Feature Adaptive Rendering of Loop Subdivision Surfaces on GPU

Yuncen Huang, Jieqing Feng, and Baoguang Yang

State Key Lab of CAD&CG, Zhejiang University
Hangzhou, 310058, P.R.China

March 25, 2013

Zusammenfassung
We present a feature adaptive rendering method for Loop subdivision surfaces on GPU. Both positions and normals are evaluated accurately on the true limit Loop subdivision surfaces. The main idea is to render regular regions of the surface directly with tessellation shader and recursively subdivide irregular regions with compute shader till the prescribed maximum depth. Our algorithm accommodates input triangular control mesh of arbitrary topology. Furthermore, we reinforce our algorithm with a watertight evaluation method, i.e., ensuring positions and normals on the boundary of adjacent patches are bitwise identical. As a result, surfaces generated are suitable for displacement mapping. Finally, we also develop an LOD approach for feature adaptive rendering that can adjust the subdivision depth or the patch tessellation factors adaptively. Besides, it admits hierarchical surface editing. The implementation results show that our method outperforms the rendering approaches of Li’s approximate patching and Stam’s direct evaluation.

Keywords Subdivision surfaces, GPU, tessellation symbolic computation

1 INTRODUCTION

The pioneer work by Catmull and Clark [CC78] and Doo and Sabin [DS78] introduced an era of surface modeling, since the control meshes of subdivision surfaces can be of arbitrary topology. After that, many subdivision surface approaches were proposed [ZS99]. The Loop subdivision surface is one of such representative approach [Loo87]. It admits a triangular mesh as the control net, that is prevalent geometric representation in industrial applications. After the milestone work by Stam[Sta98, Sta99], that provides exact evaluation method for the subdivision surfaces at arbitrary parameter, the subdivision surfaces have been widely used in movie and 3D game industries, as well as CAD/CAM area.
Traditionally, a subdivision surface is generated by recursively refining its control mesh. With the advent of modern GPU techniques, how to efficiently apply the formidable computing power of GPU to the subdivision surfaces is becoming an attractive research topic in animation and game industries. Among these GPU techniques, hardware tessellation is unique in that it tessellates original meshes into much more tile primitives, and generates triangle geometry for the next rasterization stage. The feature is suitable for generating subdivision surfaces. Generating and rendering Loop subdivision surfaces in an exact and efficient manner by exploiting the advanced feature on GPU is still an open problem.

Nießner et al. [NLMD12] recently presented a creative method of feature adaptive GPU rendering Catmull-Clark subdivision. Inspired by their work, we propose a novel method to generate and render the true limit Loop subdivision surfaces efficiently on GPU by exploiting compute shaders and tessellation shaders, without requiring extra refinement in advance.

The main contributions of our article are

- Generating and rendering Loop subdivision surfaces precisely up to the prescribed maximum depth on GPU.
- Accommodating control meshes of arbitrary topology, as well as semi-sharp creases and hierarchical details.
- The first table-driven Loop subdivision method with feature adaptive refinement on GPU.
- A bitwise identical evaluation positions and normals along the common boundary between adjacent patches which ensures the watertightness of the subdivision surfaces.
- A level-of-detail method balancing quality and efficiency within the maximum subdivision level and tessellation factors.

The rest of this paper is arranged as follows: related work and preliminary of Loop subdivision surfaces are given in Section 2; The details of the proposed algorithm are described in Section 3; Implementation results and discussions are given in Section 4; Conclusions are drawn in Section 5.

2 RELATED WORK AND PRELIMINARY

2.1 Related Work

Efficiently rendering subdivision surfaces is always a hot topic in computer graphics, especially with the emergence of parallel computing schemes on GPU. In this paper, we only review the related work on rendering Loop subdivision surfaces on GPU. These can be classified into three types: global refinement, direct evaluation, and approximate patching algorithm.
Global refinement approaches handle subdivision surfaces according to its original definition, i.e., recursively applying refinement rules. A control mesh is refined recursively and globally up to the maximum subdivision depth. In general, the approaches always require significant memory and bandwidth to store and transfer the refined mesh data, which tends to limit the performance.

Pulli and Segal [PS96] proposed a compact data structure to minimize the memory cost, i.e., packing two adjacent triangles into a quadrilateral. On the other hand, Bolz and Schröder [BS02] adopted a lookup table to store valence-related basis functions for surface evaluation. However, it can not be adapted to the control mesh of arbitrary topology.

Shuie et al. [SJP05] generated the subdivision surfaces with valence-relevant tabulate stencils to organize the control mesh. It can not be applicable to the control meshes of arbitrary topology, either. Furthermore, it requires an extra subdivision step in advance. Minho et al. [KP05] extended Shuie’s method to Loop subdivision surfaces. However, watertightness of the resulting surface after one level of refinement is not guaranteed.

In 1998, Stam proposed the milestone work to evaluate the Catmull-Clark and Loop subdivision surfaces exactly via eigen analysis of subdivision matrix [Sta98, Sta99]. In his scheme, the extraordinary vertices should be isolated as a prerequisite. To this end, two additional subdivision steps should be performed on the input control mesh in general. This will generate much more patches as the input of rendering step. Although Stam’s algorithm can be ported onto GPU, code branches in domain shader will limit the performance. It is also cumbersome to extend the algorithm to the semi-sharp creases case and hierarchical editing. Rather than evaluating polynomials, Bischoff et al. [BKS00] computed Loop subdivision surfaces on GPU via forward difference to accelerate the evaluation.

Since generating Loop subdivision surfaces on GPU is expensive in general, researchers began to seek approximate subdivision surface as an alternative, rather than true limit surface, with sacrifice of surface quality and accuracy to some extent. Among those solutions, approximate patching approach is the representative one.

To achieve good visual effect, a curved PN-triangle decouples the geometry and its surface normal, i.e., a cubic triangular Bézier surface equipped with a quadratic normal field [VPBM01]. This idea is widely applied to 3D games, and inspires the work of subdivision surface approximation.

Boubekeur and Schlick [BS07] proposed QAS to render Loop subdivision surfaces, where each patch is approximated by two quadratic triangular Bézier patch, for geometry, and normal field respectively. Ashish et al. [AFF12] adopted Gregory patches to approximate subdivision surfaces. Due to the fact that the Loop subdivision surface is derived from the definition of quartic box spline, Li et al. [LRZM11] adopted a quartic triangular Bézier patch to approximate the geometry of Loop subdivision surfaces by interpolating 15 uniformly sampled points. To remedy the artifacts along the boundary of irregular patches, the normal field of the surface is also approximated via two quartic triangular Bézier patches.
2.2 Preliminary of Loop Subdivision

A Loop subdivision surface takes a coarse triangular control mesh as input. After several refinements, a smooth mesh is obtained, consisting of more vertices and triangles [Loo87]. The more the subdivisions performed, the smoother the output mesh is. A generated vertex in next subdivision level is defined as a convex combination of old ones according to the subdivision rules, as well as the limit positions and normals [HDD+94]. The limit Loop subdivision surface is $C^2$ continuous almost everywhere except at extraordinary points where it is $C^1$ continuous.

To meet the requirements in 3D game and film applications, users can adjust the refinement rules to generate semi-sharp creases in the Loop subdivision surface, which is $C^0$ continuous.

A semi-sharp crease feature is defined by tagging an edge a sharpness, namely $es$. A vertex as ends of one or more sharp edges is also considered sharp, and its sharpness, namely $vs$, is the average of the sharpness of all its incident edges [DKT98]. The Loop subdivision rules are defined as follows:

**Vertex Point Rules.** A vertex can be classified into one of following types based on its $vs$ and the number of its incident sharp edges, namely $vk$, refer to Table 1. The new vertex is evaluated according to its type, refer to Figure 1, where $\alpha = \frac{5}{8} - \left(\frac{3+2\cos(2\pi/n)}{n}\right)^2$, $v_{\text{smooth}}$ and $v_{\text{crease}}$ are the generated edge vertices by using smooth and crease edge rules respectively.

<table>
<thead>
<tr>
<th>$vk$</th>
<th>$vs$</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2</td>
<td>$\geq 1.0$</td>
<td>smooth rule</td>
</tr>
<tr>
<td>=2</td>
<td>$[0.0, 1.0)$</td>
<td>crease rule</td>
</tr>
<tr>
<td>=2</td>
<td>$\geq 1.0$</td>
<td>corner rule</td>
</tr>
<tr>
<td>&gt;2</td>
<td>$[0.0, 1.0)$</td>
<td>$(1 - vs) \cdot v_{\text{smooth}} + vs \cdot v_{\text{crease}}$</td>
</tr>
</tbody>
</table>

**Table 1: Vertex type classification according to $vk$, and $vs$**

![Abb. 1: smooth (a), crease (b), and corner (c) rules of evaluating a vertex point. Double lines denote sharp edges.](image-url)

Abbildung 1: smooth (a), crease (b), and corner (c) rules of evaluating a vertex point. Double lines denote sharp edges.
Edge Point Rules. The edge point evaluation rules are determined according to the types of its two end vertices, refer to Table 2. Subdividing a sharp edge will generate two child sharp edges with sharpness defined as parent sharpness minus one. If $0 \leq \varepsilon s \leq 1.0$, the edge point should be evaluated as:

$$(1 - \varepsilon s) \cdot v_{ep\text{smooth}} + \varepsilon s \cdot v_{ep\text{crease}}.$$  

where $v_{ep\text{smooth}}$ and $v_{ep\text{crease}}$ are the generated edge vertices by using smooth and crease edge rules respectively.

Tabelle 2: Edge points are evaluated with the rules according to the type of its two end vertices. Masks are shown in Figure 2.

<table>
<thead>
<tr>
<th></th>
<th>smooth/regular</th>
<th>irregular</th>
<th>corner</th>
</tr>
</thead>
<tbody>
<tr>
<td>smooth/dart</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
</tr>
<tr>
<td>regular crease</td>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
</tr>
<tr>
<td>irregular crease</td>
<td>(a)</td>
<td>(c)</td>
<td>(b)</td>
</tr>
<tr>
<td>corner</td>
<td>(a)</td>
<td>(c)</td>
<td>(b)</td>
</tr>
</tbody>
</table>

3 FUTURE ADAPTIVE LOOP SUBDIVISION SURFACES RENDERING ON GPU

3.1 Overview of the Algorithm

Using all the subdivision rules in Section 2.2, a coarse control mesh can be refined in a recursive way. Its limit surface is composed of quartic box spline patches, except for the extraordinary points surrounded by an infinite number of box spline patches.

In our setting, Loop subdivision rules are performed in a feature adaptive way. Here, Features indicate extraordinary vertices or semi-sharp vertices. A patch is regular if its three vertices are regular (valence 6) and all of them have sharpness of zero. Otherwise, the patch is irregular.
The framework of the proposed feature adaptive rendering of Loop subdivision surfaces on GPU, as shown in Figure 3, consists of four steps as follows:

**Subdivision Table Generation.** Subdivision tables are designed and generated for the irregular patches, containing all topology and feature information of irregular patches. Details are described in Section 3.2.

**Parallel Computing.** We upload the the geometry data as well as all subdivision tables to compute shader. The upload of subdivision tables is one-time only. We then execute some kernels to parallel compute the new vertices in each level of subdivision. Details are given in Section 3.3.

**Patch Reconstruction.** For the purpose of eliminating potential T-junctions, we reconstruct the control net of each patch. In this way, tessellation factors of patches need to be a power of two. Details are given in Section 3.4.

**Tessellation.** The control net of each patch is transmitted to GPU. Tessellation shader is employed to tessellate each patch. The details are described in Section 3.5.

We also develop a variant of patch evaluation scheme to guarantee watertightness of tessellated Loop subdivision surface in Section 3.6. Tessellation factors of each patch can be determined in a level-of-detail way, which is described in Section 3.7.

### 3.2 Subdivision Table Generation

Firstly, we identify irregular patches, and then generate subdivision tables for them in each subdivision level, subdivide them recursively up to the maximum depth. Global refinement approaches subdivide the initial control mesh recursively and globally to $4^k N_p$ patches or triangles, where $N_p$ is the number of triangles in control mesh and $k$ is the subdivision depth. Since only irregular patches in each level is adaptively subdivided, less memory and bandwidth is required.
The subdivision table abstracts the topology of the irregular patches, and guides the evaluation of kernels during parallel computing. The specific required buffers for subdivision table are described Section 3.3. The generation of the subdivision tables is unrelated to the geometry of the control mesh, thus the tables are generated once, and can be reused if the topology of the control mesh remains unchanged in surface editing and animation.

For the example in Figure 5, we identify those irregular patches of depth 0, 1, and 2, and then generate their subdivision tables. The corresponding subdivision table of depth 0 and part of subdivision table of depth 1 are shown in Figure 4. If an irregular patch contains sharp edges, the corresponding sharpness values should also be included in the corresponding subdivision table.
3.3 Parallel Computing of Edge Points and Vertex Points

There are two types of kernels in compute shader, edge point kernel and vertex point kernel. In this step, the irregular patches are subdivided in parallel on GPU according to the subdivision rules. Buffers in the subdivision table store indices of relevant vertices, refer to Figure 4.

**Edge Point Kernel.** The edge point kernel requires an index buffer containing 4 vertex indices of two adjacent triangles of the edge.

**Vertex Point Kernel.** The vertex point kernel requires four buffers, an index buffer to store the incident vertices, a valence buffer, an offset buffer, and a predecessor index of the vertex.

3.4 Patch Reconstruction

The purpose of patch reconstruction is to represent the patches properly and eliminate T-junctions along the boundary between two adjacent patches of different subdivision levels. To this end, two types of patches are defined and reconstructed: plain patches and non-plain patches.

In the proposed algorithm, two adjacent patches are either in the same subdivision level or in the subdivision levels that differ by one.

3.4.1 Plain Patches

A plain patch is the one with no neighbouring patches (adjacent to edges) from next subdivision level. They can be further classified into plain regular patches and plain irregular patches. **Plain Regular Patches**, PRP for short, will be directly rendered with tessellation shader. **Plain Irregular Patches**, PIP for short, exist only in the maximum subdivision level. PIP will be rasterized as a triangle, with a fixed tessellation factor 1.0. Tessellation details will be given in Section 3.5.

Abbildung 6: Illustration of patch reconstruction. Plain regular patches are green, plain irregular patches are purple, and non-plain patches are orange.
3.4.2 Non-plain Patches

Non-Plain Patch, NPP for short, is one with adjacent patch in one more subdivision level. All NPPs are regular, since irregular patches are always surrounded by patches of the same subdivision level. In each level of adaptive subdivision, we tab those irregular patches, whose sub-patches will turn into NPPs and irregular patches. The problem is how to eliminate the T-junctions between a NPP and its neighbour patches.

To this end, we have to set a common boundary of adjacent patches with the same tessellation factor, in general. The simplest solution is to adopt a power of two tessellation factor to line up the tessellation of two adjacent patches. A power of two tessellation factor is a limitation for user’s interactions. To avoid it, we split up an NPP into sub-patches, according to the configuration of its neighbour patches, as shown in Figure 7. The sub-patches share the same control net with its parent NPP, but have different parametric sub-domains.

By treating one NPP as several independent sub-patches, we can eliminate all possible T-junctions between patches of different subdivision levels. This means as long as we set shared boundary edges the same tessellation factor, which need not be a power of two, a crack-free tessellation will be generated.

![Abbildung 7: There are 3 possible patterns splitting an NPP into several sub-patches. While NPPs are purple, the patches of the current level of subdivision are yellow, and the patches of the next level of subdivision is green.](image)

3.5 Hardware Tessellation

There are two parameters that can control the mesh resolution of Loop subdivision surface, i.e., tessellation factor for the patches, denoted as $F_{tess}$, and the maximum subdivision depth, denoted as $D_{max}$. In theory, two parameters are independent. However, it is more convenient to determine the maximum depth level according to the user specified tessellation factor, that will be discussed in detail in Section 3.7.

In tessellation shader, we adopt Stam’s method to evaluate exactly the points on the PRPs and NPPs, since they are the regular patches, which are in fact quartic box spline patches. Readers can also adopt other methods to evaluate those regular patches [LRZM11, Pet01]. We evaluate the points on PIPs, with tessellation factors set as 1.0, with Hoppe’s [HDD+94] formulae.
3.6 Watertightness

Sharing same tessellation factor between two adjacent patches is necessary but can not guarantee the watertightness of the tessellated surface because the floating point multiplication is neither associative nor distributive, due to machine floating point representation. The imprecise computations in the machine may cause visible cracks between two adjacent patches, especially when the two adjacent patches are in different subdivision levels.

Thanks to that floating point addition and multiplication are communicative under the IEEE floating point strictness as a compiler flag, we can tessellate Loop subdivision surfaces with GPU in a watertight way. In this subsection, we reinforce our feature adaptive algorithm to a watertight version so that the sampling points and their normals along the shared edge between adjacent patches are bitwise identical for both the same subdivision level and different levels.

3.6.1 Same Subdivision Level

As described earlier, we evaluate the positions and derivatives of a regular patch as a quartic box spline patch, by using Stam’s method [Sta99] as follows:

\[
S(u, v) = P^T B(u, v), (u, v) \in \Omega \\
\frac{\partial S}{\partial u}(u, v) = P^T \frac{d B}{d u}(u, v) \\
\frac{\partial S}{\partial v}(u, v) = P^T \frac{d B}{d v}(u, v)
\]

where \( P \) is the control vertices of the patch as shown in Figure 8a, and \( B(u, v) \) is the basis functions. The parameter domain is a “unit triangle” \( \Omega \) defined as follows

\[
\Omega = \{(u, v) | u \in [0, 1], v \in [0, 1 - u]\}.
\]

Abbildung 8: (a) A regular patch(yellow) of Loop subdivision surface defined by 12 control points; (b) Control points used to evaluate the shared boundary by patch A and B.
Cracks caused by floating point evaluation errors appear on the shared boundary between adjacent patches \((u = 0, \text{ or } v = 0, \text{ or } u + v = 1)\) or shared corner \((u = 1, \text{ or } v = 1, \text{ or } u + v = 0)\), if each patch is evaluated and tessellated independently. We handle the boundary and the corner cases respectively.

**Boundary Case.** Consider these two neighbour patches A and B, sharing same boundary as shown in Figure 8a, suppose the boundary curve in A is \(u = 0\), the boundary curve in B is also \(u = 0\) without loss of generality. Obviously, the parametric directions of two boundary curves are opposite. If we evaluate the same sampling point \(p_A\) at \((0, 0.3)\) on patch A and \(p_B\) at \((0, 0.7)\) on patch B, the crack will probably be produced. To overcome this problem, we design a bitwise identical evaluation approach according to the symmetry of basis functions along the boundary, we have

\[
\sum_{i=0}^{11} p_i^A \cdot b_i(0, v) = \sum_{i=2}^{11} p_i^A \cdot b_i(0, v),
\]

where \(\{p_i^A\}\) are the control points respectively of patch A and patch B, and \(b_i(u, v)\) is i-th basis function. Particularly \(b_0(0, v)\) and \(b_1(0, v)\) equal zero due to boundary case.

Due to the symmetry of basis functions along the boundary, we have \(b_i(0, v) = b_{13-i}(0, 1 - v), i = 2, 3, ..., 11\). Further more, we also have \(p_i^B = p_{13-i}^A\). Thus the formula for boundary curve of the patch B can be rewritten as:

\[
S_B(0, v) = \sum_{i=2}^{11} p_i^A \cdot b_{13-i}(0, 1 - v).
\]

To achieve bitwise identical evaluation, we rearrange the orders of terms in the two summations (2) and (4) as follows:

\[
S_A(0, v) = \left[ p_2^A b_2(0, v) + p_{11}^A b_{11}(0, v) \right] + \left[ p_3^A b_3(0, v) + p_{10}^A b_{10}(0, v) \right] + \left[ p_4^A b_4(0, v) + p_9^A b_9(0, v) \right] + \left[ p_5^A b_5(0, v) + p_8^A b_8(0, v) \right] + \left[ p_6^A b_6(0, v) + p_7^A b_7(0, v) \right] = S_A(0, 1 - v)
\]
In this way, bitwise identical points on the shared boundary are obtained. The partial derivatives and normal can be processed similarly, under the IEEE floating point strictness.

**Corner Case.** To evaluate a corner vertex, its 1-ring neighbour vertices will be used. To compute limit vertices with its 1-ring neighbour vertices sequentially, we identify the neighbour vertices with an extra *System-Value Semantics* in HLSL, SV_VertexID. In our implementation, we always evaluate the limit corner vertex in each patch sharing it, counterclockwise starting from the same vertex with the smallest(or largest) SV_VertexID.

### 3.6.2 Different Subdivision Levels

![Illustration of watertightness between two adjacent patches of different levels of subdivision.](image)

Abbildung 9: Illustration of watertightness between two adjacent patches of different levels of subdivision.

Special attention must be paid to boundary edges whose adjacent patches are of different level of subdivision. To achieve watertight tessellation, we pack the 12 control points of "small"patch together with the 12 control points of its parent patch, and pass them all to the domain shader, which can still be handled efficiently in the tessellation shader. For example in Figure 9, patch B is the parent patch of patch B₀ and patch B₁, we pack each 12 control points of B₀ and B together for rendering patch B₀. When evaluating points on the boundary or corner points of B₀, we use the control points of B, and we use the control points of B₀ inside the patch.

The watertight evaluation can produce high quality tessellation results. However, it is a time-consuming operation, since it involves some code branches, which limits the performance. Thus, the watertight rendering is regarded as an option.

### 3.7 ADAPTIVE LEVEL-OF-DETAIL

By default, a global F_tess is specified by user, and the maximum subdivision level is determined as \(\lceil \log_2 F_{tess} \rceil \), where \(\lceil x \rceil\) is the smallest integer not smaller than \(x\). In each subdivision level, the tessellation factor is half its previous level. As a result, in depth \(\lceil \log_2 F_{tess} \rceil\), the tessellation factor is 1.0.
The framework of our algorithm is capable of being adapted to the level-of-detail (LOD) applications. Two factors can be adjusted to balance rendering quality and cost.

**Adaptive tessellation factors.** Tessellation factors can be adaptively determined on-the-fly according to user defined metrics. For each triangular patch, there are three edge tessellation factors and one inside tessellation factor. Of course, the edge tessellation factors of a shared edge in two adjacent patches have to be the same to eliminate potential T-junction.

The tessellation factors can be assigned globally for the entire mesh or locally for edges or patches. In our implementation, the global tessellation is determined according to the distance between the camera and the centroid of the control mesh, while the local tessellation factor is determined according to the edge length projected on the screen. Obviously local tessellation factors can produce better LOD effects, at the cost of extra calculation time to determine these factors.

**Adaptive max subdivision depth.** The maximum subdivision depth can also be determined adaptively according to the distance between the camera and centroid of the control mesh. In this case, the maximum subdivision is independent of the tessellation factor. The patches in the maximum subdivision level may have tessellation factor over 1.0. To tessellate these patches well, a patching approximation method should be employed [LRZM11, BS07].

### 4 RESULTS

The proposed algorithm has been implemented on a PC with 2.80GHz Intel Core i5-2300 CPU, 4G Memory and NVIDIA GeForce GTX 570 GPU. The OS is Windows 7 and GPU API are DirectX 11. The GPU code is written in HLSL. In all the examples, different colors indicate different subdivision levels, as shown in Figure 11a.

#### 4.1 Various Implementation Results

In semi-sharp case, the subdivision table generated in each subdivision level records required sharpness attributes, and reinforced kernels which are designed to evaluate semi-sharp vertex points and edge points. Two semi-sharp examples with cubical control meshes are shown in Figure 10.

In our implementation of the displacement mapping, Loop subdivision is adapted as the base mesh. An example is shown in Figure 12.

In hierarchical editing case, we generate subdivision tables and adaptively subdivide around hierarchical details. Examples are shown in Figure 13.

#### 4.2 Implementation Statistics

The timing statistics of the proposed algorithm include three parts: CPU subdivision table generation time, GPU parallel computing time and GPU rendering
Abbildung 10: Semi-sharp creases are tagged as bold lines. (left) control mesh; (middle) sharp edges have a sharpness of 3.5; (right) sharp edges have a sharpness of 10.

The detailed statistics are given in Table 3 and Table 4. From Table 4, we can infer that the proposed algorithm is suitable for realtime or interactive application.

GPU memory needs to be allocated for subdivision tables and vertex buffers for all vertices position of patches of all subdivision levels. Figure 14 shows the GPU memory consumption for some models.

4.3 Comparison with the Approximate Patching Algorithm and Direct Evaluation Algorithm

As shown in Figure 15, it is obvious that the artifacts occur around the extraordinary vertices in Li’s approximate approach, while our results are exact.

The runtime comparison among our method, Li’s method and Stam’s method are given in Figure 16. It shows that the proposed algorithm is more efficient.

5 CONCLUSION AND FUTURE WORK

We presented a novel method of GPU-based efficiently rendering true limit Loop subdivision surfaces. Triangular control meshes of arbitrary topology can be adopted as input. Our algorithm can also handle semi-sharp creases. We also presented an elaborate solution to guarantee the watertightness of the surfaces,
Abbildung 11: Results of our method applied to the bigguy and monster frog. (a) the color table of different subdivision level; (b)-(c) are the control meshes of bigguy and monster Frog; (d)-(e) are the feature adaptive results of (b) and (c).

Tabelle 3: CPU Subdivision Table generation time(ms) of each subdivision level performed on big guy, monster frog, and head.

<table>
<thead>
<tr>
<th>Subdivision Level</th>
<th>BigGuy</th>
<th>MonsterFrog</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>56.64</td>
<td>48.82</td>
<td>11.71</td>
</tr>
<tr>
<td>2</td>
<td>199.21</td>
<td>138.61</td>
<td>33.20</td>
</tr>
<tr>
<td>3</td>
<td>500.00</td>
<td>250.00</td>
<td>70.31</td>
</tr>
<tr>
<td>4</td>
<td>605.46</td>
<td>287.10</td>
<td>83.98</td>
</tr>
<tr>
<td>5</td>
<td>603.51</td>
<td>285.15</td>
<td>83.98</td>
</tr>
</tbody>
</table>
allowing for crack-free displacement mapping. Implementation results show that the proposed feature adaptive method outperforms Li’s approximate patching algorithm and Stam’s exact evaluation, and can achieve accurately realtime rendering Loop subdivision surface.

Literatur

Tabelle 4: GPU Timing Statics including the number of generated vertices (NV), GPU parallel computing time (DCT) (ms), the number of control patches (NCP), the number of final rendering triangle primitives (NP), and GPU rendering time (RT) (ms) of different tessellation factors (TF) are collected for the big guy, monster frog and head with our algorithm.

<table>
<thead>
<tr>
<th>TF</th>
<th>NV</th>
<th>DCT</th>
<th>NCP</th>
<th>NP</th>
<th>RT</th>
<th>NV</th>
<th>DCT</th>
<th>NCP</th>
<th>NP</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0</td>
<td>0</td>
<td>3k</td>
<td>3k</td>
<td>0.23</td>
<td>0</td>
<td>0</td>
<td>2k</td>
<td>2k</td>
<td>0.19</td>
</tr>
<tr>
<td>2.0</td>
<td>5k</td>
<td>1.68</td>
<td>10k</td>
<td>18k</td>
<td>0.32</td>
<td>4k</td>
<td>0.98</td>
<td>7k</td>
<td>23k</td>
<td>0.28</td>
</tr>
<tr>
<td>4.0</td>
<td>18k</td>
<td>3.71</td>
<td>28k</td>
<td>125k</td>
<td>0.64</td>
<td>11k</td>
<td>1.86</td>
<td>12k</td>
<td>125k</td>
<td>0.46</td>
</tr>
<tr>
<td>8.0</td>
<td>35k</td>
<td>7.87</td>
<td>46k</td>
<td>601k</td>
<td>1.12</td>
<td>16k</td>
<td>3.76</td>
<td>24k</td>
<td>486k</td>
<td>0.83</td>
</tr>
<tr>
<td>16.0</td>
<td>36k</td>
<td>10.30</td>
<td>64k</td>
<td>2109k</td>
<td>2.89</td>
<td>17k</td>
<td>5.78</td>
<td>33k</td>
<td>1707k</td>
<td>1.99</td>
</tr>
<tr>
<td>32.0</td>
<td>36k</td>
<td>14.62</td>
<td>82k</td>
<td>7480k</td>
<td>7.34</td>
<td>17k</td>
<td>7.80</td>
<td>41k</td>
<td>6187k</td>
<td>5.60</td>
</tr>
</tbody>
</table>

Abbildung 14: Memory requirement to store vertex buffers and subdivision tables for three models.


Abbildung 15: Comparison between Li’s approximate patching algorithm (a) and ours (b) on the head model.

Abbildung 16: Rendering time comparison of our method with Stam’s and Li’s on the big guy model.


