Catmull–Clark Subdivision Surfaces on GPU

Juraj Konečný*
Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava, Slovakia

Abstract

In this paper we describe our approach for computing subdivision surfaces. We proceed in two ways. The first is computing on CPU and the second is computing using GPU. To achieve these goals we use OpenGL. Also we want to compare results of computing on CPU and GPU. We use two different methods for computing on CPU and three different methods of computing on GPU. The difference between these methods on GPU is in the way how we use textures for computation. The results of these algorithms are compared and evaluated. We use only one subdivision algorithm - Catmull-Clark on regular meshes for demonstration purposes.

Keywords: CG, subdivision surface, GPU, OpenGL, Catmull-Clark

1 Introduction

Subdivision surfaces are not new, but their use in high-end CG production has been limited. Traditionally, shapes like hands, heads, etc. have been modeled with NURBS surfaces despite the severe topological restrictions that NURBS imposes. Few years ago things were changed. Subdivision surfaces are replacing NURBS in modeling and animation. There are many reasons for using subdivision: efficiency, compact support, local definition [1]. Subdivision surfaces are used for example in famous short film Geri's game, movies Bug's Life, and Toy Story 2. Figure 1 shows a screenshot showing detail of Geri's finger which was computed using piecewise smooth Catmull-Clark surface [2].

Figure 1: Geri's Game – Geri's fingers

On SIGGRAPH 2005 was published a paper about subdivision surfaces computed mostly using GPU. Authors succeeded in implementation of Catmull-Clark subdivision scheme also for extraordinary vertices [3]. Another related work about subdivision using GPU is even older [4]. A result of this is a realtime rendering of subdivision surfaces using GPU. Our approach is a little bit different because we want to compare more methods and implementations of the same algorithm.

Even though CPUs performance has increased in last years and also CPUs with more than one core in desktop computers can be found, there are still many reasons why to use GPU for many purposes. For example new operating systems with all applications can consume nearly all the resources we have. A regular user is running more than one application at the time. "Today, parallel GPUs have begun making computational inroads against the CPU, and a subfield of research, dubbed GPGPU for General Purpose Computing on GPU has found its way into fields as diverse as oil exploration, scientific image processing, and even stock options pricing determination. There is increased pressure on GPU manufacturers from "GPGPU users" to improve hardware design, usually focusing on adding more flexibility to the programming model" [5]. That is why we have decided to implement this subdivision algorithm and compare performance of CPU and GPU. For working with GPU we use OpenGL shading language [6] and programmable shaders.

The paper is structured as follows. Section 2 describes the subdivision. In Section 3 we deal with OpenGL and GPU. Section 4 reports on comparison of time at different computers. Future work we discuss in Section 5.

2 Subdivision

There are plenty of subdivision algorithms being used these days. As mentioned before, the main motivation of subdivision is to replace NURBS, because subdivision surfaces have many properties similar to NURBS but computation is faster and easier. Most important properties of subdivision surfaces are:

- **Efficiency:** the location of new points should be computed with a small number of floating point operations what makes it easy to move the computing to GPU and also makes huge

* konecny.juraj@gmail.com

difference between subdivision surfaces and NURBS

- **Compact support:** the region over which a point influences the shape of the final curve or surface is small and finite, what gives us a lot of freedom that is essential for modeling
- **Local definition:** the rules used to determine where new points go should not depend on "far away" places so it is easy to work with small fragments of entire mesh independently
- **Affine invariance:** if the original set of points is transformed, e.g., translated, scaled, or rotated, the resulting shape should undergo the same transformation
- **Simplicity:** determining the rules themselves should preferably be an offline process and there should only be a small number of rules what enables us to set up fragment shader for computing
- **Continuity:** Each subdivision algorithm is specified also by its continuity as it can be seen in Table 1

In Table 1 there are the most known stationary subdivision schemes generating C^1 continuous surfaces on arbitrary meshes using splitting faces. Approximating schemes converge faster then interpolating and also produce surfaces of higher quality especially when talking about continuity.

	Triangular mesh	*Quadrilateral mesh*
Approximating	Loop C^2	Catmull-Clark C^2
Interpolating	Mod. Butterfly C^1	Kobbelt C^1

Table 1: Face split subdivision schemes

A natural choice for implementation of subdivision algorithm is Catmull-Clark.
Main reasons are:

- Control mesh doesn't need to be quadrilateral (It will be quadrilateral after the first mesh refinement using more general form of Catmull-Clark rules)
- Simple data representation for regular control meshes (in regular control mesh the valence of interior vertices is 4, boundary 3 and valence of corner vertices is 2)
- Resulting surface is C^2 continuous everywhere except for extraordinary vertices where it is C^1 continuous (for purpose of display we can use coefficient originally suggested by Catmull-Clark but to achieve formal C^1 continuity we need a bit more complicated coefficients [7])

The final subdivision surface is a limit of repeated subdivision. While we are working on regular meshes standard rules for Catmull-Clark subdivision scheme are used in the form displayed in Figure 2 and Figure 3. It is obvious, that we need only a few floating point operations to find the positions of new vertices and new positions for old vertices. Handling of extraordinary vertices requires other rules, but for now we stay

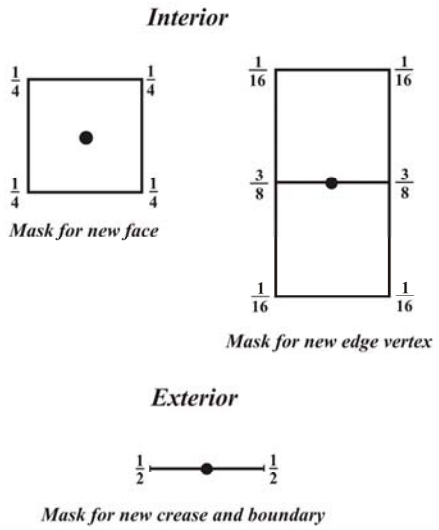

Figure 2: Standard rules for regular control mesh for new vertices

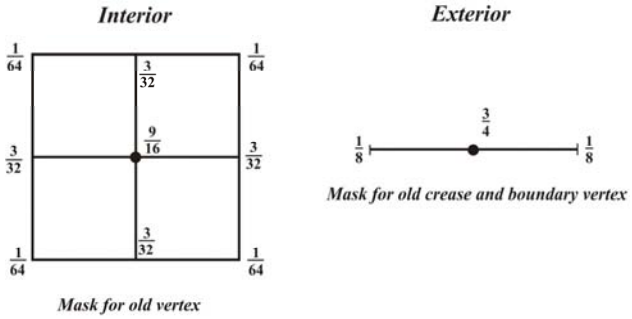

Figure 3: Standard rules for regular control mesh for old vertices

working with regular meshes what brings us various benefits. On the other hand it also decreases the amount of surfaces we can generate. But one of our main goals is to compare performance of CPU and GPU so regular control mesh is enough to get results which are worth to be taken into consideration.

For storing data of the entire mesh it is useful to have complex data structure which is flexible for reconfiguring the entire mesh and interacts with mesh processing algorithm. Half–edge data structure is costly for maintaining, reconfiguring and even for refining used in subdivision algorithm, but still is a good choice because of its adaptability. In our algorithm we use it only for storing mesh in the OpenGL scene and for the first iteration of the subdivision algorithm. It means, that our work with this initializing data structure is limited and the influence of its processing is small [3]. For this, we use half-edge structure described in [12]. This half-edge structure is considered to be one of the fastest. This half-edge is one of the structures we use for subdivision. Even trough it is costly to maintain, its adaptability will help us in future work. Another data structure used for computing on CPU is a two dimensional array in which we store all important data we need. We can even use static structure because the size of a regular mesh after

refining is easy to compute. When the size of the regular input mesh is $m \times n$ then after refinement it is $(2m-1) \times (2n-1)$. Then we compute new positions of vertices using Catmull-Clark subdivision rules. Knowing indices of vertex stored in this array means that we even know which rule is to be used so we can do it in few cycles computing new internal face and edge vertices separately, then new values of old vertices and, at last, crease and boundary vertices. In Figure 4 we can see how to determine type of new vertex.

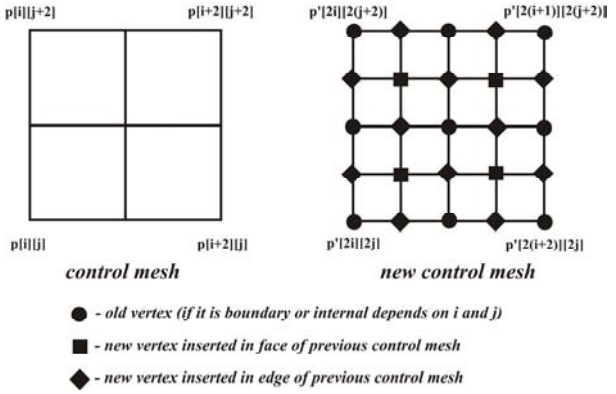

Figure 4: Array refinement

The last thing we perform on CPU is to create 2D floating point texture for storing coordinates of vertices in RGB space and one 2D texture for storing additional data (for example whether it is crease or not). Because OpenGL supports textures with size $2^w \times 2^h$ we need to store the actual size of used part of these textures in values w (width) and h (height). There are some extensions in OpenGL supporting textures with "non power of two" size but because we also need to divide by size of texture it is better to divide by power of two.

3 OpenGL and GPU

For GPU programming we use OpenGL and OpenGL shading language. Another alternative is to use Direct3D but we have decided for OpenGL because we want our program to work also under other operating systems like Linux. Most computation is done using programmable fragment, vertex and geometry shader. Fragment shader is used to compute values to update frame-buffer or texture memory. Vertex shader computes homogenous position of a single vertex independently of other vertices [6]. For now we use fragment shader to render all

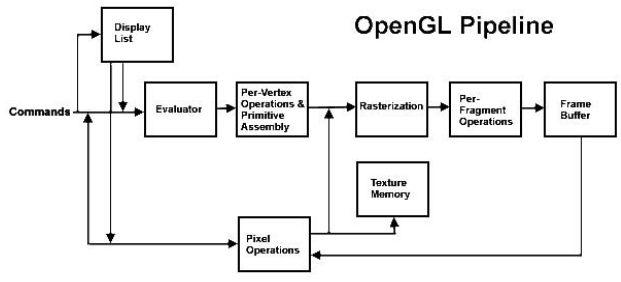

Figure 5: OpenGL pipeline [8]

important data. On the other side geometry shader begins with a single primitive (in our case point). It can read its attributes and use them to generate new primitives which are clipped and then processed like an equivalent OpenGL primitive specified by the application [9]. This new geometry shader (released in November 13, 2006) should be included in future work. Other information about programming for GPU could be found in [10].

To get 2D floating point texture with 32-bits for each RGB channel we use OpenGL extension GL_ARB_TEXTURE_FLOAT. In this texture there are stored coordinates of vertices. To know which part of texture is active we also have stored two variables m, n. We also store all subdivision masks we need in arrays. For additional data we have look_up texture where we store data like point is boundary, corner or crease. These are all data we need to create fragment shader computing new iteration. For storing new mesh we render it to texture [11]. We compare we use three methods of rendering. The first one is rendering to frame buffer object (FBO), second one is using copy to texture [13] and third one is direct rendering to texture using pixel buffer (pbuffer) [14]. Using FBO which is similar to texture object makes working with rendering target easier [15]. Copy to texture is the slowest and the oldest of these methods but it is easy for implementation. Direct rendering using pbuffer is rendering to new rendering context. This last method has some "bottle necks" which influence runtime and makes computing slower.

When we have rendering target we need to set up viewport to see entire rendered image. We already know its size. It is $(2m-1) \times (2n-1)$ where $m \times n$ is size of input texture. When we have the rasterized output (as it can be seen in Figure 5) we can easily use fragment shader to compute the new image. To determine which subdivision rule is to be used we divide current fragment coordinates by *2* and then we can find out which subdivision mask to use:

```
x1=x div 2;  y1=y div 2;       //x and y are coordinates
x2=x mod 2;  y2=y mod 2;   //of fragment
```

While *x1* and *y1* give us coordinates of neighboring vertices in input texture, *x2* and *y2* provide us with information if it is a new vertex or not.

1. old vertex: *x2==0* and *y2==0*
2. face vertex: *x2==1* and *y2==1*
3. edge vertex: *x2==1* and *y2==0,*
   x2==0 and *y2==1*

To avoid branching in this case we need new 1D function that will separate all this cases and also works with look_up texture. For this we multiply $(2 \times x2)+(3 \times y2)$ and choose part of array where is stored scheme for this case. Now we know which mask we use for computing coordinates of vertex *p[x][y]* and also indices *i,j* of neighboring vertices in old mesh. To get their texture coordinates, we need to divide their indices by the size of texture $i=i/2^h$, $j=j/2^w$. Using these values, we compute new RGB coordinates of vertex *p[x][y]*. This procedure is repeated for each fragment. Independence of these operations brings an opportunity to use parallelism provided by fragment processors because each fragment depends only on its coordinates, input texture, and values

h,w. If we want to continue in repeating subdivision we just have to replace input texture with computed output and also change *h, w, m, n* values. A difference between this method and computing on CPU using array instead of texture is, that here we have to determine which mask we have to use while with array we just perform a cycle for each mask computing only vertices we want. For rendering and displaying of resulting mesh we use vertex shader which is used to transform coordinates from RGB coordinates.

4 Comparing

There are two ways how to compute Catmull-Clark subdivision surfaces in our work. To get more values for comparing, we use 2 computers with different configuration. Both of them are running Windows XP Professional Edition and also Linux. Here are their hardware configurations:

1. HP Pavilion zd8000, Intel Pentium 4 540 / 3.2 GHz, FSB 800MHz, Hyper-Threading Technology, 1GB DDR II SDRAM - 400 MHz, graphic card ATI Mobility Radeon x600 - PCI Express x16, 256 MB video memory, SuSe Enterprise 10.0
2. Toshiba Satellite M70, Intel Centrino M 760 /2.00 GHz, FSB 533 MHz, 512MB DDR II SDRAM - 533 MHz RAM, graphic card ATI Mobility Radeon x700 - PCI Express x16, 256 MB video memory, Ubuntu 6.10

These two computers are very different. The first one has a faster CPU while the other configuration has a faster GPU. Also we want to use some computers with NVIDIA graphic cards, but their configuration is not known yet.

At first we run the algorithm using only CPU for half-edge and array data structure on both computers. The result is the time to render the final surface. This is repeated several times for different control meshes. Problem on GPU is, that rendering subdivision surface of depth 6 from control mesh 8×8 stored in array gives us during scene transformations average frame-rate about *4 fps* on computer running all common utilities. It is even slower when we use half-edge data structure. We repeat the same using computing on GPU with the same control meshes. Another part is rendering while there is user interaction with mesh in scene. In this part our result is the frame rate. User is changing coordinates of control mesh vertices in OpenGL scene and after each change the surface is rendered. We hope to achieve realtime rendering at this part. Average frame rate after 5 minutes of modeling is another result. These values will be compared and displayed in a table. This part is written in advance, because shader implementation is not finished yet. In related work [3] they got frame-rate about *1 fps* using half-edge data structure on CPU for subdivision surface of depth 4 and about *20+ fps* using fragment shaders.

5 Future work

This work is not finished in several directions. For working with irregular meshes we need to make many changes. Entire algorithm is extended for computing irregular meshes as well as regular. For this extension we have to use modified coefficients to achieve continuity of final surface. Also we have to compute new masks for each valence of extraordinary vertices and store them in another texture. For data representation we do not use array any more and CPU part is done using half-edge data structure described before. For computing data on GPU we use 1D texture for storing coordinates and another texture which serves like a look up table for finding neighbours [3],[4]. Loop subdivision scheme is another scheme which has simple rules and can be computed similar way. But even for regular meshes it is more difficult to store data in texture than for regular meshes for Catmull-Clark subdivision. Data are stored in textures the similar way like for irregular meshes mentioned before. Last extension is computing on GPU with using geometry shaders which provides us with many interesting features we can use. Even through it is new extension in OpenGL there are already some algorithms already implemented using geometry shader. One of them is marching cubes. All this future work is about to bring us new information about performance and real use of GPU for computing subdivision and its benefits.

References

[1] ZORIN, D. et al. 2000. ***Subdivision for Modeling and Animation.*** Siggraph 2000 Course Notes.

[2] DEROSE, T. et al. 2000. ***Subdivision Surfaces in Character Animation.*** Part of [1]. Siggraph 2000 Course Notes. pp. 185-194

[3] SHIUE, L.-J. et al. 2005. A Realtime GPU Subdivision Kernel**.** In *SIGGRAPH 2005 Conference Proceedings*.

[4] BOLZ, J. et al. 2007. Evaluation of Subdivision Surfaces on Programmable Graphics Hardware. Available from http://www.multires.caltech.edu/pubs/GPUSubD.pdf. Accessed January 19, 2007.

[5] Graphics processing unit, Available from http://en.wikipedia.org/wiki/GPU. Accessed January 19, 2007.

[6] KESSENICH, J. et al. 2006. The OpenGL Shading Language. Document Revision: 8. 07-Sept-2006.

[7] BOLZ, J. et al. 2002. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. In *Proceedings of the Web3D 2002Symposium*, pp. 11-18.

[8] LIPCHAK, B. 1997. Overview of OpenGL. Available from http://web.cs.wpi.edu/~matt/courses/cs563/talks/OpenGL_Presentation/. Accessed January 19, 2007.

[9] YONGMING, X. 2006. Geometry Shader Tutorials. Version 15.11.2006. Available from http://appsrv.cse.cuhk.edu.hk/~ymxie/Geometry_Shader/. Accessed January 19, 2007.

[10] PHARR, M. *ed.* 2005. ***GPU GEMS 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.*** ISBN-10: 0321335597. ISBN-13: 978-0321335593. Publisher 3. 3. 2005.

[11] BAKER, P. 2007. Render To Texture. Available from http://www.paulsprojects.net/opengl/rtotex/rtotex.html. Accessed January 19, 2007.

[12] KETTNER, L. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry 13*, 1 (May), 65–90.

[13] CORNO, D. 2007. Nehe Productions: OpenGL Lesson #36. Available from http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=36. Accessed January 19, 2007.

[14] WYNN, CH. 2007. OpenGL Render-to-Texture. Available from http://developer.nvidia.com/object/gdc_oglrtt.html. Accessed January 19, 2007.

[15] JONES, R. 2006. OpenGL Frame Buffer Object 101. Posted 11/22/2006. Available from http://www.gamedev.net/reference/articles/article2331.asp. Accessed January 19, 2007.