Patch-Based Recursive Catmull-Clark Subdivision on the GPU

Daniel Mlakar
Supervised by: Markus Steinberger

Graz University of Technology
Graz / Austria

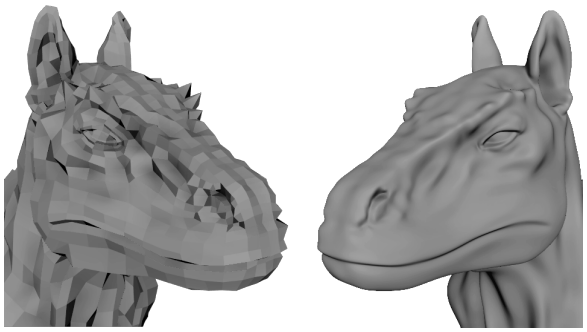


Figure 1: The prominent Killeroo model (left) subdivided with Catmul-Clark subdivision to level 5 (right).

Abstract

Catmull-Clark subdivision is an algorithm that takes a coarse mesh of a 3D model as input and outputs a smooth mesh. It has many different applications from level of detail rendering to feature film production. Starting from the coarse control mesh a series of increasingly smooth meshes is produced in an iterative way until some stopping criterion is met. Each level of subdivision depends on the previous level or, to be more precise, the positions of the new vertices are determined by its local neighborhood in the previous mesh. This property leads to patch-based approaches. In most applications Catmull-Clark subdivision is a preprocessing step and therefore has to be fast. In this work we present a parallel patch-based approach which utilizes the GPU. Patch based approaches have the advantage of enabling good possibilities for parallelization such as subdividing multiple patches to different levels at the same time. We also do not have to do preprocessing steps as some other approaches do. We use a dynamic scheduling framework which enables us to write different procedures to subdivide different types of patches. Therefore we can optimize each procedure's code by taking away the generality which would be needed if patches with different topology are subdivided by the same procedure. Good performance is reached by using matrix multiplication for the subdivision of regular quad-patches.

Keywords: Catmull, Clark, Subdivision, Patch, GPU, GPGPU

1 Introduction

Mesh subdivision is the process of refining a coarse control mesh into a dense and smooth mesh representation as shown in Figure 1. Catmull-Clark Subdivision is an iterative process where a smoother mesh is created by adding new vertices to faces and edges of the model and adapting the positions of existing vertices. Repeating these steps yields a smoother mesh representation each time. There is a need to do the necessary computations as fast as possible because Catmull-Clark subdivision is finding its way to real time applications such as computer games. It can for example be used to refine the model step by step while the camera gets closer. Using a coarser representation for models which are farther away can increase the framerate drastically. Only a few years ago Catmull-Clark was mainly used in offline rendering such as feature film rendering. In offline rendering the performance is also important but not crucial. Another scenario which requires fast subdivision is animation. Animating a subdivided model requires a large number of vertices to be moved. It is more comfortable to animate the control mesh and do the subdivision afterwards. We want interactive rendering of the fully subdivided model to give the animator instant feedback. It is an advantage to do the computations directly on the GPU because storing a subdivided model requires vast amounts of disk space and loading it to the GPU to be rendered would require large amounts of data to be transferred. Section 2 discusses related work. Section 3 provides an introduction to Catmull-Clark. We used a patch-based approach as it facilitates parallelization across different levels. It is possible to subdivide multiple patches to different levels in parallel because a patch holds all the information needed for subdivision. In Section 4 we describe the dynamic GPU scheduling framework we use for our implementation. We elucidate the importance of a fast way to subdivide regular quad-patches in Section 5.4. Section 6 compares our approach to OpenSubdiv [1]. We conclude the paper in Section 7.

2 Related Work

Subdivision surfaces have received a lot of attention in the last few years. One of the most prominent and important algorithms is Catmull-Clark subdivision [2]. There

are plenty approaches on fast implementations which use approximations instead of precise computations. Loop and Schaefer [3] used bicubic patches to approximate the subdivision surface which can be evaluated efficiently using hardware-supported tessellation. There are also adaptive subdivision approaches such as done by Nießner et al. [4]. They only subdivide in the neighborhood of features like creases and generate in each step bicubic patches which are directly rendered using hardware tessellation. An approach of fast direct evaluation was presented by Stam [5]. In this paper he first showed that Catmull-Clark subdivision surfaces can be evaluated without explicit subdivision.

Patney et al. [6] developed a data structure that can be used to do the subdivision in parallel on the GPU while maintaining the mesh structure, thus without the use of patches. They perform adaptive subdivision. Adaptive subdivision produces cracks and T-junctions and avoiding or repairing them needs additional computations. A real-time GPU kernel was presented by Shiue et al. [7]. They used the OpenGL Shading Language to do the subdivision and they rely on fragment meshes. A fragment mesh is a vertex with two layers of surrounding faces. If there are irregular vertices that are not surrounded by at least one layer of regular vertices they have to perform one step of subdivision on the CPU to make the mesh fulfil this requirement.

Many approaches have to do some kind of preprocessing as the aforementioned one. In comparison we avoid all preprocessing in our work. Our algorithm supports subdivision of meshes with arbitrary vertex valence and arbitrary face sizes but without boundaries. We use a GPU scheduling framework and a patch-based approach to do the computation of the Catmull-Clark subdivision scheme to perform uniform subdivision while maintaining good performance without the use of hardware tessellation.

3 Patch-Based Catmull-Clark Subdivision

Different strategies on how to apply the Catmull-Clark algorithm to a control mesh exist. The main difference in the approaches is the representation of the mesh. While it is possible to do the subdivision on the connected mesh we decided to extract a patch for each face in the mesh and perform the subdivision on these patches. A patch consists of the face to be subdivided and one layer of surrounding faces. We distinguish between three types of patches. The first and most general one is the arbitrary patch. It can consist of any number of faces which may have an arbitrary number of vertices with an arbitrary valence. The second type is the irregular quad-patch which is an arbitrary patch with the restriction that all faces are quads. The third and most common type is the regular quad-patch which is an irregular quad-patch with the additional property that each vertex that is not on the border of the patch has a valence of four.

3.1 Subdividing a Patch

Subdividing a patch involves four steps which are depicted in Figures 2b to 2e:

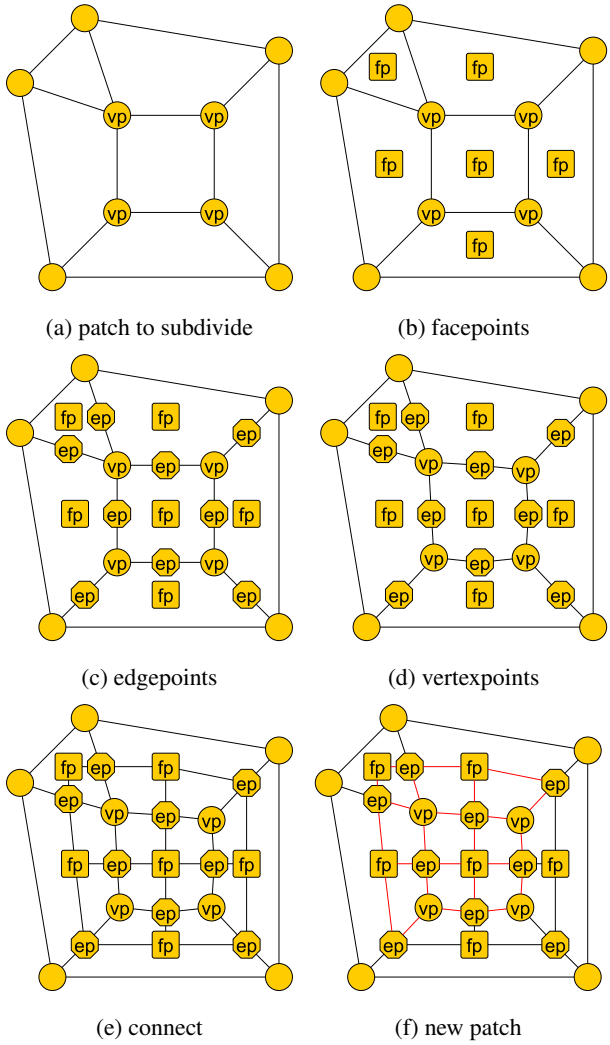


Figure 2: Steps to subdivide a patch using the Catmull-Clark algorithm

Starting with an arbitrary patch in Figure 2a, we add facepoints to the patch as shown in Figure 2b. The facepoint fp_i of the i-th face is calculated by taking the average of the face's vertices v_j as denoted in Equation 1.

$$fp_i = \frac{1}{n}\sum_{j=1}^{n} v_j \tag{1}$$

In the next step (Figure 2c) an edgepoint is added to each edge that is adjacent to one of the center vertices. The edgepoint ep_i of the i-th edge is calculated by averaging the two facepoints fp_l and fp_m of the neighboring faces and the two endpoints v_j and v_k of the edge.

$$ep_i = \frac{1}{4}(v_j + v_k + fp_l + fp_m) \quad (2)$$

In Figure 2d the adjustment of the positions of the center vertices is shown. They are moved to the positions calculated by Equation 3, where F and R are the average of adjacent facepoints and edge-midpoints respectively. v_{old} is the old position of the vertexpoint and n is its valence.

$$v_{new} = \frac{F + 2R + (n-1)v_{old}}{n} \quad (3)$$

Subsequent the newly created facepoints are connected to all edgepoints of the corresponding face edges. This is depicted in Figure 2e. Now the patch center is subdivided to a new level. The number of new faces that emerge in a step of subdivision is equal to the number of vertices of the subdivided face. As we subdivided a quad, we get four new faces. If we want to do another step of subdivision, we build a new patch for each of them. The patch for the top left face is shown in red in Figure 2f. The Patch-Based Catmull-Clark subdivision has some useful properties. First of all, the subdivision of a face exclusively yields quads. Each patch center after the first subdivision step consists of one vertex from the control mesh, two edgepoints and one facepoint. Each patch holds all the information needed to subdivide the center completely independent of other patches. Therefore, we have more possibilities for parallelization. We are able to subdivide multiple patches to different levels at the same time. Therefore, it is also possible to divide different regions of a model to different levels. Subdividing regions around features to a higher level than regular regions might be useful.

4 Whippletree

Whippletree [8] is a dynamic GPU scheduling framework which enables us to write procedures which take a specific input and provide a specific output. These procedures can be executed by a specified number of threads in parallel. The input data can be split into small work packages which can then be processed independently in different instances of the procedures. Each procedure has an input queue. A scheduler decides when a new instance of a procedure is launched with an element from its queue. Which procedures and how many of them are executed at the same time is decided with respect to using the GPU resources efficiently. The general structure of the GPU program is shown in Figure 3.

The inserter puts the patches created from the control mesh into the queue for the ”Subdivide”-procedure. This procedure is capable of subdividing patches with any topology. As it has to handle many different types of patches it has to be very general. This procedure exclusively outputs quad-patches. The ”Subdivide Non-Regular Quad Patch”-procedure handles these quad-patches. If an input element is a regular-quad patch it gets forwarded to

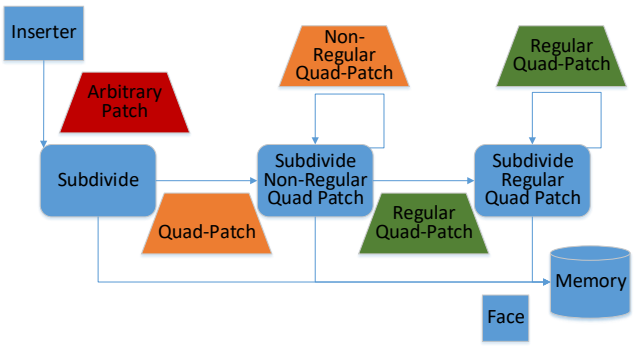


Figure 3: Structure of the subdivision program

the ”Subdivide Regular Quad Patch”-procedure. Designing this procedure in an efficient way is important because in later iterations the majority of patches are regular quad-patches. The objective was to make this procedure fast while using a minimum of resources. The stopping criterion is reaching a specified level of subdivision.

5 Implementation

In this Section we describe how shapes are represented in our implementation and how this representation is created. We also discuss the three types of patches we distinguish in our implementation. For each type we give a description of their properties and state some implementation specific details such as the data structure used to represent them.

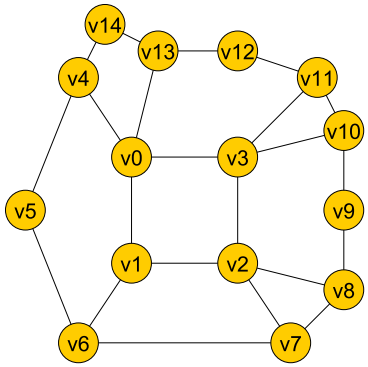
5.1 Shape Representation

We now discuss how a shape is represented in our implementation. After loading the model, a patchtable for the shape is created. A patchtable is a structure holding an array of patches and a variable that stores the size of that array which is equal to the number of faces in the control mesh. For the patches we used the data structure for arbitrary patches which we describe in Section 5.2. This is necessary, as the input patches can be of any type. In this way we can keep the part which fills the input queues of the procedures very simple as it only has to enqueue one type of patch to one procedure. Performing the first iteration with the least efficient procedure has no big impact on the runtime because the number of patches grows exponentially with the subdivision level. Therefore the number of patches for the first iteration is really just a small fraction of the total number of patches that have to be subdivided. The generation of the patchtable and therefore of the patches is done entirely on the CPU because it is only done once and therefore does not affect the per frame performance. Each patch is created iteratively by first adding the face for which we want to create a patch, the patch center, and then adding one layer of neighboring faces. Neighbouring faces are found using a map which stores the adjacent faces for each vertex. Using this map we can iterate

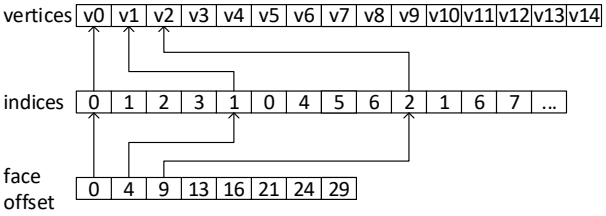
over the vertices of the already added patch center and add those adjacent faces of the vertex to the patch which were not already added before. When the patchtable is complete we can upload it to the GPU to perform the subdivison. We do not consider the creation of the patchtable a preprocessing step, because we neither change the input 3D mesh nor generate new information in this process. Therefore, the patchtable is just a different representation of the same data.

5.2 Arbitrary Patches

Properties This type of patch is the most general one. It can have any number of faces with a random number of vertices. This makes it hard to handle them efficiently. An example for an arbitrary patch is depicted in Figure 4a. Because of the quad-generation property of the Catmull-Clark subdivision algorithm only the first iteration of subdivision has to be done using this procedure. The number of patches grows exponentially with the subdivision level and therefore the performance of the subdivision procedure which handles these patches does not carry much weight when subdividing to higher levels. When just subdividing to level one there are sufficiently few patches that the model can still be subdivided fast.



(a) Arbitrary patch



(b) Arrays which represent the arbitrary patch

Figure 4: An arbitrary patch and the corresponding data structure

Data Structure The data structure used to represent an arbitrary patch is fairly large in size. This is because we can not make any assumptions about the topology. The structure consists of three arrays. The first one holds the vertex positions. The second array consists of multiple indices for each face into the first array. The third array holds an offset into the index array for each face. Additionally there are variables which store the sizes of the aforementioned arrays as well as variables for the current level and the desired level of subdivision. Figure 4b shows the arrays which represent the patch in Figure 4a. The indices for the last faces are omitted here.

5.3 Non-Regular Quad-Patches

Properties This type of patch only consists of quads which takes away some generality compared to arbitrary patches. We still have to take into account that they can be comprised of an arbitrary number of faces. When we get this type of patch from subdividing an arbitrary patch they all look as shown in Figure 5a. We can see that they only differ in the valence of the two vertices colored in red. We will discuss this property in more detail in Section 5.4. Figure 5b shows an example for a non-regular quad-patch with valences three and five. If both of this valences were four this would be a regular quad-patch.

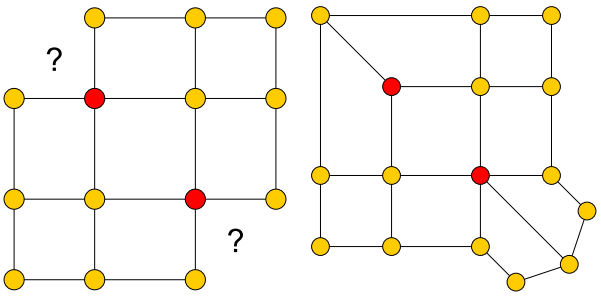
Data Structure Non-regular quad patches are represented by two arrays. The first one holds the vertex positions. The second array is comprised of four indices into the first array for each face. We can omit the face offset array as we know that all the faces in such a patch have to be quads. There is a variable that holds the number of faces and two variables for current and desired level. Figure 6a shows an example non-regular quad-patch which emerges when creating the patch for a face of a cube. The arrays which represent this patch are depicted in Figure 6b. The indices for the last two faces are omitted here.

5.4 Regular Quad-Patches

Properties A regular quad-patch only consisting of quads where each vertex of the center has a valence of four. In this Section we analyze why a majority of patches will be regular quad-patches in later iterations of Catmull-Clark subdivision and thereby emphasize the importance of subdividing this type of patch in a fast and resource indulging way.

First we state why the majority of the patches in the process of subdividing are regular quad-patches. The first reason is that subdividing a regular quad-patch yields four new regular quad-patches. Second, also non-regular quad-patches produce at least two regular quad-patches in each iteration, starting from the second one. This fact is depicted in Figure 7.

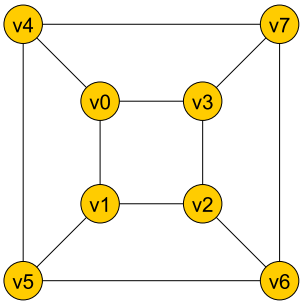
The left part of Figure 7 shows a patch center after the first iteration. On the right we have the four centers of the patches which emerge from subdividing the face on the left. To identify the faces which are the center of a regular quad-patch, we only have to check the valence of the faces



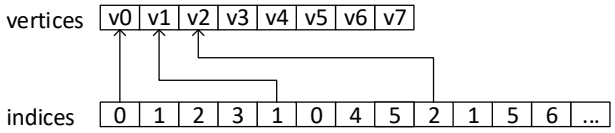
(a) Non-regular quad-patch with unknown valence vertices in red

(b) Non-regular quad-patch with example valences three and five

Figure 5: All non-regular quad-patches share some properties but differ in the valence of the vertices marked in red



(a) Non-regular quad-patch



(b) Arrays which represent the non-regular quad-patch

Figure 6: A non-regular quad-patch and the corresponding data structure

vertices as we already know that we only have quads. All edgepoints have a valence of four because they are always connected to the two facepoints of the bordering faces and to the two vertices of the edge. We also know that the facepoint in the center has a valence of four because we subdivided a quad. So the faces on the top right and on the bottom left are regular quad-patches. We do not know if the remaining two are also regular because the valence of the vertex point did not change from the control mesh. Also the valence of the facepoint in the bottom right corner of the bottom right face depends on the number of vertices of the face in the control mesh. Readers may notice that each of the four new faces has at most one vertex with a possible valence not equal to four. Due to this fact, the subdivision yields at least three regular quad patches after the third iteration.

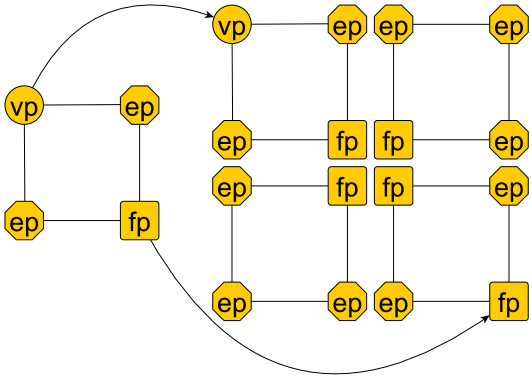
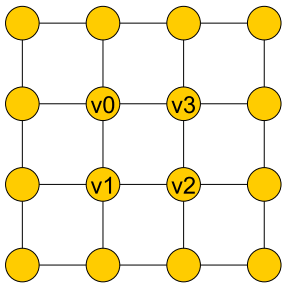
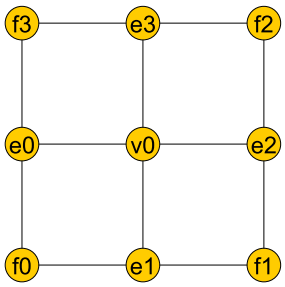


Figure 7: Left: center after first subdivision; Right: centers after second subdivision

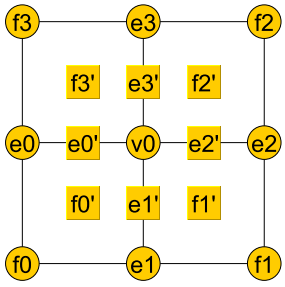
Subdivision In Section 5.4 we stressed the importance to subdivide regular quad-patches efficiently. We now discuss how this can be done. Figure 8a shows a regular quad-patch which we want to subdivide. Because of the symmetry of regular quad patches it is sufficient to explain the approach for the neighborhood of v0 which is depicted in Figure 8b. If we want to subdivide this region, we have to calculate eight vertices, which are shown as rectangles in Figure 8c and subsequently we have to move the original vertex v0.



(a) Regular quad-patch



(b) Neighbourhood of v0



(c) Unknown vertices

Figure 8: Unknowns in the neighborhood of a vertex in a regular quad-patch

As described in Section 3, edgepoints and facepoints can be calculated by averaging some known vertices. Also, the new position of the vertexpoint v0 is a weighted average of edgepoints, facepoints and the old vertex posi-

tion. Therefore, we can express this computations as dot product of two vectors as in Equations 4 to 6.

If we now define the two vectors

$$V = (v0\ e0\ e1\ e2\ e3\ f0\ f1\ f2\ f3)^{\mathrm{T}}$$
$$V' = (v0'\ e0'\ e1'\ e2'\ e3'\ f0'\ f1'\ f2'\ f3')^{\mathrm{T}} \quad (7)$$

we can find a matrix

$$S = \frac{1}{16}\begin{pmatrix} 9 & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 6 & 6 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 6 & 1 & 6 & 1 & 0 & 1 & 1 & 0 & 0 \\ 6 & 0 & 1 & 6 & 1 & 0 & 1 & 1 & 0 \\ 6 & 1 & 0 & 1 & 6 & 0 & 0 & 1 & 1 \\ 4 & 4 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 4 & 0 & 4 & 4 & 0 & 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 4 & 4 & 0 & 0 & 4 & 0 \\ 4 & 4 & 0 & 0 & 4 & 0 & 0 & 0 & 4 \end{pmatrix} \quad (8)$$

such that

$$V' = S \cdot V. \quad (9)$$

Now the subdivision of the neighborhood around v0 is reduced to this matrix multiplication. This was proposed by Doo and Sabin [9] and the matrix for regular vertices was given by Halstead et al. [10]. To subdivide the whole patch we also have to do this computation for the neighborhoods of the three remaining vertices v1, v2 and v3. Matrix multiplications can be paralleled very efficiently and are therefore well suited to be computed on the GPU.

Data Structure All regular quad-patches have the same topology. They consist of exactly sixteen vertices in nine quad-faces. Therefore, the data structure used to represent them is small. It only consists of one array holding the vertex positions. We can omit any index or face offset array as we know which vertex is connected to other vertices and which vertices are part of a specific face. Figure 9a shows a regular quad patch. The vertex array representing a regular quad-patch is shown in Figure 9b. The fact that we do not need any additional information to subdivide this type of patch is a big advantage because the structure is small in size and therefore the overall memory bandwidth requirements are reduced.

6 Results

In this Section we discuss our results and compare our approach to OpenSubdiv, which is a well known subdivision API by Pixar. All measurements were captured on a system with an Intel i5 4690k, 8GB of RAM and a Nvidia GTX 980.

6.1 OpenSubdiv

We want to shortly describe how we used OpenSubdiv as there are many different ways how to subdivide a model using this API.

The first step is to refine the topology. This does not include variables associated with vertices such as positions or normals but only connection informations. With the refined topology a stencil table is created. This stencil table holds weights which are later used to refine the vertex data. Now we can create a CUDA evaluator using this stencil table. The steps up to this point are considered preprocessing, preparing data for efficient subdivision on the GPU. We can now hand coarse vertex data, e.g. positions, to the CUDA evaluator which then calculates the refined data for each vertex in the refined topology.

6.2 Ours vs. OpenSubdiv

For the evaluation we used two simple models namely a cube and a pyramid and also some prominent models that are often used to evaluate subdivision approaches such as the Killeroo, Bigguy and Monsterfrog. Figure 10 shows their control mesh and the subdivided model.

Table 1 shows the timings we measured for the per frame computations. We can see that our approach performs pretty similar to OpenSubdiv. The values for our approach also include the subdivision of the topology which is one of two preprocessing steps in OpenSubdiv. The second preprocessing step is the calculation of the stencils. This timings can also be found in table 1. They take pretty long dependent on the complexity of the model because they are executed on the CPU and not on the GPU. It turns out that our approach seems to be also well suited for applications where models with frequently changing topologies have to be subdivided. This is the case in the 3D modeling process if we want to show the artist a live preview of the subdivided mesh. Using OpenSubdiv in such a scenario would require doing the preprocessing each time the topology changes which would result in low framerates. If the changes in topology are known beforehand it would be possible to precompute the refined topology and the stencil table and store it for later use but that would use big amounts of disk space depending on the number of different topologies.

	Ours	**OpenSubdiv**			
	PFE	sum	PFE	RT	CST
Cube lvl 9	2.081	1551.687	2.519	141.680	1407.488
Pyramid lvl 10	4.664	2838.814	3.543	391.277	2443.994
Killeroo lvl 4	1.805	1757.346	1.656	84.056	1671.634
Bigguy lvl 4	0.710	683.239	0.583	32.243	650.413
Bigguy lvl 5	2.005	2889.844	2.216	135.861	2751.767
Monsterfrog lvl 4	0.718	601.266	0.561	28.992	571.713
Monsterfrog lvl 5	1.904	2519.056	2.198	119.914	2396.944

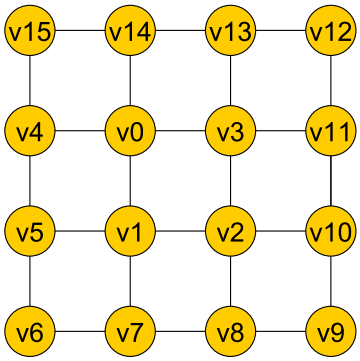
Table 1: Mean of measurements taken over fifteen runs in [ms]. Per Frame Evaluations (PFE), RefineTopology (RT), ComputeSencilTable (CST)

While stencil tables have obvious advantages, using

$$f0' = \left(\tfrac{1}{4}\ \tfrac{1}{4}\ \tfrac{1}{4}\ \tfrac{1}{4}\right) \cdot (v0\ e0\ e1\ f0)^{\mathrm{T}} \tag{4}$$

$$e0' = \left(\tfrac{3}{8}\ \tfrac{3}{8}\ \tfrac{1}{16}\ \tfrac{1}{16}\ \tfrac{1}{16}\ \tfrac{1}{16}\right) \cdot (v0\ e0\ e1\ e3\ f0\ f3)^{\mathrm{T}} \tag{5}$$

$$v0' = \left(\tfrac{9}{16}\ \tfrac{3}{32}\ \tfrac{3}{32}\ \tfrac{3}{32}\ \tfrac{3}{32}\ \tfrac{1}{64}\ \tfrac{1}{64}\ \tfrac{1}{64}\ \tfrac{1}{64}\right) \cdot (v0\ e0\ e1\ e2\ e3\ f0\ f1\ f2\ f3)^{\mathrm{T}} \tag{6}$$



(a) Regular quad-patch

vertices	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15

(b) Array which represents the regular quad-patch

Figure 9: A regular quad-patch and the corresponding data structure

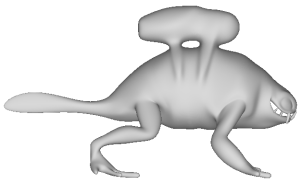
(a) Killeroo at level 0

(b) Killeroo at level 4

(c) Bigguy at level 0

(d) Bigguy at level 5

(e) Monsterfrog at level 0



(f) Monsterfrog at level 5

Figure 10: Models used for performance evaluations

them to perform view dependent subdivision causes some problems. Therefore, OpenSubdiv also supports patchtables. In the adaptive subdivision case only non-regular patches are subdivided using a stencil table as in the uniform case but the subdivision of the regular quad-patches is done using a patchtable and tessellation shaders. Regular patches are stored as b-spline patches in the patchtable. Each of them is converted into a bezier patch in a control tessellation shader and then evaluated in the evaluation tessellation shader. To get a setup comparable to the one used in OpenSubdiv we adapted our approach such that only non-regular patches are subdivided further and regular patch centers are written to the GPU memory after the first subdivision step. In OpenSubdiv we used a stencil table to do adaptive refinement where also only the non-regular patches are subdivided without the use of tessellation shaders. With this experiment we wanted to compare the performance in subdividing non-regular patches to get an estimate of the fraction of computation time spent on this type of patch when subdividing the whole model. The result of this experiment can be found in 2. Screen space or view dependent subdivision in OpenSubdiv is also done using the described setup. With our approach it would be possible to do screen space subdivision without the use of shaders as we can dynamically decide which patches we have to subdivide further and which ones we want to render.

As we do all the computations in software on the GPU and refrain from using hardware tessellation we can overcome some limitations: Using displacement mapping on a large input patch could lead to under tessellated displacements. The opposite is also possible: the displacement is sufficiently subdivided but the surrounding is over tessellated. Our approach not only allows for dynamic adaption of the tessellation levels for different patches but also for regions within a patch, in contrast to OpenGL where inner and outer tessellation levels have to be specified, which determine the amount of tessellation for the whole patch. This can also be used in other applications than displacement mapping where it is desirable and beneficial that different regions are subdivided to different levels. Using OpenGL the maximum number of output vertices is also limited which restricts the maximum tessellation level while the maximum amount of tessellation in our approach is only limited by the available GPU memory. Using a software approach for the mesh subdivision increases the flexibility in general and therefore makes it more versatile.

	Ours	OpenSubdiv			
	PFE	sum	PFE	RT	CST
Cube lvl 9	0.274	11.684	0.008	0.470	11.206
Pyramid lvl 10	0.313	11.358	0.007	0.522	10.829
Killeroo lvl 4	0.635	84.918	0.044	16.211	68.663
Bigguy lvl 4	0.318	42.853	0.025	7.319	35.509
Bigguy lvl 5	0.364	55.378	0.025	9.634	45.719
Monsterfrog lvl 4	0.320	51.857	0.030	9.074	42.753
Monsterfrog lvl 5	0.376	67.980	0.030	12.439	55.511

Table 2: Mean of measurements taken over fifteen runs in [ms] in the adaptive case. Per Frame Evaluations (PFE), RefineTopology (RT), ComputeSencilTable (CST)

7 Conclusion and Future Work

Subdivision has a wide range of applications. A commonly used algorithm is the Catmull-Clark algorithm. There are many promising approaches to use subdivision in real time applications which increases the necessity of fast Catmull-Clark subdivision. We achieved good performance by implementing a patch-based approach which was parallelized on a GPU. A dynamic scheduling framework was used, which helps utilizing the GPU resources efficiently. We compared our approach to OpenSubdiv, which spends significant time on data preprocessing - approximately 1000× the time it requires for the processing in each frame. Even though our approach does not require any preprocessing, our per frame processing times are comparable. This fact also points towards the ability of our approach to handle topology changes efficiently and to be integrated into a modeling software. We focused on optimizing the procedure which handles regular quad-patches because they form the majority of patches. This was done by expressing the subdivision of such a patch as a single matrix multiplication. This approach could also be used to subdivide non-regular quad-patches by adjusting the used matrices. In that way the performance could be improved but presumably not by far. Our approach could be extended to perform screen space subdivision because it is possible to dynamically decide the subdivision level of each face independently.

References

[1] Pixar. http://graphics.pixar.com/opensubdiv/. accessed: 05.02.2017.

[2] E. Catmull and J. Clark. Seminal graphics. chapter Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes, pages 183–188. ACM, New York, NY, USA, 1998.

[3] Charles Loop and Scott Schaefer. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Trans. Graph.*, 27(1):8:1–8:11, March 2008.

[4] Matthias Nießner, Charles Loop, Mark Meyer, and Tony Derose. Feature-adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Trans. Graph.*, 31(1):6:1–6:11, February 2012.

[5] Jos Stam. 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, 1998. ACM.

[6] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. Parallel View-dependent Tessellation of Catmull-Clark Subdivision Surfaces. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 99–108, New York, NY, USA, 2009. ACM.

[7] Le-Jeng Shiue, Ian Jones, and Jörg Peters. A Real-time GPU Subdivision Kernel. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1010–1015, New York, NY, USA, 2005. ACM.

[8] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.*, 33(6):228:1–228:11, November 2014.

[9] D. Doo and M. Sabin. Seminal graphics. chapter Behaviour of Recursive Division Surfaces Near Extraordinary Points, pages 177–181. ACM, New York, NY, USA, 1998.

[10] Mark Halstead, Michael Kass, and Tony DeRose. Efficient, Fair Interpolation Using Catmull-Clark Surfaces. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 35–44, New York, NY, USA, 1993. ACM.