Procedural Rendering on Playstation® 2

Robin Green
R&D Programmer
Sony Computer Entertainment America

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:

1. **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.
2. **It can generate a lot of polygons quickly.** Procedural models are scalable to the point where they can swamp even the fastest graphics system.
3. **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.
4. **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.
5. **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.

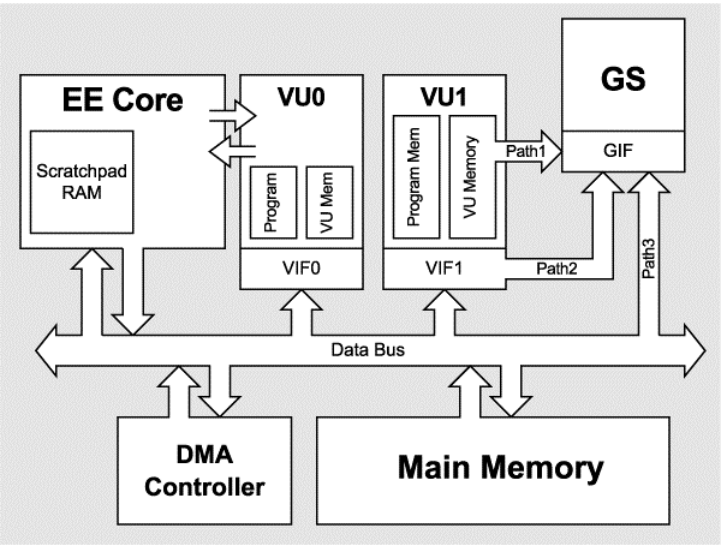


Figure 1: Simplified PS2 Block diagram

Lifeforms

First, a look at the type of model we're going to be rendering. The ideas for these models 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.

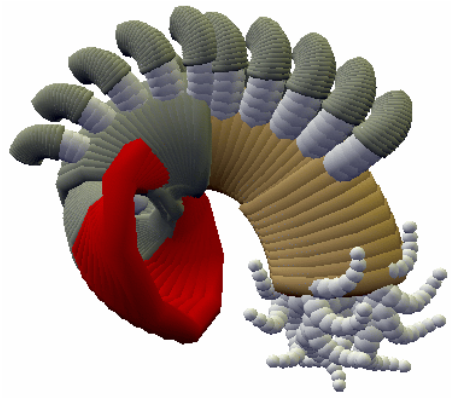


Figure 2. An instance of the Lobster class.

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) ;
```

Figure 3. 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%. This is not to say that each rib will be 90% 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.

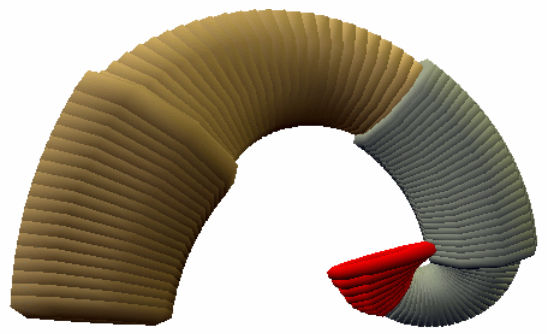


Figure 4: A backbone made from chained horns.

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.

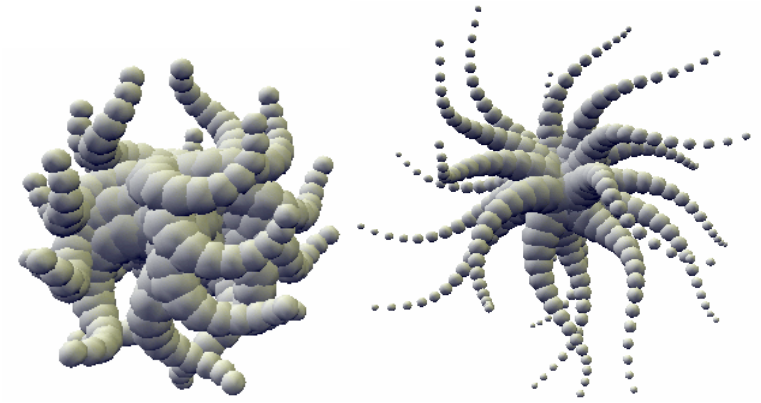


Figure 5. Examples of a branch structure.

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, 360 is the whole sphere
       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:

$$root_{rib}(m) = T_{backbone}(m) * reflection * root$$

where

$$reflection = \begin{vmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \text{ and } \begin{vmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

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

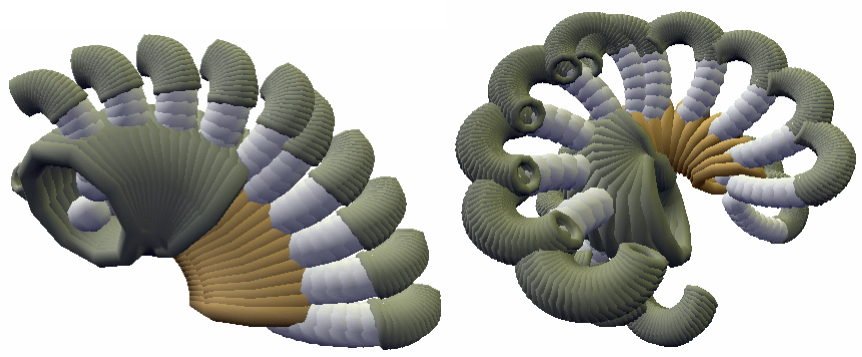


Figure 6. Examples of a ribcage (with backbone).

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.

```
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);
}
```


10 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.

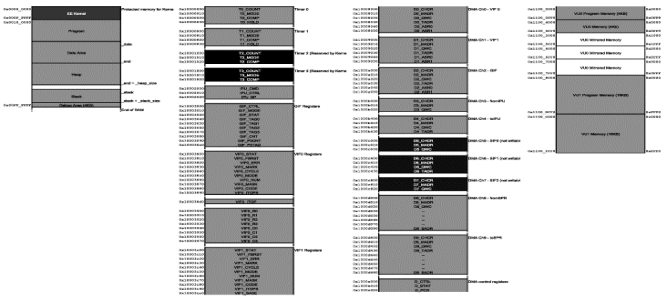
3. All the hardware is memory mapped.

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.

Having said that, here is a thumbnail diagram of the hardware memory map as described in `eeregs.h`:



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 directly 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% 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
```

More on VCL later…

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);
}
```

VCL – 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.

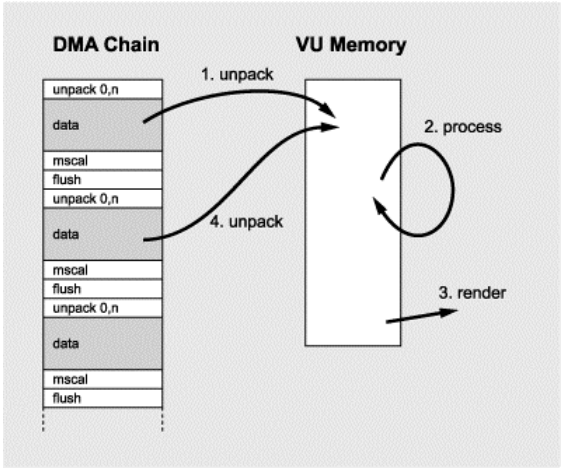
Benefits: Process a large amount of data in one go.

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

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.



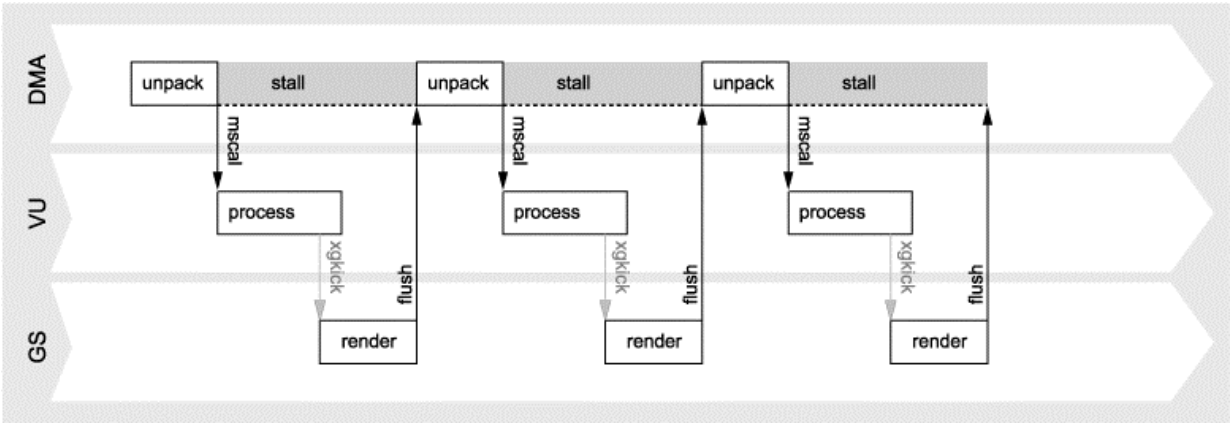


Figure 7: Single Buffer layout and timing

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 is 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.

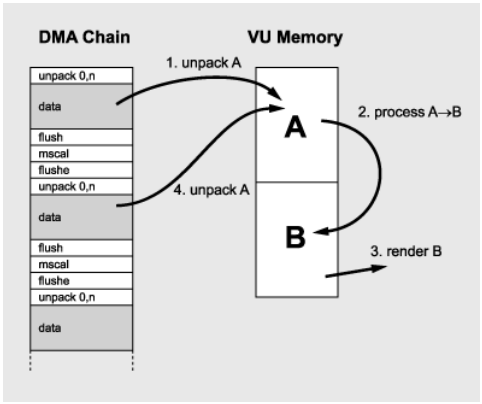
Although more parallel, VU calculation is still serialized.

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.



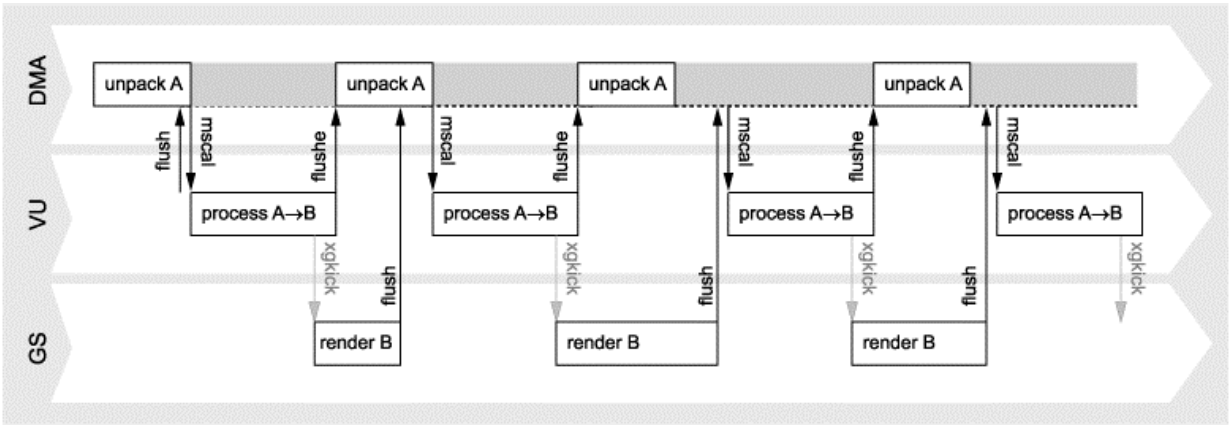


Figure 8: Double Buffer layout and timing

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 bhuffered. 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.

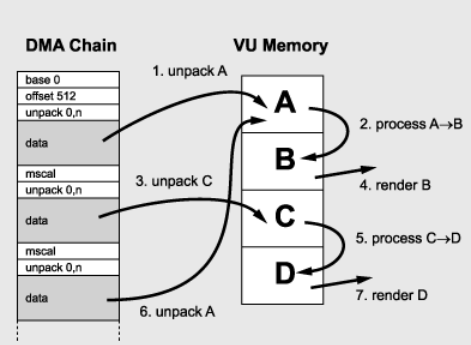
Works well with the double buffering registers TOP and TOPS, which may have caching advantages.

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.

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.

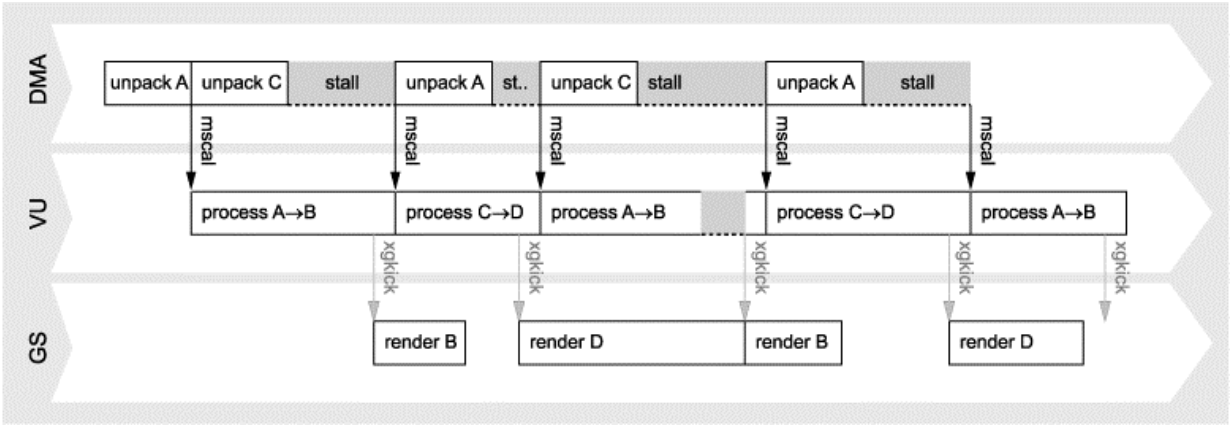


Figure 9: Quad Buffer layout and timing

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.

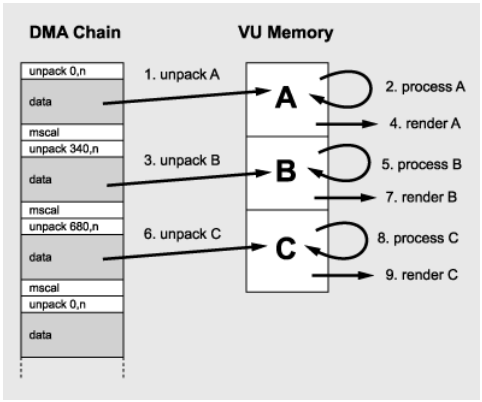
Three streams of read/writes again introduce hidden wait states.

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.



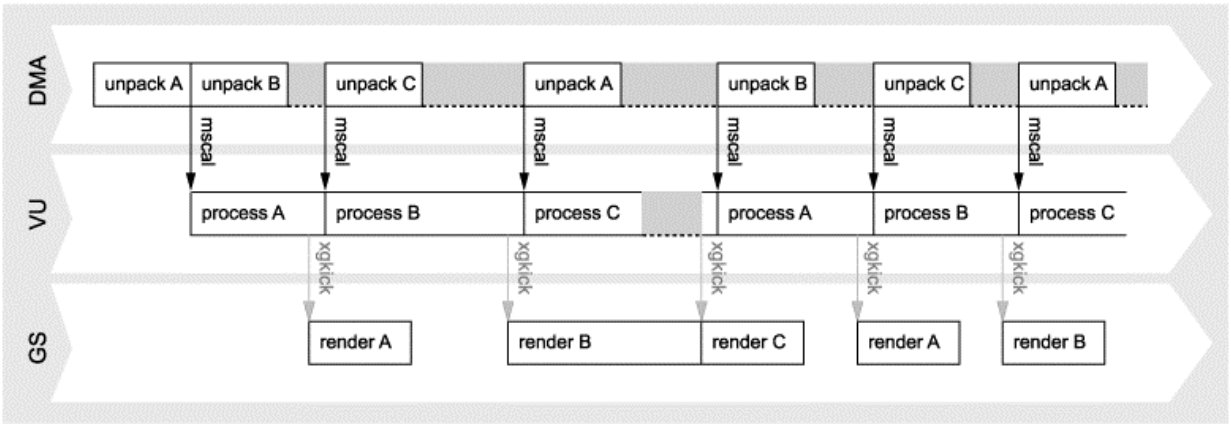


Figure 10: Triple Buffer layout and timing

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.

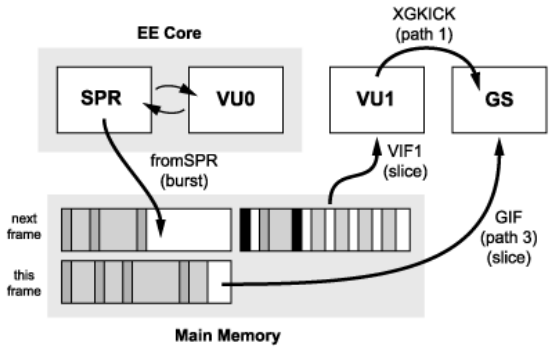
Drawbacks: Moving data from VU0 to Scratchpad efficiently is a complex issue.

Large amounts of main memory are needed as buffers.

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.



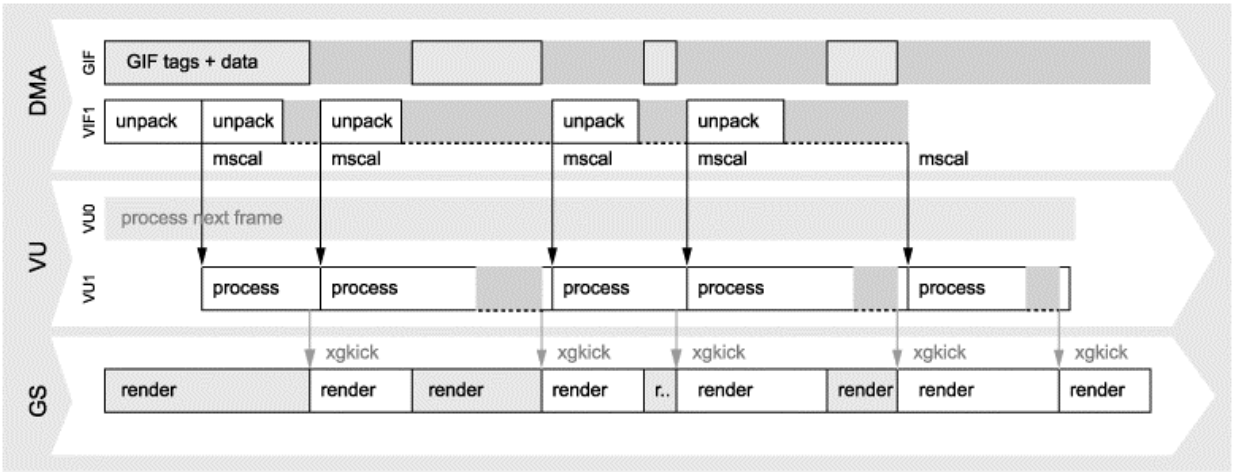
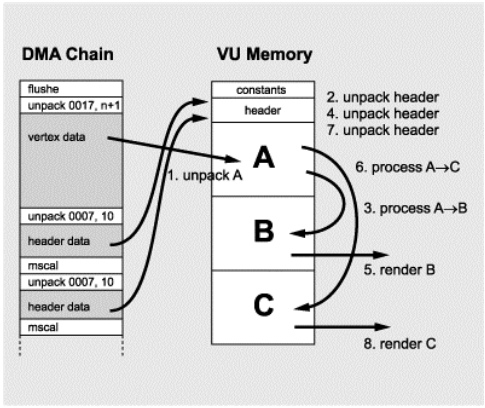


Figure 11: Parallel Processing data flow and timing

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.



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 `xgkick`s 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:

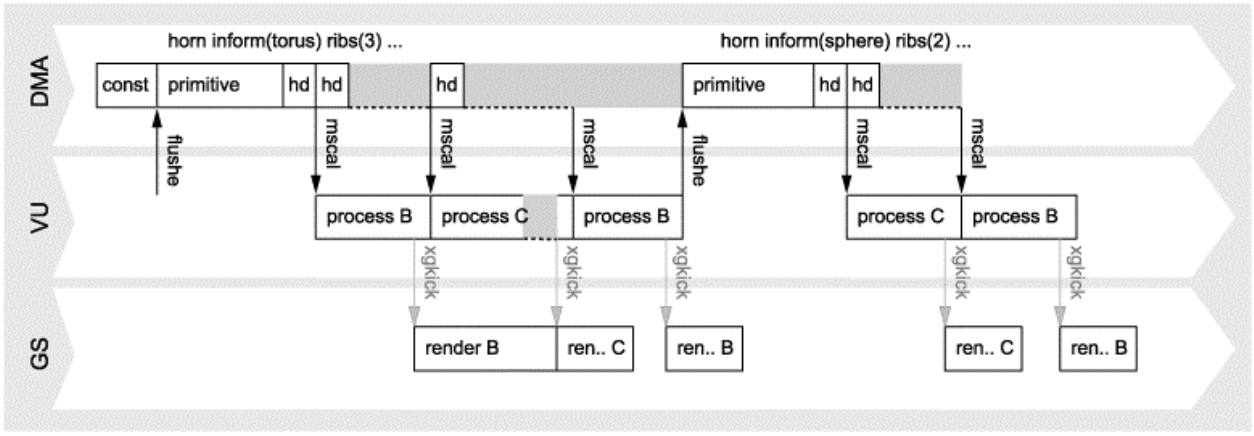


Figure 12: Timing diagram for the Lifeform renderer.

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.

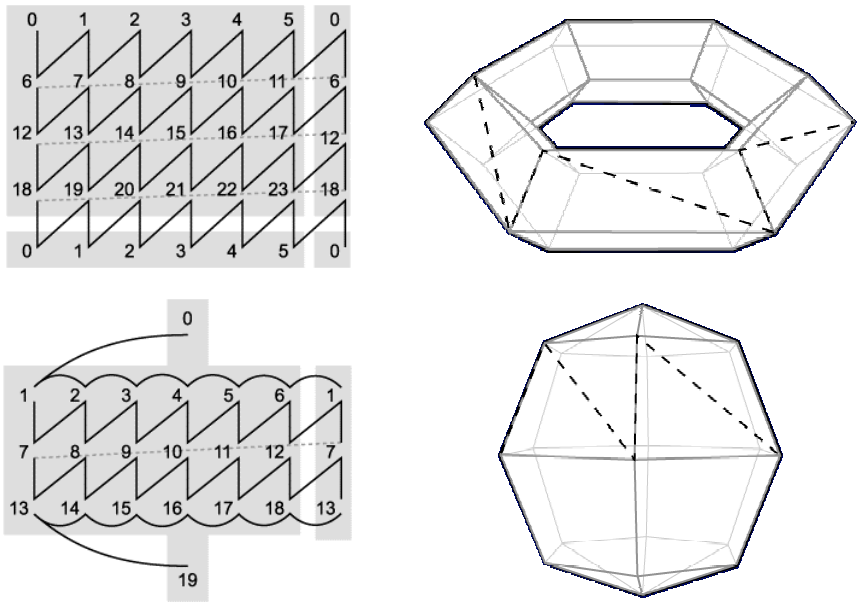


Figure 13: tristrip vertex orders for 6x4 torus and a 6x4 sphere.

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:

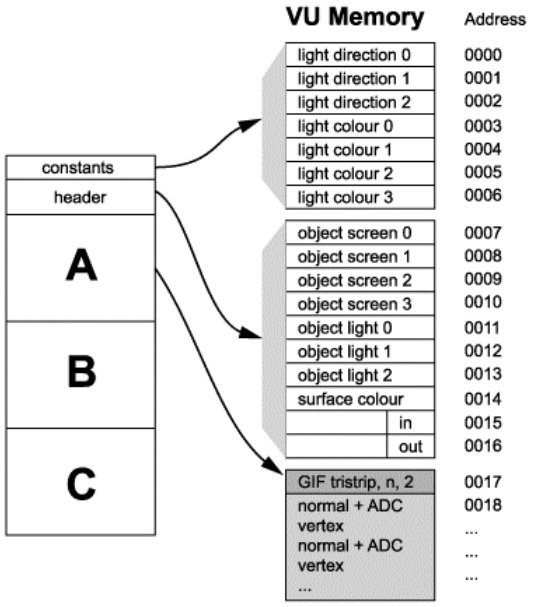
```
vif_packet->Ret();
{
   vif_packet->Pad128();
   vif_packet->Nop();
   vif_packet->Nop();
   vif_packet->Flushe();  // wait for previous program to finish before uploading new vertices.
   vif_packet->OpenUnpack(Vifs::UnpackModes::v4_32, 17, Packet::kSingleBuff);
   {
      // unpack this data to 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:

$$color = K_{surface} * (I_{light} * N.L)$$

$$= K_r * (I_r * N.L)$$
$$K_g * (I_g * N.L)$$
$$K_b * (I_b * N.L)$$

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:

$$color = K_{surface} * Sum(n, I_n * N.L_n)$$

$$= K_r * (I_{0,r}*N.L_0 + I_{1,r}*N.L_1 + I_{2,r}*N.L_2)$$
$$K_g * (I_{0,r}*N.L_0 + I_{1,r}*N.L_1 + I_{2,r}*N.L_2)$$
$$K_b * (I_{0,r}*N.L_0 + I_{1,r}*N.L_1 + I_{2,r}*N.L_2)$$

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

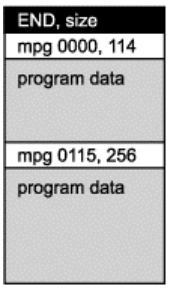
Now we have all the information needed to design the VU program. We know where the matrices are going to be stored, we know where to get our GIF tags from and we know how many verts need to be transformed for each primitive (it's the last 8 bits of the GIF tag). We will need to:

1. Transform the vertex to screen space, divide by W and covert the result to an integer.
2. Transform the normal into light space (a 3x3 matrix calculated by the horn function).
3. Calculate N.L and multiply it by the light colors.
4. Multiply the resulting light intensity by the surface color.
5. 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:

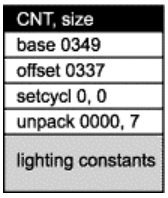
```
CVifSCDmaPacket *packet = new CVifSCDmaPacket(80000,
                                  DMAC::Channels::vif1,
                                  Packet::kDontXferTags,
                                  Core::MemMappings::UncachedAccl );
CVifSCDmaPacket *last_packet = new CVifSCDmaPacket(80000,
   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

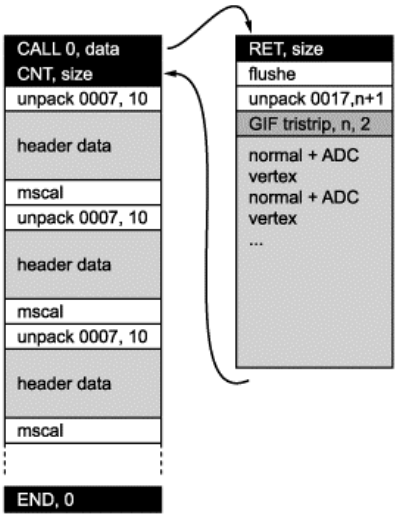
From this point on all data for the next frame gets appended to the end of the current DMA packet called, usefully, `packet`.

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.

Conclusion

In conclusion, we covered:

- Some details of why programming PS2 is different to programming a PC application, coming up with some guidelines for thinking in PS2.
- Introduced a number of new tools for programming PS2.
- Presented a set of basic VU dataflow diagrams with both space and time diagrams to show how different algorithms work with VU programs and to give you a starting point for designing your own VU programs.
- Adapted the triple buffering algorithm for a specific purpose – rendering multiple instances of a single primitive using different transformation matrices for each instance – and shown a significant speed up.

Results