Inside the Playstation 2:
Architecture, Graphics Rendering, and Programming

from a series of articles at GamaSutra.com -- the site for game developers

PS2 Architecture: Benefits of a Micro-Programmable Graphics Architecture

PS2 Rendering: Procedural Rendering on Playstation 2

From PS2 to XBox: Porting the Game "State of Emergency" from PS2 to XBox

see also these interesting PDFs by an amateur PS2 "hacker"
PS2 Basics PS2 I/O Processor PS2 Bitmaps 1 PS2 Bitmaps 2 PS2 Animation (2D) PS2 Fonts and Text

---

Benefits of A Micro-programmable Graphics Architecture

By Dominic Mallinson

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: the Playstation 2.

Playstation 2 Graphics Architecture

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

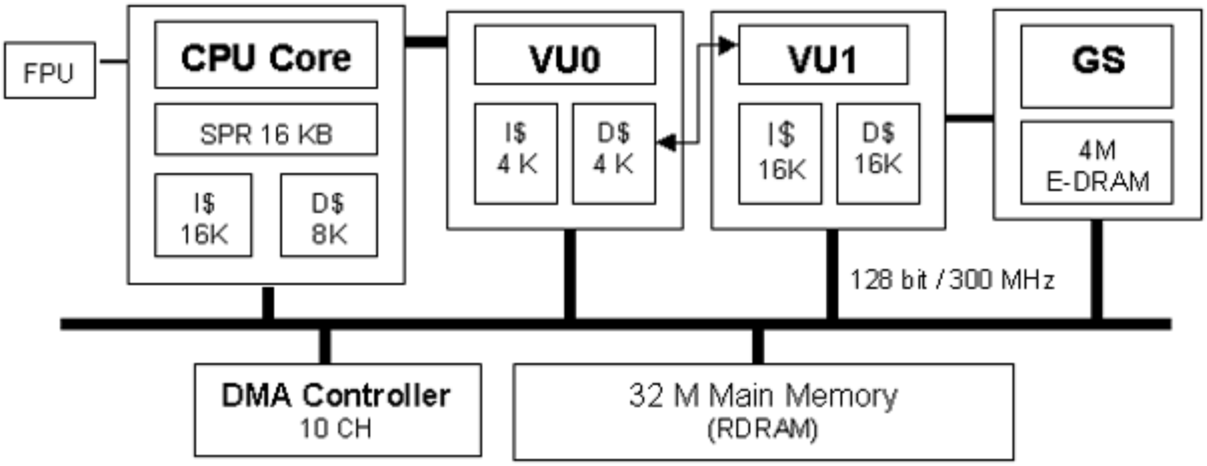


Figure 2. Playstation 2 Graphics Architecture

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.

The 3D Graphics Pipeline

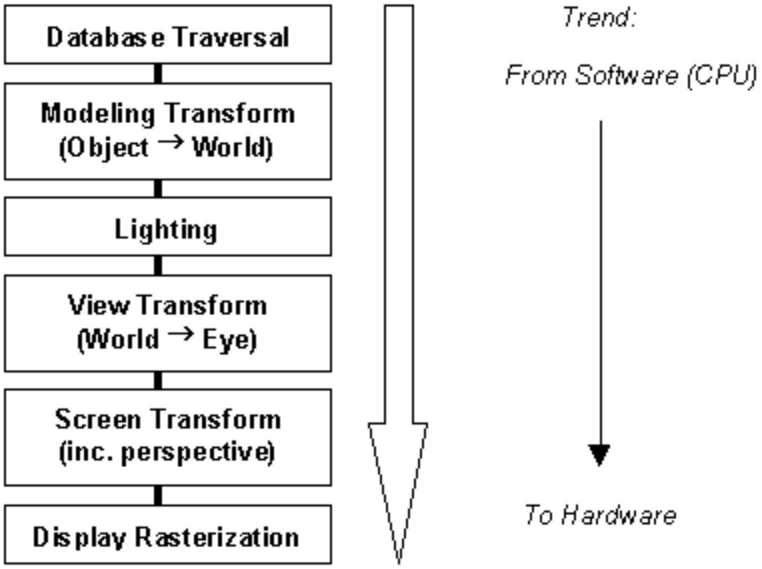

Database Traversal
Modeling Transform (Object → World)
Lighting
View Transform (World → Eye)
Screen Transform (inc. perspective)
Display Rasterization
Trend:
From Software (CPU)
To Hardware

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.

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.

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)

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

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

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

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

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

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

E. Catmull & J. Clark, Recursively generated b-spline surfaces on arbitrary topological meshes. Computer Aided Design, 10:350-355, 1978

C. Loop, Smooth spline surfaces based on triangles. Master's Thesis, University of Utah, Department of Mathematics, 1987

K. Pulli, Fast Rendering of Subdivision Surfaces. In SIGGRAPH 96 Technical Sketch, ACM SIGGRAPH, 1996.

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.

Procedural Rendering on Playstation 2

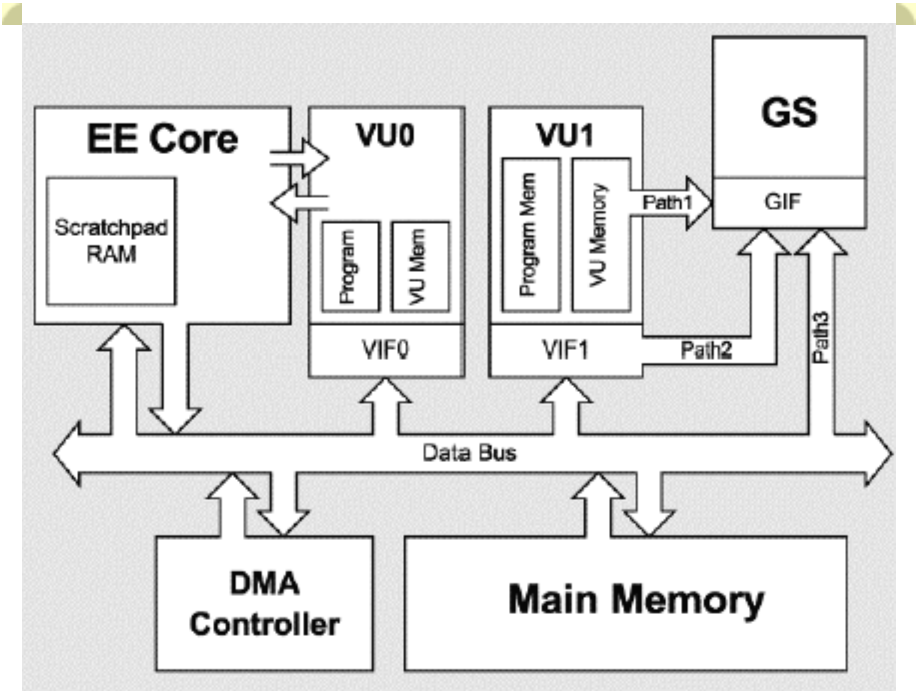
By Robin Green

This paper describes the design and coding of a program to generate instances of a procedural Lifeform, based on the work of William Latham in the late 1980’s. The sneaky thing is that this tutorial is not really about procedural rendering, it’s more about the real-world experience of designing and writing programs on PS2.

Why choose a procedural model? Nearly all of the early example code that ships with the T10000 development kit is something along the lines of “send this magic lump of precalculated data at this magic bit of VU code”. It does show off the speed of the machine and shows that techniques are possible but it’s not very useful if you want to learn how to produce your own code. By choosing a procedural model to program there are several benefits:

- Very little can be precalculated. The program has to explicitly build up all the data structures, allowing more insights into how things are put together.
- It can generate a lot of polygons quickly. Procedural models are scalable to the point where they can swamp even the fastest graphics system.
- It’s more interesting than yet another Fractal Landscape or Particle System. I just wanted to be a little different. The algorithm is only marginally more complex than a particle system with emitters.
- The algorithm has many ways it can be parallelized. The algorithm has some interesting properties that could allow large parts of it to be run on a VU, taking just the numerical description of a creature as inputs and generating the polygons directly. I have only space to explore one way of producing the models, but there are several other equally good separation points.
- It’s been over ten years. I thought it would be fun to try an old technique with modern hardware. Evolutionary Art was once a painstaking, time consuming search through the genome space, previewing and finally rendering with raytracing. Being able to generate 60 new Lifeforms a second opens up whole new areas of animation and interaction.

This paper will start by assuming you are familiar with the internal structure of the PS2 – EE Core, VIF0, VIF1, GIF, DMAC and Paths 1, 2 and 3. It will also assume you know the formats for GIF, VIF and DMA tags as they have been discussed endlessly in previous tutorials.



Simplified PS2 Block Diagram

Lifeforms

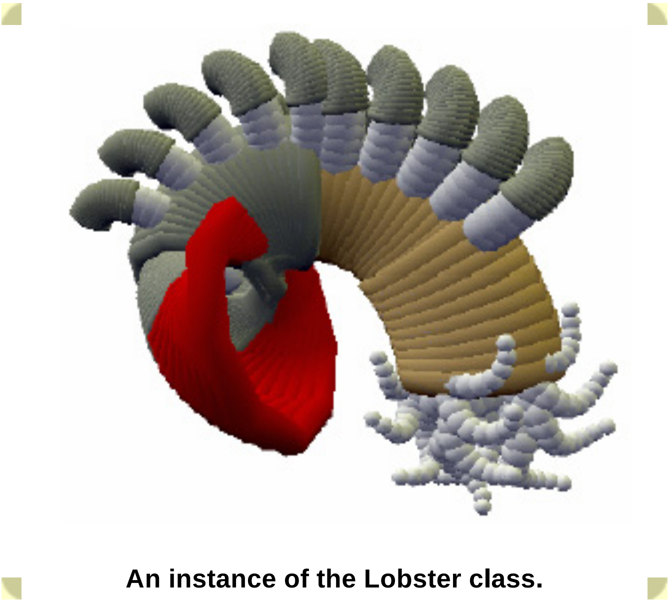
First, a look at the type of model we're going to be rendering. The ideas for thesemodels come from the organic artworks of William Latham.

Latham (in the days before his amazing sideburns) was recruited in 1987 from a career lecturing in Fine Art by a team of computer graphics engineers at the IBM UK Scientific Centre at Winchester, UK. He was recruited because of his earlier work in "systems of art" where he generated huge paper-based tree diagrams of forms that followed simple grammars. Basic forms like cubes, spheres and cylinders would have transformations applied to them – "scoop" took a chunk out of a form, "bulge" would add a lobe to the side, etc. These painstaking hand drawn production trees grew to fill enormous sheets of paper and were displayed in the exhibition "The Empire of Form".

The team at IBM Winchester wanted to use his insights into rule based art to generate computer graphics and so Latham was teamed with programmer Stephen Todd in order to make some demonstrations of his ideas. The resulting programs Form Build and Mutator were used to create images for the exhibitions "The Conquest of Form" and "Mutations".

The final system was documented in a 1989 IBM Systems Journal paper (vol.28, no.4) and in the book "Evolutionary Art and Computers", from which all the technical details in this tutorial are taken. (Latham later went on to found the company Computer Artworks who have recently released the game *Evolva*).

The aim of this tutorial is to render a class of Lifeform that Latham called a "Lobster" as efficiently as possible on the Playstation2. The Lobster is a useful object for exploring procedural rendering as it is a composite object made three parts, head, backbone and ribcage, each of which is a fully parameterized procedural object.



An instance of the Lobster class.

Definitions

In this talk I shall be using the Renderman standard names for coordinate spaces:

- Object space – Objects are defined relative to their own origin. For example, a sphere might be defined as having a unit radius and centered around the origin (0,0,0).
- World Space – A modeling transformation is used to position, scaled and orient objects into world space for lighting.
- Camera Space – All objects in the world are transformed so that the camera is sitting at the origin looking down the positive z-axis.
- Screen Space – A 2D space where coordinates range from –1.0..1.0 along the widest axis and takes into consideration the aspect ratio along the smallest axis.
- Raster Space – A integer 2D space where increments of 1 map to single pixel steps.

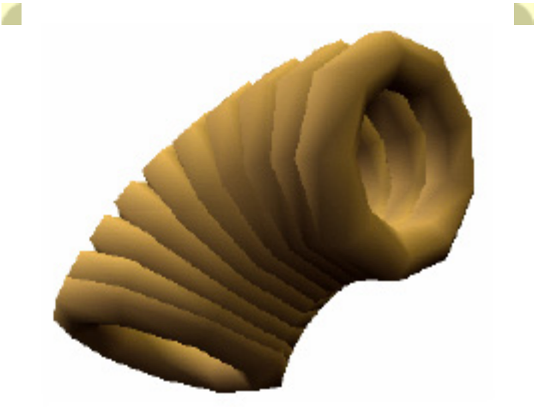
The order of spaces is: `object -> world -> camera -> screen -> raster`

Defining A Horn

The basic unit of rendering is the horn, named because of it's similarity to the spiral forms of animal horns. A control parameter is swept from 0.0 to 1.0 along the horn and at fixed intervals along this line primitives are instanced – a sphere or a torus – according to an ordered list of parameterized transformation rules.

Here's an example declaration from the book:

```
neck := horn
  ribs (18)
  torus (2.1, 0.4)
  twist (40, 1)
  stack (8.0)
  bend (80)
  grow (0.9) ;
```



An example horn.

The above declaration, exactly as it appears in the book, is written in a high level functional scripting language the IBM team used to define objects. We will have to translate this language into a form we can program in C++, so let's start by analyzing the declaration line by line:

```
neck := horn
```

The symbol "neck" is defined by this horn. Symbols come into play later when we start creating linear lists of horns or use a horn as the input primitive for other production rules like "branch" or "ribcage". At this point in the declaration the horn object is initialized to zero values.

```
ribs (18)
```

This horn has 18 ribs along it's length. To generate the horn, an iterator will begin at the `start position` (see later, initially 0.0) and count up to 1.0, taking 18 steps along the way. At each step a copy of the current `inform` (input form) is generated. Internally, this interpolator value is known as m. The number of ribs need not be an integral number – in this implementation the first rib is always at the start position, then a fractional rib is generated, followed by the integer ribs. In effect, the horn grows from the root.

```
torus (2.1, 0.4)
```

This horn will use for it's `inform` a torus of major radius 2.1 units and minor radius 0.4 units (remember that these values are just the radii, the bounding box size of the object is twice these values). In the original `Form Build` program this declaration can be any list of horns, composite forms or primitives which are instanced along the horn in the order they are declared (e.g. sphere, cube, torus, sphere, cube, torus, etc.), but for our purposes we will allow only a single primitive here. Extending the program to use generalized primitive lists is quite simple. Primitives are assumed to be positioned at the origin sitting on the x-z plane, right handed coordinate system.

```
twist (40, 1)
```

Now we begin the list of transformations that each primitive will undergo. These declarations specify the transform at the far end of the horn, so intermediate ribs will have to interpolate this transform along the horn using the iterator value *m*.

`twist(angle,offset)` is a compound transform that works like this:

- translate the rib offset units along the x-axis.
- rotate the result m*angle degrees around the y-axis.
- translate that result –offset units along the x-axis.

The end result, at this stage, produces a spiral shaped horn sitting on the x-z plane with as yet no elevation.

```
stack (8.0)
```

This translates the rib upwards along the +y-axis. Note that although we have 18 ribs that are 0.8 units high (18 * 0.8 = 14.4), we’re packing them into a space 8.0 units high. Primitives will therefore overlap making a solid, space filling form.

```
bend (80)
```

This rotates each rib some fraction of 80 degrees around the z-axis as it grows. This introduces a bend to the left if you were looking down the –z-axis (right handed system).

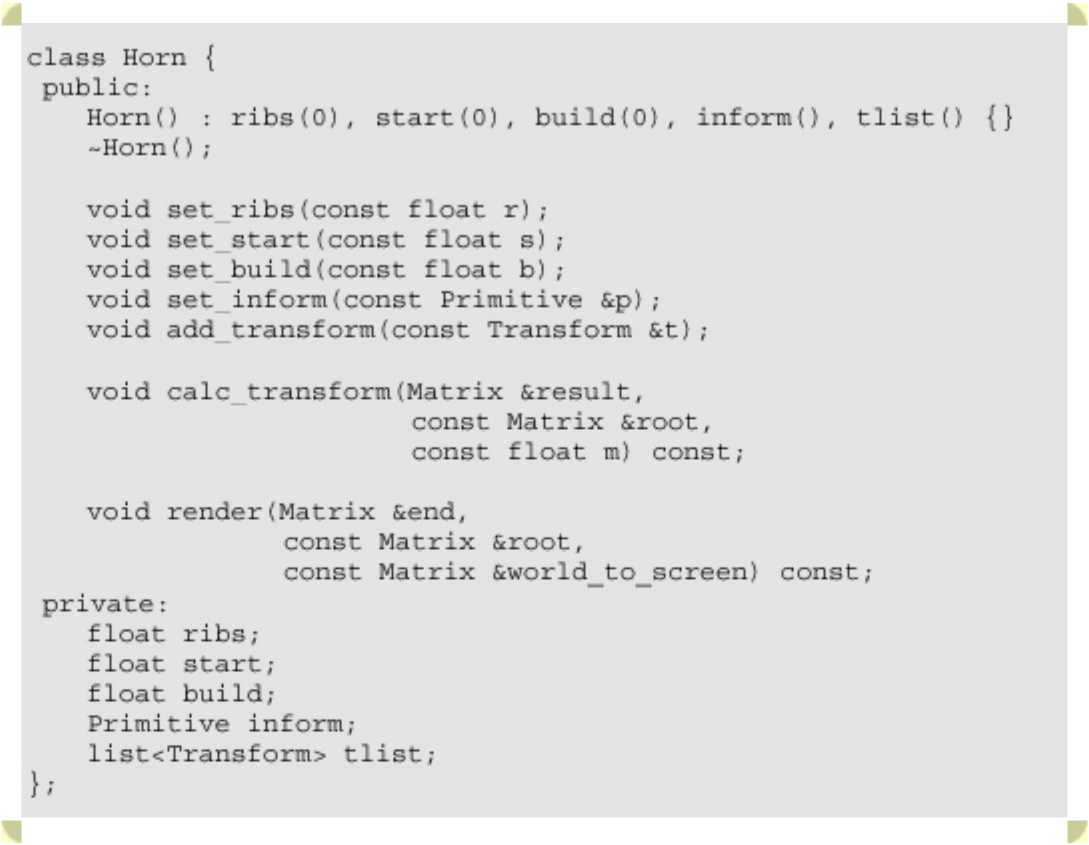
```
grow (0.9)
```

This scales the rib by a factor of 0.9, shrinking each primitive by 10 percent. This is not to say that each rib will be 90 percent of the size of the previous rib, the 0.9 scale factor is for the whole horn transformation. In order to interpolate the scale factor correctly we need to calculate `powf(0.9,m)` where m is our interpolator.

At the end of this horn declaration we have defined three areas of information:

1. The horn’s attributes (18 ribs, start counting at zero)
2. The ordered list of input forms (in this case a single torus).
3. The ordered list of transformations (twist, stack, bend, grow).

The order of declarations inside each area is important e.g. twist comes before stack, but between areas order is unimportant e.g. ribs(18) can be declared anywhere and overrides the previous declaration. This leads us to a design for a horn object, using the C++ STL, something like this:



```
class Horn {
 public:
    Horn() : ribs(0), start(0), build(0), inform(), tlist() {}
    ~Horn();

    void set_ribs(const float r);
    void set_start(const float s);
    void set_build(const float b);
    void set_inform(const Primitive &p);
    void add_transform(const Transform &t);

    void calc_transform(Matrix &result,
                        const Matrix &root,
                        const float m) const;

    void render(Matrix &end,
                const Matrix &root,
                const Matrix &world_to_screen) const;
 private:
    float ribs;
    float start;
    float build;
    Primitive inform;
    list<Transform> tlist;
};
```

The Transformation List

To render a single rib we need to know where to render the inform, at what size and at what orientation. To use this information we need to be able to generate a 4x4 matrix at any point along the parametric line.

Remembering that matrices concatenate right to left, the horn we defined earlier generates this list of operations:

```
    T = bend * (stack * (twist (grow * identity)))
      = bend * stack * twist * grow * identity
```

(As a side note, one oddity of the horn generation algorithm presented in the book “Evolutionary Art” is that it seems the programmers have altered the order of scale operations to always prepend the transformation queue. Every other transformation is appended to the left hand end of a transformation queue, e.g.

```
     M = rotate * M
```

whereas all scale operations (the grow function) are prepended on the right hand side:

```
     M = M * scale
```

This detail is never discussed in the book but turns out to be necessary to get the diagrams to match the declarations in the book.)

For simplicity’s sake we’ll leave off the identity matrix from the far right and just assume it’s there. If we parameterize this list of transforms w.r.t. the iterator m, the equation expands to this:

```
  T(m) = bend(80*m) * stack(8.0*m) * twist(40*m, 1) * grow(0.9^m)
```

Each of the transforms has to generate a 4x4 matrix which again expands the equation into the more familiar matrix form (ignoring details of the coefficients):

```
T(m) = B * S * T * G

                   |c s 0 0|   |1 0 0 0|   |c 0 s 0|   |s 0 0 0|
                   |s c 0 0| * |0 1 0 0| * |0 1 0 0| * |0 s 0 0|
                   |0 0 1 0|   |0 0 1 0|   |s 0 c 0|   |0 0 s 0|
                   |0 0 0 1|   |0 t 0 1|   |t 0 t 1|   |0 0 0 1|
```

We are going to be wanting to move and orient the whole horn to some position in world space to allow us to construct composite objects, so we will have to prepend a modeling transformation we shall call the “root” transform (positioned after the identity):

```
   T(m) = B * S * T * G * root
```

And finally we will have to transform the objects from world to screen space to generate the final rendering matrix `M`:

```
  M = world_to_screen * T(m) * root
```

This description is slightly simplified as we have to take into consideration transforming lighting normals for the primitives. Normals are direction vectors transformed by the transpose of the inverse of a matrix, but as every matrix in this list is simple rotation, translation or scale then this matrix is the same as the modeling matrix T(m) except for a scale factor. We can use the transpose of the adjoint (the inverse not yet divided by the determinant) and renormalize each normal vector after transformation or build a special lighting matrix that ignores translations and scales, retaining only the orthogonal operations that change orientation (i.e. record only rotations and reflections).

```
     void Horn::render(const Matrix &root,
                        const Matrix &world_to_screen);
```

Chaining Horns and Build

One operation we’re going to want to do is to chain horns together to allow us to generate long strings of primitives to model the backbones and tentacles.

A backbone made from chained horns.

This is where the root transformation comes into play. To chain horns together we need to transform the root of a horn N+1 by the end transformation of the previous horn N. We do this by getting the horn generation function to return the final matrix of the current horn for use later:

```
void Horn::render(Matrix &newroot,
                  const Matrix &root,
                  const Matrix &world_to_screen);
```

Another detail of horn generation arises in animations. As the number of ribs increases all that happens is that ribs become more closely packed – if you want the horn to appear to grow in length by adding horns you will have to re-specify the `stack` command in the transformation list to correctly extend your horn for more ribs. The way around this problem is to tell the iterator how many ribs this list of transforms was designed for, allowing it to generate the correct transformations for more ribs. This is done using the build command:

```
myhorn := horn
   ribs (9)
   build (5)
   torus (2.1, 0.4)
   stack (11);
```

This declaration declares a horn where the first 5 ribs are stacked 11 units high, but the horn is designed to transform 9 ribs -the remaining four ribs will be generated with interpolation values greater than 1.0.

Initially the horn has a `build` member variable set to `0.0`, telling the interpolator to generate interpolation values of `0..1` using ribs steps. If the `build` variable is not equal to 0.0, the interpolator will map 0..1 using build steps, allowing requests for fewer or greater ribs to work with the correct spacing:

```
Matrix T;
float base = (build == 0.0) ? ribs : build;
```

```
for(int i=0; i<ribs; ++i) {
   float m = i / base;
   calc_matrix(T, m);
   ...
}
```

Branch and Ribcage

In order to construct the Lobster there are two more structures to generate – the head and the ribcage.

The head is an instance of a branch object – horns are transformed so that their near end is at the origin and their far ends spread out over the surface of a sphere using a spiral pattern along the sphere's surface. You can control the pitch and angles of the spiral to produce a wide range of interesting patterns.

Examples of a branch structure.

The branch algorithm just produces a bunch of root transformations by:

- Generate a point on a unit sphere.
- Calculating the vector from the origin of the sphere to that point on the surface.
- Return a 4x4 transform matrix that rotates the Object space y-axis to point in that direction.

To correctly orient horns twist could be taken into consideration by rotating horns so that the x-axis lies along the tangent to the spiral, but this is unimportant for the look of most Lifeforms. (To do branches correctly we could use a quaternion slerp along the spiral, but we'll leave that as an exercise). The function declaration looks like this:

```
branch(const Matrix &root,
      const Horn &horn,   // inform to instance
      const int number,   // number of informs to instance
      const float angle,  // solid angle of the sphere to use
      const float pitch); // pitch of the spiral
```

The ribcage algorithm takes two inputs – a horn for the shape of the rib and a second horn to use as the backbone. To place a horn on the ribcage the ribcage algorithm queries the backbone horn for transformations at specific points along it’s length and uses those transforms as the root for instancing each ribcage horn.

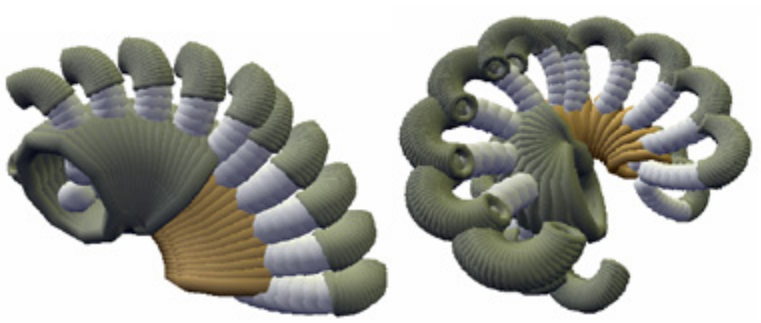
Assuming for the moment that the backbone is a simple stack up the y-axis. Ribs must stick out at 90 degrees from the backbone so we introduce a 90 degree rotation about the z-axis, followed by a 90 degree rotation about the y-axis. Also ribs come in pairs so the second rib must be a reflection of the first about the y-z plane. This can all be done by adding in a single reflection matrix. Think of the resulting composite transform as repositioning the rib flat onto the x-z plane pointing in the correct direction before moving it along the backbone:

$$\mathrm{root}_{rib}(m) = T_{backbone}(m) * \mathrm{reflection} * \mathrm{root}$$

where

```
reflection =   |-1 0 0 0 |     |-1 0 0 0 |
               |0 0 –1 0 |     |0 0 –1 0 |
               |0 –1 0 0 |     | 0 1 0 0 |
               | 0 0 0 1 |     | 0 0 0 1 |
```

Note that this reflection must be applied to the shading normals too otherwise lighting for the ribcage will be wrong.



Examples of a ribcage (with backbone).

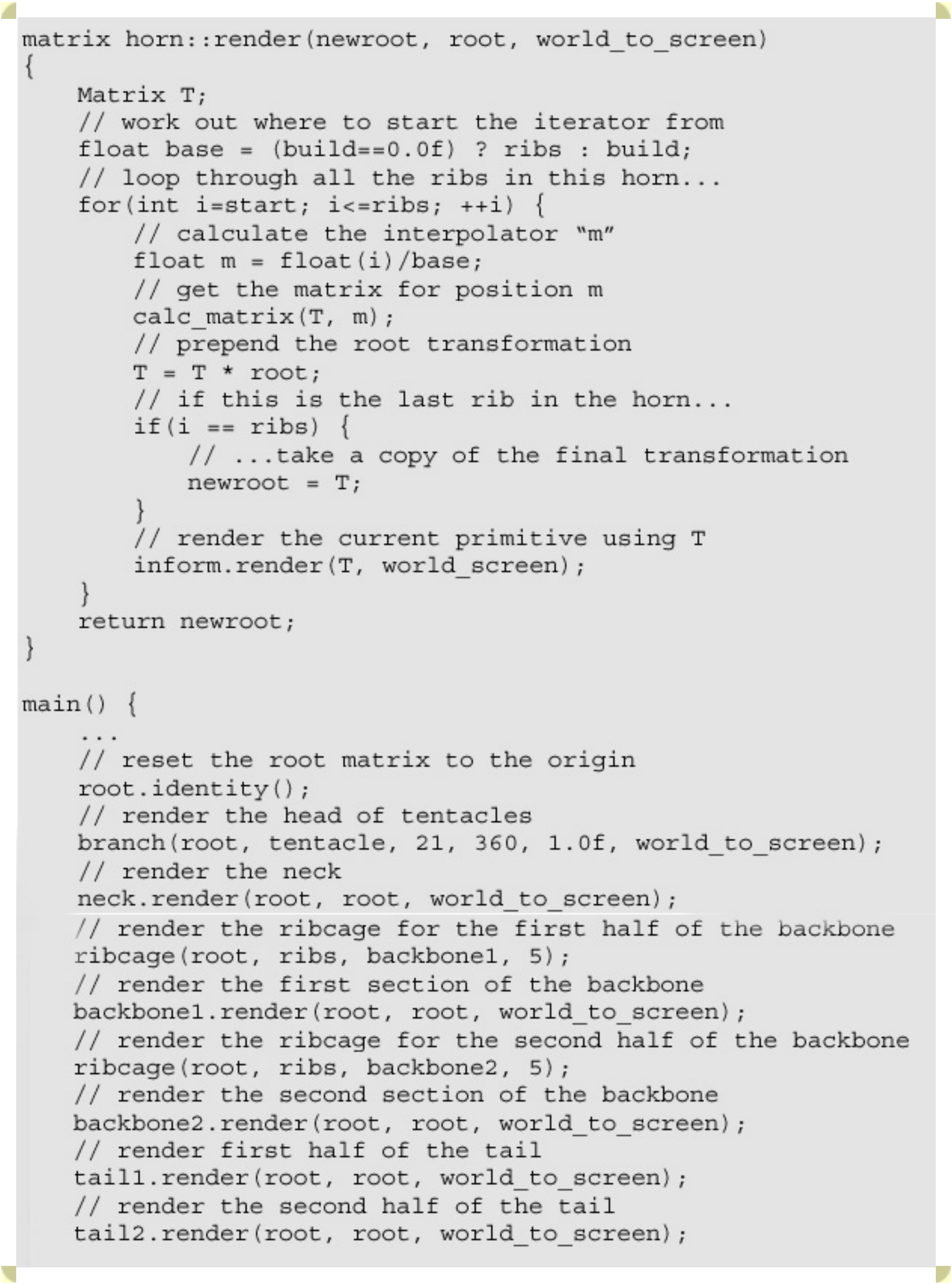
To help this process we must be able to randomly access transformations along a horn, hence the generalised calc_transform() member function in the Horn class.

The Basic Rendering Process Here, finally, is an outline of the basic rendering process for a Lobster. We will take this as the starting point for our investigation of procedural rendering and start focussing on how to implement this process more efficiently on the Playstation 2 using the EE-Core, vector units and GS together.

```
ribcage(const Matrix &root,
        const Horn &horn,     // inform to use as ribs
        const Horn &backbone, // inform to query as the backbone
        const int number);    // number of pairs of ribs to generate
```

The Basic Rendering Process

Here, finally, is an outline of the basic rendering process for a Lobster. We will take this as the starting point for our investigation of procedural rendering and start focussing on how to implement this process more efficiently on the Playstation 2 using the EE-Core, vector units and GS together.



```
matrix horn::render(newroot, root, world_to_screen)
{
    Matrix T;
    // work out where to start the iterator from
    float base = (build==0.0f) ? ribs : build;
    // loop through all the ribs in this horn...
    for(int i=start; i<=ribs; ++i) {
        // calculate the interpolator “m”
        float m = float(i)/base;
        // get the matrix for position m
        calc_matrix(T, m);
        // prepend the root transformation
        T = T * root;
        // if this is the last rib in the horn...
        if(i == ribs) {
            // ...take a copy of the final transformation
            newroot = T;
        }
        // render the current primitive using T
        inform.render(T, world_screen);
    }
    return newroot;
}

main() {
    ...
    // reset the root matrix to the origin
    root.identity();
    // render the head of tentacles
    branch(root, tentacle, 21, 360, 1.0f, world_to_screen);
    // render the neck
    neck.render(root, root, world_to_screen);
    // render the ribcage for the first half of the backbone
    ribcage(root, ribs, backbone1, 5);
    // render the first section of the backbone
    backbone1.render(root, root, world_to_screen);
    // render the ribcage for the second half of the backbone
    ribcage(root, ribs, backbone2, 5);
    // render the second section of the backbone
    backbone2.render(root, root, world_to_screen);
    // render first half of the tail
    tail1.render(root, root, world_to_screen);
    // render the second half of the tail
    tail2.render(root, root, world_to_screen);
```

Ten Things Nobody Told You About PS2

Before we start translating this Lifeform algorithm into a PS2 friendly design, I'd like to cover some more about the PS2. Later we'll use these insights to re-order and tweak the algorithm to get some speed out of the machine.

1. You *must* design before coding.

Lots of people have said this about PS2 – you cannot just sit down and code away and expect high speed programs as a result. You have to plan and design your code around the hardware and that requires insight into how the machine works. Where do you get this insight from? This is where this paper comes in. The aim later is to present some of the boilerplate designs you can use as a starting point.

2. The compiler doesn't make things easy for you.

Many of the problems with programming PS2 come from the limitations of the compiler. Each General Purpose Register in the Emotion Engine is 128-bits in length but the compiler only supports these as a special case type. Much of the data you need to pass around the machine comes in 128-bit packets (GIF tags, 4D float vectors, etc.) so you will spend a lot of time casting between different representations of the same data type, paying special attention to alignment issues. A lot of this confusion can be removed if you have access to well designed Packet and 3D Vector classes.

Additionally the inline assembler doesn't have enough information to make good decisions about uploading and downloading VU0 Macro Mode registers and generating broadcast instructions on vector data types. There is a patch for the ee-gcc compiler by Tyler Daniel and Dylan Cuthbert that update the inline assembler to add register naming, access to broadcast fields and new register types which are used to good effect in our C++ Vector classes. It's by no means perfect as you're still limited to only 10 input and output registers, but it's a significant advance.

3. All the hardware is memory mapped.

Nearly all of the basic tutorials I have seen for PS2 have started by telling you that, in order to get anything rendered on screen, you have to learn all about DMA tags, VIF tags and GIF tags, alignment, casting and enormous frustration before your "Hello World" program will work. The tutorials always seem to imply that the only way to access outboard hardware is through painstakingly structured DMA packets. This statement is not true, and it greatly complicates the process of learning PS2. In my opinion this is one of the reasons the PS2 is rejected as "hard to program".

Much of this confusion comes from the lack of a detailed memory map of the PS2 in the documentation. Understandably, the designers were reticent to provide one as the machine was in flux at the time of writing (the memory layout is completely reconfigurable by the kernel at boot time) and they were scared of giving programmers bad information. Let's change this.

All outboard registers and memory areas are freely accessible at fixed addresses. Digging through the headers you will come across a header called eeregs.h that holds the key. In here are the hard-coded addresses of most of the internals of the machine. First a note here about future proofing your programs. Accessing these registers directly in final production code is not advisable as it's fully possible that the memory map could change with future versions of the PS2. These techniques are only outlined here for tinkering around and learning the system so you can prove to yourself there's no magic here. Once you have grokked how the PS2 and the standard library functions work, it's safest to stick to using the libraries.

Let's take a look at a few of the values in the header and see what they mean:

```
    #define VU1_MICRO      ((volatile u_long *)(0xNNNNNNNN))
    #define VU1_MEM        ((volatile u_long128 *)(0xNNNNNNNN))
```

These two addresses are the start addresses of VU1 program and data memory if VU1 is not currently caclulating. Most tutorials paint VU1 as "far away", a hands off device that's unforgiving if you get a single instruction wrong and consequently hard to debug. Sure, the memory is unavailable if VU1 is running a program, but using these addresses you can dump the contents before and after running VU programs. Couple this knowledge with the DMA Disassembler and VCL, the vector code compiler, and VU programming without expensive proprietary tools and debuggers is not quite as scary as it seems.

```
        #define D2_CHCR      ((volatile u_int *)(0xNNNNNNNN))
        #define D2_MADR      ((volatile u_int *)(0xNNNNNNNN))
        #define D2_QWC       ((volatile u_int *)(0xNNNNNNNN))
        #define D2_TADR      ((volatile u_int *)(0xNNNNNNNN))
        #define D2_ASR0      ((volatile u_int *)(0xNNNNNNNN))
        #define D2_ASR1      ((volatile u_int *)(0xNNNNNNNN))

        #define D3_CHCR      ((volatile u_int *)(0xNNNNNNNN))
        #define D3_MADR      ((volatile u_int *)(0xNNNNNNNN))
        #define D3_QWC       ((volatile u_int *)(0xNNNNNNNN))
```

If you have only read the SCE libraries you may be under the impression that “Getting a DMA Channel” is an arcane and complicated process requiring a whole function call. Far from it. The DMA channels are not genuine programming abstractions, in reality they’re just a bank of memory mapped registers. The entries in the structure `sceDmaChan` map direc tly onto these addresses like a cookie cutter.

```
  #define GIF_FIFO      ((volatile u_long128 *)(0xNNNNNNNN))
```

The GIF FIFO is the doorway into the Graphics Synthesizer. You push qwords in here one after another and the GS generates polygons - simple as that. No need to use DMA to get your first program working, just program up a GIF Tag with some data and stuff it into this address.

This leads me to my favorite insight into the PS2…

4. The DMAC is just a Pigeon Hole Stuffer.

The DMA Controller (DMAC) is a very simple beast. In essence all it does is read a qword from a source address, write it to a destination address, increments one or both of these addresses, decrements a counter and loops. When you’re DMAing data from memory to the GIF all that’s happening is that the DMA chip is reading from the source address and pushing the quads through the GIF_FIFO we mentioned earlier – that DMA Channel has a hard-wired destination address.

5. Myth: VU code is hard.

VU code isn’t hard. Fast VU code is hard, but there are now some tools to help you get 80 percent of the way there for a lot less effort.

VCL (Vector Command Line, as opposed to the interactive graphic version) is a tool that preprocesses a single stream of VU code (no paired instructions necessary), analyses it for loop blocks and control flow, pairs and rearranges instructions, opens loops and interleaves the result to give pretty efficient code. For example, take this simplest of VU programs that takes a block of vectors and in-place matrix multiplies them by a fixed matrix, divides by W and integerizes the value:

```
      ; test.vcl
      ; simplest vcl program ever
      .init_vf_all
      .init_vi_all
      --enter
      --endenter
      .name start_here
      start_here:
         ilw.x srce_ptr 0(vi00)
         ilw.x counter, 1(vi00)
```

```
   iadd counter, counter, srce_ptr
   lq v_transf0 2(vi00)
   lq v_transf1 3(vi00)
   lq v_transf2 4(vi00)
   lq v_transf3 5(vi00)
loop:
   --LoopCS 6, 1
   lq vec, 0(srce_ptr)
   mulax.xyzw ACC, v_transf0, vec
   madday.xyzw ACC, v_transf1, vec
   maddaz.xyzw ACC, v_transf2, vec
   maddw.xyzw vec, v_transf3, vf00
   div Q, vf0w, vecw
   mulq.xyzw vec, vec, Q
   ftoi4.xyzw vec, vec
   sq vec, 0(srce_ptr)
   iaddiu srce_ptr,srce_ptr,1
   ibne srce_ptr, counter, loop
--exit
--endexit
. . .
```

VCL takes the source code, pairs the instructions and unwrap the loop to this seven instruction inner loop (with entry and exit blocks not shown):

```
loop__MAIN_LOOP:
; [0,7) size=7 nU=6 nL=7 ic=13 [lin=7 lp=7]
 maddw VF09,VF04,VF00w   lq.xyz  VF08,0(VI01)
 nop                     sq      VF07,(0)-(5*(1))(VI01)
 ftoi4 VF07,VF06         iaddiu  VI01,VI01,1
 mulq VF06,VF05,Q        move    VF05,VF10
 mulax ACC,VF01,VF08x    div     Q,VF00w,VF09w
 madday ACC,VF02,VF08y   ibne    VI01,VI02,loop__MAIN_LOOP
 maddaz ACC,VF03,VF08z   move    VF10,VF09
```

6. Myth: Synchronization is complicated.

The problem with synchronization is that much of it is built into the hardware and the documentation isn’t clear about what’s happening and when. Synchronization points are described variously as “stall states” or hidden behind descriptions of queues and scattered all over the documentation. Nowhere is there a single list of “How to force a wait for X” techniques.

The first point to make is that complicated as general purpose synchronization is, when we are rendering to screen we are dealing with a more limited problem: you only need to keep things in sync once a frame. All your automatic processes can kick off and be fighting for resources during a frame, but as soon as you reach the end of rendering the frame then everything must be finished. You are only dealing with short bursts of synchronization.

The PS2 has three main systems for synchronization:

- synchronization within the EE Core
- synchronization between the EE Core and external devices
- synchronization between external devices.

This whole area is worthy of a paper in itself as much of this information is spread around the documentation. Breaking the problem down into these three areas sheds allows you to grok the whole system. Briefly summarizing:

Within the EE Core we have sync.l and sync.e instructions that guarantee that results are finished before continuing with execution.

Between the EE Core and external devices (VIF, GIF, DMAC, etc) we have a variety of tools. Many events can generate interrupts upon completion, the VIF has a mark instruction that sets the value of a register that can be read by the EE Core allowing the EE Core to know that a certain point has been reached in a DMA stream and we have the memory mapped registers that contain status bits that can be polled.

Between external devices there is a well defined set of priorities that cause execution orders to be well defined. The VIF can also be forced to wait using flush, flushe and flusha instructions. These are the main ones we'll be using in this tutorial.

7. Myth: Scratchpad is for speed.

The Scratchpad is the 16KB area of memory that is actually on-chip in the EE Core. Using some MMU shenanigans at boot up time, the EE Core makes Scratchpad RAM (SPR) appear to be part of the normal memory map. The thing to note about SPR is that reads and writes to SPR are uncached and memory accesses don't go through the memory bus – it's on-chip and physically sitting next to (actually inside) the CPU.

You could think of scratchpad as a fast area of memory, like the original PSX, but real world timings show that it's not that much faster than Uncached Accelerated memory for sequential work or in-cache data for random work. The best way to think of SPR is as a place to work while the data bus is busy - something like a playground surrounded by roads with heavy traffic.

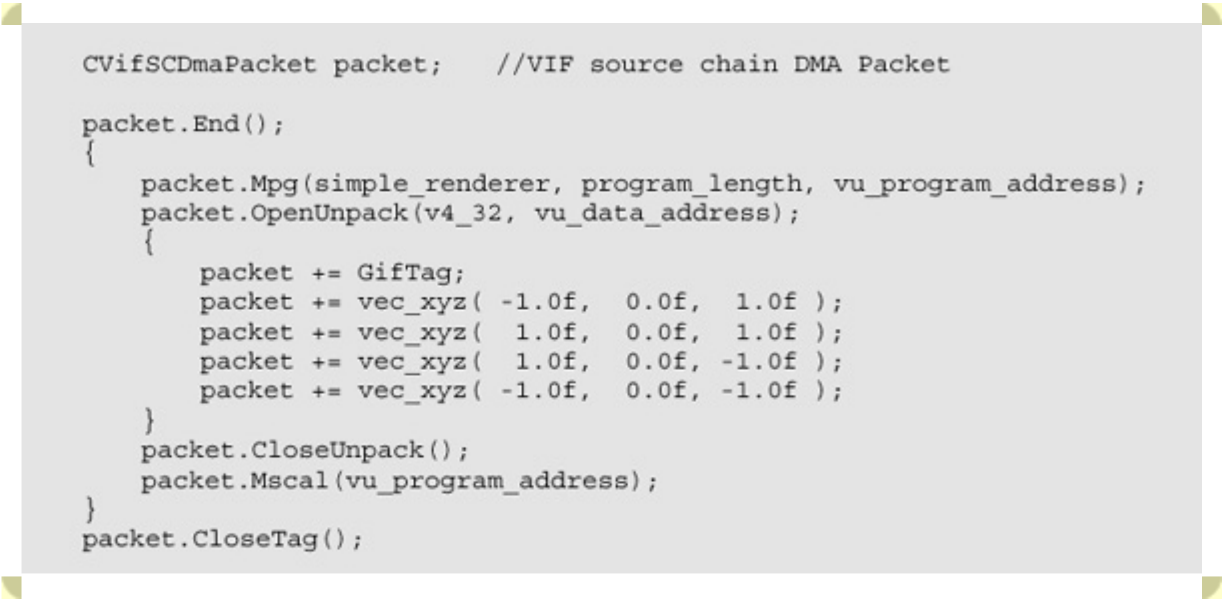
Picture this: Your program has just kicked off a huge DMA chain of events that will automatically upload and execute VU programs and move information through the system. The DMAC is moving information from unit to unit over the Memory Bus in 8-qword chunks, checking for interruptions every tick and CPU has precedence. The last thing the DMAC needs is to be interrupted every 8 clock cycles with the CPU needing to use the bus for more data. This is why the designers gave you an area of memory to play with while this happens. Sure, the Instruction and Data caches play their part but they are primarily there to aid throughput of instructions.

Scratchpad is there to keep you off the data bus – use it to batch up memory writes and move the data to main memory using burst-mode DMA transfers using the fromSPR DMA channel.

8. There is no such thing as "The Pipeline".

The best way to think about the rendering hardware in PS2 is a series of optimized programs that run over your data and pipe the resulting polygon lists to the GS. Within a frame there may be many different renderers – one for unclipped models, one for procedural models, one for specular models, one for subdivision surfaces, etc.

As each renderer is less than 16KB of VU code they are very cheap to upload compared to the amount of polygon data they will be generating. Program uploads can be embedded inside DMA chains to complete the automation process, e.g.



```
CVifSCDmaPacket packet;   //VIF source chain DMA Packet

packet.End();
{
    packet.Mpg(simple_renderer, program_length, vu_program_address);
    packet.OpenUnpack(v4_32, vu_data_address);
    {
        packet += GifTag;
        packet += vec_xyz( -1.0f,  0.0f,  1.0f );
        packet += vec_xyz(  1.0f,  0.0f,  1.0f );
        packet += vec_xyz(  1.0f,  0.0f, -1.0f );
        packet += vec_xyz( -1.0f,  0.0f, -1.0f );
    }
    packet.CloseUnpack();
    packet.Mscal(vu_program_address);
}
packet.CloseTag();
```

9. Speed is all about the Bus.

This has been said many times before, but it bears repeating. The theoretical speed limits of the GS are pretty much attainable, but only by paying attention to the bus speed. The GS can kick one triangle every clock tick (using tri-strips) at 150MHz. This gives us a theoretical upper limit of:

150 million verts per second = 2.5 million verts / frame at 60Hz

Given that each of these polygons will be flat shaded the result isn’t very interesting. We will need to factor in a perspective transform, clipping and lighting which are done on the VUs, which run at 300MHz. The PS2 FAQ says these operations can take 15 – 20 cycles per vertex typically, giving us a throughput of:

5 million verts / 20 cycles per vertex
= 250,000 verts per frame
= 15 million verts per second
5 million verts / 15 cycles per vertex
= 333,000 verts per frame
= 20 million verts per second

Notice the difference here. Just by removing five cycles per vertex we get a huge increase in output. This is the reason we need different renderers for every situation – each renderer can shave off precious cycles-per-vertex by doing only the work necessary.

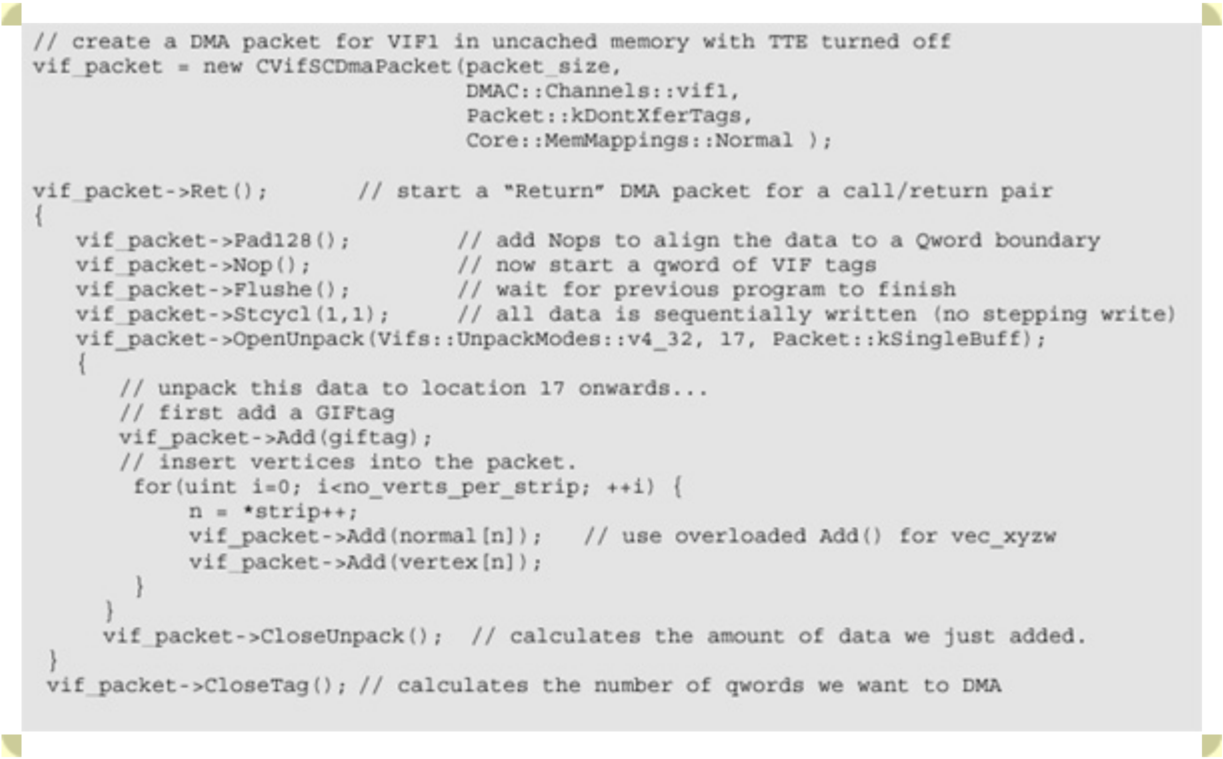
This is also the reason we have two VUs – often VU1 is often described as the “rendering” VU and VU0 as the “everything else” renderer, but this is not necessarily so. Both can be transforming vertices but only one can be feeding the GIF, and this explains the Memory FIFO you can set up: one VU is feeding the GS while the other is filling the FIFO. It also explains why we have two rendering contexts in the GS, one for each of the two input streams.

10. There are new tools to help you.

Unlike the early days of the PS2 where everything had to be painstakingly pieced together from the manuals and example code, lately there are some new tools to help you program PS2. Most of these are freely available for registered developers from the PS2 support websites and nearly all come with source.

DMA Disassembler. This tool, from SCEE’s James Russell, takes a completes DMA packet, parses it and generates a printout of how the machine will interpret the data block when it is sent. It can report errors in the chain and provides an excellent visual report of your DMA chain.

Packet Libraries. Built by Tyler Daniel, this set of classes allows easy construction of DMA packets, either at fixed locations in memory or in dynamically allocated buffers. The packet classes are styled after insertion-only STL containers and know how to add VIF tags, create all types of DMA packet and will calculate qword counts for you.



```
// create a DMA packet for VIF1 in uncached memory with TTE turned off
vif_packet = new CVifSCDmaPacket(packet_size,
                                 DMAC::Channels::vif1,
                                 Packet::kDontXferTags,
                                 Core::MemMappings::Normal );

vif_packet->Ret();       // start a "Return" DMA packet for a call/return pair
{
   vif_packet->Pad128();        // add Nops to align the data to a Qword boundary
   vif_packet->Nop();           // now start a qword of VIF tags
   vif_packet->Flushe();        // wait for previous program to finish
   vif_packet->Stcycl(1,1);     // all data is sequentially written (no stepping write)
   vif_packet->OpenUnpack(Vifs::UnpackModes::v4_32, 17, Packet::kSingleBuff);
   {
       // unpack this data to location 17 onwards...
       // first add a GIFtag
       vif_packet->Add(giftag);
       // insert vertices into the packet.
        for(uint i=0; i<no_verts_per_strip; ++i) {
            n = *strip++;
            vif_packet->Add(normal[n]);   // use overloaded Add() for vec_xyzw
            vif_packet->Add(vertex[n]);
        }
     }
     vif_packet->CloseUnpack();  // calculates the amount of data we just added.
 }
 vif_packet->CloseTag(); // calculates the number of qwords we want to DMA
```

Vector Libraries and GCC patch. The GCC inline assembler patch adds a number of new features to the inline assembler:

- Introduces a new j register type for 128-bit vector registers, allowing the compiler to know that these values are to be assigned to VU0 Macro Mode registers
- Allows register naming, so more descriptive symbols can be used.
- Allows access to fields in VU0 broadcast instructions allowing you to, say, template a function across broadcast fields (x, xy, xyz, xyzw)
- No more volatile inline assembly, the compiler is free to reorder instructions as the context is properly described.
- No more explicit loading and moving registers to and from VU registers, the compiler is free to keep values in VU registers as long as possible.
- No need to use explicit registers, unless you want to. The compiler can assign free registers

The patch is not perfect as there is still a limit to 10 input and output registers per section of inline assembly, and that can be a little painful at times (i.e. three operand 4x4 matrix operations like a = b * c take 12 registers to declare), but it is at least an improvement.

The Matrix and Vector classes showcase the GCC assembler patch, providing a set of template classes that produce fairly optimized results, plus being way easier to write and alter to your needs:

```
vec_x dot( const vec_xyz rhs ) const
{
  vec128_t result, one;
  asm(
    " ### vec_xyzw dot vec_xyzw ### \n"
    "vmul          result, lhs, rhs \n"
    "vaddw.x      one, vf00, vf00 \n"
    "vaddax.x     ACC, vf00, result \n"
    "vmadday.x ACC, one, result \n"
    "vmaddz.x    result, one, result \n"
    : "=j result" (result),
    "=j one"   (one)
    : "j lhs"   (*this),
    "j rhs"    (rhs)
  );
  return vec_x(result);
}
```

VU Command Line preprocessor. As mentioned earlier, one of the newest tools to aid PS2 programming is VCL, the vector code optimizing preprocessor. It takes a single stream of VU instructions and:

- Automatically pairs instructions into upper and lower streams.
- Intelligently breaks code into looped sections.
- Unrolls and interleaves loops, producing correct header and footer sections. . Inserts necessary nops between instructions.
- Allows symbolic referencing of registers by assigning a free register to the symbol at first use. (The set of free regs is declared at the beginning of a piece of VCL code).
- Tracks vector element usage based on the declared type – it can ensure a vector element that has been declared as an integer but held in a float is treated correctly.

No more writing VU code in Excel! It outputs pretty well optimized results that can be used as a starting point for hand coding (It can also be run on already existing code to see if any improvements can be made).

VCL is not that intelligent yet (it will happily optimize complete rubbish). For the best results it's worth learning how to code in a VCL friendly style, e.g.:

- Instead of directly incrementing pointers:

  ```
  lqi vector, (address++)
  lqi normal, (address++)
  ```

  You should use offset addressing:

  ```
  lq vector, 0(address)
  lq normal, 1(address)
  iaddi address, address, 2
  ```

- Make sure that all members of a vector type are accounted for, e.g. when calculating normal lighting only the xyz part of a vector is needed, so remember to set the w value to a constant in the preamble, thus breaking a dependency chain that prevents VCL from interleaving unrolled loop sections:

  ```
  sub.w normal, normal, vf00
  ```

More of these techniques are in the VCL documentation. It's really satisfying to be able to cut and paste blocks of code together to get the VU program you need and not need to worry about pairing instructions and inserting nops.

Built-in Profiling Registers. The EE Core, like all MIPS processors, has a number of performance registers built into Coprocessor 0 (the CPU control unit). The PerfTest class reads these registers and can print out a running commentary on the efficiency of any section of code you want to sample.

Performance Analyzer. SCE have just announced it's hardware Performance Analyzer (PA). It's a hardware device that samples activity on the busses and produces graphs, analysis and in depth insights into your algorithms. Currently the development support offices are being fitted with these devices and your teams will be able to book consultation time with them.

VU Dataflow Designs

In this section we'll look behind some of the design decisions we'll need to make when translating the Lifeform algorithm to PS2. The first decision is where to draw the line between CPU and VU calculations. There are several options, in order of difficulty:

1. The main program calculates an abstract list of triangle strips for the VU to transform and light.
2. The main program calculates transform and lighting matrices each primitive (sphere, torus). The VU is passed a static description of an example primitive in object space which the VU transforms into world space, lights and displays.
3. The main program sends the VU a description of a single horn plus an example primitive in object space. The VU iterates along the horn, calculating the transform and lighting matrices for each rib and instances the primitives into world space using these matrices.
4. The main program sends the VU a numerical description of the entire model, the VU does everything else.

For this tutorial I chose the second option as a halfway house between full optimization and simple triangle lists. Following down this path is not too far removed from ordinary engine programming and it leaves the final program wide open for further optimizations.

Keep Your Eye On The Prize

Before setting out it's useful to remember what the ultimate aim of the design is - to send correct GS Tags and associated data to the GS. A GIF Tag contains quite a bit of information, but only a few of them are important.

GIF Tags are the essence of PS2 graphics programming. They tell the GS how much data to expect and in what format and they contain all the necessary information for an xgkick instruction to transfer data from VU1 to the GS automatically. Create the correct GIF Tag for your set of N vertices and everything else is pretty much automatic.

The complications in PS2 programming arise when you start to work out how to get the GIF Tags to the GS. There are three routes into the GS. One is direct and driven by DMA (Path 3), one is indirect and goes via VIF1 (Path 2) and is useful for redirecting part of a DMA stream to the GS and the third is very indirect (Path 1) and requires you to DMA data into VU1 memory and executed a program that ends in a `xgkick`.

Choosing a Data Flow Design

The first problem is to choose a dataflow design. The next few pages contain examples of basic data flows that can be used as starting points for your algorithms. Each dataflow is described two ways – once using it's physical layout in memory showing where data moves to and from, and once as a time line showing how stall conditions and VIF instructions control the synchronization between the different pieces of hardware.

The diagrams are pretty abstract in that they only outline the data movement and control signals necessary but don't show areas for constants, precalculated data or uploading the VU program code. We'll be covering all these details later when we go in-depth into the actual algorithm used to render the Lifeform primitives.

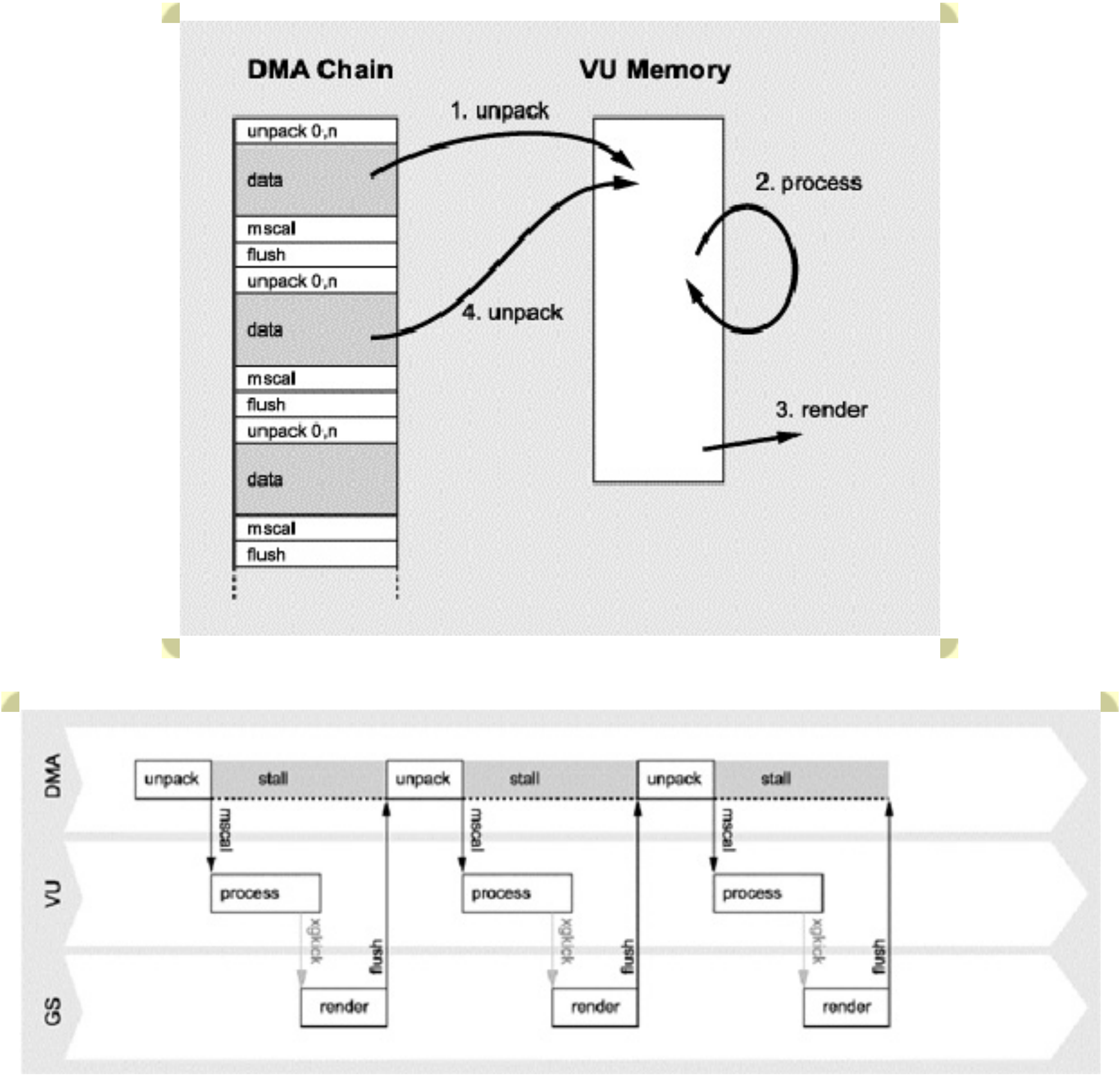
The other point to note is that none of these diagrams take into consideration the effect of texture uploads on the rendering sequence. This is a whole other tutorial for another day...

Single Buffer

Single buffering takes one big buffer of source data and processes it in-place. After processing the result is transferred to the GS with an xgkick, during which the DMA stream has to wait for completion of rendering before uploading new data for the next pass.

Benefits: Process a large amount of data in one go.

Drawbacks: Processing is nearly serial – DMA, VU and GS are mostly left waiting.



Single Buffer Layout and Timing

First, the VIF unpacks a chunk of data into VU1 Memory (`unpack`).

Next the VU program is called to process the data (`mscal`). When the data has been processed the result is transferred from VU1 Memory to the GS by an `xgkick` command. Because the GIF is going to be reading the transformed data from the VU memory we can't upload more data until the `xgkick` has finished, hence the need for a `flush`. (there are three VIF `flush` commands `flush`, `flushe` and `flusha`, where `flush` waits for the end of both the VU program and the data transfer to the GS.)

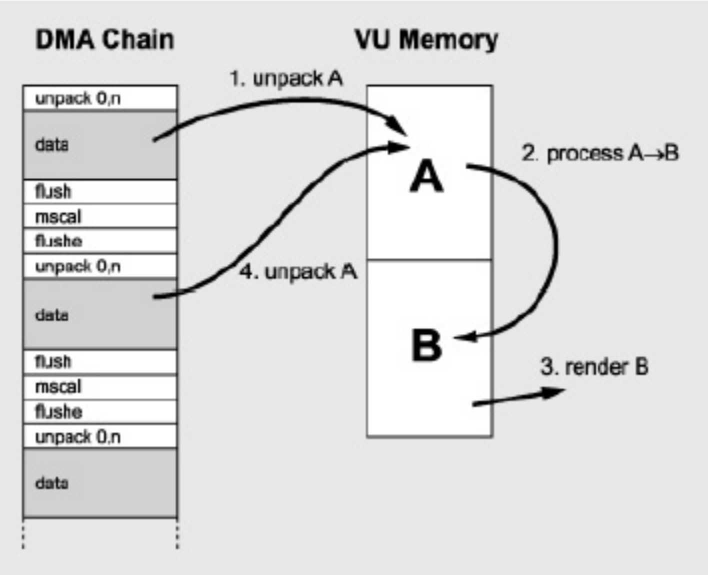
When the `flush` returns the process loops.

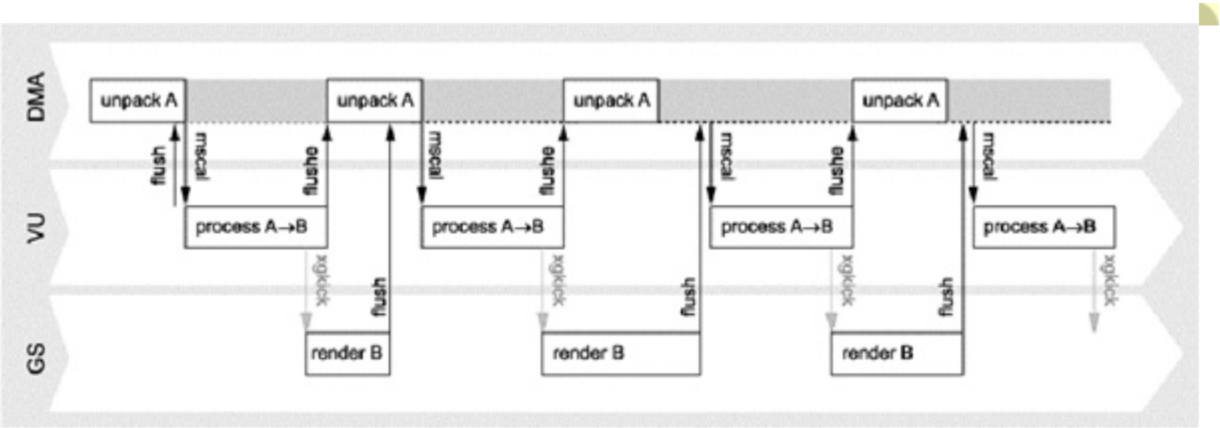
Double Buffer

Double Buffering speeds up the operation by allowing you to upload new data simultaneously with rendering the previous buffer.

Benefits: Uploading in parallel with rendering. Works with Data Amplification where the VU generates more data than uploaded, e.g. 16 verts expanded into a Bezier Patch. Areas A and B do not need to be the same size.

Drawbacks: Less data per chunk.





Double Buffer Layout and Timing

Although more parallel, VU calculation is still serialized. Data is unpacked to area A. DMA then waits for the VU to finish transferring buffer B to the GS with a flush (for the first iteration this should return immediately).

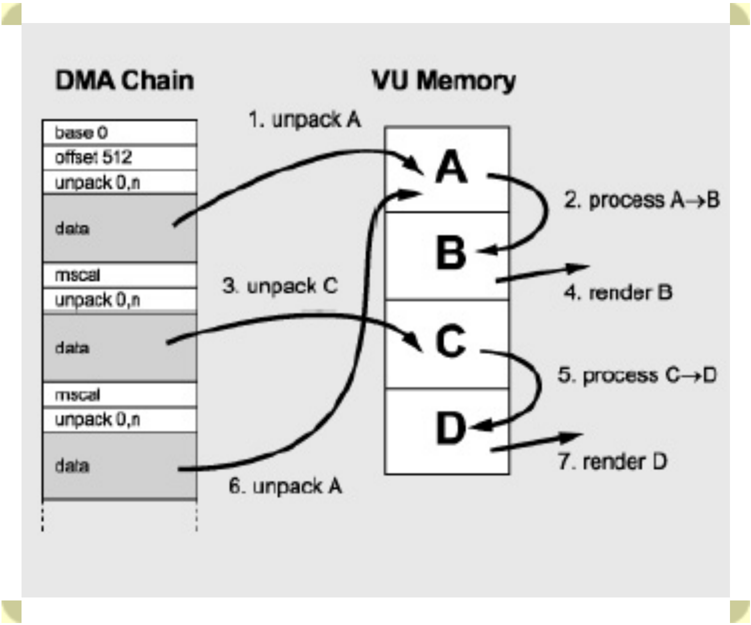
The VU then processes buffer A into buffer B (`mscal`) while the DMA stream waits for the program to finish (`flushe`). When the program has finished processing buffer A the DMA is free to upload more data into it, while simultaneously buffer B is being transferred to the GS via Path 1 (`xgkick`). The DMA stream then waits for buffer B to finish being transferred (`flush`) and the process loops back to the beginning.

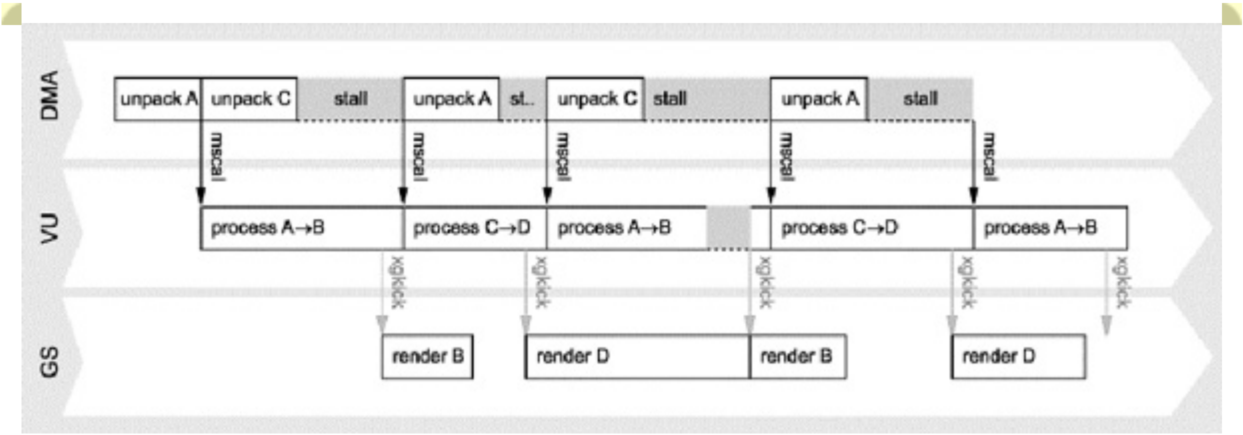
Quad Buffer

Quad buffering is the default choice for most PS2 VU programs. The VU memory is split into two areas of equal size, each area double buffered. When set up correctly, the TOP and TOPS VU registers will automatically transfer data to the correct buffers.

Benefits: Good use of parallelism – uploading, calculating and rendering all take place simultaneously, much like a RISC instruction pipeline. Works well with the double buffering registers TOP and TOPS, which may have caching advantages. The best technique for out-of-place processing of vertices or data amplification.

Drawbacks: Data can only be processed in <8KB chunks. There are three devices accessing the same area of memory at the same time – VU, VIF and GIF. The VU has read/write priority (at 300MHz) over the GIF (150MHz) which has priority over the VIF (150MHz). Higher priority devices cause lower priority devices to stall if there is any contention meaning there are hidden wait-states in this technique.





Quad Buffer Layout and Timing

First, the DMA stream sets the base and offset for double buffering – usually the base is 0 and the offset is half of VU1 memory, 512 quads.

The data is uploaded into buffer A (unpack), remembering to use the double buffer offset. The program is called (mscal) which swaps the TOP and TOPS registers, so any subsequent unpack instructions will be directed to buffer C.

The DMA stream then immediately unpacks data to buffer C and attempts to execute another mscal. This instruction cannot be executed as the VU is already running a program so the DMA stream will stall until the VU has finished processing buffer A into B.

When the VU has finished processing, the mscal will succeed causing the TOP and TOPS registers to again be swapped. The VU program will begin to process buffer C into D while simultaniously transferring buffer B to the GS.

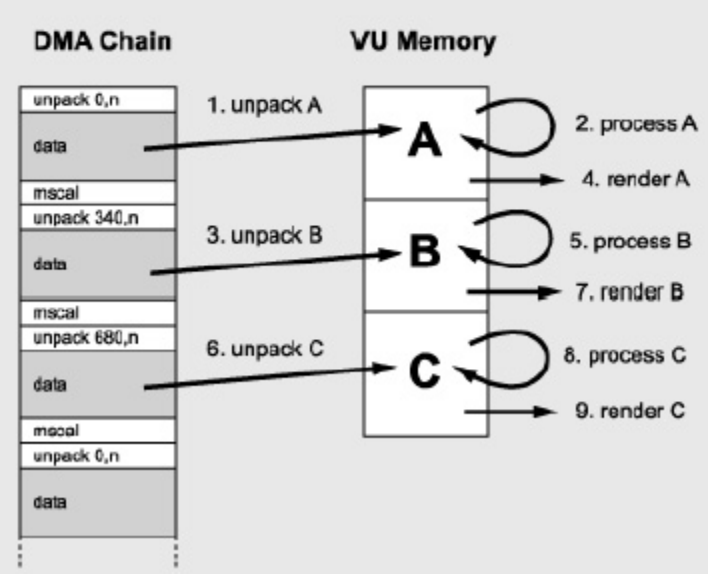
This process of stalls and buffer swaps continues until all VIF packets have been completed.

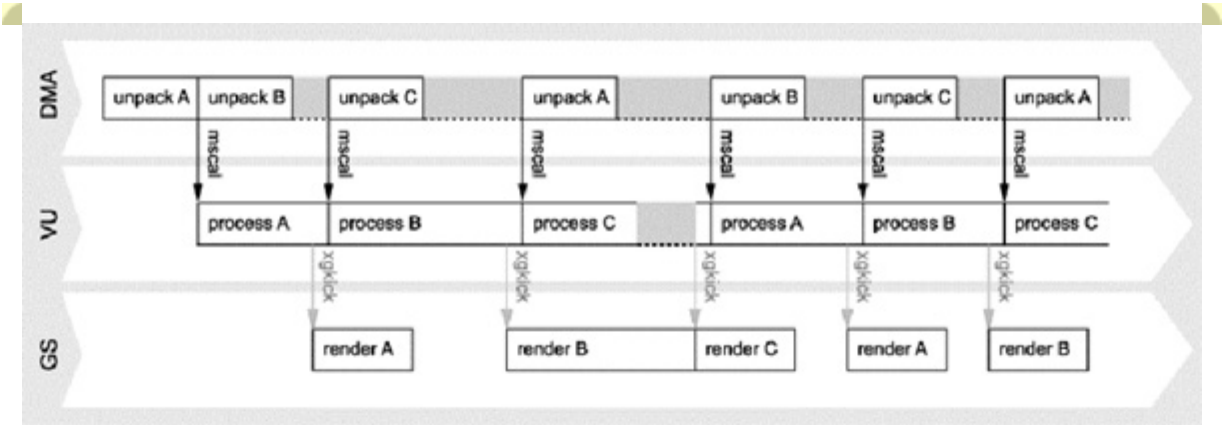
Triple Buffer

Another pipeline technique for parallel uploading, calculation and rendering, this technique relies on in-place processing of vertices.

Benefits: All the benefits of quad buffering with larger buffer sizes. Best technique for simple in-place transform and lighting of precalculated vertices.

Drawbacks: Cannot use TOP and TOPS registers – you must handle all offsets by hand and remember which buffer to use between VU programs. Three streams of read/writes again introduce hidden wait states.





Triple Buffer Layout and Timing

Data is transferred directly to buffer A (all destination pointers must be handled directly by the VIF codes – TOP and TOPS cannot be used) and processing is started on it.

Simultaneously, data is transferred to buffer B and another mscal is attempted. This will stall until processing of buffer A is finished.

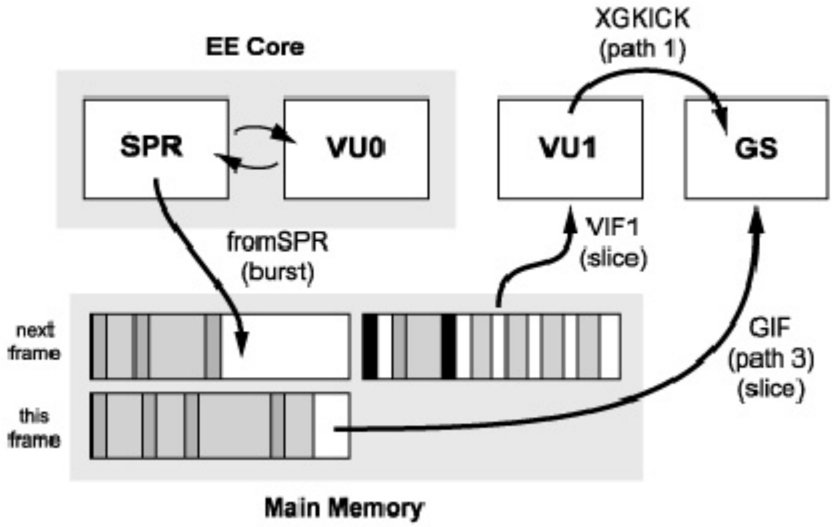
Processing on Buffer B is started while buffer A is being rendered (xgkick). Meanwhile buffer C is being uploaded. The three-buffer pipeline continues to rotate A->B->C until all VIF packets are completed.

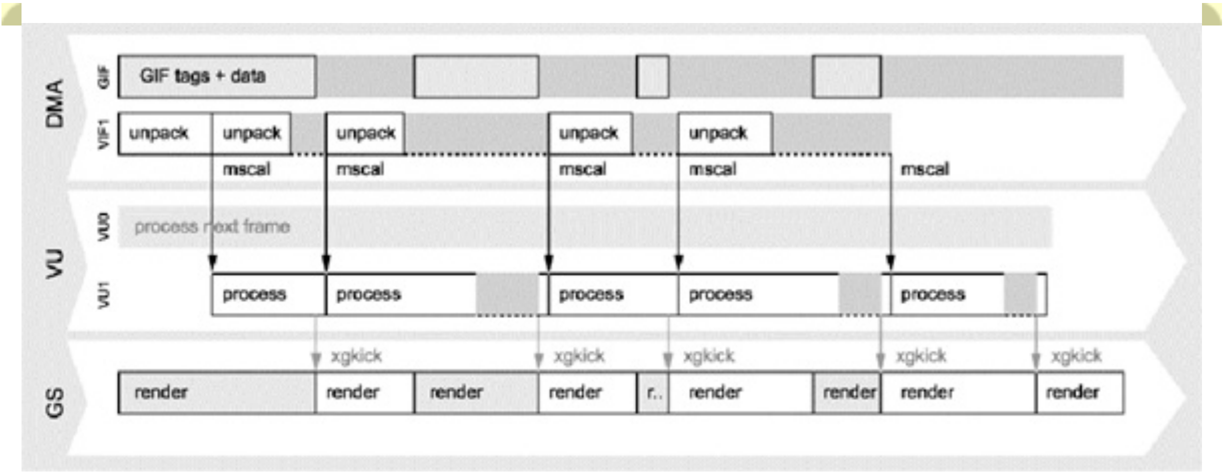
Parallel Processing

This technique is the simplest demonstration of how to use the PS2 to it's maximum – all units are fully stressed. All the previous techniques have used just one VU for processing and the GS has been left waiting for more data to render. In this example we use precalculated buffers of GIF tags and data to fill the gaps in GS processing, at the cost of large amounts of main memory. Many of the advanced techniques on PS2 are variations optimized to use less memory.

Benefits: All units are fully stressed. VU1 can be using any of the previous techniques for rendering.

Drawbacks: Moving data from VU0 to Scratchpad efficiently is a complex issue. Large amounts of main memory are needed as buffers.





Parallel Processing Data Flow and Timing

With VU1 running one of the previous techniques (e.g. quad buffering), the gaps in GS rendering are filled by a Path 3 DMA stream of GIF tags and data from main memory. Each of the GIF tags must be marked EOP=1 (end of primitive) allowing VU1 to interrupt the GIF tag stream at the end of any primitive in the stream.

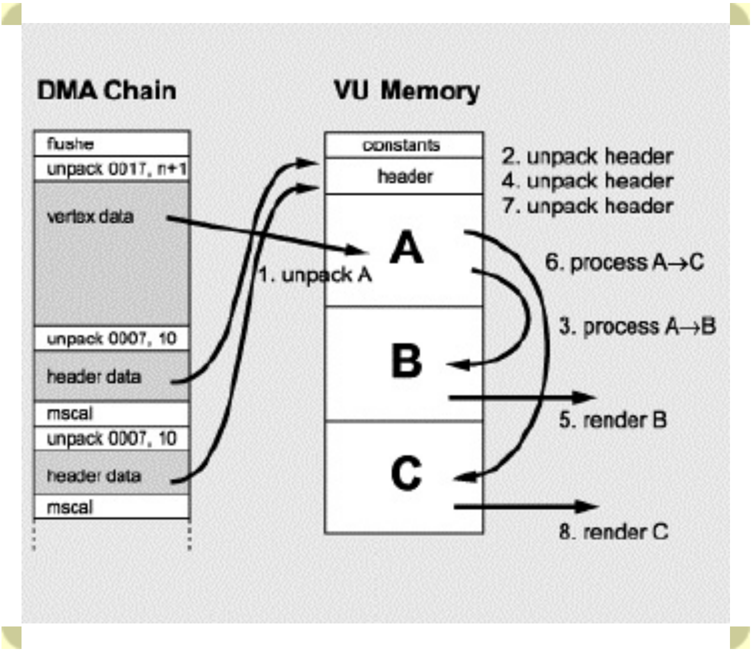
Data is moved from Scratchpad (SPR) to the next-frame buffer using burst mode. Using slice mode introduces too many delays in bus transfer as the DMAC has to arbitrate between three different streams. Better to allow the SPR data to hog the bus for quick one-off transfers.

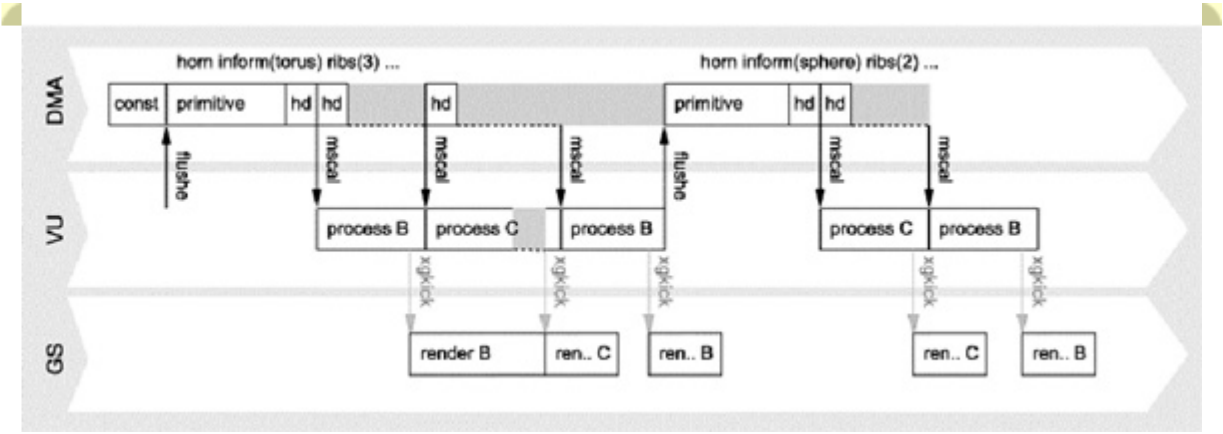
Note in the diagram below how both the VIF1 mscal and the VU1 xgkick instructions are subject to stalls if the receiving hardware is not ready for the new data.

The Design

The Lifeform program has to instance many copies of the same primitive, each one with a different object to world transformation matrix. To achieve this efficiently we can use a variation on the Triple buffering system. This explanation will add a few more practical detail than the earlier dataflow examples.

First, we unpack the data for an example primitive into buffer A. This data packet contains everything we need to render a single primitive – GIF tag and vertices – except the vertices are all in object space. We will need to transform the verts to world space and calculate RGB values for the vertices for Gouraud shading.





Timing diagram for the Lifeform renderer.

Next we upload an object-to-screen transformation matrix which is the concatenation of:

```
camera-screen * world-camera * object-world
```

where the object-world matrix was calculated by the Horn algorithm. Multiplying the object space vertices with this matrix will transform them directly to screen space ready for conversion to raster space.

We then execute the VU program which transforms the object space verts in buffer A to buffer B and `xgkicks` them.

Simultaneously we upload a new header to VU memory and attempt to start processing buffer B with an `mscal`, stalling until the VU has finished processing.

Here is a diagram of the data flow rendering two horns, the first a horn of three torii and the second a horn of two spheres. Due to the first torus, say, taking up a lot of screen space, it causes the xgkick of the second torus to wait until rendering is complete:

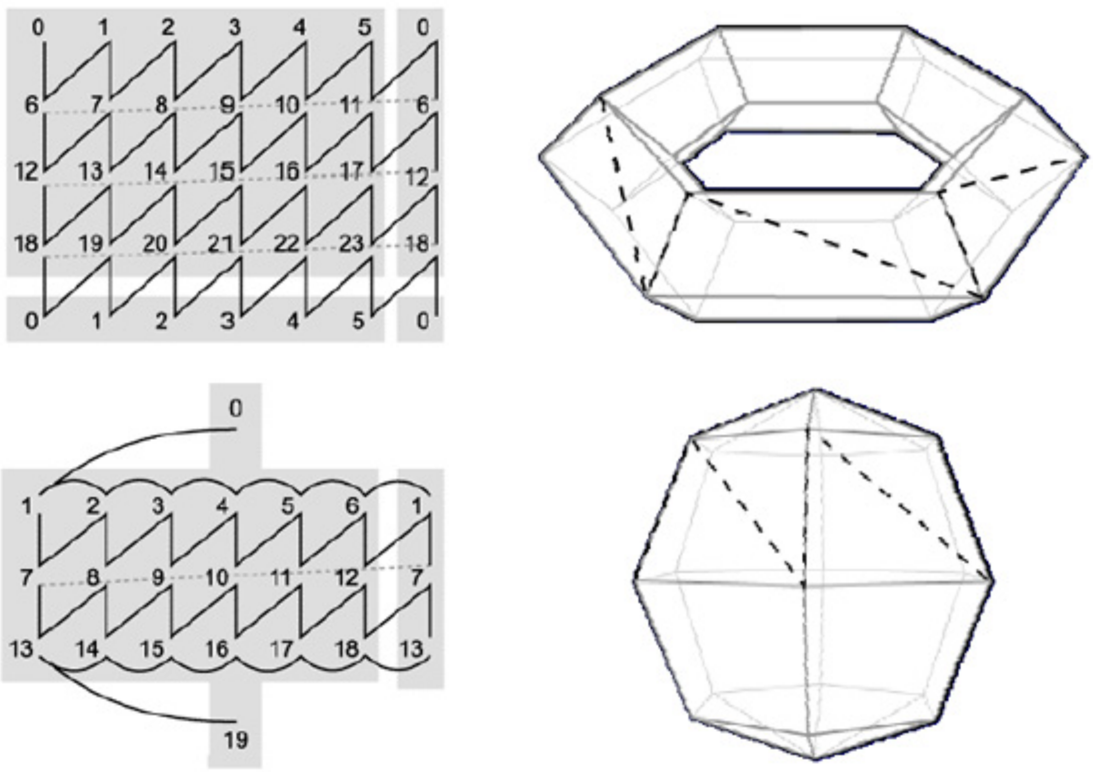
Untransformed Primitives

Because the models are going to be procedural, we have to calculate instances of the untransformed primitives to upload when they are needed. It would be useful if we could package the data up into a single DMA packet that could just be referenced (like a procedure call) rather than copying them every time they are needed. Using the DMA tags call and ret we can do just that.

We want the spheres and toruses to render as quickly as possible. Here we come to a dilemma. In PC graphics we are always told to minimize the bus bandwidth and use indexed primitives. It turns out that VU code is not well suited to indirection – it's optimized for blasting through VU instructions linearly. The overhead of reordering indexed primitives far outweighs the benefits in bus upload speed so, at least for this program, the best solution is just to produce long triangle strips.

A torus can simply be represented as one single long strip if we are allowed to skip rendering certain triangles. The ADC bit in can achieve this – we pass the ADC bit to the VU program in the w component of our surface normals, but you could just as easily pass it in the lowest bit of any 32-bit float. The accuracy is almost always not required.

Spheres cannot be properly described by a single tristrip without duplicating a lot of vertices and edges. Here I opted to produce spheres as (in this order) two trifans (top and bottom) plus one tristrip for the inner “bands” of triangles. This is the reason we have two VU programs – the sphere must embed three GIF tags in it's stream rather than just one for the torus.



Tristrip vertex orders fro 6x4 torus and a 6x4 sphere.

We are free to calculate the vertices as for an indexed primitive, but they must be stored as triangle strips in a DMA packet. Here is the code used to do the conversion for the torus and there is very similar code for the sphere.

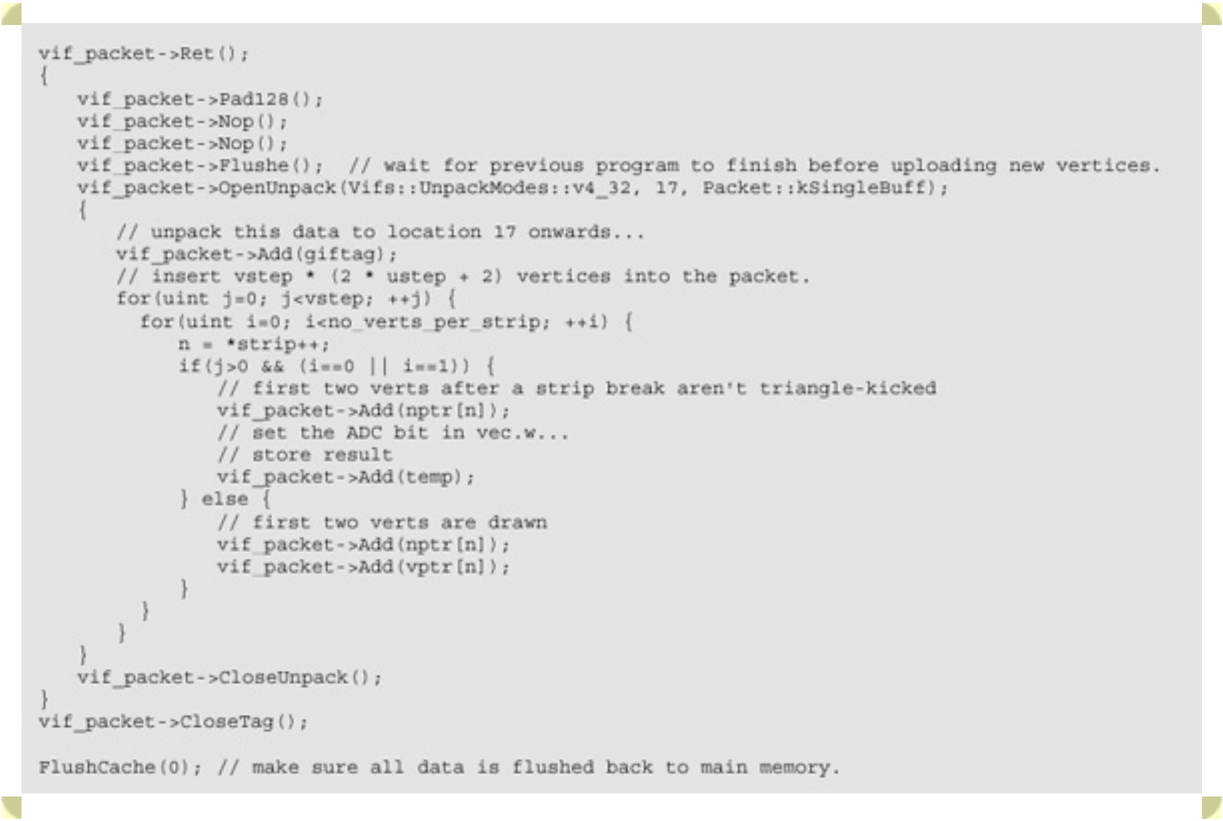
First we dynamically create a new DMA packet and the GIF tag:

```
            vif_packet = new CVifSCDmaPacket( (512,
                                              DMAC::Channels::vif1,
                                              Packet::kDontXferTags,
                                              Core::MemMappings::Normal );
```

Then we setup the pointers to our indexed vertices. Vertices are in `vertex_data[]` , normals are in `normal_data[]` and indices are in `strip_table[]`.

```
     uint128 *vptr = (uint128 *)vertex_data;
     uint128 *nptr = (uint128 *)normal_data;
     uint *strip = strip_table;
     uint n = 0;
```

Finally we loop over the vertex indices to produce a single packet of vertices and normals that can be used to instance toruses:



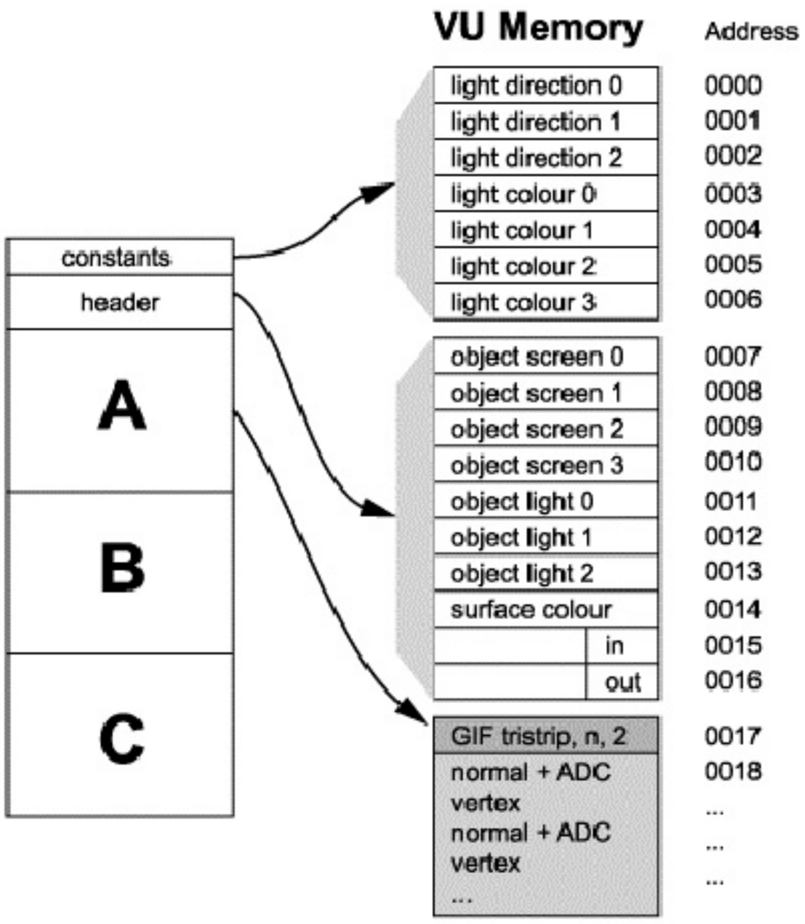
```
vif_packet->Ret();
{
   vif_packet->Pad128();
   vif_packet->Nop();
   vif_packet->Nop();
   vif_packet->Flushe();  // wait for previous program to finish before uploading new vertices.
   vif_packet->OpenUnpack(Vifs::UnpackModes::v4_32, 17, Packet::kSingleBuff);
   {
      // unpack this data to location 17 onwards...
      vif_packet->Add(giftag);
      // insert vstep * (2 * ustep + 2) vertices into the packet.
      for(uint j=0; j<vstep; ++j) {
        for(uint i=0; i<no_verts_per_strip; ++i) {
           n = *strip++;
           if(j>0 && (i==0 || i==1)) {
              // first two verts after a strip break aren't triangle-kicked
              vif_packet->Add(nptr[n]);
              // set the ADC bit in vec.w...
              // store result
              vif_packet->Add(temp);
           } else {
              // first two verts are drawn
              vif_packet->Add(nptr[n]);
              vif_packet->Add(vptr[n]);
           }
        }
      }
   }
   vif_packet->CloseUnpack();
}
vif_packet->CloseTag();

FlushCache(0); // make sure all data is flushed back to main memory.
```

VU Memory Layout

In order to actually write VU program to run this algorithm, we need to first block out the VU memory map so we can work out where to upload the matrices using VIF unpack commands.

Here is the layout I used. The VU memory is broken into five areas – the constants, the header, the untransformed vertices with GIF tags (buffer A) and the two buffers for transformed vertices (buffers B & C).



Constants. The constants for rendering a horn are a 3x3 light direction matrix for parallel lighting, and a 3x4 matrix of light colors (three RGB lights plus ambient).

The light direction matrix is a transposed matrix of unit vectors allowing lighting to be calculated as a single 3x3 matrix multiply. To light a vertex normal we have to calculate the dot product between the surface normal and direction to the light source. In math, the end result looks like this:

```
color = Ksurface * ( Ilight * N.L )
 = Kr * ( Ir * N.L )
   Kg * ( Ig * N.L )
   Kb * ( Ib * N.L )
```

where N is the unit surface normal, I is illumination and K is a reflectance function (e.g. the surface color).

Because the VU units don’t have a special dot product instruction we have to piece this together out of multiplies and adds. It turns out that doing three dot products takes the same time as doing one so we may as well use three light sources:

```
color = Ksurface * Sum(n, In * N.Ln)
 = Kr * (I0,r*N.L0 + I1,r*N.L1 + I2,r*N.L2 )
   Kg * (I0,r*N.L0 + I1,r*N.L1 + I2,r*N.L2 )
   Kb * (I0,r*N.L0 + I1,r*N.L1 + I2,r*N.L2 )
```

So, first we calculate the dot products into a single vector – this only works because our light vectors are stored in a transposed matrix:

```
                 x    y    z   w
lighting0 = | L0.x L1.x L2.x 0 |
lighting1 = | L0.y L1.y L2.y 0 |
lighting1 = | L0.z L1.z L2.z 0 |
```

```
N.L0 =   L0.x * N.x      + L0.y * N.y     + L0.z * N.z
N.L1 =   L1.x * N.x      + L1.y * N.y     + L1.z * N.z
N.L2 =   L2.x * N.x      + L2.y * N.y     + L2.z * N.z

         mulax.xyz …     madday.xyz …     maddz.xyz …
```

Then we multiply through by the light colors to get the final vertex color:

```
r =  I0r * N.L0      + I1r * N.L1      + I2r * N.L2      + 1.0 * Ar
g =  I0g * N.L0      + I1g * N.L1      + I2g * N.L2      + 1.0 * Ag
b =  I0b * N.L0      + I1b * N.L1      + I2b * N.L2      + 1.0 * Ab

     mulax.xyz …     madday.xyz …      maddaz.xyz …      maddw.xyz …
```

Header. The header contains the additional information needed to render this particular instance of the primitive – a 4x4 object-screen matrix, a 3x3 matrix to transform the normals into world space for lighting calculations, the surface color for this primitive, the address of the input matrix and the address of where to put the transformed vertices.

All this information is calculated during the previous frame, embedded in a DMA packet and uploaded once per primitive at rendering time.

Untransformed Vertices. After the header is stored a GIF Tag (from which we can work out the number of vertices in the packet) and the untransformed vertices and normals.

The VU Program

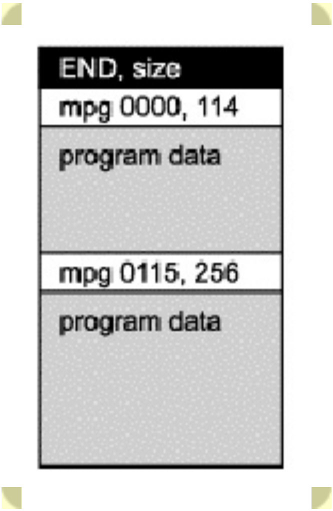
Now we have all the information needed to design the VU program. We know where the matrices are going to be stored, we know where to get our GIF tags from and we know how many verts need to be transformed for each primitive (it's the last 8 bits of the GIF tag). We will need to:

- Transform the vertex to screen space, divide by W and covert the result to an integer.
- Transform the normal into light space (a 3x3 matrix calculated by the horn function).
- Calculate N.L and multiply it by the light colors.
- Multiply the resulting light intensity by the surface color.
- Store the results in the output buffer and loop.

VCL compiles the program resulting in an inner loop of 22 cycles. This can be improved (see later) but it's not bad for so little effort.

Running Order

The first job the program has is to upload the VU programs to VU Program Memory. There are two programs in the packet, one for transforming and lighting toruses and one for transforming and lighting spheres, positioned one after the other. Uploading is achieved using an `mpg` VIF Tag that loads the program starting at a specified location in VU Program Memory.



A short script generates an object file that can be linked into your executable, and also defines four global variables for you to use as extern pointers. `vu1_packet_begin` and `vu1_packet_end` allow you to get the starting address and (should you want it) the length of the of the DMA packet. `torus_start_here` and `sphere_start_here` are the starting addresses of the two programs *relative to the start of VU Program Memory*. You can use these values for the mscal VIF instruction.

```
void upload_vu1_programs()
{
     CSCDmaPacket vu1_upload((uint128*) (&vu1_packet_begin),
                                        ((uint32)vu1_packet_end) /
                                        16,
                                        DMAC::Channels::vif1,
                                        Packet::kXferTags,
                                        Core::MemMappings::Normal,
                                        Packet::kFull );
 vu1_upload.Send();
}
```

The program then enters it’s rendering loop. The job of the rendering loop is to render the previous frame and calculate the DMA packets for the next frame. To do this we define two global DMA lists in uncached accelerated main memory:

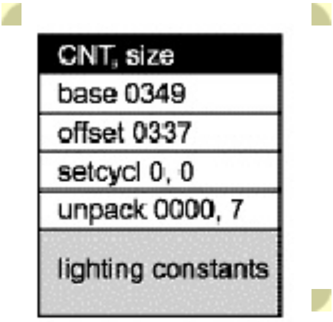
```
      CVifSCDmaPacket *packet = new CVifSCDmaPacket(80000,
                               DMAC::Channels::vif1,
                               Packet::kDontXferTags,
                               Core::MemMappings::UncachedAccl );
  CVifSCDmaPacket*last_packet = new CVifSCDmaPacket(80000,
                               DMAC::Channels::vif1,
                               Packet::kDontXferTags,
                               Core::MemMappings::UncachedAccl );
```

For the first iteration we fill the previous frame with an empty DMA tag so that it will do nothing.

```
last_packet->End();
 last_packet->CloseTag();
```

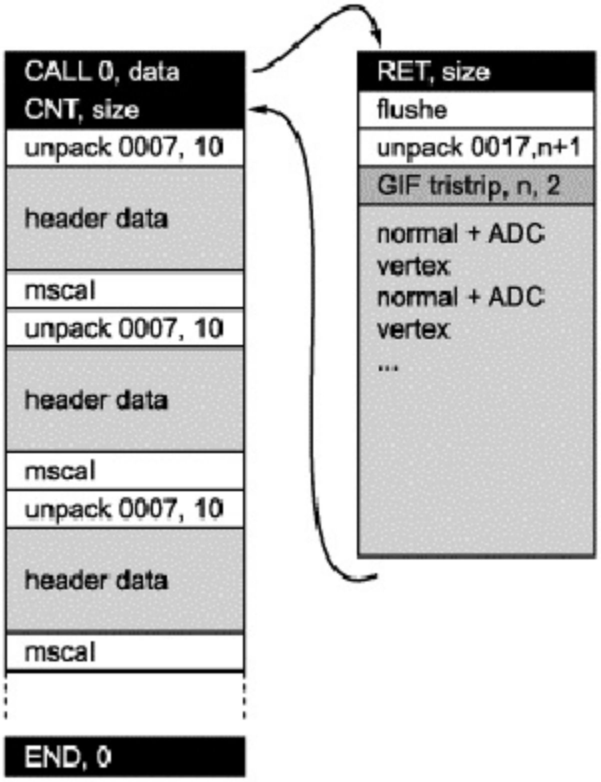
From this point on all data for the next frame gets appended to the end of the current DMA packet called, usefully, packet.

The next job is to upload the constants. This is done once per frame, just in case you want to animate the lighting for each render. Also in this packet we set up the double buffering base and offset.



```
void upload_light_constants(CVifSCDmaPacket *packet,
mat_44 &direction, mat_44 &color)
{
   // upload light constants at location 0 in VU1 memory
   packet->Cnt();
   {
    packet->Base(349);
    packet->Offset(337);
    packet->Stcycl(1,1);
    packet->OpenUnpack(Vifs::UnpackModes::v4_32, 0, Packet::kSingleBuff);
    {
     packet->Add(direction.get_col0());
     packet->Add(direction.get_col1());
     packet->Add(direction.get_col2());
     packet->Add(color.get_col0());
     packet->Add(color.get_col1());
     packet->Add(color.get_col2());
     packet->Add(color.get_col3());
    }
    packet->CloseUnpack();
   }
   packet->CloseTag();
  }
```

After all this it’s time to actually render some primitives. First we have to upload the untransformed vertices in to buffer A. These verts are calculated once, procedurally, at the beginning of the program and stored in a VIF RET packet, allowing the DMA stream to execute a call and return something like a function call.



```
 if(inform->type == torus) {
   packet->Call(*(inform->m_torus.vif_packet));
```

```
    packet->CloseTag();
 } else if(inform->type == sphere) {
    packet->Call(*(inform->m_sphere.vif_packet));
    packet->CloseTag();
 }
```

After the data had been uploaded to buffer A we can set about generating instances of the primitive. To do this, all we have to set the header information at and call the program. Lather, rinse, repeat.

```
  void Torus::add_header_packet(CVifSCDmaPacket *packet,
                                mat_44 &object_screen,
                                mat_44 &object_light,
                                vec_xyz surface_color)
  {
    packet->Nop();
    packet->Nop();
    packet->Flushe();
    packet->OpenUnpack(Vifs::UnpackModes::v4_32, 7, Packet::kSingleBuff);
    {
      packet->Add(object_screen); // 4x4 matrix
      packet->Add(object_light.get_col0());
      packet->Add(object_light.get_col1());
      packet->Add(object_light.get_col2());
      packet->Add(surface_color * 255.0f);
      packet->Add(input_buffer);
      packet->Add(output_buffer);
    }
    packet->CloseUnpack();
    packet->Mscal((uint32)torus_start_here >> 3);
    packet->Nop();
    packet->Nop();
    packet->Nop();
  }
```

So we've generated all the horns and filled the DMA stream for the next frame. All that's left to do is to flip the double buffered screen to show the previous render, swap the buffer pointers (making the current packet into the previous packet) and render the previous frame.

```
     // wait for vsync
     wait_for_vsync();

     // wait for the current frame to finish drawing (should be done by
     now)...
     wait_for_GS_to_finish();

     // ...then swap double buffers...
     swap_screen_double_buffers();

     // ... and send the next frame for rendering.
     packet->Send(Packet::kDontWait, Packet::kFlushCache);

     // swap active and previous packets.
     CVifSCDmaPacket *temp_packet = packet;
     packet = last_packet;
     last_packet = temp_packet;

     // clear the current packet for new data
     packet->Reset();
```

Further Optimizations and Tweaks

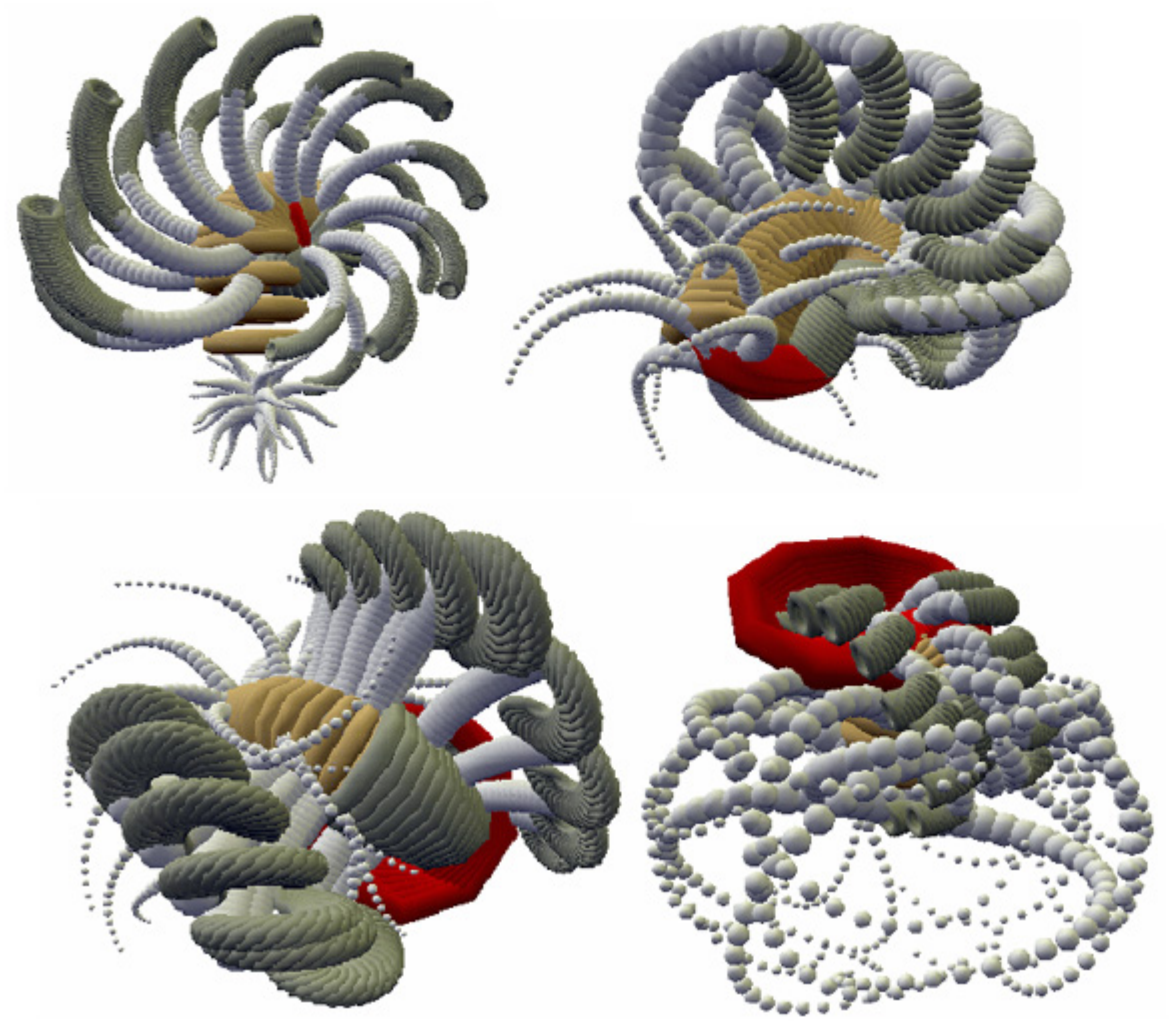
The program as it stands is not as optimal as it could be. Here are a couple of ideas for increasing the speed of the program.

- **Move the light transform out of the rendering inner loop.** The inner loop currently stands at 22 cycles per vertex, mainly because each vertex normal has to be transformed from object space into world space for lighting. There are tens of normals per primitive but only one lighting matrix. It would be more efficient to transform the light direction matrix to object space once per primitive and use that matrix for lighting calculations. This would save at least 4 cycles per vertex.
- **Double buffer the Header.** Double buffering the header would allow you to remove the `Flushe()` in rendering the primitives.
- **Load the constants only once.** A tiny optimization with little real effect on polygon throughput (seven cycles per primitive), but it would tidy things up a little.
- **Code the horn algorithm as a VU0 Micro Mode program.** The slowest part of the program are the string of matrix multiplies used to position and scale each primitive. Each matrix multiply, although executed using Macro Mode VU0 code, is not using VU0 to it's full. The routine could be coded as a VU0 Micro Mode program that takes the variables needed to specify a horn and generates the 3 to 50 matrices needed per horn in one go. The major problem with this conversion is that VU0 has no instructions for calculating `sin()` and `cos()` or calculating a `powf(),` but these are just programming problems. Algorithms and table based approximations for these functions are simple to find on the net. For example, we only have to evaluate `sin()` and `cos()` at fixed intervals allowing us to use forward differencing, Taylor Series approximations or Goertzels algorithm to generate the values. Other functions can be derived from more primitive power series:

  ```
  powf(float x,float y) = exp( y * log(x) )
  ```

  The only drawback of this technique is that the EE Core will have to wait for the full set of matrices to be returned before it can generate the VU header packets. However, if you think about it, the EE Core is already waiting for VU0 in tiny pieces scattered throughout the programs execution. The work has to be done anyway so why not batch it together and use the computation time constructively.
- **Move the whole Horn algorithm into VU1.** It's possible to move the whole algorithm into VU1, even to the point of generating the vertices of each primitive at runtime. The benefits to bus bandwidth are obvious – all you send across are a handful of instructions and floats per horn and nothing more, plus there would be a lot of satisfaction in creating such a monster program. The drawback is more sticky though **-** you would again be serializing the operation to only one unit. It may ultimately be more efficient to distribute the job over several processors.

Results



Game Developer Conference 2003: Porting a PS2centric Game to the Xbox: A Case Study of *State of Emergency*

By Jonathan Dobson and Peter Brace

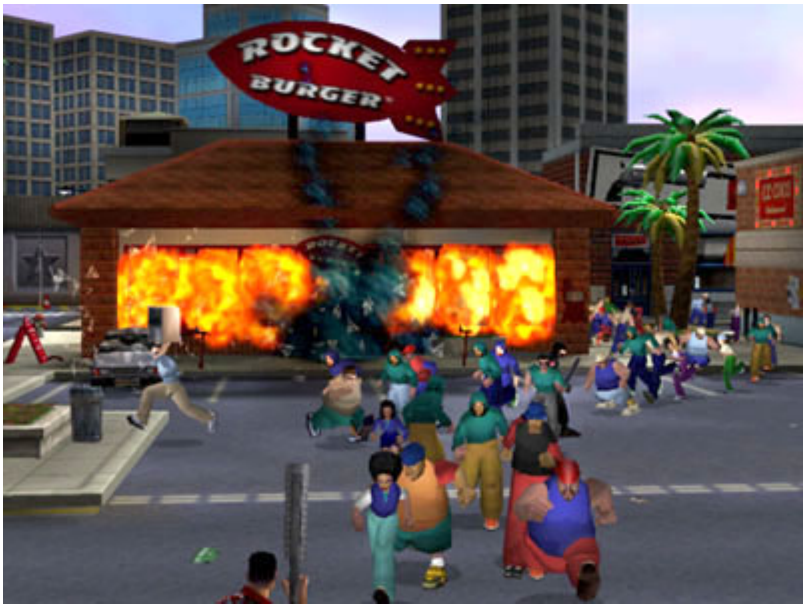
This article describes the challenges that were encountered when porting *State of Emergency (SOE)* from the PlayStation 2 to the Xbox. *SOE* was developed as a PS2 game; the underlying rendering and animation technology consists of over 5000 lines of hand written vector unit assembly code. The biggest hurdle to overcome was the fact the entire game was built around code that was hand-tuned to derive the maximum performance from the PS2 architecture.

This article shows that the approach taken was to develop a very specialized Xbox rendering engine that was optimized to perform extremely well in areas of the engine that had relied on the high-performance of specific PS2 hardware. In particular, the code that previously had utilized the vector unit processors.

The resulting Xbox engine is capable of displaying the same number of characters as the PS2 version, but at 60 frames-per-second (fps), as opposed to 30fps. The article also shows that different high and low-level optimizations were required make *SOE* work on the contrasting architecture of the Xbox, and to achieve higher levels of performance.

The Original Concept

The original concept for *State of Emergency (SOE)* was that of a riot simulator. The player would control hundreds of characters, who in turn, could fight hundreds of other characters. The game would be a hybrid of the strategy and action genres. Development on *SOE* started on the Dreamcast, the intention was that *SOE* would be a game that was truly "next-generation". The core feature of the game, namely the number and complexity of the characters contained within it, would simply not have been possible on previous generations of consoles.



***SOE*'s target audience more close fitted the PS2's target demographic than the Dreamcast for which it was originally planned.**

A 2D prototype of the game was produced. This demonstrated the gameplay, but not the graphics. However the gameplay was not engaging enough, so the style of game was changed to a 3rd person action game, rather than a mix of two quite different genres.

It was also decided to change the lead platform. During the pre-production period on *SOE*, Sony announced the development of the PlayStation 2 (PS2). The developers of *SOE*, VIS entertainment, and the publishers, Rockstar Games, decided that the target demographic for the PS2 more closely fitted with the target audience of *SOE*.

We saw *SOE* as an opportunity to develop a high-performance PS2 engine, close to the release of the PS2, which could be used as the basis for a number of titles throughout the PS2's lifetime.

Requirements For PS2 Version

The main requirement of the game on the PS2 was that it should be able to display as many characters as is possible. This meant that a highly specialized renderer was needed. Not only was it a requirement that large number of animated characters should be displayed on the screen, but these characters had to have complex Artificial Intelligence (AI) and route finding code controlling their behavior.

The approach taken was to develop a PS2 renderer with an integrated character animation and rendering subsystem. This was designed to exploit the relationship between the PS2's main CPU (the Emotion Engine, or EE) and its Graphics Processor Unit (GPU, made of up two Vector Units, or VUs). The PS2 renderer performs as much of the character animation as possible on the VUs. This maximizes the available EE time, so that the AI computations can happen in parallel with the rendering.

The renderer was developed to minimize the amount of online processing needed for scene rendering. To achieve this, an offline exporter was written that converted geometry data from Maya, into the final format that would be sent via Direct Memory Access (DMA) to the VUs on the PS2. This introduced a certain amount of inefficiency in terms of the memory usage, since the data was optimized for speed, and not necessarily storage. In addition, the geometry data was not in a format that was readily accessible by the game engine, which meant that game data, such as collision information, needed to be stored separately, and in addition to the display data.

The maximizing of the parallelization, and the offline generation of renderer specific data, especially with the character rendering and AI, was the key feature of the PS2 renderer that allows *SOE* to display and process so many characters simultaneously. In a typical scene within *SOE*, over 100 characters are visible, with the behavior of up to 250 characters being processed within a single frame.

Conversion Timeline

It should be noted that the conversion of the code to other platforms, specifically the Xbox, was not considered during the development of the original PS2 game. The reason for adopting this development methodology was because the game engine, and the renderer were specifically geared towards the hardware of the PS2. To achieve the required number of visible characters, over 5000 lines of VU assembly language was produced. If cross-platform requirements had been considered at that point, then the performance of the PS2 version would almost certainly have been compromised.

Following the release, and subsequent success of *SOE* on the PS2, the decision was made to port the game to the Xbox. The decision was spurred on by investigations, implemented on the PS2, into multiplayer split-screen modes, immediately following the games release. The best focus testing available for a game is achieved by releasing it to the public, and in the Xbox version an opportunity was seen to evolve the gameplay of *SOE*, based on feedback from the people who actually bought it.

The content of the Xbox version, and the scope of work involved, was also influenced by the fact that the game had a twelve-month exclusivity deal on the PS2. From the outset it was known that the game would not be released until February 2003 at the earliest, so this allowed more time than is often available for an average game port.

SOE Xbox was an eight-month project, from May to December 2002. There was a core team of fourteen people, although a number of these people were part time, and the exact size of the team fluctuated throughout the development, depending on the needs of the project. On average there were eight full-time people on the project. The majority of the work was software development, with between three and four programmers working on the renderer engine, and the conversion of the game code itself. In addition, there was one full-time game designer, who was responsible for producing the mission scripts, a Director and Producer, and finally up to seven artists, who were responsible for increasing the resolution of both textures, and meshes.

The eight months were broken down into five months production, two months alpha-to-beta, and one month beta-to-gold-master, with the game passing Microsoft final submission on Christmas Eve 2002.

The Initial Conversion

The first three months of development involved work progressing on three different fronts. Firstly the Research and Development team started to develop the Xbox renderer; secondly the game designer and Director began reviewing and revising the missions on a PS2, and thirdly the game programmers started converting the game engine to run on the Xbox.
Since the game engine did not directly access the render data for the PS2 version, it was possible to compile and run the game by simply stubbing out the renderer function calls. In addition, the virtual memory

management system on the Xbox allowed the conversion of the game code to be achieved very quickly. It was possible to allocate memory for the game engine data at exactly the same address used on the PS2, this meant that the binary map data could be loaded in completely unchanged, even though it included absolute memory addresses. The game was compiling and running on the Xbox within a few hours - although without anything visible on the TV screen!

Since the Xbox development kit is based around DirectX, a functional (albeit un-optimized) renderer was produced within a short space of time. The renderer functions used by SOE on the PS2, which had been stubbed out during the conversion of the game code, were now re-implemented in a wrapper library that converted the DirectX based renderer, to function calls that were identical to those used on the PS2.

This initial conversion took about three months, with much of the work being the development of the DirectX renderer, and the wrapper library that allowed *SOE* to use the renderer without many changes to the game code.

The result of the first playable version on the Xbox was disappointing. It had been assumed (incorrectly, with hindsight), that the raw power of the Xbox would result in an implementation that was as fast, if not faster than the PS2 version. However, this was the not case. *SOE* on the Xbox actually ran slower than on the PS2. While the PS2 maintained 30 frames-per-second (fps) in nearly every scene, and about 50% of the time could theoretically render at 60fps, on the Xbox *SOE* maintained 30fps less than half the time, and often dropped down to 15fps, or even 10fps on many occasions.

The Architectural Differences

This initial conversion of *SOE* to the Xbox instantly showed how the different architectures of the PS2 and the Xbox resulted in a radically different performance.

On the PS2, the relationship between the game engine CPU requirements, and the renderer performance was finely balanced. In fact, as described in Section 2, *SOE* achieved the results on the PS2 because the parallelism between the EE and the VUs was maximized. The EE processor cycle requirements for the rendering, of everything within the game, was very small. This left a large amount of CPU time available for the game engine, specifically the AI behavior. This was very important, since the AI processing and route finding needed to be performed for many Non-Player Characters (NPCs) in every frame. The nature of the AI engine, and the design of the PS2 meant that this behavioral processing required as much processor time as could be made available. The AI engine did not perform particularly efficiently in terms of data and instruction cache usage. At the most basic level, the AI engine is a loop, with a large amount of complex code to be performed on a different data set, for each iteration. The EE does not aggressively attempt to improve the performance of this sort of code automatically, in the same way that some other processors do. So, if anything, the bottleneck on the PS2 was the AI processing.

The complete reverse was true on the Xbox. The rendering side of the application was written in DirectX, which is an additional layer of abstraction from the hardware level. This means that the rendering of geometry had a significant CPU requirement, and the same scenes actually took longer, in time, than the PS2 version to render. Conversely the game engine, and the AI code in particular, performed significantly faster without any changes. This was to be expected, since the raw power of the Xbox CPU is arguably higher than the PS2's EE, and the AI code was more suited to Xbox processor.

However, the overall performance on the PS2 had been achieved by balancing the CPU and GPU load to maximize parallelism. This parallelism was not immediately apparent on the Xbox, so initially, the same performance levels could not be reached.

Xbox Specific Optimizations

To reach the same level of performance the game achieved on the PS2, it was apparent that some significant optimizations would be required. It was also thought that by spending some additional time on the optimization, performance levels could be increased so that the game could run at 60fps for the single player mode, and 30fps for the split-screen multiplayer modes.

***State of Emergency* Screenshot (PS2)**

There were three main optimizations that were implemented: pushbuffers, animation caching, and reorganizing the frame processing to maximize parallelization. Before any optimization was started, the main bottleneck was the CPU usage.

- Pushbuffers. Pushbuffers are the native rendering format of the Xbox GPU. When DirectX is used on the Xbox, the rendering instructions are converted into pushbuffers for execution on the GPU. A small subset of commands within the main render loop: draw primitive, z-writes, set vertexbuffer, select vertexshader, set pixelshader, etc, were reverse engineered to determine the resulting pushbuffer code. This was then used to develop an offline converter application that constructs a large pushbuffer for the whole level. Each object in the pushbuffer is preceded by a null operation (NOP), which is conditionally set as a NOP, if the following object is visible, and a jump, if the following object is not visible. The concept is similar to "execute buffers" rendering techniques in DirectX 3.0. This pushbuffer technique resulted in a reduction in the CPU set up time required for the rendering, essentially halving the frame time.
- Animation Caching. The animation system on the PS2 was implemented using VU assembly code, which had a negligible CPU requirement. However, on the Xbox the animation code is implemented on the CPU, which contributed to the CPU bottleneck.

  To optimize the animation code, it was assumed that many characters would be using the same animations, since the majority of the visible NPCs spend their timing running about in a panic! In addition, there are only three different skeleton types in *SOE*; therefore most of the animation matrices could be pre-calculated even with very limited extra storage. A chunk of memory was set up to store the final animation matrices, half of this storage is used as a cache. Each cached set of matrices has a header describing which skeleton, animation, and frame is stored, together with a score of how useful this set had been so far and a reference counter. The score is decremented each frame, and the slot is considered free if the reference count reaches zero, and the score is low. To improve the search speed, the first x slots were set aside for frames 0, 10, 20, the second x slots were set aside for frames 1, 11, 21, etc. If the cache is full, or the matrices need changing, there is non-cached space at the end of the memory; these entries contain the set of joint matrices, but no additional information, except the reference count.

- Renderloop Restructuring. The two solutions I just described massively reduced CPU usage, but

> towards the end of the project, the GPU usage in the game started to increase. This was partly due to the change from DirectX primitives to pushbuffers, but also as the performance of the engine increased, we increased the volume of data rendered. There was an additional problem, during the first third of a frame there was a large amount of CPU usage, followed by a large amount of GPU usage, with small amounts of CPU code, which blocked waiting for the GPU to become free.
>
> The final optimization was to spread the processing over two frames, so that at the start of a frame, the pushbuffer is executed. This takes lots of GPU, but requires no CPU after it has started. While this is running, all of the CPU intensive, game engine code is run, and the render lists are stored for the next frame. This balanced the CPU and GPU usage in a similar way to the PS2, maximizing the parallelization, and allowing the Xbox to maintain 60fps for most of the gameplay.

Differences In The Xbox Version

Since *SOE* Xbox was not a direct port from the original we had the opportunity to evolve the game beyond what achieved with the PS2 version. Specifically we revised the missions contained within the "Revolution" mode, we also made some improvements to the art within the game, and finally we added a multiplayer, split-screen extension of the arcade mode.

Missions Revisited

The core mechanic of *SOE* was the crowd, and the behavior of the NPCs. Specifically NPCs would react to events, and other NPCs would react to those reactions. From essentially simplistic building blocks, fairly complex, and non-design behavior would evolve. This made the world of *SOE* feel alive. The player might follow some enforcers around a street corner, and discover that they were running to break up a fight between two rival gangs.

However, since this behavior was emergent, and not designed into the game from the start, it was quite incidental, and did not play a significant part during the mission game. For *SOE* Xbox, the mission game was reviewed extensively. Because of the three month gap between finishing the PS2 game, and starting the port to the Xbox, the development team could take a fairly detached view concerning the quality, and "enjoyablity" of the missions.

Missions that the majority of the development team did not enjoy, were either removed, or significantly revised, with the intention of making the game more fun. For example, a mission that required the player to prevent an unarmed, pacifist from being killed by large number of armed enforcers was not enjoyable! Missions that involved the player being aided by a large number of armed allies, killing an even larger number of enemy NPC were much more fun! Unfortunately this latter type of mission was in the minority, specifically because the mechanic that allowed that type of gameplay had evolved from the emergent AI implementation.

Art Improvements

The Xbox port gave the art team the opportunity to revise some of the game art in *SOE*. On the PS2, the memory usage was very tight, with the majority of levels using all but a few kilobytes of the 32Mbytes of main memory available. Since the Xbox has 64Mbytes of RAM, it was possible to include some additional artwork.

Firstly, the textures were increased in resolution. For memory and texture cache issues, the PS2 predominantly had textures with the dimensions of 64x64 texels. In many cases the texel to pixel ratio in *SOE* was actually 1:1, but an in depth review of the textures revealed a number of instances where this wasn't the case, and those textures were increase to 128x128 texels. In addition, the PS2 used palletized texture to reduce memory, and increase rendering speed. The Xbox version uses the original 24-bit source textures, with DXT compression.
Secondly, the additional memory allowed the team to produce some mission specific characters to add variation to the game. For example, in *SOE* PS2 all of the mission givers were identical, but on the Xbox there are six variations.

Finally, some elements of the maps were augmented with some exotic shaders. Selected cars, windows, and street signs had environment, and specular maps added, and in some cases an additional gloss map component. Specifically the floor in The Mall level was given a reflective environment map surface. To improve the appearance of these surfaces, some of the object using them, such as the cars, were increased in polygon resolution.

Multiplayer Modes

The biggest addition to *SOE* Xbox was the multiplayer versions of the arcade mode. From the beginning the renderer was designed with split-screen capabilities, even on the PS2. In fact, the game modes were initially prototyped on the PS2 before the Xbox renderer was functional.

It was quickly found that some of the elements of *SOE* do not lend themselves to multiplayer gaming. For example, the hand-to-hand combat works in the single player game when the opponent is as "up for the fight" as the main player. However, even in the single player *SOE*, fighting against civilians who try and run away when attacked, is quite difficult. It would have be possible to redesign the combat so that it is more applicable to multiplayer fighting, however this would have resulted in a the mechanic that was different from the single player game. It was decided to bias the multiplayer modes away from ones that concentrated on hand-to-hand fights with human opponents.

As noted in Section 5, the scene rendering in the *SOE* Xbox requires more CPU time that the game engine processing of the same scene. Much of this is the geometry setup time, so the multiplayer modes run at 30fps, as opposed to 60fps.

Unanticipated Challenges

This section describes a number of issues that arose during the development of *SOE* Xbox, that were not originally anticipated during the planning stages of the project.

Incorporating Multiplayer Functionality

SOE was designed as a single player game from the start. This meant that the multiplayer element had to be retrofitted to the existing game engine. On top of this, the existing game was required to continue to work with all of the single player game modes.

The initial extension of the game to multiple players was fairly straightforward, and was implemented in roughly two days. This was aided by the inclusion of split-screen functionality in the renderer. Also, the main player character, and the NPCs were stored as the same basic datatype within the engine. This meant that it was relatively straightforward to alter characters that were previously AI controlled NPCs, to be player controlled characters. While this allowed the initial multiplayer code to be added quickly, a lot of game code assumed that the only user controlled player was character zero, and that all other characters where under AI control. A significant amount of QA and programmer time was spent identifying and fixing instances where the game logic didn't behave consistently for all of the human controlled characters.

Bugs Due to Hardware Differences

When converting a project from one platform to another, during the conversion process it is not unusual to discover bugs that had existed in the original. However the effect of these bugs was not anticipated during the initial stages of converting *SOE* to the Xbox. Specifically a number of code errors were discovered that produced undesirable effects (i.e. crashes) on the Xbox.

In many of these cases, these bugs didn't result in anywhere near as fatal behavior on the PS2. Most of these bugs were due to errors in floating point calculations, specifically ones that resulted in "not-a-number" (NaN) values. On closing inspection, it was found that the floating point unit (FPU) on the PS2 is much more robust when dealing with arithmetic operations that result in, and subsequently operate on, NaN values.

In addition to the FPU errors, bugs due to the slightly different memory map in *SOE* Xbox resulted in memory corruption errors that could not have occurred on the PS2. As usual, these proved particularly tricky to track down.

Results

For most of the scenes in *SOE* Xbox, the game runs at 60fps in the single player mode, and 30fps in the multiplayer split-screen modes. This compares favourably with the single player only PS2 version, which mostly maintains 30fps.

There are some areas and occasions within the game where the framerate drops below a constant 60fps, although these are also areas where the PS2 version sometimes has issues maintaining a consistent framerate.

The two player game is capable of running at 60fps for about 50% of the time, but it was decided that the optimizations required to maintain this performance would have resulted in a reduction of quality in both the art and gameplay, so it is locked at 30fps. The three and four player games also maintain 30fps for much of the game.

State of Emergency **Screenshot (XBOX)**

Conclusions

This article has described the conversion of *SOE* from the PS2 to the Xbox. It has shown that the original game engine was highly optimized for the PS2 architecture, and the resulting performance was due to the development of an in-house rendering written primarily in VU assembly language. This allowed the engine to maximize the parallelization between the CPU and the GPU. The resulting performance on the PS2 would not have been achievable without this finely tuned low-level renderer that was designed to perform very efficiently in the areas that *SOE* required.

Since the performance of the game was a result of this exploitation of the PS2 architecture, the initial conversion to Xbox, resulted in lower overall performance than the PS2. Even though, based on raw processing power, the Xbox might have been expected to perform at least the same, if not better, than the PS2. This necessitated a change to the Xbox renderer, specifically converting the code from using DirectX drawing primitives, to pushbuffers, the native rendering format of the Xbox GPU. The result is that *SOE* on the Xbox runs at 60fps in the single player mode, twice the frame-rate of the original PS2 version.

Overall, the performance of *SOE* on both platforms was only achieved by the development of a specific renderer that performed well in areas that required dedicated low-level processing, namely the character animation. In addition, the increased frame rate on the Xbox could only be achieved by processing the geometry offline into a format ready for executing directly on the GPU. This increased the available CPU time for the game engine code, and maximized the parallelization of the engine.

ACKNOWLEDGMENTS

Dr. Jonathan M. Dobson and Peter A. Brace thank both the original *State of Emergency* team, and the team responsible for the conversion of the title to the Xbox, for their support and dedication. Specifically Richard Reavy, Head of R&D Department; Software Developer Chris Trewartha; Artists Iain McNaughton, Max Cant, Malcolm Shortt, Anna Chalk, Russell Hughes, Rachel Segans, and Paul Webb; Director Jamie Bryan, and Producer Alison Wilson. Finally, thanks go to Rockstar Games, for their invaluable support and advice throughout the development of *State of Emergency*, and the conversion of the game to the Xbox

---

From a series of articles at **GamaSutra.com**

Back to **VazGames** www.PhilVaz.com/games