Conceptual design of a programmable geometry generator

Jesús Gumbau , Miguel Chover

Universitat Jaume I, Castellón, Spain

Abstract

This document describes the conceptual design of a user-programmable geometry generator unit. This unit is capable to generate 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.

Key words: 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 not able to generate new primitives inside the graphics pipeline.

This work is an intend to introduce 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 completely transparent to the present design of the graphics pipeline, so that the generated geometry can be treated normally by the pipeline. This issue will be largely explained along this paper.

Email addresses: `jesus.gumbau@anubis.uji.es` (Jesús Gumbau), `chover@lsi.uji.es` (Miguel Chover).

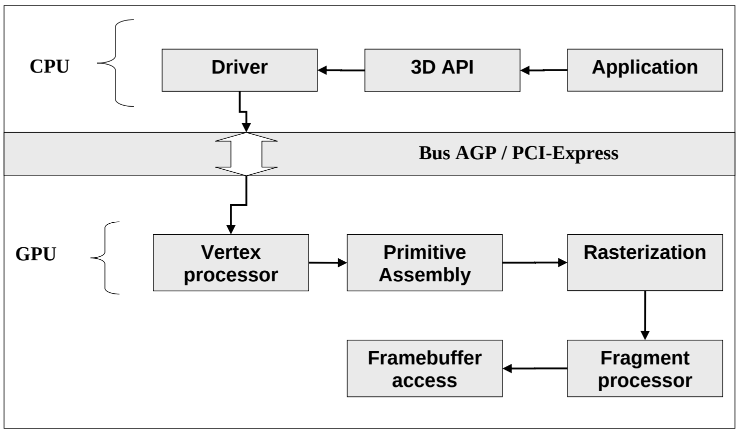

Figure 1. Graphics pipeline

2 Previous work

Some proposals for geometry generators have been published, but not in the same way the GPGG has been designed. One of them is a pipeline extension (11) that proposes a new first stage of the pipeline. This stage acts like a bicubic surface tesselation unit, by converting a set of input control points defining a surface to a set of vertices aproximating that surface. That output tesselated geometry is then sent to the normal graphics pipeline.

That unit can be considered as a geometry generation unit, as it generates vertices from a set of control points. However, it uses a fixed generation function: generates the geometry that aproximates the input bicubic surface represented as control points. In contrast, the GPGG is conceived as a *programmable* geometry generation unit. Moreover, it is also designed to be as *general-purpose*, meaning it is be able to generate geometry to accomplish a large set of tasks (view section 5). This will be more explained in section 3.

In the other hand, Microsoft is designing a new geometry generation unit for the next DirectX version (2). The design proposed by this company is an integration of a geometry generation unit into the graphics pipeline. In other words: extending (changing) the present pipeline so that it can be able to generate new geometry from the GPU.

The original pipeline itself is designed to transorm the input primitives. The vertex shader itself, is designed as a one-vertex to one-vertex transformer, that works completely on the GPU (see figure 1), it can't generate new vertices. So adding the ability to create new geometry requieres a completely restructuration of the original pipeline by introducing lots of changes to it. Figure 3 shows this changes. In the conclusions (section 7) this design will be compared to the GPGG aproach to be analyzed more deeper.

An indirect way to implement a simple geometry generator in the current hardware can be accomplished by using the “render to vertex” method. This is done by performing a render to texture (stored in a Pixel Buffer Object) and reinterpreting that buffer as a Vertex Buffer Object. So, The grahics pipeline itself can “generate” new geometry that is stored in the graphics hardware itself. However, this method will not be taken into account as it is an indirect consequence of the OpenGL API.

3 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. So, there is a minimal AGP/PCIE bus traffic and thus, no limitations due to the common bus bandwith bottleneck.

This will be detailed separately in subsections 3.1, 3.2 and 3.3.

3.1 Input data

The input data 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 doens’t need any kind of input data, so the input data stream is not mandatory.

The input data stream is divided in a number of input data channels. So, these data can be separated into these channels in a logic manner to make the data processing more easy or efficient. These input data channels are implemented as different buffers in GPU memory.

Note we are talking about general input data, without defining any kind of format for it. This is due to the fact the GPGG is a general-purpose geometry generator, so it can generate geometry for any kind of applications and thus, the is no way to define an input data format. So, input data can be expressed in any kind of format. There is only one restriction to this: an entire buffer must store the input data in the same format (having multiple data channels for multiple format data).

3.2 Data processing and generation

The processing data unit must have the ability to iterate between input channels, compute the new geometry and generate an output. This physically should be a user-programmable processor specially designed to work this way. It needs a generic enough instruction set to do any kind of computation from input data (which can be in a very flexible format) and the ability to write into buffers stored in graphics hardware memory: output data channels (see section 3.3):

3.3 Output data

Output data are the result of the geometry generation process. Unlike input data, these are not considered unformatted data, but vertices and indices. These output data are dumped over memory buffers accesible from the 3D API, so the result of the generation can be used to feed the graphics pipeline so that 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*.

4 Integration into the present arquitecture

The goal is to integrate this technology into the present graphics hardware making the fewer changes possible to it, and looking for the best integration scheme. This is done by conceiving the geometry generator as a processor separated from the pipeline binding its input and output systems to buffers stored in graphics hardware. This aproach allows this integration without any modification to the pipeline: we only add, do not change anything.

The GPGG is designed externally to the graphics pipeline because the integration into the pipeline requieres a completely new redesign (see figure 3), and this changes are not really needed, as this work stands for. Conceptually, the pipeline is designed to transform input primitives, that are generated previously in software (CPU side). The scheme presented in this work completelly respects this idea, separating the *geometry generation stage* (GPGG) and the *geometry representation stage* (graphics pipeline). That makes this integration scheme conceptually clenaer.

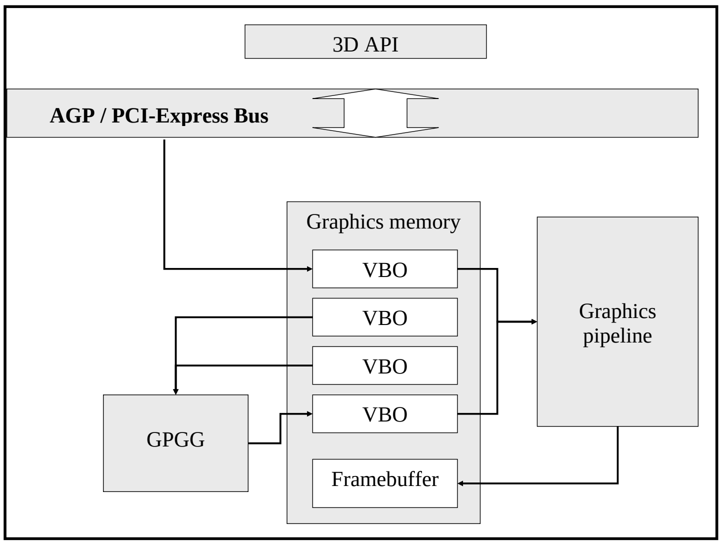

Figure 2. GPGG integration scheme into the current graphics arquitecture.

By making this separation, we get an inherent direct benefit: Generated geometry can be rendered multiple times without having to generate it every time, as it is stored in a memory buffer.

A comparative analysis is shown in section 7 between a geometry generation design integrated into the pipeline (the proposed by Microsoft) and the one designed for this article, which is separated from the pipeline.

4.1 Direct sending to the pipeline

Though 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 could be interesting. This can save a lot of memory for applications that doesn't really need to store anything in graphics memory because they need to generate the geometry every time. Or because the cost of generating that geometry is fewer than the possible storing cost.

To add this interesting capability, an output channel can be configured as a *bridge* to the pipeline, instead as a binding to a memory buffer. Having the GPGG separated from the pipeline is beneficial one more time, because that way the GPGG and the graphics pipeline can work in parallel: while one unit is generating new geometry, the other one can process the already generated geometry.

5 GPGG applications

The following is a list of possible real world applications for the GPGG that shows its entire functionality.

Some real time shadow rendering methods (as *Stencil Buffer*(3)(6) technique) need to build a shadow volume to perform its calculations. That geometry is often recalculated in real time when the object/light that casts the shadow are dynamic. The GPGG is a perfect tool to recalculate this volumes all inside the GPU. In this case, GPGG inputs could be the geometry of the shadow caster (which is often stored in a VBO) and the light position. Using this data, the shadow volume for a caster object could be generated.

Every continous LOD technique (9) 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. Any software LOD algorithm can be programmed into this unit to perform this task.

Terrain generation (8) could use the GPGG to generate the piece of terrain seen at a given time and a camera position. The geometry generation unit should be fed with an input data that represents the terrain to be generated (i.e. a heightmap).

A surface tesselator could be programed into a geometry program, setting as input data the control points that describes a bicubic surface. So the geometry generator can generate all the triangles needed to aproximate that surface. As an optional parameter, the GPGG could take the level of detail of the terrain to be generated. Not only bicubic surfaces could be tesselated this way, but low polygonal density models, similar to the ATI *TruForm*(1) technology.

Displacement mapping (4) (shown as a geometry tellesation technique from a heightmap) could be entirely implemented in hardware this way, as GPGG programs can access to texture data.

Marching Cubes (7) is a general solution to approximate polygonal surfaces to arbitrary volumetric objects. The GPGG could accept as parameters the coeficients of an equation that represents a volume of an object.

GPGPU (5) is the ability to use the GPU as a general purpose processor, because it has becamed a very powerful and programmable piece of hardware. So the GPGG could be a perfect generic data processor: it has a powerfull GPU-integrated I/O system and the fact it is designed to generate geometry from “unformatted” data, makes it a more general purpose computation unit (with vector extensions) instead of a completely vector-oriented unit (as a vertex or fragment program).

By specifying a string, derived from an L-system, as a GPGG input, the unit could be configured to instantiate the geometric primitives for plant generation. The GPGG should iterate over the input data, interpreting them in the right way in order to generate the geometry.

One thing that makes the GPGG more powerfull is its design itself: being an entity separeted from the graphics pipeline instead of an extension of the pipeline itself makes it totally independent from it. So, this design is easily portable to other graphic architectures.

The article (10) defines a new architecture for geometry representation based on a raytracing algorithm (12). This method does not have the concept of graphics pipeline who process vertices and transform them in potential pixels. However, with small changes, the GPGG could be integrated into this architecture.

6 Discussion

Experimenting with a non-existing architecture is pretty difficult, so there aren't real results we can take from it. However, a theoretical dissertation can be made about the benefits the GPGG can offer against the current architecture.

Examples of benefits this technology can supply are shown in section 5, but now its time to talk about its possible disadvantages. One of them concerns to execution cost of a typical GPGG program, as all the previously possible applications needs relatively complex programs who must iterate over a big amount of input data and make complex calculations on them.

As a general rule, the execution time inverted on the GPGG to render a scene can be compared to the execution time inverted on the vertex and fragment units. The idea is: to became the GPGG the bottleneck of the system by generating some parts of a scene, it must be more computational expensive than the cost of processing the vertices and fragments of the same scene. If we find the vertex and fragment processing can be more expensive in a typical scene than using the GPGG to generate some parts of it, so the GPGG has a rasonable cost.

Thinking in pixels, to render a scene at a resolution of 1024x768, a minimum of 786.432 pixels are being processed each frame (with an unreal 0% pixel redrawal). So, to became the bottleneck, a typical GPGG execution per frame should be more computationally expensive than 786.432 typical fragment programs.

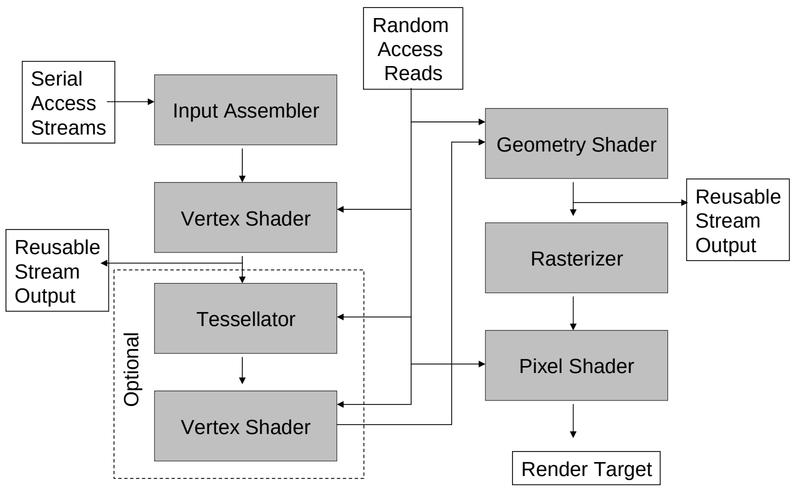

Figure 3. New pipeline as proposed by Microsoft

Even the current graphics hardware has multiple vertex and fragment shader units runing in parallel (lowering the cost of vertex and fragment processing), the same approach could be taken in a hardware implementation of the GPGG by setting up multiple GPGG units running in paralel, in a similar way multi-CPU nowadays systems do.

Thereby, the GPGG is a tool that complements the graphics pipeline architecture by adding an acceptable computing cost (relative to the fragment shader unit costs).

7 Conclusion

By implementing a geometry generator in the graphics hardware a large amount of benefits emerge, as previously explained (see section 5).

This work introduces a conceptual design of a hardware geometry generation unit that operates in a completelly transparent way for the current pipeline design. The pipeline is an excelent design for a 3D primitive transformation system, but introducing the ability to generate new primitives into the pipeline is not trivial due to the way it operates.

The changes introduced by Microsoft to implement a geometry generator into the pipeline for DirectX (2), forces a completelly redesign of the pipeline. This contrasts with the GPGG design that introduces no changes to the traditional pipeline, only additions to the hardware graphics and the APIs. More over, theoretically the Microsoft approach is not more powerfull than this as both designs can do the same thing. The difference is that the GPGG approach is simpler to integrate into the current graphics architecture and more simple and efficient to use.

This simpler and efficient way to use it is due to the global scope of input data (one program over all input data), against the pipelined programming model proposed by Microsoft (one program for each input element). This makes easier and efficient the access to input data and the iteration over them.

The only disadvantage the GPGG could have is the fact that the generation stage is executed before the vertex processing stage. So the vertex shader unit must operate over any new vertex created on the geometry generator. This may be a problem whether the GPGG generates large amounts of geometry but the solution to this is quite easy: the geometry generator can generate each vertex with the final information on them, so there is no need to process that vertex in the vertex shader stage. This is done by simply specifying a minimum vertex program that outputs untransformed vertices, so its execution cost has no impact on performance.

8 Future work

This work is a first step by introducing a conceptual design og a geometry generation unit. From this point, the future work go towards defining a complete instruction set (as the GPGG is a programmable processor) and the implementation the the GPGG into a real open 3D API.

8.1 OpenGL extension definition

OpenGL can be choosed to implement all the GPGG functionality on it, as it is an open API. A new OpenGL extension can be defined to integrate this concepts into the API. This extension should be a built over other extensions that partial funcionality we want to extend.

8.2 Pixel Buffer Object compatibility

This is a possible extension to the GPGG concept of geometry generation. As vertex and pixel data can be stored in graphic hardware memory, the GPGG could be extended to make it capable to generate images (textures) in addition to vertex data. This is theoretically easy to integrate as PBO are data buffers just as VBO are, so that could be completelly transperent to the GPGG.

References

[1] ATI, *TruForm Technology*, http://www.ati.com, 2001.

[2] Blythe, D., *Windows Graphics Foundation*, Microsoft Corporation, 2004.

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

[4] Doggett, M. and Hirche, J., *Adaptive view dependent tessellation of displacement maps*, Siggraph, 2000.

[5] Harris, M., *SIGGRAPH 2004 GPGPU Course*, Siggraph, 2004.

[6] Hornus, S., et al., *ZP+: Correct Z-pass Stencil Shadows*, Artis, 2005.

[7] Lorensen, W. E. and Cline, H. E., *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Computer Graphics, (Proc. SIGGRAPH), 1987.

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

[9] Luebke, D., et al., *Level Of Detail For 3D Graphics*, Morgan Kaufmann, 2002.

[10] Schmittler, J., *Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip*, Graphics Hardware, 2004.

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

[12] Whitted, T., *An Improved Illumination Model for Shaded Display*, Communications of the ACM, 1980.