Dynamic Feature-Adaptive Subdivision

H. Schäfer[1] J. Raab[1] B. Keinert[1] M. Meyer[2] M. Stamminger[1] M. Nießner[3]
[1]University of Erlangen-Nuremberg [2]Pixar Animation Studios [3]Stanford University

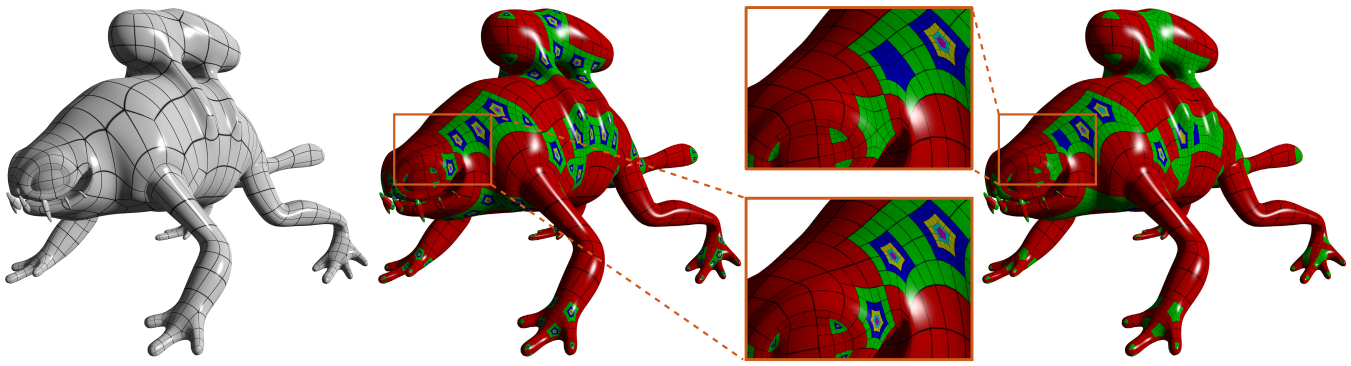

Figure 1: *Rendering of the Frog model (left), using feature-adaptive subdivision [Nießner et al. 2012a] takes 0.68ms (middle); our method only takes 0.36ms by performing locally adaptive subdivision (right); colors denote different subdivision levels.*

Abstract

Feature-adaptive subdivision (**FAS**) is one of the state-of-the art real-time rendering methods for subdivision surfaces on modern GPUs. It enables efficient and accurate rendering of subdivision surfaces in many interactive applications, such as video games or authoring tools. In this paper, we present dynamic feature-adaptive subdivision (**DFAS**), which improves upon FAS by enabling an independent subdivision depth for every irregularity. Our subdivision kernels fill a dynamic patch buffer on-the-fly with the appropriate number of patches corresponding to the chosen level-of-detail scheme. By reducing the number of generated and processed patches, DFAS significantly improves upon the performance of static FAS.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations

Keywords: subdivision surfaces, rendering

1 Introduction

Subdivision surfaces [Catmull and Clark 1978; Doo and Sabin 1978; Loop 1987] have been the standard for representing three-dimensional content in the movie industry for many years. While subdivision surfaces provide significant advantages in surface quality, modeling, and animation, their evaluation is computationally expensive. This typically restricts their use to offline rendering methods such as those used in feature films, e.g., [Pixar Animation Studios 2005]. However, the rapid rise of massively parallel graphics hardware in recent years has opened up new possibilities for accurate rendering of subdivision surfaces in interactive scenarios. Feature-adaptive subdivision (**FAS**) [Nießner et al. 2012a] is a method for efficiently rendering subdivision surfaces with high quality features such as creases and irregular vertices (i.e., valence $\neq 4$) without approximation. This makes it ideal for content creation tools (e.g., Maya) and even video games (e.g., Call of Duty Ghosts). As a result, feature-adaptive subdivision has emerged as a state-of-the-art real-time rendering method for subdivision surfaces on modern GPUs, and is the basis for Pixar's industry standard *OpenSubdiv*[1] platform.

Feature-adaptive subdivision adaptively subdivides around irregularities only where needed to produce regular patches, and leverages hardware tessellation to process these regular patches. The key to maintaining high performance is to generate as few patches as possible while still accurately representing the surface. Although the adaptivity greatly reduces the number of patches needed, the amount of patches generated around irregularities is still a performance bottleneck. The most significant problem is that feature-adaptive subdivision restricts each model to a single adaptive subdivision level around all irregular vertices. That is, every irregular patch of a model is adaptively subdivided to the same subdivision depth. This can cause tremendous overhead when distant regions are forced to subdivide to the same depth as nearby regions – resulting in an unnecessarily large amount of patches as well as over-tessellation.

Contributions In this paper, we present dynamic feature-adaptive subdivision (**DFAS**), enabling an independent subdivision level for each irregular vertex, thus greatly reducing the number of patches required. Our method uses GPU compute kernels to dynamically identify and subdivide the patches around an irregular vertex if necessary. In addition, unified patch buffers naturally allow us to minimize overhead by processing patches on all subdivision levels together, thereby reducing the amount of separate render calls. Overall, our easy-to-implement method significantly improves upon the performance of the feature-adaptive subdivision rendering method, as shown in timings of our test scenes.

In the remainder of this paper, we refer to the original method [Nießner et al. 2012a] by feature-adaptive subdivision (**FAS**), and to our new method by dynamic feature-adaptive subdivision (**DFAS**).

[1] `http://graphics.pixar.com/opensubdiv/`

2 Previous Work

Subdivision Surfaces Subdivision surfaces were originally introduced by Catmull and Clark [1978] as a generalization of bicubic B-Splines. The core idea is that repeated refinement of a control mesh with arbitrary connectivity according to a set of subdivision rules will result in a smooth surface. Subsequently, many subdivision schemes have appeared, with each scheme providing different surface properties regarding continuity and primitive setup [Loop 1987; Doo and Sabin 1978]. While we present our method on the basis of Catmull-Clark subdivision surfaces, our ideas extend easily to other schemes.

Global Refinement using GPGPU With the rapid development of massively parallel graphics hardware in the last decade, the evaluation and rendering of curved surfaces has become feasible for real-time applications. While early approaches use the graphics pipeline to evaluate parametric surfaces [Vlachos et al. 2001; Shiue et al. 2005; Bunnell 2005], the introduction of general purpose programming [Nvidia 2007] (GPGPU) opened up new possibilities. As the parallel architecture of GPUs directly maps to the subdivision rules, attention turned to the problem of crack-free view-dependent adaptive surface rendering [Eisenacher et al. 2009; Schwarz and Stamminger 2009; Patney et al. 2009; Fisher et al. 2009]. These methods use a two-pass approach per frame: 1) a GPGPU kernel writes out refined vertex data to global GPU memory 2) the graphics pipeline loads the data back into local streaming multiprocessor memory where it is processed for rasterization. As the GPGPU refinement is densely refining the control mesh, this results in significant memory I/O between geometry generation and primitive processing, severely limiting the performance of modern GPUs.

Patch Evaluation using Hardware Tessellation Hardware tessellation [Moreton 2001; Andrews and Baker 2006; Microsoft Corporation 2009] eliminates the two-pass problem of traditional GPGPU methods by evaluating and rasterizing the surface geometry directly on-chip, thus minimizing global memory I/O. That is, meshes are interpreted as a set of parametric patches, with each patch defined by a fixed number of control points [Schäfer et al. 2014]. The key requirement is to have a closed-form solution in order to evaluate each patch at an arbitrary uv domain location. While bi-cubic Bézier and B-Spline patches fit into this paradigm, the direct evaluation of subdivision surfaces around extraordinary patches is unfortunately quite challenging. Direct evaluation methods exist [Stam 1998; Bolz and Schröder 2002], but are relatively costly in practice.

Patching Subdivision Surfaces For efficient patch evaluation, many methods approximate subdivision surfaces [Loop and Schaefer 2008; Myles et al. 2008b; Ni et al. 2008; Myles et al. 2008a; Ni et al. 2009; Loop et al. 2009]. While these approaches are ideally suited to real-time processing using hardware tessellation, the generated surfaces differ from the original surface definition. In many applications, such as authoring tools for feature films, this is not acceptable. In addition, the surface properties of these approximations are problematic for some scenarios, e.g., in the context of displacement mapping [Nießner and Loop 2013].

Feature-adaptive subdivision [Nießner et al. 2012a] avoids approximation and efficiently handles features such as semi-sharp creases [DeRose et al. 1998; Nießner et al. 2012b]. It achieves the same performance as approximate schemes while producing accurate results. The key idea is to apply subdivision only in regions where direct evaluation is costly or infeasible, and process all regular patches using the hardware tessellator. Unfortunately, the number of adaptive subdivision levels is static for each mesh, which significantly limits the quality of level-of-detail. Note that FAS relies on the execution of GPGPU kernels; however, in contrast to global refinement methods, the amount of transfered memory to the graphics pipeline is minimal due to the adaptive nature of the method.

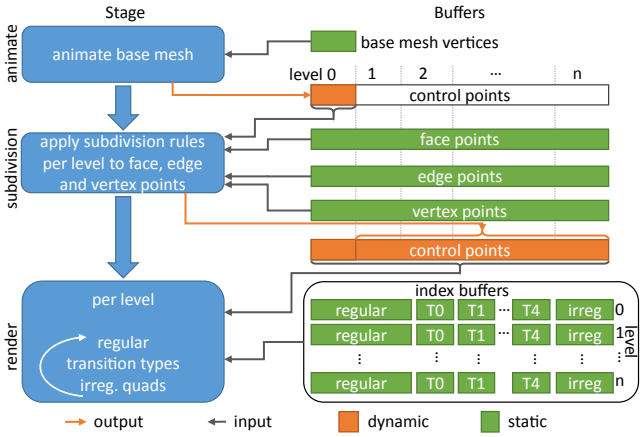

Figure 2: *Overview of the original feature-adaptive subdivision algorithm; processing stages are shown on the left and data structures on the right.*

3 Feature-Adaptive Subdivision

As our algorithm builds and extends upon the feature-adaptive subdivision method [Nießner et al. 2012a], we provide an overview of the original method. We refer to [Nießner 2013] for additional implementation details. Feature-adaptive subdivision consists of the following steps (see Fig. 2):

Preprocess This step analyzes the connectivity and features of the mesh, and determines where to adaptively subdivide in order to produce patches ready for tessellation. This analysis gives the patches which need to be tessellated as well as the computation of the control points defining these patches. Patch data is stored in *index buffers* describing the type of the patch as well as the indices of all control points needed to define the patch. *Subdivision tables* store all of the data necessary to compute one subdivided point via a subdivision rule (indices of all points used by the subdivision rule, valence, sharpnesses, etc.). These tables are separated into three types according to the rule they describe: face, edge, or vertex.

In order to deal with the T-junctions produced by adaptive subdivision, FAS produces *transition patches* which connect patches at differing levels of subdivision. These transition patches are simply regular patches whose parametric domain is triangulated so that the tessellation stage can produce watertight meshes in areas of differing subdivision levels. Due to symmetries, this results in five different template transition patches (see Fig. 3).

Adaptive Subdivision This stage iteratively runs subdivision kernels at runtime – for each level – to adaptively subdivide a mesh in order to generate the patch control points. At each level, the control points and the subdivision tables are used to compute the control points of the next finer level, following the subdivision rules encoded in the table entries. This step fills in a pre-allocated control point buffer that was created in the preprocess for a maximum subdivision level. The subdivision level can be adjusted at runtime; however, there is a single subdivision level for each mesh.

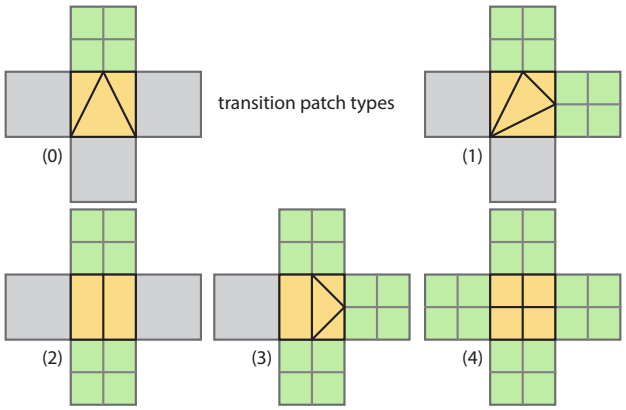

Figure 3: *Transition patch types of feature-adaptive subdivision. These patches are used to bridge transitions between different subdivision levels where T-junctions would cause cracks otherwise. Transition patches are colored yellow, regular patches of the coarser subdivision level gray, and those of the next finer level green.*

Patch Tessellation This stage uses the patch control points computed in the previous stage along with the index buffers created in the preprocess to produce the tessellated patches. For each level of subdivision, each patch type (regular, transition types, and irregular) describes all of its patches (control point buffer, control point indices, tessellation factor) to the GPU tessellation unit. The patches are then densely subdivided by the tessellator, and generated triangles are directly sent to the rasterization units.

One of the key benefits using GPU hardware tessellation is the ability to assign a dynamic tess factor to each patch. Feature-adaptive subdivision supports the assignment of tess factors t on the base patch level to control the local tessellation rate. For adaptively subdivided patches, a tess factor $\hat{t}$ is automatically computed according to the patch level d: $\hat{t} = \max(1,\ t/2^d)$. As irregular patches cannot be further tessellated, their tess factor must be exactly 1. To this end, the adaptive subdivision depth of a mesh is set according to the maximum base patch tess factor: depth $= \max_i \lceil \log t_i \rceil$. Unfortunately, a fixed subdivision level among all irregular patches of a mesh introduces significant over-tessellation in regions where the base patch tess factors are small. In our method, we remove this restriction, allowing a dynamic and independent subdivision depth for each irregular patch.

4 Dynamic FAS: Algorithm Overview

Our algorithm follows the table-driven approach of [Nießner et al. 2012a], which statically pre-computes subdivision tables up to a predefined maximum subdivision level (see Sect. 5). In contrast to static table-driven subdivision, the novelty of our algorithm is to dynamically adapt the subdivision depth around each irregular vertex independently, thus reducing the amount of computation and the number of processed patches at runtime. An overview of our method is shown in Fig. 6.

In every frame, the first step of our approach is a *metric* kernel running on the GPU, which computes the tessellation density for each base patch (regular and irregular), following a user-defined level-of-detail scheme (see Sect. 6.1). For regular patches of the base mesh, this directly defines the tess factor t used by the hardware tessellator for rendering. In the case of an irregular patch, the subdivision depth d around a vertex v is set accordingly: $d_v = \lceil \log t \rceil$.

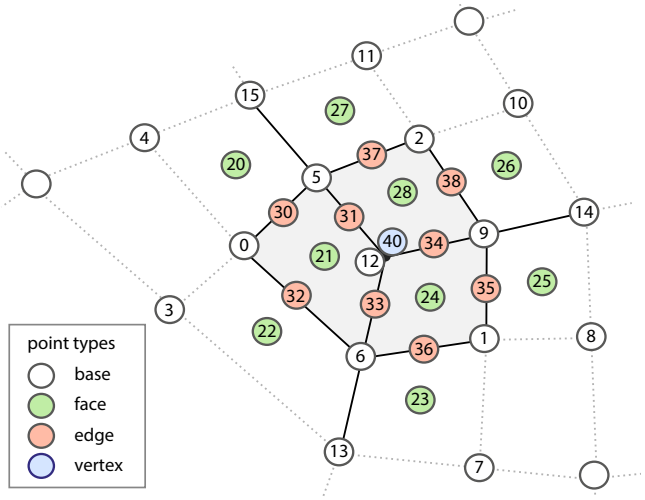

Figure 4: *Simplified characteristic map for an irregular vertex with a valence of three. White points show the isolated base mesh for this characteristic map. Colored points refer to face points (red), edge points (green), and vertex points (blue), according to the Catmull Clark subdivision rules.*

Since a patch may share multiple irregular vertices, it is possible that the subdivision levels for two vertices of a patch disagree. To avoid this situation, we isolate irregular vertices by a single static feature-adaptive subdivision step if necessary.

The next step is the actual dynamic adaptive subdivision using a set of GPU compute kernels (see Sect. 6.2). While we have precomputed subdivision tables for all potential levels, we only need to access those corresponding to the levels d_v with $v \in \mathit{irregular}$. The output buffer has space allocated for control points until the maximum subdivision depth; however, data is only generated for a subset of control points depending on the local dynamic subdivision depth defined by the *metric* kernel. In addition, we fill a dynamic patch buffer, which describes locations of the generated sub-patches in a precomputed index buffer.

Finally, we send all regular patches – from the base mesh and those generated by the subdivision kernels – to the hardware tessellator for rendering. T-junctions between different subdivision levels are stitched using the concept of transition patches as introduced by [Nießner et al. 2012a]. In contrast to FAS, we render all patches of the same type (including different levels) in a single render pass, thus minimizing the number of draw calls.

5 Data Structure Generation

As in the original FAS algorithm, we pre-compute data structures describing the adaptive subdivision of an input mesh up to a maximum subdivision level. However, to support dynamic adaptation at each irregularity, we must augment our data structures. In the following, we describe these augmentations: Sect. 5.1 describes extended subdivision tables which are required to execute the subdivision; patch tables, identifying patch indices, are introduced in Sect. 5.2.

5.1 Extended Subdivision Tables

Similar to FAS, we encode the Catmull-Clark [1978] subdivision rules for face, edge, and vertex points in pre-computed subdivision tables. To this end, we store the linear combination for each control

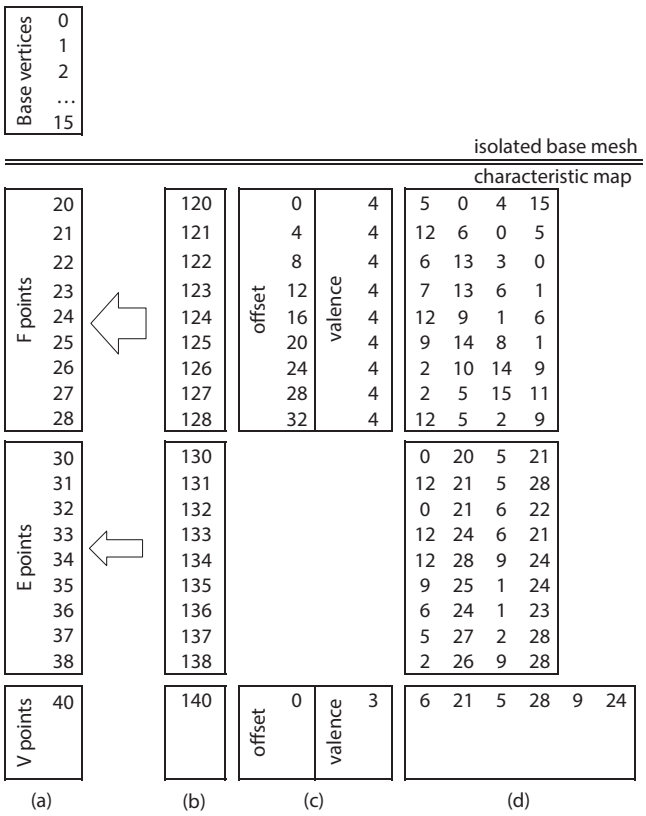

Figure 5: *Extended subdivision tables for the configuration shown in Fig. 4: (a) IDs of the vertices, (b) offsets to the vertex buffer for writing the updated positions, (c) topology information, and (d) indices of vertices needed as input for the subdivision.*

point type with respect to the corresponding parent patch control points. As our subdivision is locally dynamic, we structure the subdivision tables according to the characteristic maps of extraordinary vertices, see Fig. 4 for an example characteristic map. Thus, the 1-ring table entries of all levels of a non-valence four vertex are clustered together. Note that tables are pre-computed up the maximum possible subdivision depth; however, only the entries corresponding to the local subdivision depth are used (see Sect. 6.1).

In contrast to the original FAS method, only a subset of the points in the global control point buffer is updated at runtime by the subdivision kernels. Hence, there is no implicit mapping from the kernel threadID to the output location (see Sect. 6.2). To this end, each subdivision table row stores the target address to the global control point buffer; i.e., the ID of the control point the row generates (see Fig. 5(b)).

5.2 Patch Tables

Once we have computed the subdivision depth for each extraordinary vertex (see Sect. 6.1), we also need to identify associated patches. To this end, we pre-compute patch tables which map a given vertex v with a subdivision depth d_v to a set of patches. For each level of an extraordinary vertex, the patch table refers to four patches (see Fig. 8 and Sect. 6.1). For instance, if $d_v = 0$, we need to render one regular, one irregular, and two transition patches. We also pre-compute a static global index buffer for all possible patch types and final patches for all levels. A final patch is a regular patch on the finest subdivision level which would have been a transition

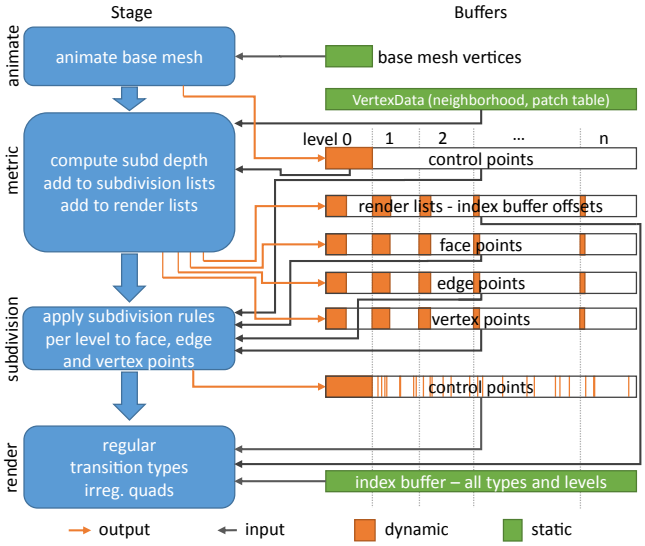

Figure 6: *Overview of our dynamic feature-adaptive subdivision: the metric kernel is run per vertex; it determines the local subdivision depth and marks control points needed for subdivision in the point lists. For rendering the patches of each level, offsets to a static pre-computed index buffer are appended to a render list. Then, the sparse set of required control vertices is updated in the subdivision kernels. Finally, the patches are rendered using the updated control points and index buffer offsets from the metric kernel.*

patch otherwise; see Fig. 7 for the memory layout. The patch table entries are all base offsets to this index buffer. For patch rendering (see Sect. 6.3), these indices specify the corresponding control points in the global vertex buffer.

6 Dynamic Subdivision

Our algorithm allows each irregular vertex to choose an independent subdivision level at runtime. First, we estimate the subdivision depth for irregular patches. Then, the subdivision rules are applied to dynamically refine these patches, and generate the corresponding control points. Finally, we need to render all regular patches of the base mesh as well as the newly generated patches. An overview of our pipeline is depicted in Figure 6. In the following, we describe the technical details of three main stages of our algorithm.

6.1 Metric Stage

For every frame, we need to perform adaptive subdivision on patches with extraordinary vertices before these patches can be processed by the hardware tessellator. The amount of adaptive subdivision around a specific irregular vertex is dependent on the local patch tessellation density. To this end, we determine a tessellation density t for each patch at the beginning of every frame. We have chosen to project the control points of the base mesh into screen space and measure edge lengths in pixels – though our algorithm is independent of the specific level-of-detail metric. For all regular base patches, we can simply assign these pixel measures as edge tess factors t_e. The inner tess factors of patches are computed according to the maximum length of adjacent edges. In the case of irregular patches, we need to perform adaptive subdivision before patches can be processed by the hardware tessellator. We determine the local subdivision depth d_v around a vertex v based on the surrounding edges e_i: $d_v = \text{argmax}_i \lceil \log t(e_i) \rceil$.

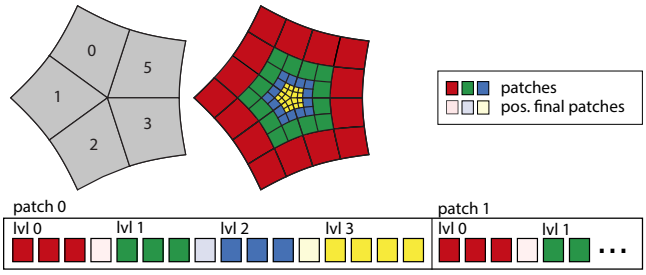

Figure 7: *Memory layout of the patch table buffer with three subdivision levels. We store all subdivision tables and render patches around an extraordinary vertex sorted by level and order within each patch. This includes a final patch at each level for rendering when no further subdivision is necessary.*

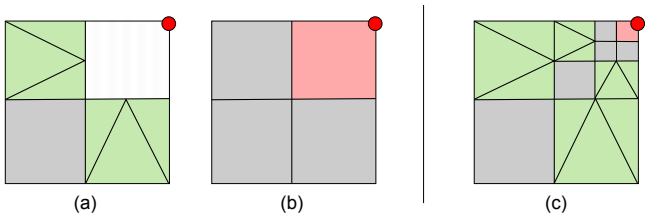

Figure 8: *Patches around an extraordinary vertex (red dot) are sorted into render lists by the metric kernel: regular patches (grey), transition patches (green), and patch-filling quads (red). (a) one regular and two transition patches are emitted for levels $l \in [0; d_v - 1]$ and the irregular patch is further subdivided (top right), (b) on the final level $l = d_v$, three regular patches and one patch-filling quad is generated. (c) the patch layout at an example subdivision depth $d_v = 3$.*

Both patch tessellation densities and local subdivision depths are computed by a GPU compute program, which we call the *metric* kernel. The inputs to this kernel are the control points – i.e., the (animated) base mesh vertices – and the extended subdivision tables encoding the subdivision rules around extraordinary vertices. As a result, we obtain patch tess factors, a subdivision control point list, and a patch list for rendering. These lists are synchronized using an atomic counter, which we found to be faster than a prefix sum compaction. Pseudocode of the metric kernel for extraordinary vertices is shown in Listing 1.

Note that our tess factor computation is not necessarily optimal in terms of the pixel size of generated triangles. However, our method is completely orthogonal to the used level-of-detail scheme, as we abstract the tessellation density computation away from mesh irregularities. From an API and user perspective, this enables the use of arbitrary tessellation metrics, such as those proposed by [Yeo et al. 2012] or [Nießner and Loop 2013], which can be efficiently computed on a per base patch level.

Subdivision Lists In addition to identifying the tessellation densities, the metric kernel identifies control points that need to be updated by adaptive subdivision. To this end, the metric kernel appends selected subdivision table entries to linked lists, which we call *subdivision lists*. Each entry corresponds to the characteristic map of a vertex tagged for subdivision. The lists are structured by the subdivision point type; i.e., face-, edge-, vertex points (see Fig. 5). We obtain a linked list of subdivision table entries for every point type per subdivision level. The list entries are simple copies of the subdivision tables, but dynamically appended according to the d_v's. For every list, we maintain an atomic counter to manage its size. The counters are also used to launch the subdivision kernels with the correct number of threads (see Sect. 6.2).

Patch Render Lists The metric kernel also generates the patch *render lists*, where each entry corresponds to a patch to be rendered. Overall, we have three different render lists: one for regular, one for transition, and one for irregular patches. Every list entry corresponds to a specific base or sub-patch storing its level l, the level-corrected patch tess factor t, and an offset referring to the control point index buffer. The control point index buffer is static and keeps indices for patch rendering; i.e., every possible patch knows its indices and thus the control point locations (see Sect. 6.3). As shown in Fig. 8, for every level $l \in [0; d_v - 1]$, one regular patch and two transition patches are generated, respectively. At the final level $l = d_v$, one patch-filling quad and three regular patches are generated. Transition and regular patches are processed by the tessellator whereas patch-filling quads are rendered by standard triangle rasterization; cf. [Nießner et al. 2012a].

```
// one thread per vertex
metric_kernel(uint Tid /*thread ID*/) {
  //neighborhood around the vertex
  VertexData& v = getVertexData(Tid);

  //determine tessellation density and subd depth
  computeSubdDepth(v); //t_e_i and d_v

  if (v.valence == 4) { //regular patch
    //add to renderlist; no subd required
    append_regular_points(v);
  } else {
    for(uint patch=0; patch < v.valence; patch++) {
      for (uint level=0; level < v.d_v; level++) {
        // add points to subdivision lists
        append_face_points(v,patch,level);
        append_edge_points(v,patch,level);
        append_vert_points(v,patch,level);

        // add patches to renderlists
        if (level < v.d_v-1) { //all but the last
          // 2 transition patches
          append_transit_patches(v,patch,level);
          // 1 regular patch
          append_regular_patches(v,patch,level);
        } else { //final dynamic level
          // 3 regular patches on last level
          append_regular_patches(v,patch,level);
          // 1 final patch-filling quad
          append_irregular_patch(v,patch,level);
        }
      }
    }
  }
};
```

Listing 1: *Pseudocode of the metric kernel.*

6.2 Subdivision Stage

Once the metric kernel has identified all patches that need to be subdivided – i.e., generated the subdivision lists – we compute the corresponding control points. Following the Catmull-Clark [1978] subdivision rules, we run three different GPU compute kernels processing face, edge, and vertex points (see Fig. 4). While we apply table-driven subdivision as proposed by Nießner et al. [2012a], we only subdivide according to the local subdivision depth. We iteratively process the subdivision levels, running the three kernels at each level, where each thread generates a single output control point. Overall, we run $(3 \cdot \#\text{levels})$ kernels, with each kernel processing one of the subdivision lists. The control point buffer is shared by all kernels and has space allocated for all potential levels; however, we only update a subset of the points up to a dynamically-

computed subdivision depth. In order to keep host-device interaction at a minimum, we use indirect dispatches to run the kernels.

At every subdivision level, we launch threads based on the sizes of the subdivision lists generated by the metric kernel (see Sect. 6.1). List entries, which contain a copy of the corresponding subdivision table rows, are then indexed by the kernel threadIDs. Each thread then computes its associated control point by obtaining the linear combination of the parent control points and applying the respective subdivision rule. The result is written back to the global control point buffer at the location corresponding to the output index stored in the subdivision table.

Since the subdivision tables are structured by the characteristic map of irregular vertices, it is required that there is at most one irregularity within the 1-ring of a vertex; i.e., there must be no overlap in the tables. Therefore, we perform a static feature-adaptive subdivision step to isolate the irregular vertices if required by the connectivity of the mesh. This also removes the issue of irregular patches potentially containing multiple irregular vertices which disagree in their subdivision depth d_v. If this subdivision on the first level is required, we employ a special subdivision table and follow the original static table-driven subdivision scheme [Nießner et al. 2012a]. Note that for meshes which already have isolated extraordinaries and for all levels above one, subdivision is always locally dynamic.

6.3 Patch Rendering

The metric stage defines the locally adaptive subdivision depth around extraordinary vertices and the subdivision kernels compute the corresponding control point data. The final step is the rendering of regular and adaptively subdivided patches. We process the dynamically generated patch render lists (see Sect. 6.1) and issue indirect draw calls for each list. As patches of all levels of the same type – regular, transition, or irregular – are contained in a single list, we only need a few draw calls. The tessellation factor t for any edge is determined on a per frame basis with the same screen space metric that we use in metric kernel (see Sect. 6.1). This differs from the original feature-adaptive subdivision [Nießner et al. 2012b], where draw calls are issued for each level independently. In particular, for scenes with many small models, fewer draw calls increase performance, as the GPU occupancy is higher.

In addition, draw calls are not directly dispatched with a vertex and index buffer. Instead, in a shader program, we access the patch render lists using the pipeline-generated primitiveID values. The patch render lists store base offsets to a static index buffer which contains corresponding indices into the global control point buffer. Compared to traditional indexed draw calls, this introduces another indirection; however, we found this to be much more efficient than the dynamic generation of an index buffer.

Regular Patches Regular patches are the simplest case; the render list of regular patches comprises regular patches of the base mesh as well as regular patches of higher levels (see Fig. 4). They can be directly processed by the hardware tessellator, as they are bi-cubic B-Splines defined by 16 control points each. The render lists store the base offset to the static index buffer and patch-corrected tess factors for further tessellation. The static index buffer is precomputed once at model loading time and stores 16 indices for each regular patch. As mentioned above, the indices reference the actual control points which are stored in the global control point buffer. Once control points are obtained and the tess factors are set (hull shader), regular patches are directly evaluated using the B-Spline basis functions (domain shader). Note that the computation of the subdivision depth d_v guarantees that the tess factor of patches at the finest level is always equal to 1.

Transition Patches Transition patches are analytically defined as regular patches; i.e., they are bi-cubic B-Splines with 16 control points each. However, the parameter domain of transition patches is split into several sub-patches in order to bridge the T-junctions between adjacent subdivision levels for crack-free rendering. On the first subdivision level, there are five possible domain splits of transition patches depending on the mesh connectivity. In the case of isolated extraordinary vertices as well as after the first subdivision level, there is only a single domain split; for details see [Nießner et al. 2012a]. We require a render pass for every domain split; however, we process all subdivision levels together. Aside from the domain split, the patch evaluation and rendering with the hardware tessellator is the same as with regular patches.

Patch-filling Quads Even at the finest level, any patches adjacent to an irregular vertex are still irregular. The key idea of feature-adaptive subdivision is reduce the size of these surface pieces such that no further tessellation is required; i.e., a tess factor of 1.0. While the direct evaluation of irregular patches at arbitrary parameter locations is challenging, it is relatively straightforward to compute the limit positions of the control points themselves. This allows us to render irregular patches on the finest level simply as quads. We apply the limit stencil masks of Halstead et al. [1993] to the vertices in a shader program employing the neighborhood information of the subdivision tables. We refer to the original feature-adaptive subdivision method since the rendering of patch-filling quads is unchanged, despite that the finest levels are now locally different within a mesh.

7 Results

We have implemented our dynamic feature-adaptive subdivision method in DirectX running on an Nvidia GTX 980 under Windows 8. The metric and subdivision kernels are realized in Direct Compute and rendered using the Direct3D 11 graphics pipeline with hardware tessellation. All measurements are provided in milliseconds rendering at a 720p resolution.

Example Scenes We run our dynamic subdivision method on the models visualized in Fig. 9 and the terrain surface shown in Fig. 10. The models and terrain are characterized by different numbers of base mesh patches, the ratio between regular and irregular patch configurations and the distribution of irregular vertices. The terrain scene consists of 17136 input patches of which 2248 are adjacent to irregular vertices. The data for the other test models is shown in

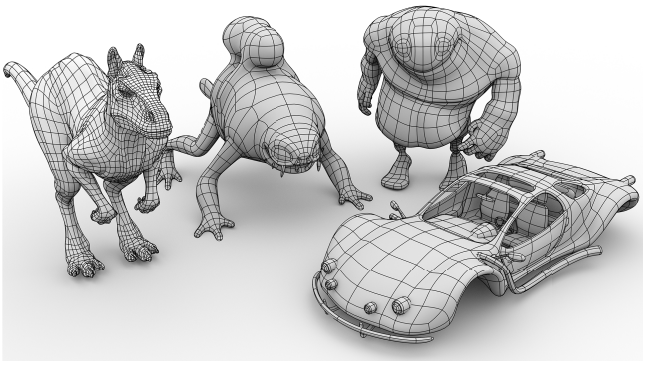

Figure 9: *Visualization of a set of test models used in the performance comparison between static and dynamic feature adaptive subdivision shown in Table 1.*

Model	Killeroo		Frog		Bigguy		Car	
base patches	3762		1292		1450		1992	
irregular patches	1330		764		592		948	
irregular vertices	425		292		196		538	
method	FAS	DFAS	FAS	DFAS	FAS	DFAS	FAS	DFAS
draw	0.764	0.416	0.508	0.261	0.443	0.290	0.671	0.332
subdivision	0.189	0.154	0.160	0.105	0.142	0.099	0.181	0.102
total	0.953	0.570	0.668	0.366	0.585	0.389	0.852	0.434

Table 1: *Performance comparison between static (FAS) and dynamic (DFAS) feature-adaptive subdivision in terms of subdivision and total processing time including rendering on the set of test models shown in Fig. 9. The subdivision time of our dynamic feature-adaptive subdivision algorithm includes the time for executing the metric kernel.*

Table 1. We also refer to the accompanying video.

Performance Measurements In Fig. 11, we compare our dynamic algorithm to the static method in terms of subdivision and total rendering performance on the terrain scene shown in Fig. 10. The graph shows a performance plot corresponding to a camera flight through the valleys of the terrain (see the accompanying video). We configured the subdivision and tessellation metric such that the generated triangles each roughly cover an area of 4x4 pixels. In contrast to original FAS, we are able to produce a uniform tessellation density over the entire mesh, whereas FAS over-tessellates around irregularities since it cannot adapt the subdivision depth locally. Finally, we provide a quantitative evaluation for our test models in Table 1. Note that the subdivision timings of our dynamic algorithm includes the execution of the metric kernel. Despite the additional kernel for managing local adaptivity, we are able to significantly improve both subdivision and rendering performance.

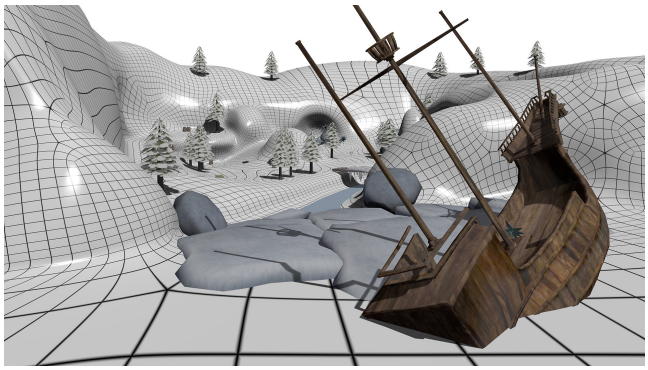

Figure 10: *Rendering of a terrain scene consisting of 17k input patches showing the base mesh connectivity visualization of the subdivision surface.*

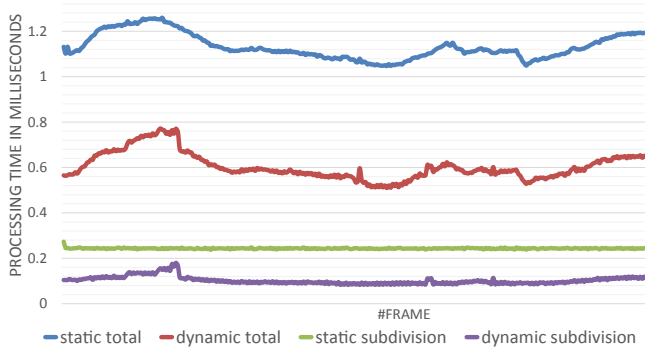

Figure 11: *Performance comparison between static and dynamic feature-adaptive subdivision measuring the processing time during a camera flight through the terrain scene shown in Fig 10.*

8 Conclusion

In this paper, we have presented a dynamic feature-adaptive subdivision surface rendering algorithm, which significantly improves performance over state of the art. With our algorithm, we are able to dynamically decide the subdivision depth for each irregular vertex individually. Thus, we are able to realize efficient level-of-detail rendering without causing over-tessellation.

Overall, we provide an abstraction over patch regularities; i.e., a tess factor can be assigned to any patch of the base mesh irrespective of whether it is irregular or not. We believe that this will help future hardware generations to support native subdivision surface rendering since the underlying technique can be abstracted away from the user by an API. Our method will be available in Pixar's OpenSubdiv framework, aiming to bring this goal one step closer.

Acknowledgements

We would like to thank Bay Raitt for the BigGuy, Sportscar and Monsterfrog models. The Killeroo model is courtesy of Headus (metamorphosis) Pty Ltd. We also thank the anonymous reviewers for their comments and suggestions for improving this paper. This research was supported by the German Research Foundation (DFG), grant GRK-1773 Heterogeneous Image Systems, and the Max Planck Center for Visual Computing & Communication.

References

ANDREWS, J., AND BAKER, N. 2006. Xbox 360 System Architecture. *IEEE Micro 26*, 2, 25–37.

BOLZ, J., AND SCHRÖDER, P. 2002. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. In *Proceeding of the International Conference on 3D Web Technology*, 11–17.

BUNNELL, M. 2005. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. In *GPU Gems 2*. 109–122.

CATMULL, E., AND CLARK, J. 1978. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer-aided design 10*, 6, 350–355.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision Surfaces in Character Animation. In *Proceedings of SIGGRAPH 98*, Annual Conference Series, ACM, 85–94.

DOO, D., AND SABIN, M. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design 10*, 6, 356–360.

EISENACHER, C., MEYER, Q., AND LOOP, C. 2009. Real-Time View-Dependent Rendering of Parametric Surfaces. In *Proceedings of I3D'09*, 137–143.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. DiagSplit: Parallel, Crack-Free, Adaptive Tessellation for Micropolygon Rendering. In *ACM Transactions on Graphics (TOG)*, vol. 28, ACM, 150.

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, Fair Interpolation using Catmull-Clark Surfaces. In *Proceedings of SIGGRAPH 93*, Annual Conference Series, ACM, 35–44.

LOOP, C., AND SCHAEFER, S. 2008. Approximating Catmull-Clark Subdivision sSurfaces with Bicubic Patches. *ACM Transactions on Graphics (TOG) 27*, 1, 8.

LOOP, C., SCHAEFER, S., NI, T., AND CASTAÑO, I. 2009. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. In *ACM Transactions on Graphics (TOG)*, vol. 28, ACM, 151.

LOOP, C. 1987. Smooth Subdivision Surfaces Based on Triangles.

MICROSOFT CORPORATION, 2009. Direct3D 11 Features. http://msdn.microsoft.com/en-us/library/ff476342(VS.85) .aspx.

MORETON, H. 2001. Watertight Tessellation using Forward Differencing. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ACM, New York, NY, USA, 25–32.

MYLES, A., NI, T., AND PETERS, J. 2008. Fast Parallel Construction of Smooth Surfaces from Meshes with Tri/Quad/Pent Facets. *Computer Graphics Forum 27*, 5, 1365–1372.

MYLES, A., YEO, Y. I., AND PETERS, J. 2008. GPU Conversion of Quad Meshes to Smooth Surfaces. In *SPM '08: ACM Symposium on Solid and Physical Modeling*, 321–326.

NI, T., YEO, Y. I., MYLES, A., GOEL, V., AND PETERS, J. 2008. GPU Smoothing of Quad Meshes. In *SMI '08: IEEE International Conference on Shape Modeling and Applications*, 3–9.

NI, T., CASTAÑO, I., PETERS, J., MITCHELL, J., SCHNEIDER, P., AND VERMA, V. 2009. Efficient substitutes for subdivision surfaces. In *ACM SIGGRAPH 2009 Courses*, ACM, 13.

NIESSNER, M., AND LOOP, C. 2013. Analytic Displacement Mapping using Hardware Tessellation. *ACM Transactions on Graphics (TOG) 32*, 3, 26.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics (TOG) 31*, 1, 6.

NIESSNER, M., LOOP, C. T., AND GREINER, G. 2012. Efficient Evaluation of Semi-Smooth Creases in Catmull-Clark Subdivision Surfaces. In *Eurographics (Short Papers)*, EG, 41–44.

NIESSNER, M. 2013. *Rendering Subdivision Surfaces using Hardware Tessellation*. Dissertation, Computer Graphics Group, Department of Computer Science, University of Erlangen-Nuremberg, Germany. Verlag Dr. Hut, Munich, Germany.

NVIDIA, C. 2007. Compute unified device architecture programming guide.

PATNEY, A., EBEIDA, M. S., AND OWENS, J. D. 2009. Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, 99–108.

PIXAR ANIMATION STUDIOS, 2005. The RenderMan Interface version 3.2.1. (https://renderman.pixar.com/products/rispec/-index.htm).

SCHÄFER, H., NIESSNER, M., KEINERT, B., STAMMINGER, M., AND LOOP, C. 2014. State of the Art Report on Real-time Rendering with Hardware Tessellation. In *Eurographics 2014 (State of the Art Reports)*, Wiley, 93–117.

SCHWARZ, M., AND STAMMINGER, M. 2009. Fast GPU-based Adaptive Tessellation with CUDA. *Computer Graphics Forum 28*, 2, 365–374.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime gpu subdivision kernel. *ACM Transactions on Graphics (TOG) 24*, 3, 1010–1015.

STAM, J. 1998. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. In *Proceedings SIGGRAPH 98*, Annual Conference Series, ACM, 395–404.

VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. 2001. Curved PN triangles. In *Proceedings of I3D'01*, ACM, 159–166.

YEO, Y. I., BIN, L., AND PETERS, J. 2012. Efficient pixel-accurate rendering of curved surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, 165–174.