CS780: Topics in Computer Graphics - Scalable Graphics/Geometric Algorithms

Mid-term report

Subdivision Meshes in GPU

20075026 Young-Jun Kim

Introduction

The most of objects in our real life has smooth shape. The subdivision meshes are developed for representing the characters and the objects naturally in animations and games. The subdivision meshes are methods of representing the smooth surfaces using iterative operation with a few number of data (control meshes).

The graphics processing unit (GPU) in nowadays has more flexible structure and programmability while the traditional graphic processors with the fixed pipeline were optimized for rendering the image to the frame buffer. These programmable elements in GPU are called the programmable shaders. Thanks to the programmable shaders, GPU can operate the complex calculation or flow control like CPU. GPU is suitable for processing independent and parallel data due to its structural characteristic. The various evaluations of subdivision meshes on GPU have been researched since the algorithm of subdivision meshes is matched well with GPU structure.

Many people want to use the GPU for solving general problems – physical simulation, calculations, etc. – because the GPU is very powerful parallel processing machine. The general purpose graphics processing unit (GPGPU) fields handle those problems.

Even though GPU has a plentiful of programmable functionalities, the GPU has a limitation of data feedback – reusing the processed data – since GPU is unidirectional processing unit for rendering image.

In this research, we survey the structural features of GPU and find the most efficient memory handling for resolving the bottle-neck of memory transferring during evaluating of the subdivision meshes.

Related work

There are a few issues for representing subdivision meshes.

One is how fast calculate the subdivision arithmetic. The standard recursive evaluations or the tabulated driven evaluations exist.

Another is data transactions and load balancing problem between CPU and GPU. In general, the regular subdivision can be processed parallel and within GPU but the processing of irregular vertices or at the case of adaptive subdivision requires handling by CPU. The load balancing between CPU and GPU are addressed in many research. The good approach is maximizing the GPU load and minimizing the CPU load.

The efficient evaluating the subdivision meshes are advanced in many research, and the recent issues are the deformation of subdivision meshes. The deformation of subdivision meshes is useful for dynamic scene in animations.

The major researches for the evaluations of subdivision meshes on GPU are followings.

[Bolz et al. 2002] This is the version of the implementation on GPU using tabulated driven method. The tabulated driven method has the characteristic of very regular and cacheable. They insist that it is faster due to its regular memory access pattern even though it has more operation counts than classical recursive method.

Pros.

1. Faster operations due to proper structure of memory and cache
2. Maximum use of GPU parallel architecture

Cons.

1. The lack of flexibility
2. Need many tables
3. More operation counts
4. Result data should be feed through to CPU and CPU send data to GPU for rendering

[Bunnel et al. 2005] It is using the patch-based data processing. Adaptive subdivision scheme is implemented by using flatness test. The animation is showed using the model data of several frames. This is a good practical subdivision meshes example.

Pros.

1. Adaptive subdivision using flatness test
2. Implement many practical features (displacement mapping, etc.)

Cons.

1. Too complex data structure (many lists)
2. Too many interactions between CPU-GPU.

[Shiue et al. 2005] It is suggested for overcoming the data dependency problem of irregular pattern by adopting fragment mesh conception. It is maximizing the parallel processing in GPU. It may be the best solution for static subdivision meshes except adaptive subdivision features.

Pros.

1. Almost operation is executed in GPU. (No dependency between fragment meshes)
2. No redundant transaction for rendering using p-buffer
3. Good flexibility

Cons.

1. The lack of the description of adaptive subdivision
2. P buffer context switching (target to the texture memory and target to the frame buffer) has a large overhead.

[Kim et al. 2005] It is the version of the implementation of Loop subdivision algorithm using the method of [Shiue et al. 2005]. This is a good explanation of the data transaction between local host memory and video memory in GPU.

Pros.

1. Good reference of the description of the data transaction between local host memory and video memory.

Cons.

1. Not implemented features using VBO/PBO (superbuffer in ATI GPU)

[Zhou et al. 2007] The research of the deformation of subdivision meshes. It is useful for dynamic scene.

Pros.

1. Real-time deformation of subdivision surfaces model
2. Good approach for subdivision of dynamic scene

Cons.

1. Only supported for uniform subdivision
2. Not yet collision-free deformation

Out research examine the efficient memory usage including adaptive subdivision features.

Overview

The conception of evaluating the subdivision meshes is very simple. We use Catmull-Clark subdivision with this research since Catmull-Clark subdivision is most famous and having a nice-look result at most cases. All other schemes are conceptually similar.

Subdivision meshes consists of two phase processes – refinement phase and smoothing phase. New vertices are created and reconnected during refinement phase and all the old and new vertices are moved to new positions during smoothing phase. Simple rules are used for calculating new positions for vertices. These averaging rules of weight-parameters are called as 'masks'.

These processes are described in Fig. 1 and Fig. 2.

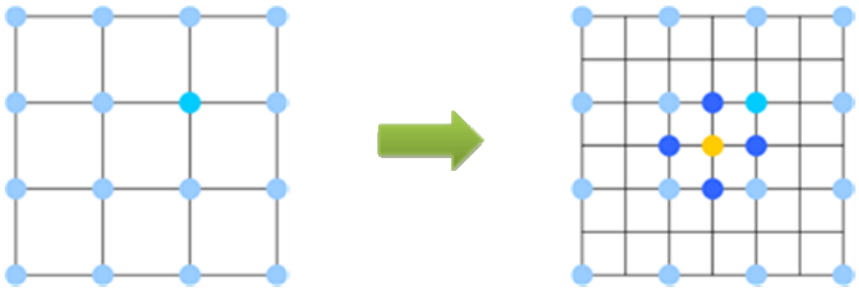

Fig. 1 Face split (Refinement phase)

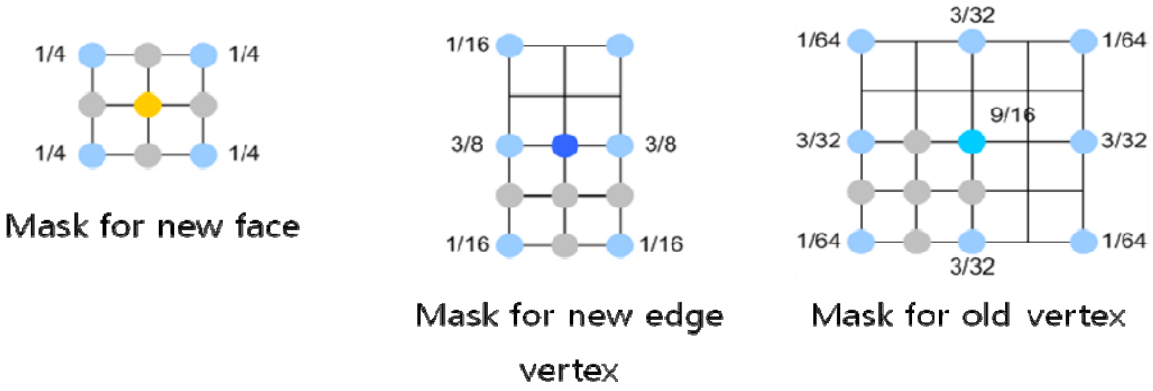

Fig. 2 Standard rules for regular control mesh (Smoothing phase)

These two process are iteratively executed and $(2^n+1) * (2^n+1)$ vertices are managed at n-th iteration for one original quad face. These iterations require the recursive operation for getting more detail. The efficient memory management is required since the calculation of new positions requires the information of neighbors. There are more complex cases for the irregular cases (vertex having more than 4 or less than 4). The irregular cases make memory access pattern irregular.

The components of GPU are showed in Fig. 3. There are GPU front-end (input buffer), vertex shader, primitive assembler, rasterizer, pixel shader, raster operator, and frame buffer. The data from CPU run through all this graphics pipelines.

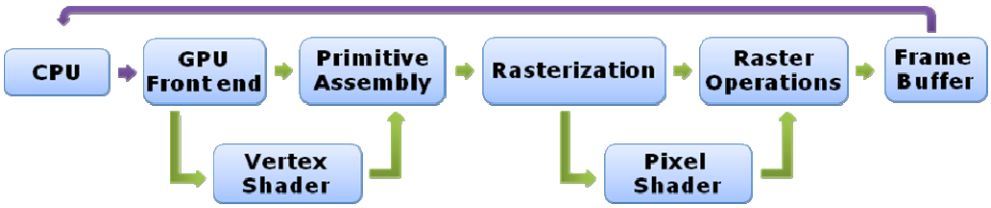

Fig. 3 Graphics Pipelines

The video memory in GPU consists of texture memory and frame buffer memory. CPU can read and write texture memory and frame buffer memory. The final rendered image is stored in the

frame buffer memory. The pixel shader in GPU can only read texture memory and write to raster operator (finally frame buffer). Generally, CPU cannot read the data in the middle of vertex shader and pixel shader. That is the reason that the pixel shader is mainly used for GPGPU.

The previous result of calculation is needed as the input data of the next operating stage for evaluating subdivision meshes. CPU write down the initial input data into the texture memory and pixel shader reads the texture memory and processes those data. The calculated data are written into the frame buffer memory. CPU reads the data in the frame buffer because the pixel shader cannot read from the frame buffer memory. And CPU re-sends the processed data into GPU. This process is described in Fig. 3. The data transaction between the host (CPU) local memory and the video memory is occurred during this process and it causes a very expensive traffic cost because there is slower interface between them (PCI express or AGP bus).

The standard OpenGL API supports the feature of copying the frame buffer memory into the texture memory (Fig. 4). This feature enables that the memory transaction is only within the video memory. But the data copied from the frame buffer data are redundant.

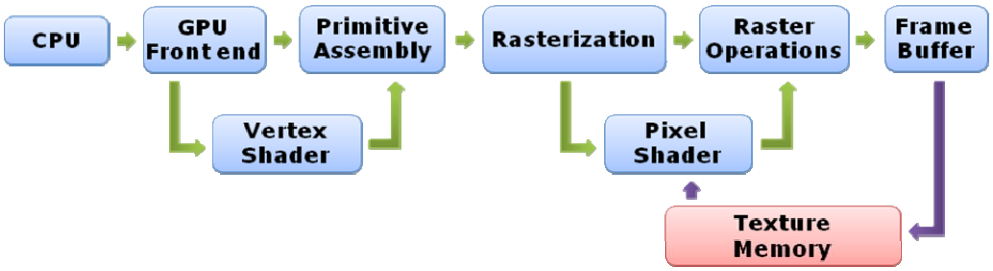

Fig. 4 Copy-to-Texture

The extended OpenGL API also supports the render-to-texture method using the pbuffer. The pixel shader can write data to the pbuffer, and this pbuffer can be used as the texture memory. But the read operation and the write operation cannot be occurred at the same time. The context switching between binding as frame buffer and binding as texture memory is very expensive in API.

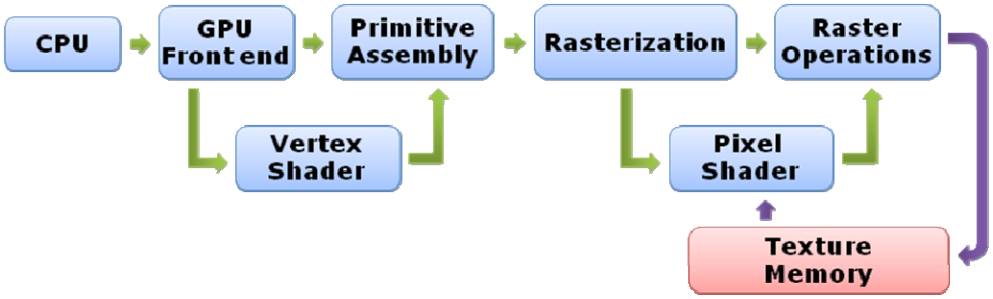

Fig. 5 Render-to-Texture

There is another OpenGL extension for resolving this structural problem. The frame buffer object

(FBO) function in Fig. 6 enables setting the rendering destination as the off-screen buffer (renderbuffer) or the texture. It is the fastest method since there is no context switching overhead.

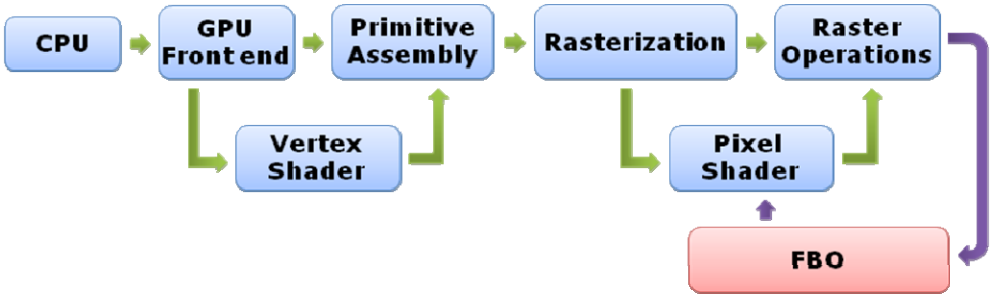

Fig. 6 Render-to-FBO

The immediate data during evaluation of subdivision meshes can be transferred back using above configurations.

There is the other issue of the memory transaction for rendering. The final result of subdivision processing should be send to vertex shader input due to the characteristic of GPU. CPU can send the vertex data after reading the frame buffer. The subdivision meshes are invented for representing the complex and smooth surfaces using a few data. But we lose the advantage of subdivision mesh in this case since there are a lot of data transactions from the result of subdividing.

The extension of OpenGL API also supports vertex buffer object (VBO) and pixel buffer object (PBO) for resolving this problem (Fig. 7). The data in the texture memory can be copied to VBO or PBO and these data are sent to vertex shader for rendering. It is known that this method is used in [Zhou et al. 2007].

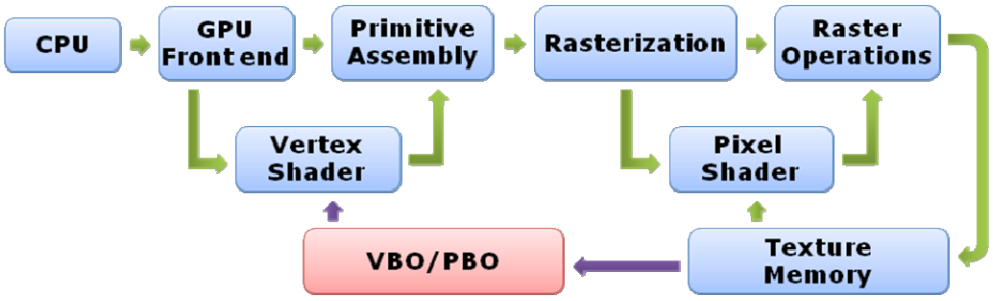

Fig. 7 Using VBO/PBO

There is another way for sending the result data to the vertex shader using vertex texture feature supported by some GPU architectures (Fig. 8).

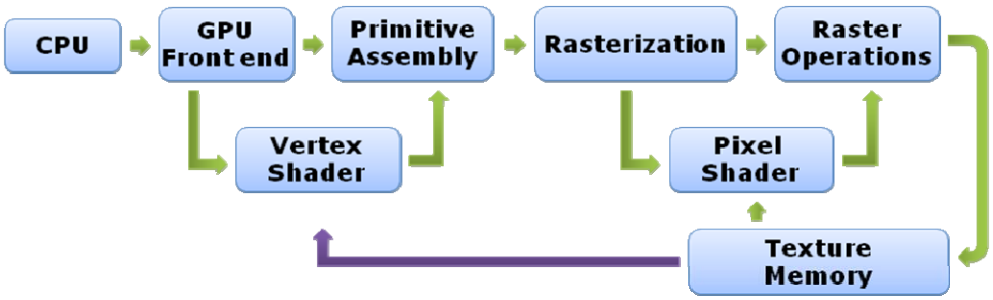

Fig. 8 Using Vertex Texture

We survey the methods of evaluating the subdivision meshes using VBO/PBO and vertex texture and implementation issues, and suggest the new architecture supporting temporary buffers for storing the indices and some information during managing data structure (Fig. 9). We also examine whether the suggested architecture is helpful or not for resolving data transaction bottle-neck including the adaptive subdivision cases.

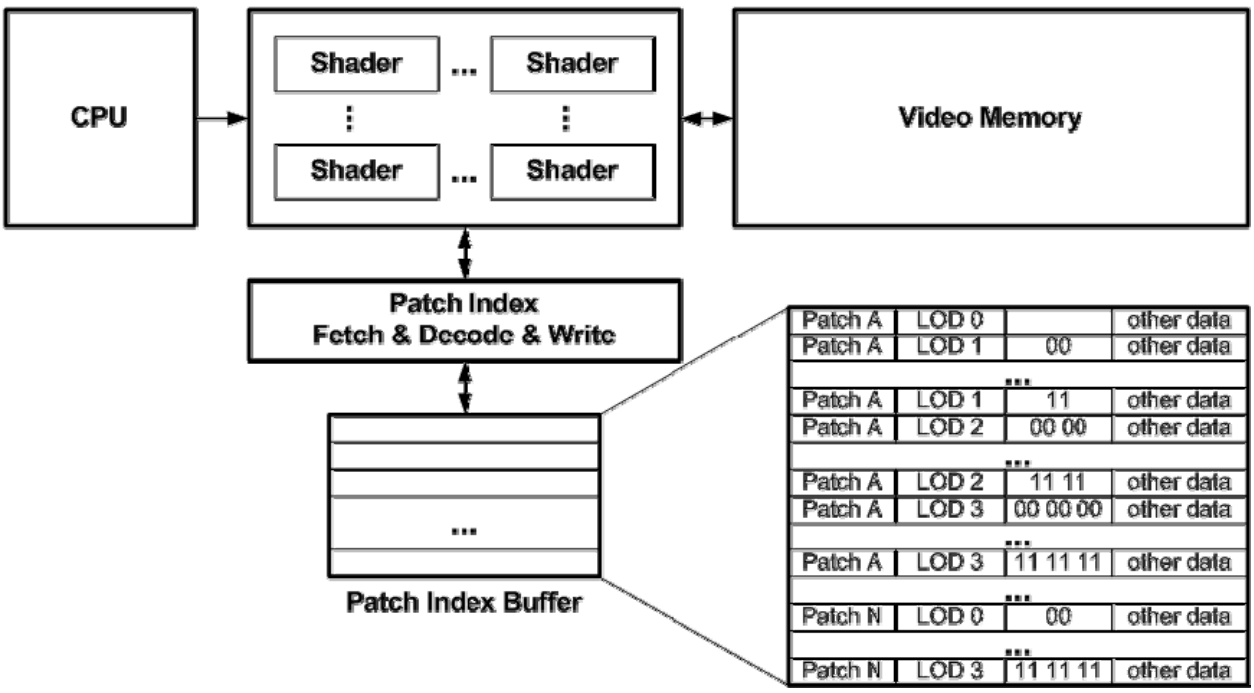

Fig. 9 New architecture including temporary buffer

References

1. D. Zorin et. al. 2000. Subdivision for Modeling and Animation, In Proceedings of SIGGRAPH '00 Course Notes.

2. Bolz, J. and Schröder, P. 2002. Evaluation of subdivision surfaces on programmable graphics hardware, http://www.multires.caltech.edu/pubs/GPUSubD.pdf.

3. Bunnel, M. 2005. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Reading, MA, ch. Adaptive Tessellation of

Subdivision Surfaces with Displacement Mapping.
4. Le-Jeng Shiue 2005. A Real-time GPU subdivision Kernel, In Proceedings of SIGGRAPH '05.
5. Minho Kim. 2005. Real-time Loop Subdivision on the GPU, In Proceedings of SIGGRAPH '05 Posters.
6. Kun Zhou. 2007. Direct Manipulation of Subdivision Surfaces on GPUs, In Proceedings of SIGGRAPH '07.
7. Simon Green, 2005. The OpenGL Framebuffer Object Extension, GDC 2005.