube model, all edges with sharpness 2.

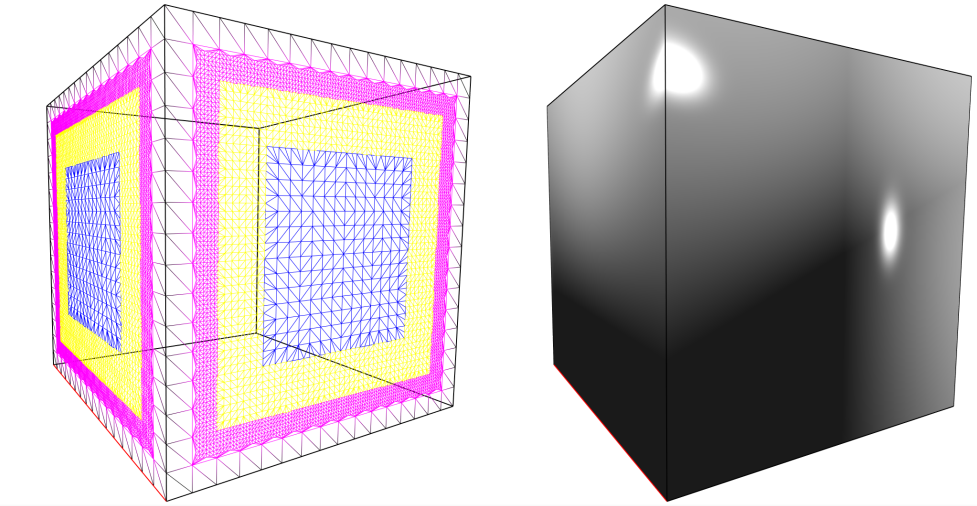


Figure 5.9: The cube model, all edges with infinite sharpness.

If the sharpness is not changed at all edges, different objects can be created from a base cube. In this case, the piecewise increase of the sharpness at 2 faces results in a cylindric shape.

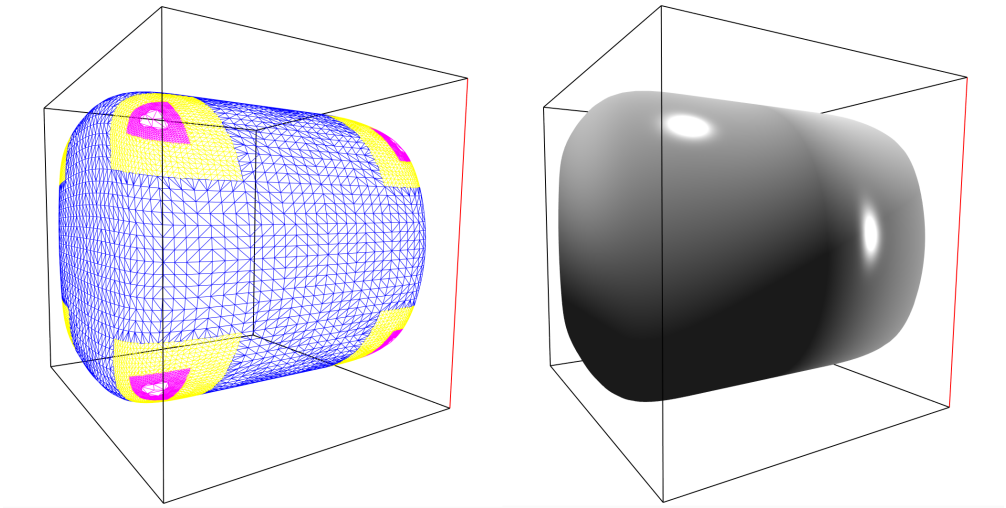


Figure 5.10: The cube model, at 2 faces the edges are tagged with sharpness 1.

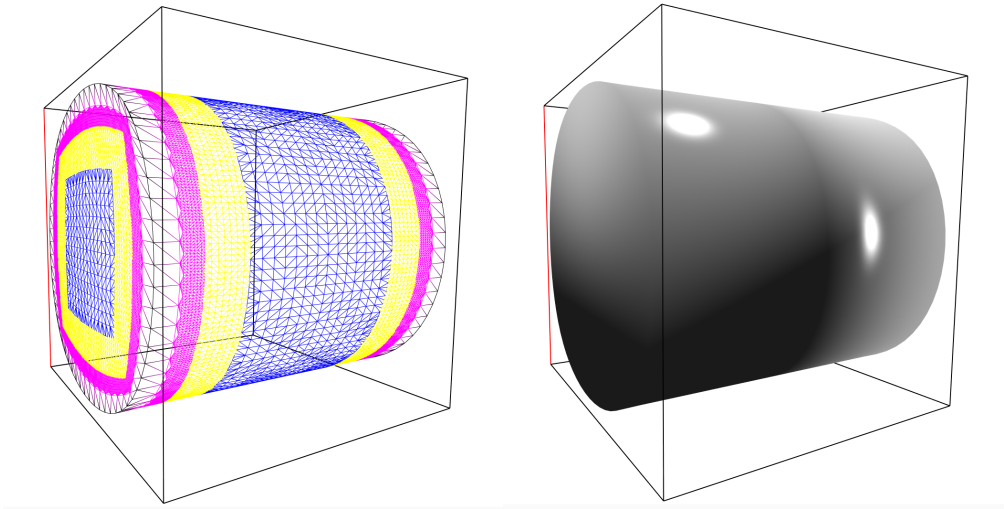


Figure 5.11: The cube model, at 2 faces the edges are tagged infinitely sharp.

The pyramid model consists of triangles only, and is quadrangulized at the first subdivision level, different sharpness values are applied.

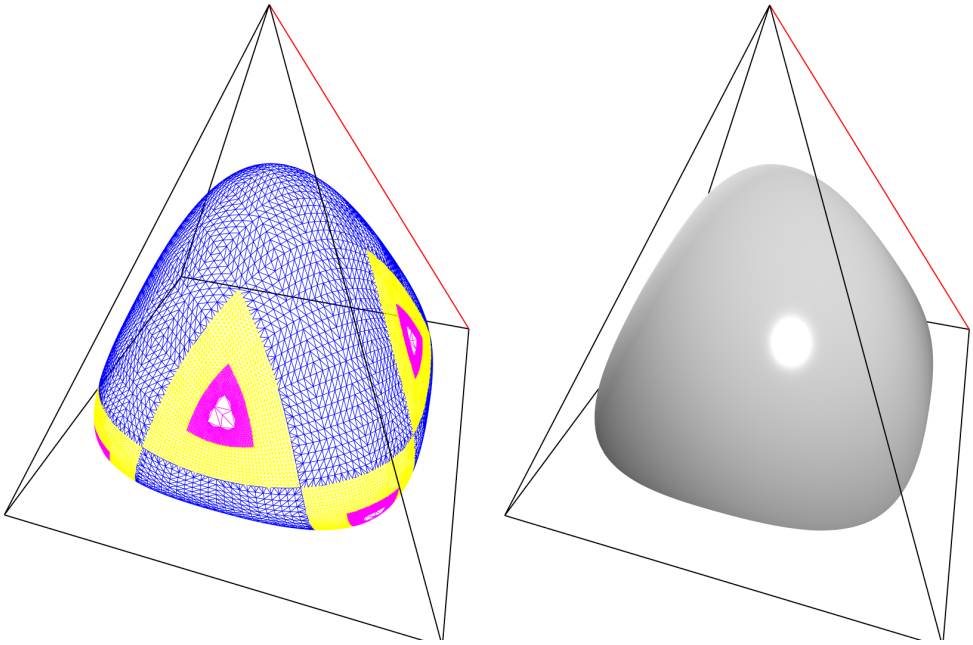


Figure 5.12: The pyramid model, all edges are tagged with sharpness 1.

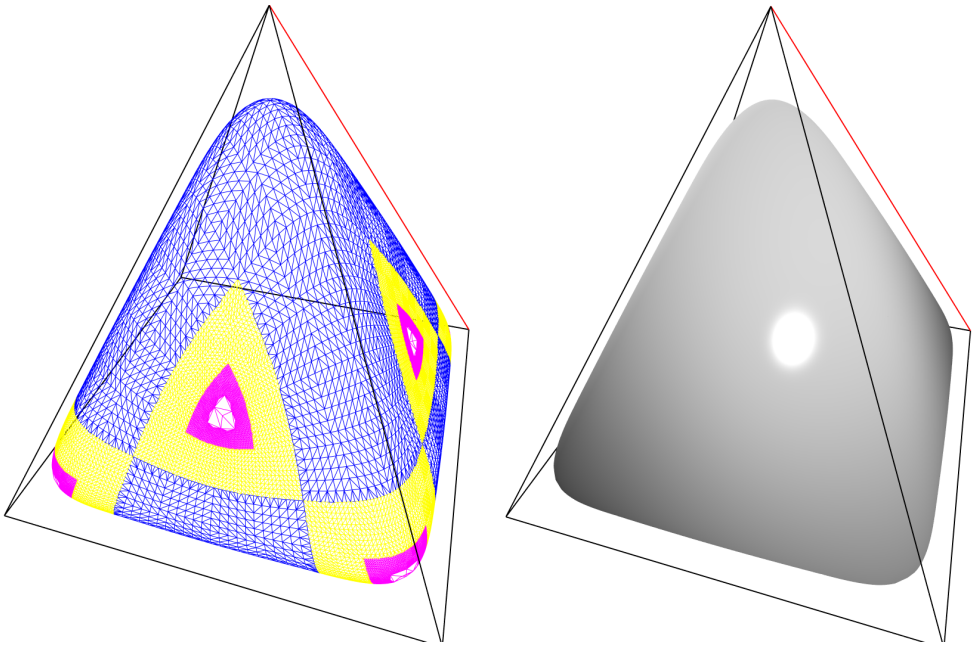


Figure 5.13: The pyramid model, all edges are tagged with sharpness 2.

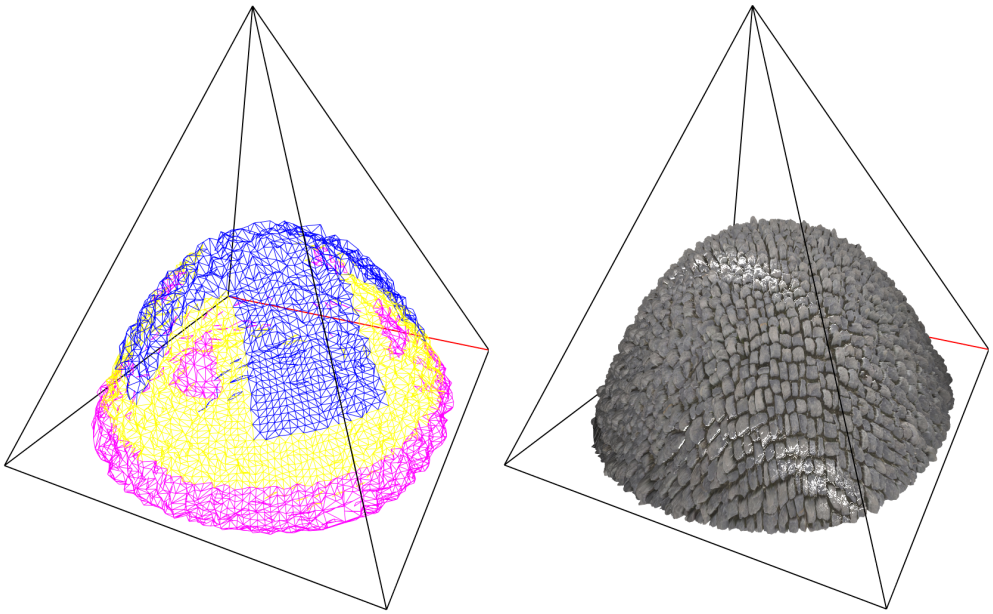


Figure 5.14: The pyramid model, the base edges are tagged with sharpness 2, at the right side diffuse-, normal- and displacement map are applied.

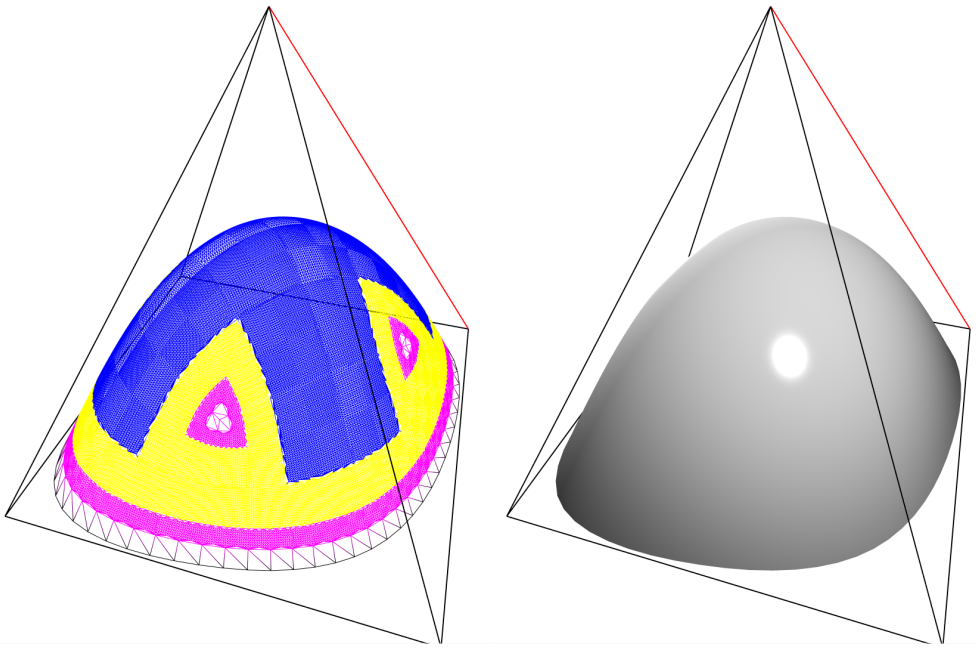


Figure 5.15: The pyramid model, edges have pairwise different sharpness tags.

The combination of different sharpness tags, diffuse-, normal- and displacementmapping creates a curved stone wall with real geometry, derived just from a coarse cube base model.

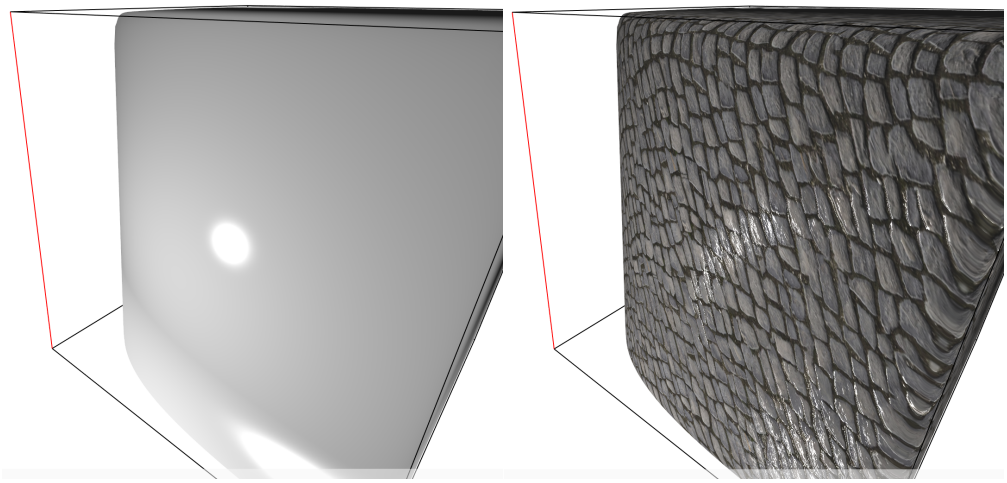


Figure 5.16: The cube model, all edges, instead the red one are tagged with infinite sharpness, at the right side diffuse- and normal map are applied.

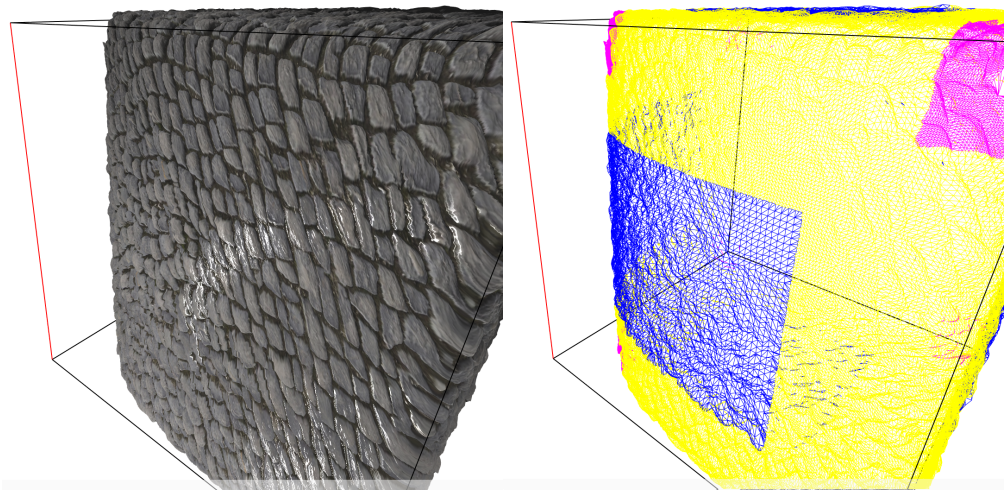


Figure 5.17: The cube model, a displacment map is additionally applied.

CHAPTER 6

Conclusion

6.1 Summary

In this thesis we focused on the Catmull-Clark subdivision scheme, and its optimized implementation on the GPU. In Chapter 2 we presented an overview of several approaches to optimize the Catmull-Clark subdivision scheme and to achieve good realtime results with current GPU technologies. The techniques of two of these methods were combined and an alternative approach to solve the issues about adjacent patches with different subdivision levels was given in Chapter 3. In Chapter 4, the resulting method was implemented with C#/DirectX into the existing Helix 3D Toolkit SharpDX Framework, the results were shown in Chapter 5.

We discovered that current GPU technologies, like the use of compute shaders or the tessellation unit, especially the combination of them are a set of powerful tools and offer a flexible way of GPU programming, that is certainly not at the end of its possibilities. The application of those state-of-the-art features, for instance the use of read/write buffers on the GPU was very interesting and helpful for future projects. Also to deal with a managed wrapper for DirectX, that enables the convenient use of managed code was an interesting task. The use of high-level user-controls within the Helix 3D Toolkit framework, that provides already the basic necessary rendering functions was appealing.

6.2 Limitations and Future Work

Although the standard Catmull-Clark subdivision rules are simple to use, there exists a wide variety of different extensions. Several mesh-topology characteristics can lead to complex variations and many exceptional cases, that have to be considered. Due to limited time resources, only some of the possibilities could be explored. The support of semi-sharp creases could be achieved, also an almost-realtime sharpness-changing mechanism was implemented. Several rendering features like normal-mapping and displacement mapping, that require seamless vertex

attributes, were achieved. The support of more input types besides of quads and triangles is a possible future task, also the support for meshes with boundaries could be implemented, since our algorithm deals only with closed meshes. The algorithm, which creates all the necessary preprocessing data could be further improved to shorten the wait while a model is changed or the sharpness of an edge is adjusted while runtime. To make the CPU-prepocessing-tasks multihreaded, could also be a benefit for that. The user-interface, where edges can be selected, could be implemented in a more convenient way. Instead to switch between edges with a button, some sort of object picking could be implemented to select edges interactively on the model. The support of hierarchical edits in our method could be done with little effort since the used technique is already prepared for that. Furthermore, our project supports adaptive tessellation based on the edge size in screen space. This technique does not work with the transition borders of different subdivision levels, so for this special case a distance-to-the-camera approach is used. If the screenspace-method would be required on all edges, another approach, like the presented technique with transition patches, and therefor the support of more different bezier patches would be necessary.

Bibliography

[Bolz and Schröder, 2004] Bolz, J. and Schröder, P. (2004). Evaluation of subdivision surfaces on programmable graphics hardware.

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

[DeRose et al., 1998] DeRose, T., Kass, M., and Truong, T. (1998). Subdivision surfaces in character animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 85–94, New York, NY, USA. ACM.

[Doo, 1978] Doo, D. (1978). A subdivision algorithm for smoothing down irregularly shaped polyhedrons. *Proceedings on Interactive Techniques in Computer Aided Design (Bologna, Italy)*, pages pp. 157–165.

[Dyn et al., 1990] Dyn, N., Levine, D., and Gregory, J. A. (1990). A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graph.*, 9(2):160–169.

[Gregory, 1974] Gregory, J. (1974). Smooth interpolation without twist constraints. *Computer Aided Geometric Design, R. E. Barnhill and R. F. Riesenfeld, Eds. Acadmic Press, New York*, pages 71–87.

[Helix 3D Toolkit, 2013] Helix 3D Toolkit (2013). http://helixtoolkit.codeplex.com. Accessed: 2013-10-05.

[Kobbelt, 2000] Kobbelt, L. (2000). √3-subdivision. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 103–112, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Kovacs et al., 2009] Kovacs, D., Mitchell, J., Drone, S., and Zorin, D. (2009). Real-time creased approximate subdivision surfaces. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 155–160, New York, NY, USA. ACM.

[Loop et al., 2009] Loop, C., Schaefer, S., Ni, T., and Castano, I. (2009). Approximating subdivision surfaces with gregory patches for hardware tessellation. In *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH Asia '09, pages 151:1–151:9, New York, NY, USA. ACM.

[Loop, 1987] Loop, C. T. (1987). Smooth Subdivision Surfaces Based on Triangles.

[Loop and Schaefer, 2008] Loop, C. T. and Schaefer, S. (2008). Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1).

[Microsoft, 2013a] Microsoft (2013a). http://msdn.microsoft.com/en-us/library/windows/desktop/ff471356 Accessed: 2013-10-05.

[Microsoft, 2013b] Microsoft (2013b). http://msdn.microsoft.com/en-us/library/windows/desktop/ff476136 Accessed: 2013-10-05.

[Microsoft, 2013c] Microsoft (2013c). http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331 Accessed: 2013-10-05.

[Microsoft, 2013d] Microsoft (2013d). http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340 Accessed: 2013-10-05.

[Microsoft, 2013e] Microsoft (2013e). http://msdn.microsoft.com/en-us/library/windows/desktop/ff476405 Accessed: 2013-10-05.

[Microsoft, 2013f] Microsoft (2013f). http://msdn.microsoft.com/en-us/magazine/dd419663.aspx. Accessed: 2013-10-05.

[Musialski et al., 2007] Musialski, P., Tobler, R. F., Maierhofer, S., and Wüthrich, C. A. (2007). Multiresolution geometric details on subdivision surfaces. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 211–218, New York, NY, USA. ACM.

[Ni et al., 2009] Ni, T., Castaño, I., Peters, J., Mitchell, J., Schneider, P., and Verma, V. (2009). Efficient substitutes for subdivision surfaces. In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 13:1–13:107, New York, NY, USA. ACM.

[Niessner et al., 2012] Niessner, M., Loop, C., Meyer, M., and Derose, T. (2012). Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Trans. Graph.*, 31(1):6:1–6:11.

[SharpDX, 2013] SharpDX (2013). http://www.sharpdx.org. Accessed: 2013-10-05.

[Stam, 1998] Stam, J. (1998). Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 395–404, New York, NY, USA. ACM.