PARALLEL MESH DECIMATION WITH GPU

Jiří Vaďura
Doctoral Degree Programme (2), FIT BUT
E-mail: xvadur00@stud.fit.vutbr.cz

Supervised by: Přemysl Kršek
E-mail: krsek@fit.vutbr.cz

Abstract: This paper outlines possibilities of parallel GPU decimation of a polygon mesh. While conventional decimation algorithms process the mesh as a whole, our proposed approach splits the mesh into sub-sections that are processed independently by a GPU. The algorithm executes entirely in the GPU but requires some pre-processing. Advantages and disadvantages of this approach and its possible use in other mesh processing algorithms are discussed.

Keywords: decimation, mesh, GPU, OpenCL

1. INTRODUCTION

As computers became faster, large polygonal data sets began to be acquired and processed. Such large meshes can come from medical imaging devices after data segmentation is completed or from a high resolution 3D scanner. Modern graphics accelerators are able to display large meshes, but sometimes the size becomes impractical and needs to be reduced. When mesh size exceeds a couple of million triangles, real-time visualization becomes nearly impossible. Mesh simplification algorithms were designed to reduce number of polygons in the model, so that it can be manipulated easily. Standard decimation may take a long time to complete. In this paper, we present a new method that aims to significantly reduce required processing time and allows more user interactivity with the mesh.

Traditional mesh decimation algorithms were primarily designed with a single-thread CPU in mind. When simplification algorithms were first introduced [1], more than a decade ago, parallel computing systems were not readily available. These algorithms use complex data structures, heavy branching and work with all polygons at the same time. Until recently, computing languages didn't allow complex algorithms like this to be easily ported to GPUs. With introduction of languages like OpenCL, the only task that remained was to redesign the decimation algorithm to take advantage of parallel computing.

2. BACKGROUND

2.1. MESH DECIMATION

Primary goal of the decimation is to reduce the number of polygons in the mesh with minimal impact on its shape. One of the first algorithms dates back to 1993 when vertex clustering [1] technique has been introduced. The mesh and its bounding box are divided into a linear grid. A new vertex is calculated from all vertices within each cube. This vertex becomes part of the new decimated mesh.

Another approach is to use an iterative method that removes one small segment of the mesh at the time. Two basic methods are the vertex elimination and the edge collapse [2]. Vertex removal can be seen as a generalization of the edge collapse algorithm. Edge collapse assigns a collapse cost to each edge in the mesh and then attempts to eliminate an edge with the lowest cost. This ensures that decimation will have lowest possible impact on object's geometry. The metrics that assigns collapse cost can vary and many different techniques have been published. Garland and Heckbert [3] introduced Quadric error matrices that represent the cost. Another notable example is thes Error

volume metrics [4] that uses volume difference between original and decimated mesh as a metrics. All edge collapse algorithms can be described in three steps:

1. **Edge collapse cost calculation** – each edge is assigned a cost that represents possible geometry deformation if the edge would be removed.
2. **Find lowest collapse cost** – this can be done by either sorting all edges or looking just for the minimum.
3. **Remove the edge** – edge with the minimal collapse cost is removed from mesh, eliminating up to two triangles and slightly changing mesh geometry.

Vertex collapse method [5] is very similar to edge collapse, but the metrics is calculated for each vertex instead of an edge. When removing a vertex, one or more edges connected to it are removed similarly to the algorithm above.

2.2. GPU AND MESH SIMPLIFICATION

Hjelmervik and Leon in *GPU-Accelerated Shape Simplification for Mechanical-Based Applications* [6] published a paper where GPU's graphics pipeline is used to accelerate some parts of vertex collapse mesh decimation. They noticed that the most time consuming parts of the algorithm are sharp edge detection, detection of large angular deviation and half-edge collapse. They decided to offload these computations to the GPU while the whole mesh structure still remains in main memory and is handled by CPU. In all passes of the algorithm, complete 1-ring neighborhood is extracted for each vertex. This allows independent processing of the vertices. Vertices and their neighborhoods are encoded into a texture and processed by generic graphics pipeline. This can be done with relative ease because the most complicated parts of the decimation are still executed on the CPU side. Final speed-up of this approach was around 8× compared to a strict CPU implementation.

Another mesh simplification algorithm designed to run on a GPU was *Real-time Mesh Simplification Using the GPU [7]*, published by DeCoro and Tatarchuk. They have chosen vertex clustering as a base algorithm. The algorithm is taking advantage of a new addition to DirectX 10 compatible cards – the geometry shader. Usage of graphics library was advantageous in this case because the simplified mesh was supposed to be rendered and displayed after simplification is done. Vertex clustering has an advantage of not needing any surrounding triangle information. All triangles are processed individually with the only distinction being their cluster ID. In the first pass, the algorithm determines cluster ID for each vertex. In subsequent passes, error quadric is calculated for each triangle and that affects final vertex position for single output vertex of the cluster. This implementation has proven to be up to 20 times faster than a CPU version.

A major problem with generic algorithms running on the GPU is that memory access tends to be relatively slow compared to the CPU architecture. The whole graphics system is build for maximizing bandwidth and not for low latency. New generic computing languages such as OpenCL allow programmers to access functionality that has been hidden from the graphics pipeline. A notable feature is the shared memory that serves as an ultra fast, user accessible on chip memory. Other feature is the ability to access video memory arbitrarily, granting real random read/write access as it's known from the traditional desktop programming.

3. ALGORITHM

Our goal was to create a fully parallel implementation of the edge collapse simplification method that would utilize all benefits that modern graphics processors offer. By experimentation, we found out that dynamic work with triangle neighborhood is a very memory intensive task and we would be unable to achieve a performance that would outperform already published work. Not only that traditional mesh decimation doesn't parallelize well and would lead to low overall utilization of

graphics processors, but the number of random memory accesses though pointers would make the computing units even less effective.

Our algorithm is designed to run entirely in the graphics processor with maximum possible GPU utilization. This requires that the input mesh is split into smaller segments that can be processed independently to each other. This resembles vertex clustering algorithm, but the key difference is that instead of clustering all vertices into a single point, edge collapse technique is performed for each cluster.

3.1. First steps from CPU to GPU

Modern GPUs are not just an array of independent processors. Internal structure of a GPU is usually more complicated depending on the actual hardware manufacturer. But some commonalities are always present. Graphics processors are composed of number of smaller blocks that we will call multiprocessors. Each multiprocessor is completely independent on other multiprocessors. They are composed of memory address units, texture units, shared memory bank and an array of processors among other features. These processors within the multiprocessor share the resources and sometimes all of them need to execute the same instruction. Their internal structure varies, some are scalar (NVIDIA [8]) and others are effectively VLIW processors (AMD [9]).

Our parallel mesh decimation divides the input mesh into smaller segments that can fit into the shared memory of the multiprocessor. At present (2011), most common shared memory size is 32kB and that is enough to fit little more than 700 triangles. When splitting the mesh, border triangles are marked so that their edges cannot be removed to ensure that the mesh could be seamlessly assembled again. This creates first level of parallelism.

Because each multiprocessor contains a set of its own processors (approximately 8-20), secondary level of parallelism is required to perform the decimation itself. We've chosen OpenCL computing language that has support for all modern GPUs as the main development tool. OpenCL allowed us to differentiate individual multiprocessors and their processors though global and local IDs. Patches are transferred by the OpenCL to main GPU memory and then processed.

3.2. GPU edge collapse

Once OpenCL kernel is executed, patches are transferred from main GPU's DRAM memory to local memory of the multiprocessor. Each mesh patch (or segment) is usually assigned several dozens of threads that map to individual processors within the multiprocessor. An iterative process starts when memory transfer is completed:

1. **Cost calculation** is the first part of the decimation process. We've chosen the error volume metrics [4] because its calculation is well suited for GPUs. An edge can collapse into one of three points along its length: the two sides and center. The exact position is determined in this stage where error volume is calculated for each possible position. The cost also includes previous decimation in the area by adding cost from nearby vertices (see point 3). This ensures that whole mesh is decimated evenly. Cost calculation executes in parallel within the multiprocessor.
2. **Minimum cost selection** is the second step. Traditionally, all edges are continuously sorted so that the ones with the lowest cost can be easily accessed. Our implementation does not sort edges, but searches for the lowest cost in each pass. This is not ineffective as it may seem, the search is fully parallelized and takes only minimal amount of cycles.
3. **Edge collapse** is the final step where the edge with the minimal collapse cost is removed from the mesh and surrounding geometry is altered so that the topology remains consistent. When the edge is removed, its cost is transferred to the point of collapse. This ensures that an area is not over decimated. Most of the code is executed only by a single processor, but some parts use the whole multiprocessor.

After successful decimation of all segments, all simplified parts are merged back into a single geometry. This can be repeated until target triangle count is reached, but every pass requires new model division.

4. RESULTS

Our initial tests were designed to determine raw decimation speed of the GPU compared to the CPU. In these tests the mesh is divided in advance so that it simulated maximum usage of shared memory. Than we compared how fast can either algorithm reduce the mesh to 10% of its original size. The test was synthetic and did not use any real data. We observed an average speed up of around 100 times compared to the pure CPU implementation. However, it should be noted that this is not a result of a complete simplification as it does not include the time needed to divide the mesh. These numbers show that patch based mesh processing inside on-chip shared memory is able to achieve very high GPU utilization.

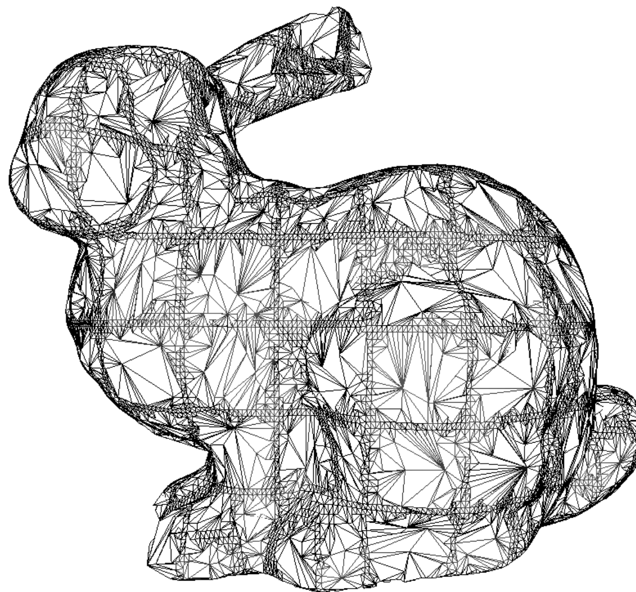

Figure 1: Decimated bunny model to 45% of the original. Patch borders are clearly visible and illustrate that final mesh is not homogeneous after a single pass.

Because the algorithm needs to preserve patch borders intact so that they can be seamlessly merged again, the active surface where decimation is performed becomes smaller. Algorithm loses efficiency because many triangles in the shared memory remain unmodified. Fig. 1 shows rectangular patches distributed over surface of a bunny. After one simplification step where total triangle count dropped below 50%, the border regions are clearly visible. Multiple passes of the algorithm will be able to produce more uniform results, but at the cost of additional mesh splitting. The implementation is not yet complete so no large data tests exist at the moment.

The error volume metrics is GPU friendly method and has good sharp edge preservation. If we take the whole mesh and process it as if it was a single patch, the output is visually comparable to the commonly used quadric matrix technique, but it still lacks fine detail. This should improve as our implementation progresses.

5. CONCLUSION AND FUTURE WORK

We have presented a new approach for a single-thread mesh simplification that takes advantage of a modern GPU. We used edge collapse method and error volume metrics as a basis for our algo-

rithm. We have demonstrated that a mesh can be processed in parallel with great efficiency even with decimation's high random memory access requirements. The cost calculation and minimum lookup have been both fully parallelized and the last stage of edge removal has been partially parallelized as well.

To divide the mesh into smaller patches, we've used simple grid method that is fast but tends to be problematic in densely curved areas of the model, because it does not produce a continuous piece of surface. An ideal solution would be to split the mesh in GPU prior to decimation using surface based algorithm. This process would be chained so that unmodified border areas can be decimated in subsequent passes. Presently, only about 700 triangles can fit into a standard shared memory that is 32kB large. Because significant portion of this memory needs to be allocated for triangles that are never modified, algorithm efficiency decreases. Any OpenCL implementation that would offer larger shared memory would also improve overall performance.

When compared to other GPU implementations, our solution should have better performance than a hybrid CPU/GPU algorithm, but it might be behind in terms of visual quality. The final speed-up can come up close to that of the vertex clustering, but with better looking output polygon mesh.

In the future, we would like to improve volume metrics to accommodate more criteria like edge length, surrounding geometry and improved handling of border regions. This would add more robustness to the edge collapse algorithm. We would also like to rework merging of the decimated patches back together so that more triangles in each patch could be decimated. Better implementation in this area should minimize impact of splitting and merging to the mesh.

ACKNOWLEDGEMENT

This article has been created with support from research grant MSM0021630528.

REFERENCES

[1] Rossignac, J., Borrel, P.: Multi-resolution 3D approximations for rendering complex scenes. Modeling in Computer Graphics: Methods and Applications (June 1993), 455-465.

[2] Garland, M., Shaffer, E.: A Multiphase Approach to Efficient Surface Simplification. In Proceedings of IEEE Visualization 2002, October 2002.

[3] Garland, M., Heckbert, P. S.:Surface simplification using quadric error metrics. Proceedings of ACM SIGGRAPH 1997, 209-216.

[4] Guéziec, A.: Surface simplification inside a tolerance volume, in Second Annual International Symposium on Medical Robotics and Computer Aided Surgery. 1995, pp. 132–139.

[5] Luebke, D.P., Reddy, M., Cohen, J.D., Varshney, A., Watson, B. ,Huebner, R.: Level of detail for 3D graphics. Morgan Kaufmann Publishers, 2003.

[6] Hjelmervik, J., Leon, JC.: GPU-Accelerated Shape Simplification for Mechanical-Based Applications. In Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007 (SMI '07). IEEE Computer Society, Washington, DC, USA.

[7] DeCoro, Ch., Tatarchuk, N.: Real-time Mesh Simplification Using the GPU. Symposium on Interactive 3D Graphics (I3D) 2007, pp. 6, April 2007.

[8] NVIDIA Developer Zone [online]. [2011] URL: <http://developer.nvidia.com/page/home.html>.

[9] AMD Developer Central [online]. [2011] URL: <http://developer.amd.com/gpu/ATIStreamSDK/documentation/Pages/default.aspx>.