A Micro-programmable Graphics Architecture

Dominic Mallinson
Director of Technology, R&D Sony Computer Entertainment America
<dominic_mallinson@playstation.sony.com>

Introduction

During this session, I want to examine the benefits of a micro-programmable graphics architecture. I will consider the graphics pipeline and where hardware and software techniques can be applied. After looking at the pros and cons of hardwired hardware vs. CPU vs. micro-coded coprocessors, I will discuss procedural vs. explicit descriptions. Finally, I hope to demonstrate some of the principles by showing examples on a specific micro-programmable graphics system; PlayStation®2.

3D Graphics Pipeline

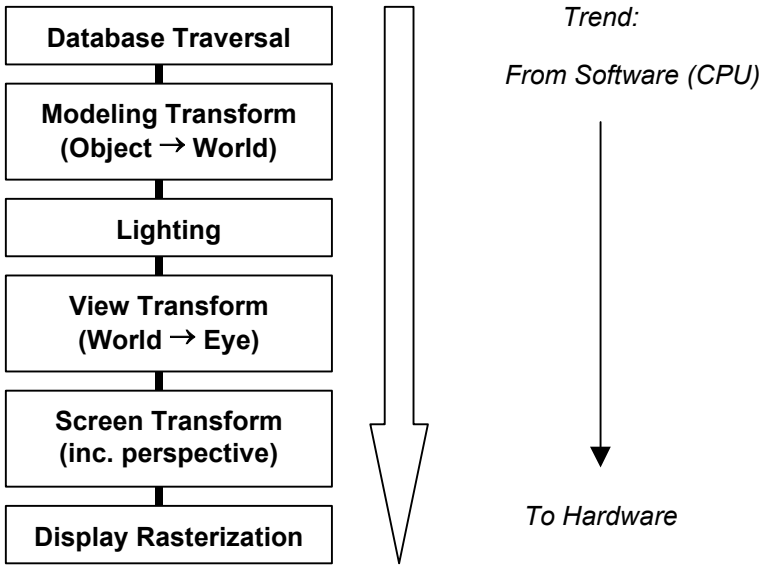

Figure 1. The classic 3D graphics pipeline.

Figure 1 shows a typical 3D graphics rendering pipeline. Notice the trend that has occurred over several years where more and more of this pipeline has been finding its way into special hardware. In the early days of CG, the display was memory mapped and the CPU wrote pixels directly into the frame buffer. In this case, the CPU was responsible for every step of the pipeline. Hardware implementations begun with the rasterization of pixels on the display and have gradually been rising up the pipeline with the latest hardware able to take geometrical descriptions in terms of vertices, normals and polygons and perform all the transformations, lighting, texturing and rasterization in hardware.

By its very nature, a hardware implementation of any section of the pipeline will be less flexible than its software counterpart. For example, most hardware rasterization is limited to a gouraud shading model and cannot be re-tasked to do a per-pixel phong for instance. The reward for this loss of flexibility is greater performance for a certain class of rendering operations.

Micro-Programmability

In this paper, I'm going to use the word micro-programmable to describe coprocessors which operate in parallel with the main system CPU which can have their own local programs and data.

The key coprocessor features that I am highlighting are :-

Local instruction memory
Local data memory
Parallel (non blocking) operation with the CPU and other system components
A direct connection to display rasterization

I'm going to concentrate on the PlayStation®2 architecture which has hardwired rasterization and micro-programmable vector processors for transformation, lighting and rendering setup.

Transition from CPU to graphics coprocessors

In the vast majority of cases, the scene database and high level world description are best managed by a general purpose CPU. The data is often highly application specific and the algorithms and data sets too big and too complex to implement on hardware.

However, a little lower down the pipeline, the data starts to become more manageable by hardware and the graphics operations start to become more constrained and consequently easier to implement in hardware or micro-program. Certainly transformation and lighting fall into this category. There are other procedures such as skinning, adaptive tessellation, level of detail, higher order surfaces and so on which may be too complex and too application specific for hardwired hardware implementations. It is in this area that a micro-programmed coprocessor is most useful.

Pros and Cons of Micro-Programmability

When compared to a fixed hardware implementation, micro-programmed hardware has the following pros and cons :

Pros : (Micro-program vs. hardwired hardware)

Flexibility and generality. Slight variations and parameterizations of an algorithm are easy. A wider set of needs can be addressed.

Tailored (and optimized) to application and data set whereas hardwired hardware tends to force application functionality to fit the available hardware.

Multiple use. Micro-programmed hardware can be tasked with performing very different tasks by using a different set of microcode.

Can reduce bus-bandwidth. Hardwired solutions typically require a fixed data format to be passed in. Often this is more verbose than a specific object or application requires. Microcode allows the data set to be passed to the hardware in a more "native" format. This typically reduces the amount of data passed across the bus from main memory to the graphics coprocessors.

Non standard effects. e.g. non-photorealistic rendering, distortion, noise, vertex perturbation etc.

Cons: (Micro-program vs. hardwired hardware)

Usually slower. Typically, dedicated hardware can get better performance than microcoded hardware. However, this is not always the case. Its worth noting that certain combinations of graphics algorithms share intermediate calculations and data. In a hardwired solution it may be necessary to redundantly repeat these intermediate calculations whereas a microcode implementation may recognize these special cases and optimize appropriately.

Performance Issues : Procedural Descriptions and Micro-programmability

In games, performance is always important. There are never enough resources to go around. As previously mentioned, procedural descriptions are economic on memory usage. However, processing performance is dependent on both algorithm and architecture implementation.

For example, a procedural algorithm may be so algorithmically complex that it will always be slower than using an explicit (polygonal) description.

Assuming a procedural technique is to be used, then the following performance characteristics are likely:

If the technique happens to be implemented directly in silicon in hard-wired hardware, it will almost certainly be the fastest implementation.

In most cases, the technique won't be hardwired into hardware and it will need to be programmed either on the CPU or on a microcoded coprocessor. In this case, assuming the same clock speed for CPU and coprocessor, the coprocessor will almost always be faster. This is because the coprocessor will have hardwired elements of common graphics routines. In addition, it is likely that the CPU can be undertaking other tasks in parallel with a coprocessor, therefore increasing the effective speed of the whole application.

Swapping Microcode

The breadth of hardwired hardware functionality is limited by the number of gates on the silicon. While this is ever increasing, it is not practical to layout every possible graphics algorithm in gates. Microcoded functionality is limited by microcode instruction space, but as with all programmable systems, this instruction space can be reused. In the case of PlayStation®2, it is possible to reload the entire microcode many times a frame allowing for a huge set of algorithms.

Explicit vs. Procedural Content

Lets consider the description of the objects and the world that a 3D graphics pipeline will be rendering. For the purposes of this session, I will split geometric/object descriptions into two categories ; procedural (implicit) and explicit (low level).

In most cases, 3D hardware ultimately renders triangles from vertices, normals and textures. When creating the description of an object in a game, one approach is to describe that object explicitly in terms of a list of vertices, normals, texture UV co-ordinates and polygons. Lets call this “explicit”.

A second approach is to abstract the description of the object to a higher level than the vertices and polygons that will ultimately be rendered. The low level vertices and polygons are still required, but they can be generated procedurally by an algorithm that takes its description of the world in a higher level form. Lets call this “procedural”. In a procedural description, some of the information of what an object looks like is transferred from the data set to the algorithm.

There are various levels of procedural descriptions. The simplest types would be surface descriptions using mathematical techniques such as Bezier surfaces which can be described in terms of a few control points instead of many vertices and triangles. An extreme procedural example might be some complex algorithm for drawing a class of objects such as trees, plants or landscapes.

Pros and cons of Procedural vs. Explicit

On the positive side, parameterized or procedural descriptions can significantly reduce the data storage requirements. This is always important in games, but even more so on consoles where there is often a smaller amount of RAM available than on a typical PC. Not only does this reduction in data size help on memory usage, it also improves the transfer of data on the bus from main memory or CPU to the graphics unit. Bus bandwidth issues can often be a bottleneck in graphics architectures and so this element of procedural descriptions is important.

On the negative side, some geometry is just not suitable for procedural description. At times, an explicit description is necessary to maintain the original artwork. In some cases, an explicit description is needed to maintain control of certain aspects of the model.

One area where there is both a positive and a negative effect is in animating geometry. Some procedural descriptions naturally lend themselves to effective animation. For example, a subdivision surface will usually work well for an object that has a bending or creasing deformation applied. A counter example might be breaking or cutting a hole in a Bezier surface. This might be difficult to accomplish when compared to performing the same operation with a polygonal mesh.

PlayStation®2 Graphics Architecture

The figure shows the architecture of the PS2 hardware for graphics. The system is essentially split into 5 components.

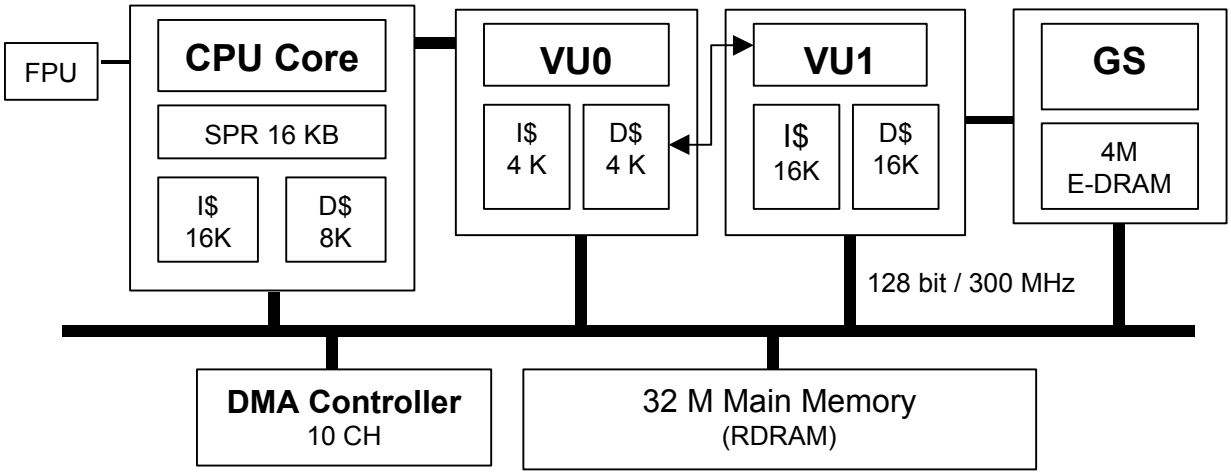

Figure 2. PlayStation®2 Graphics Architecture

PS2 : CPU

The CPU is a general purpose MIPS variant CPU with its own FPU, 128 bit SIMD integer multimedia extensions, ICACHE, DCACHE and a special on-chip “Scratch pad” memory of 16K.

PS2 : Vector Units

The vector coprocessors are SIMD floating point units. They perform multiply/accumulate operations on 4 single precision floats simultaneously with single cycle throughput. In parallel with the FMAC operations, the vector units perform single float divide, integer and logic operations.

PS2 : Vector Unit 0 (VU0)

The VU0 has 4K of instruction RAM and 4 K of data RAM.

This unit is closely coupled to the CPU and can be used as a MIPS coprocessor, allowing the CPU instruction stream to directly call vector unit instructions.

The VU0 can also be used as a stand-alone, parallel coprocessor by downloading microcode to the local instruction memory and data to its data memory and issuing execution instructions from the CPU. In this mode, the CPU can run in parallel with VU0 operations.

PS2 : Vector Unit 1 (VU1)

The VU1 has 16 K of instruction RAM and 16 K of data RAM.

This unit is closely coupled to the Graphics Synthesizer and has a dedicated bus for sending primitive packet streams to the GS for rasterization.

The VU1 only operates in stand-alone coprocessor mode and has no impact on CPU processing which takes place totally in parallel. Downloading of microcode, data and the issuing of execution commands to VU1 are all accomplished via the DMA Controller.

PS2 : Graphics Synthesizer (GS)

This unit is responsible for rasterizing an input stream of primitives. The GS has a dedicated 4M of embedded (on-chip) DRAM for storing frame buffers, Z buffer and textures. This embedded DRAM makes the GS incredibly quick at both polygon setup and fill rate operations. The GS supports points (dots), triangles, strips, fans, lines and poly-line and decals (sprites). Fast DMA also allows for textures to be reloaded several times within a frame.

PS2: DMA Controller (DMAC)

The DMA controller is the arbiter of the main bus. It manages the data transfer between all processing elements in the system. In terms of the graphics pipeline, the DMAC is able to automatically feed the VU1 with data from main system DRAM with no CPU intervention allowing the VU1 to get maximum parallel operation.

When the VU1 accepts data from the DMA, it has another parallel unit which can perform data unpacking and re-formatting operations so that the input stream is in the perfect format for VU1 microcode operation. This unit also allows for VU1 data memory to be double buffered so that data can be loaded into the VU1 via DMA at the same time as the VU1 is processing data and sending primitives to the GS.

Examples

So much for the theory. How does it work in practice? I want to demonstrate some of the points that I have been making by showing some example applications running on PlayStation®2.

Example: Higher Order Surfaces

Some curved objects can be best described by surfaces rather than polygons
The parameters describing the surface often require less data than the corresponding polygons that are rendered. Surface descriptions also allow for easier automatic level of detail.

I will show two different types of surfaces ; Bezier and Subdivision.

Example: Bezier Surfaces on PS2

The demo application shows around 10-16 M polygons/sec being drawn to render objects described by Bezier patches.

The VU1 has a micro-program which creates, lights and renders triangle strips directly from Bezier blending matrices. The blending matrices can either be pre-calculated and sent to the VU1 or the VU1 can calculate them on the fly from control points. Because of the pipelined nature of the vector units, calculating the matrices on the fly does not exact a massive penalty on performance.

By using Bezier surfaces, suitable geometry can be described using much less data. This reduces the memory storage in main RAM, lowers the system bus traffic of data from the CPU to the graphics coprocessors and also maintains a very high performance. The high performance is achieved because the microcode is able to create very efficient triangle strips thus sharing more vertices, consequently transforming and lighting less data.

Demonstration written by Mark Breugelmans, Sony Computer Entertainment Europe.
e-mail: Mark_Breugelmans@scee.net

Example: Subdivision Surfaces on PS2

The demonstration application shows the rendering of a non-modified Loop subdivision surface on PlayStation®2. [Refs 1,2,3,4]

Subdivision surfaces offer a modeling paradigm which combines the advantages of polygon modeling (precise control, local detail and arbitrary topology) and patch modeling (smoothness).

The CPU sends the VU1 the vertices, colors and texture co-ordinates and the VU1 performs the subdivision, calculating the normals if needed for lighting. The VU1 then efficiently renders the polygons resulting from subdivision.

The implementation shown, which is not fully optimized, has 4 parallel lights plus ambient, texture and fogging. Even without full optimization, the renderer is exceeding 4 million polygons per second.

As with other “surface” schemes, the amount of data required to describe the geometry is generally less than a polygon mesh and allows for level of detail to be controlled by the amount of subdivision taking place.

Demonstration written by Os, Sony Computer Entertainment Europe (Cambridge)
e-mail: os@scee.sony.co.uk

Example: Terrain Rendering on PS2

The demonstration application shows an interactive fly-through of a potentially infinite procedurally generated landscape. The terrain is calculated using noise functions as are all the textures used.

In the demo, the CPU first calculates the view frustum and works out some view dependent sampling for the terrain that is within the view. Essentially this is a level of detail which is adaptive on a tile basis. The CPU is using a metric which tries to keep the size of rendered triangles roughly the same for both distant and close parts of the terrain. It also prevents non-visible parts of the terrain from being calculated.

The CPU then hands off the terrain sampling parameters to the VU0 and VU1 coprocessors which run several micro-programs to accelerate various calculations and ultimately perform the polygon lighting and rendering.

The VU0 is used for fast generation of the terrain using noise functions. The VU1 is responsible for building highly efficient triangle strips in local coprocessor memory which are then lit and transformed. As the VU1 is constructing the triangle strips in an algorithmic fashion, it knows how best to share vertices which results in an effective saving of 46% less vertices compared to a simple polygon mesh. This reduces the amount of transform and lighting calculations keeping performance high.

In this demo, the VU0 microcode has to be swapped during a frame because of the number and complexity of operations it performs.

The terrain in this demonstration can be rendered with adequate performance remaining to add game play. If the landscape in this demo was stored as a polygon mesh, it would require huge amounts of memory and a complex streaming architecture. Instead, the procedural terrain takes a handful of parameters and about a 2M footprint. This frees up machine resources for the storage of other game models and environment which do not lend themselves to procedural generation.

Demonstration written by Tyler Daniel, Sony Computer Entertainment America R+D
e-mail: Tyler_Daniel@playstation.sony.com

Example: Particle Rendering on PS2

The demonstration application shows a non-standard rendering of a 3D model. In this case, the model (with approximately 40,000 triangles) is rendered only with dots which are themselves a particle simulation.

Instead of rendering triangles, the microcode in the coprocessors renders dots. The VU0 and CPU calculate the motion of the dots using a particle simulation. The particles are emitted from the vertices of the original model along the vertex normals of the model. The effect looks somewhat like fire or smoke. The particles are then sent to the VU1 to be transformed and rendered.

This is a good example of how special microcode can replace a standard triangle rendering to give a different appearance to in-game models. In this case, the model remains the same and the act of swapping the microcode determines whether it is rendered as triangles or as particles.

Demonstration written by Stewart Sargaison, Sony Computer Entertainment America R+D
e-mail: Stewart_Sargaison@playstation.sony.com

Example: Normal Perturbation on PS2

The demonstration application shows another non-standard rendering of a 3D model. In this case, the model (with approximately 40,000 triangles) is rendered with the vertices perturbed by a procedural amount along the normal at that vertex. This gives a kind of swirling effect. As with the previous example, the model is described in a standard way and by swapping in a different rendering microcode, this effect can be accomplished.

An ‘invisible’ light is used in the scene. The VU1 calculates the cosine of the normal with the invisible light and uses this as the amount by which to extrude or perturb the vertex from its original position. The VU1 then continues with the “standard” rendering operations except that it combines the render using a blending operation with the previous frame.

Due to the pipeline of the VU1, it is possible to completely hide the extra rendering operations for this demonstration which means that there is no performance impact for this effect vs. standard rendering.

Demonstration written by Stewart Sargaison, Sony Computer Entertainment America R+D
e-mail: Stewart_Sargaison@playstation.sony.com

Example: Post processing effect on PS2

The demonstration application shows a post-processing effect applied to the frame buffer after a first pass 3D rendering stage. In this case, the first stage render takes place to an off-screen buffer which is then applied as a texture onto a microcode generated mesh. The mesh is then rendered into the frame buffer with alpha blending while the mesh vertices are perturbed using a noise function. The final effect is a kind of under water like distortion.

Demonstration written by Stewart Sargaison, Sony Computer Entertainment America R+D
e-mail: Stewart_Sargaison@playstation.sony.com

Example: Shadow Rendering using PS2

This demonstration shows the rendering of shadows using a point light source and a blocker object which renders to a shadow texture map and then applies this shadow map to the receiving background objects.

The operation is split into two parts. First, a CPU and VU0 microcode routine calculates the shadow texture map and framing of the blocker object which casts a shadow. Note that this geometry can be a simpler version of the actual blocker object.

The second part is the rendering of the shadow onto the receiver geometry. Here, a special VU1 microcode is used which combines the standard rendering operations and the shadow calculations so that the vertices only have to be fetched once and many of the intermediate results of the transformations can be shared between the two calculations. This is a clear example of where a separate second pass of the geometry for casting the shadow would be much more expensive.

Demonstration written by Gabor Nagy, Sony Computer Entertainment America R+D
e-mail: Gabor_Nagy@playstation.sony.com

Other potential uses

Special microcode can be written for certain classes of in-game objects which lend themselves to a parametric or procedural description. Often, these descriptions embody more information about the way a class of objects should be drawn which allows for efficiency in both storage and rendering. Here are two examples that should work well:

Trees & Plants

A lot of excellent papers have been written about the procedural generation of plants. It should be possible to write a microcode renderer which would take a procedural description of a plant and render it directly – without CPU intervention.

Roads

Some in-game objects obey certain “rules” and therefore can be described in terms of those rules. One such example is a road surface in a racing game. These objects can be described in terms of splines, camber, bank, width, surface type etc. A special microcode could be written to take the procedural description and automatically tessellate a view dependent rendering of the road surface. This should be efficient both in memory use and in processing.

Summary

My aim in this session has been to demonstrate the benefits of a micro-programmable graphics architecture. Instead of a single, inflexible, monolithic set of rendering operations hard-coded in hardware, I show how microcode can allow for a multitude of different rendering

techniques including many which are specific to the application and its data set. The main advantage of these techniques is a reduction in the memory and bus-bandwidth used to describe in-game models. The secondary advantage is to allow novel, non-standard rendering techniques to be implemented more efficiently. Finally, I hope to have shown that performance of a microcoded architecture is excellent.

References

[1] E. Catmull & J. Clark, Recursively generated b-spline surfaces on arbitrary topological meshes. Computer Aided Design, 10:350-355, 1978
[2] C.Loop, Smooth spline surfaces based on triangles. Master's Thesis, University of Utah, Department of Mathematics, 1987
[3] K. Pulli, Fast Rendering of Subdivision Surfaces. In SIGGRAPH 96 Technical Sketch, ACM SIGGRAPH,1996.
[4] D.Zorin, P.Schroder, T.DeRose, J.Stam, L.Kobbelt, J.Warren, Subdivision for modeling and animation. In SIGGRAPH 99 Course Notes, Course 37, ACM SIGGRAPH, 1999.