Conceptual design of a programmable geometry generator

Jesús Gumbau
Universidad Jaume I, Spain
jgumbau@sg.uji.es

Miguel Chover
Universidad Jaume I, Spain
chover@uji.es

ABSTRACT

Current real-time graphics architecture lacks a method for procedural geometry generation inside the GPU, so that the limited bus bandwidth doesn't get involved. This document describes the conceptual design of a user-programmable geometry generator unit. This unit is capable of generating new geometry (vertices and indices) by processing a set of input data. This new geometry can be passed through the graphics pipeline to be rendered normally. This is done completely inside the GPU.

Keywords: Pipeline, buffer, GPU, shaders, vertices, fragments.

1 INTRODUCTION

Programmable parts of the present graphics hardware are designed to transform the properties of input primitives (vertices or fragments), through small user programs (also known as "shaders"). However, they are unable to generate new primitives inside the graphics pipeline. This work introduces the conceptual design of a hardware unit capable to generate geometry, inside the GPU, computed from an arbitrary set of input data. This unit will be called *General-Purpose Geometry Generator* (or *GPGG*). It is designed to run in a completely transparent way to the present design of the graphics pipeline, so that the generated geometry can be treated normally by the pipeline.

2 HOW THE GPGG WORKS

The GPGG can be defined as a programmable geometry generation unit designed to work completely inside de GPU. By having this unit inside the GPU, the generated data can be directly sent from the GPGG to the graphics pipeline. Thus, there is a minimal AGP/PCIE bus traffic and thus, no limitations due to the common bus bandwith bottleneck.

The input data that the GPGG can read is defined as a set of generic data that can be stored in any format. It is also possible that the unit doesn't require any kind of input data, so the input data stream is not mandatory. The input data stream is divided in a number of separated input data channels, implemented as different hardware buffers in the GPU. Note that we are talking about general input data, without defining any kind of format for it. This is due to the fact that the GPGG is a general-purpose geometry generator, each buffer can have its own data format.

The processing data unit must have the ability to iterate between input channels, compute the new geometry and generate an output. This should be a user-programmable processor specially designed to work this way. It needs an instruction set generic enough to do any kind of computation from input data and the ability to write into buffers stored in graphics hardware memory: output data channels.

Output data is the result of the geometry generation process. Unlike input data, output data forma must be either vertices or indices. Output data is dumped over memory buffers accessible from the 3D API, so that the result of the generation can be used to feed the graphics pipeline and then the geometry can be rendered. Output data is also divided in different data channels or streams, because the GPGG output data may need to be stored in different buffers: vertex coordinates, texture coordinates, normals, indices or vertex attributes for *vertex programs*.

3 INTEGRATION INTO THE PRESENT ARQUITECTURE

Integration is done by conceiving the geometry generator as a processor separated from the pipeline, binding its input and output channels to buffers stored in graphics hardware. This aproach allows this integration without modifications of the pipeline, just additions.

The graphics pipeline has been designed to transform primitives (render process), not to generate new ones procedurally. The scheme presented in this work respects this idea completelly, separating the *geometry generation stage* (GPGG) and the *geometry representation stage* (graphics pipeline). That makes this integration scheme conceptually cleaner.

Although the general rule is to communicate the GPGG and the pipeline through hardware buffers, the possibility to send automatically the output data to the pipeline is also interesting. This can save a lot of memory for applications that doesn't really need to

WSCG 2006 Short Communications proceedings, ISBN 80-86943-05-4
WSCG'2006, January 30 – February 3, 2006
Plzen, Czech Republic.

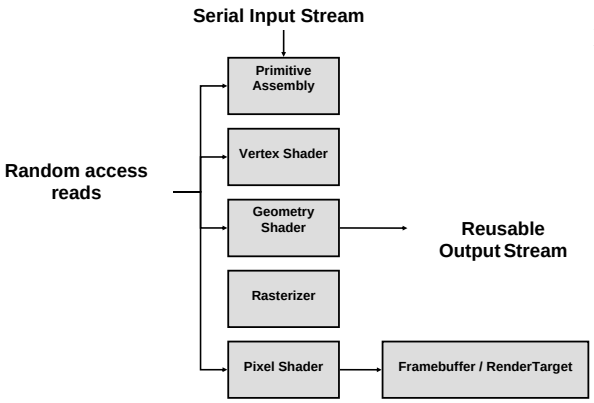

Figure 1: New pipeline proposed by Microsoft

store anything in graphics memory because they need to generate the geometry every time.

To add this capability, an output channel can be configured as a *bridge* to the pipeline, instead if a binding to a memory buffer.

4 GPGG APPLICATIONS

The following is a list of possible real world applications for the GPGG that shows its entire functionality. The GPGG is a perfect tool to calculate the shadow volumes for the *Stencil Shadowing* technique [1] entirely inside the GPU. Every continous LOD technique would benefit from the GPGG as it allows to calculate in real time the triangle list (indices) that define a mesh in an arbitrary level of detail.

Terrain generation [3] could use the GPGG to generate the piece of terrain seen at a given time and a camera position from a heightmap. Displacement mapping coulg be implemented inside the GPU using a similar approach. A surface tesselator [4] could be programmed into a geometry program, setting as input data the control points that describes a bicubic surface. The GPGG could accept as parameters the coeficients of an equation that represents a volume of an object and apply the Marching Cubes method to approximate a polygonal surface.

The unit could be configured to instantiate the geometric primitives for plant generation, by specifying a string, derived from an L-system, as a GPGG input.

5 CONCLUSIONS

Implementing a geometry generator in the graphics hardware has a large amount of benefits, as previously explained (see section 4).

Having the GPGG separated from the pipeline is beneficial in terms of parallelism: while one unit is generating new geometry, the other one can process the already generated geometry.

Even the DirectX approach (figure 1) [2] has multiple *geometry shader* units running in parallel. The same approach could be taken in a hardware implementation of the GPGG by setting up multiple GPGG units running in parallel, similar to the multicore CPU systems do.

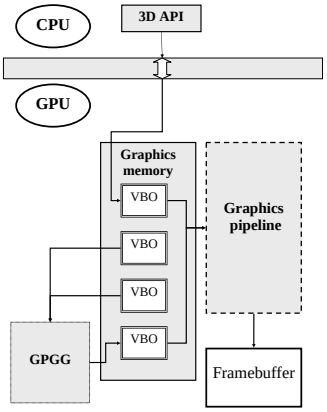

Figure 2: GPGG integration scheme into the current graphics arquitecture.

This work introduces a conceptual design of a hardware geometry generation unit that operates in a completelly transparent way to the current pipeline design.

The changes introduced by Microsoft to implement a geometry generator into the pipeline for DirectX, forces a completelly redesign of the pipeline. In contrast, the GPGG design introduces no changes to the traditional pipeline, only additions to the hardware graphics and the APIs. Moreover, the possibility to access randomly the input data makes it a more intuitive programming model, in contrast to the per-primitive pipelined one proposed by Microsoft.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Ministry of Science and Technology (TIN2004-07451-C03-03), the European Union (IST-2-004363) and FEDER funds.

REFERENCES

[1] Crow, F., *Shadow Algorithms for Computer Graphics*, Computer Graphics, 1977.

[2] Rudolph B., Glassenberg S., *DirectX and Windows Vista*, PDC, 2005.

[3] Losasso, F. and Hoppe, H., *Geometry clipmaps: Terrain rendering using nested regular grids*, Siggraph, 2004.

[4] Sfarti, A., *Bicubic surface rendering*, U.S. patent, #6.563.501.