Putting Curved Surfaces to Work on the Nintendo 64

by Mark A. DeLoura

ame console programming is largely a secret art. The technology and APIs are kept hidden by nondisclosure agreements, and you won't find development kits for game consoles at your local software store. As a result, programming for game consoles is something you just don't hear much about.

While specific techniques for programming Nintendo's current game console are well-known within that particular developer community, they are virtually unknown among PC developers, or developers looking to do cross-platform titles. This article will give you some insight into the inner workings of the Nintendo 64 (N64). Much of what I'll discuss in this article hasn't even been released to authorized N64 developers. Nintendo has chosen to

Mark A. DeLoura (markde01@noa.nintendo.com) is the software engineering lead for the Product Support Group at Nintendo. He's been working on the Nintendo 64 since the first hardware dev kits showed up, and he damn near cried the first time he booted up SUPER MARIO 64. Now he's working on high-tech wizardry for Nintendo's next-generation console, Dolphin.

pull back the covers to help developers squeeze the last ounce of performance out of the machine. We hope that this article will help N64 developers do just that, and encourage other developers to explore N64 programming.

After a quick discussion of the N64 architecture, we'll dig down deep and design some custom Reality Signal Processor (RSP) microcode, which tessellates a Bézier surface as shown in Figure 1. The RSP is a very powerful custom chip in the N64, and until now the details of programming this chip have been kept secret. In a sense, we at Nintendo have decided to let the cat out of the bag. You'll get a feel for the incredible power of this chip and see why N64 is capable of great 3D graphics with features that still aren't available in consumer 3D cards.

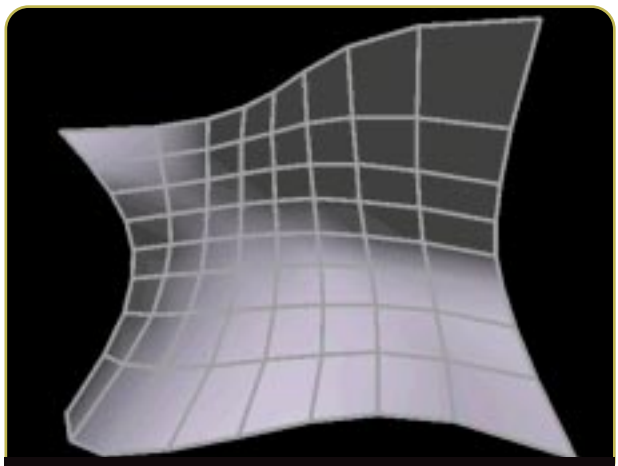


FIGURE 1. *This Bézier surface has been tessellated by the microcode we develop in this article, and rendered by a Nintendo 64.*

Nintendo 64 Architecture

The Nintendo 64 is designed around two main processing components (Figure 2). These two elements are a MIPS R4300i CPU, and the Reality Co-Processor (RCP), which is a custom chip. The simplicity of this architecture makes N64 programming very straightforward. In addition to these processors, the N64 contains 4MB of Rambus DRAM (RDRAM), four controller ports, and a cartridge port. The memory is expandable and a 4MB Expansion Pak is currently available.

The N64's custom RCP runs at 62.5MHz. It is primarily composed of two parts: the Reality Signal Processor (RSP) and the Reality Display Processor (RDP). The RSP processes display lists which are sent from the CPU. It performs all matrix and vertex computations and outputs triangle commands to the RDP. The RDP takes this information, loads the texture cache from RDRAM, and renders fully MIP-mapped, anti-aliased, Z-buffered triangles to the frame buffer. This design leaves the CPU free to perform physics calculations, advanced artificial intelligence, sound processing, and other game functions.

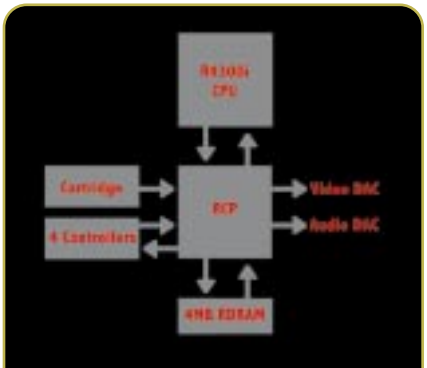


FIGURE 2. *The Nintendo 64 architecture is simple and elegant.*

RSP Architecture

The RSP is modeled on a general-purpose 32-bit RISC processor. It includes 4KB of memory for code (IMEM) and 4KB of memory for data (DMEM). Programs which execute on the RSP are known as microcode. Nintendo provides a standard suite of microcode to all N64 developers, including 3D transformation and lighting code, line-drawing code, sprite routines, and audio processing. Due to special features of the RSP, it is very well-suited for computationally heavy tasks such as 3D graphics calculation and audio mixing.

In addition to 32 32-bit scalar registers, the RSP includes 32 128-bit vector registers. These vector registers can be addressed in a variety of ways, but they are ideally used as eight shorts (also called vector slices). Each slice has a 48-bit accumulator associated with it that can be used to store intermediate results. Using the vector registers and accumulators, a vector operation can be performed which multiplies two vectors and adds the result to the current accumulators, giving 16 calculations in one cycle.

The RSP can actually execute a vector operation and a scalar operation each cycle. This means that it's possible to do 17 calculations per cycle. With carefully tuned microcode, it is possible to reach a maximum of just over one billion operations per second.

The Microcode

This high-speed programmable architecture was very forward-thinking at the time the Nintendo 64 was designed. It has enabled Nintendo to provide a set of standard microcode libraries which make 3D programming easier for the novice. At the same time, elite programmers are able to code up special routines which are optimized for their own games or enable unique functionality. During the life span of the N64, the 3D performance has nearly doubled as a result of microcode optimizations.

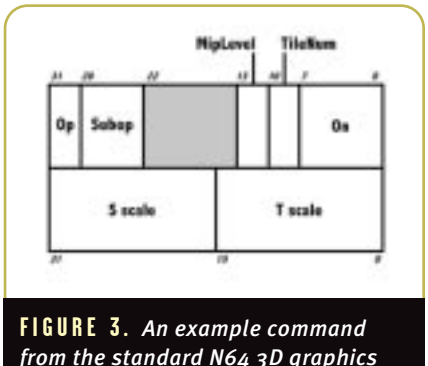


FIGURE 3. *An example command from the standard N64 3D graphics microcode. This command turns on texture mapping using a specific texture tile.*

Microcode Execution

First let's talk a little about the structure of microcode and how to use it. Microcode is exe-

cuted through the use of an RSP task. Tasks are command lists (graphics display lists or audio commands) which indicate a series of operations for the microcode to perform. They are executed in parallel with the CPU. In order to start an RSP task, you create the command list and pass it to the RSP along with pointers to the microcode and various buffers that the microcode needs. Then you call a simple function to start RSP execution and control is immediately returned to the main program while the RSP begins processing commands.

The RSP can communicate with the RDP or CPU during execution if necessary. For example, most versions of the 3D microcode communicate with the RDP, feeding it triangles and other data to render to the frame buffer. Other versions of microcode communicate with the CPU when data is ready. For example, the Z-Sort microcode can be set up to alert the CPU after a number of objects have been processed so that the CPU can work on these objects in parallel. When the RSP completes the task, it signals the CPU so that the user program can send the next RSP task or use this information for synchronization.

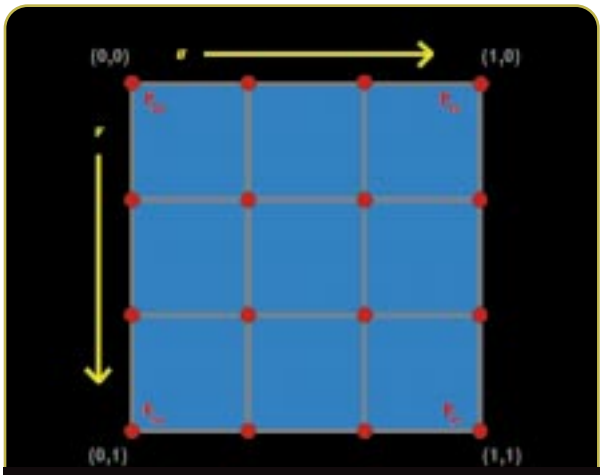


FIGURE 4. *Bézier surfaces are defined in a biparametric space. Sixteen control points are used to define the surface completely.*

The Command Loop

The microcode command loop sequentially goes through commands which have been DMA'd into DMEM from the command list. Similar to assembly language instructions, the commands have bitfields which indicate the RSP function desired. In the microcode command loop, the opcode and subopcode bitfields are masked off and used as an offset into the function jump tables (also stored in DMEM) to determine the IMEM function location.

In the standard graphics microcodes, each command is a 64-bit doubleword. The opcode and subopcode are contained in the upper bits, and lower bits are reserved for data being passed as function parameters as shown in the example in Figure 3. The data bitfields are masked off in the main loop and stored in separate registers before jumping to the function requested.

The DMA Engine

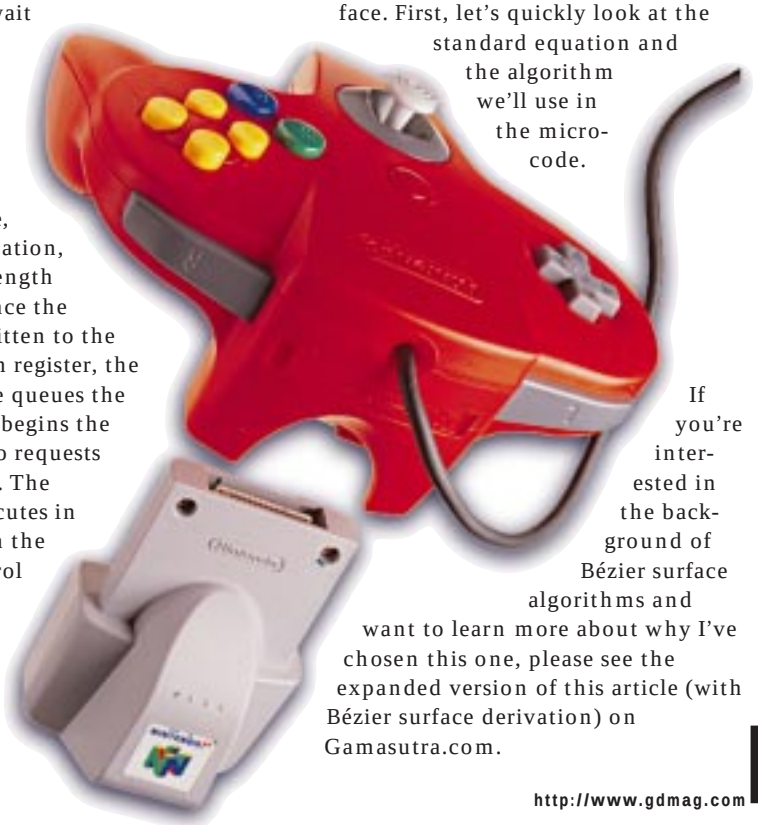
The RSP includes a set of registers which control the DMA engine. Since there may be multiple requests for DMA pending, the microcode must check the DMA Busy register before submitting its request. If a request is being processed and there is already another request pending, the microcode must wait before submitting a request. A request is made by altering the DMA Source, DMA Destination, and DMA Length registers. Once the length is written to the DMA Length register, the DMA engine queues the request and begins the transfer if no requests are pending. The transfer executes in parallel with the RSP so control is immediately returned to the microcode.

Using Curved Surfaces

With this basic understanding of the N64's workings behind us, let's move on to the main focus of this article, using curved surfaces. Curved surfaces are not supported in the standard N64 microcodes. But if you want to render curved surfaces, it makes a lot of sense to do the heavy computations required on the vector processor. Now, we're not actually going to render curved surfaces. We'll take a curved surface representation and tessellate the surface into polygons which the N64 then renders.

For our purposes here, we are going to use Bézier surfaces. A Bézier surface is a curved bicubic surface, similar to Hermite surfaces, B-spline surfaces, and NURBS. The Bézier is mathematically complex enough for us to be able to create interesting surfaces, while not being so difficult to compute that we're only going to be able to do a couple per frame. If you need to brush up on curved surface technology, check out the list of references at the end of this article.

There are a number of algorithms we could use to tessellate a Bézier surface. First, let's quickly look at the standard equation and the algorithm we'll use in the microcode.



If you're interested in the background of Bézier surface algorithms and want to learn more about why I've chosen this one, please see the expanded version of this article (with Bézier surface derivation) on Gamasutra.com.

Bézier Surface Equation

A Bézier surface is a parametric surface (u,v = [0,1], [0,1]) defined by its 16 control points p_{ij} which form a 4×4 grid, as shown in Figure 4. The common form for representing this surface is:

$$Q(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} p_{ij} B_i(u) B_j(v)$$

The functions $B_i(u)$ and $B_j(v)$ are the Bernstein polynomials which are also used for Bézier curves.

The edges of a Bézier surface are each Bézier curves. Since only the end control points of Bézier curves lie on the curve, we can extrapolate that the corner points of the surface are the only control points which lie on the surface. All twelve of the other control points influence the shape of the surface, but are not on the surface itself. For this article, we'll create a microcode that tessellates a Bézier surface into an 8×8 grid of quadrilaterals.

Tessellation by Evaluation

The most direct way to slice a Bézier surface into polygons is by calculating the above $Q(u,v)$ double summation on a regular grid. Performing this in a very optimized way, each surface vertex we calculate requires 54 additions and 108 multiplies. That's a lot of work to do when we're planning to create a 9×9 grid of vertices.

Central Differencing

The way we're going to generate points on the Bézier surface in this article is through the use of central differencing. Central differencing gives us an easy way to find the midpoint of a Bézier curve without having to keep track of control points for each subcurve. We can split the edge curves at their midpoints, and then split the surface across these midpoints to create four subsurfaces. This process can be repeated recursively to create an arbitrarily fine mesh. (For details on this algorithm please see the previously-mentioned article on Gamasutra.com, or Brian Sharp's series of articles on curved surfaces, June–July 1999.)

The central differencing algorithm has a hefty initialization cost due to the computation of second partial derivatives (Quu, Qvv, $Quuvv$) at each corner control point. But every curve subdivision after that will only cost us 18 additions and 18 multiplies. The memory footprint is 24 bytes per subdivision, and there are 77 subdivisions necessary to create our mesh. This will fit in our 4KB DMEM nicely.

Writing the Tessellation Microcode

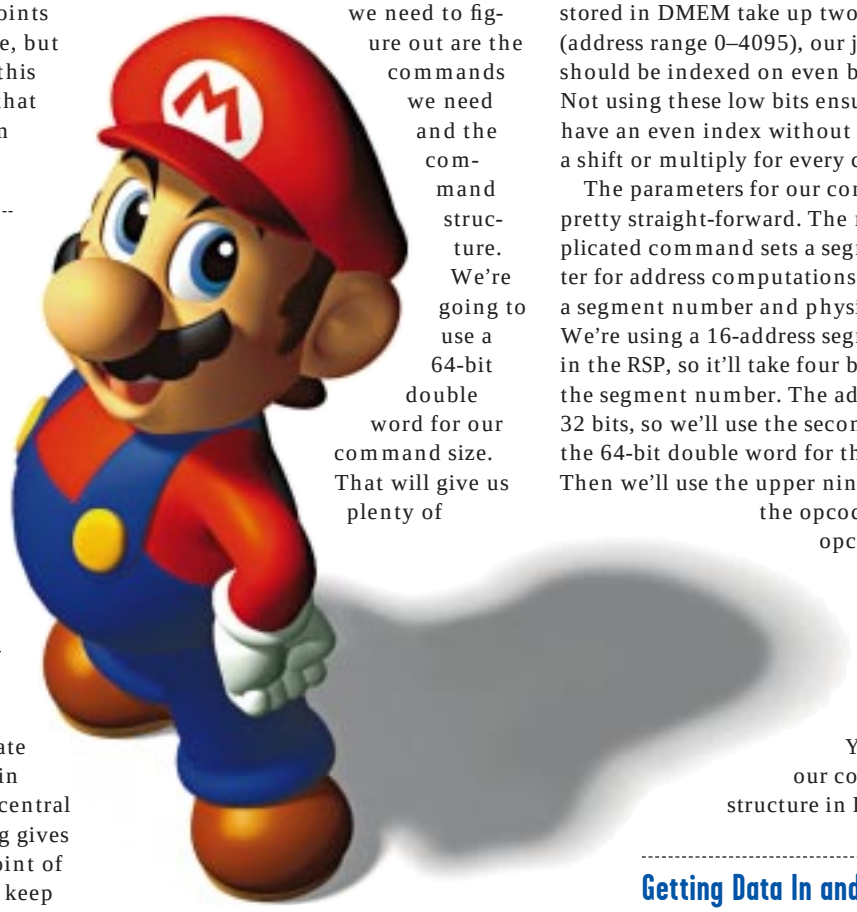
Now that we've chosen the algorithm to tessellate the surface, let's get back to work on the microcode itself. The first things we need to figure out are the commands we need and the command structure. We're going to use a 64-bit double word for our command size. That will give us plenty of room to store the data for each command inside the instruction. The commands and parameters necessary for our tessellation microcode are:

1. Set RSP segment (segment number, physical address).
2. Load control points (segment address).
3. Perform tessellation.
4. Save surface vertices (segment address).
5. End display list.

I'll describe these commands further in a moment.

Since we only have five commands, we can just use a 3-bit field for the opcode. Fortunately, the standard graphics microcodes all use a 3-bit opcode field and 6-bit subopcode field, so we'll use that. But we'll just wedge all our instructions into the subopcodes for one primary opcode. Then we can reuse a lot of the main command loop routines from the standard microcodes, including the display list DMA routine that loads commands into the DMEM buffer for us. The low bit of the opcode field and subopcode field are not used. Since microcode function addresses stored in DMEM take up two bytes (address range 0–4095), our jump table should be indexed on even bytes only. Not using these low bits ensures that we have an even index without performing a shift or multiply for every command.

The parameters for our commands are pretty straight-forward. The most complicated command sets a segment register for address computations. It requires a segment number and physical address. We're using a 16-address segment table in the RSP, so it'll take four bits to hold the segment number. The addresses are 32 bits, so we'll use the second half of the 64-bit double word for the address. Then we'll use the upper nine bits for the opcode and subopcode fields and follow it with four bits for the segment address. You can see our command structure in Figure 5.



Getting Data In and Out

Before we code up the tessellation algorithm, let's figure out how to get data in and out of DMEM. The "set RSP segment" command fills an entry of our 16-entry segment/offset table, which is stored in DMEM. This table makes some programming tasks easier, such as swapping the frame buffer each frame. The segment table stores 24-bit

FIGURE 5. *Our command structure is similar to the structure of standard N64 microcode commands. We use this structure for all of our commands.*

offsets which are added to any address sent to the RSP. The segment table index is stored in bits 24–27 of the addresses passed in. The low 24 bits of the segment address are added to the 24 bits stored in the segment table. Since our physical address range is 0–0x007fffff (8MB), 24 bits is enough.

Prior to tessellation we need to load the control points into DMEM. Our "load control points" command simply takes an address as a parameter. The address is passed to the segment address translation routine, which uses the segment table to convert the address to a physical address. The DMA engine is called to bring the 16 control points into DMEM from this physical address.

After tessellation, we need to save the surface vertices we've computed, using the "save surface vertices" command. We'll pass in an address and the segment address translation routine will convert it to a physical address. That physical address is used to program the DMA engine to copy our 81 surface vertices to RDRAM.

The "end display list" command simply flags the RSP to quit. It executes a break, which signals the CPU, and alerts our main program.

Data Formats

The first thing our "perform tessellation" command does is perform a simple translation from control point format to surface vertex format. So let's talk about these formats.

The Nintendo 64's standard vertex format uses 16-bit coordinate ranges, which are s15 quantities (one sign bit and 15 integer bits). This gives vertex coordinates an effective range of +/– 32KB. It makes sense for us to use this same format for control points, but since we're just tessellating, we really only need the point position. Rather than wasting the extra space for colors and texture coordinates when we DMA the control points into DMEM, our format will only represent the *x*, *y*, and *z* position as signed shorts.

The surface vertex format is more complicated. The central difference algorithm describes four sets of values that each vertex needs to track. These are:

1. $Q(u,v)$: Position
2. $Q_{uu}(u,v)$: Second partial derivative in *u* at this vertex.
3. $Q_{vv}(u,v)$: Second partial derivative in *v* at this vertex.
4. $Q_{uuvv}(u,v)$: Second partial derivative in *u* of the second partial derivative in *v* at this vertex.

All of these values are vectors of *x*, *y*, and *z*. Since the vector slice size of the RSP is 16 bits, and the control point coordinates are 16 bits, we're going to stick with 16 bits for these coordinate values as well. We'll have to tweak our math to minimize overflow and underflow, but it will pay off in performance.

One final note on formats. Each vector register contains eight vector slices. But each of our points contains three values (*x*, *y*, and *z*). We're really just going to make things confusing if we try to stuff two of three coordinates from one vertex into a vector register, along with two other vertices. So let's insert a junk (we'll call it *j*) field at the end of each of these vertices. This will also give us much better alignment in DMEM.

Now that we have our formats defined as in Listing 1, it's a simple task to convert from one to the other. Actually, all we need to do is copy the 64 bits from each corner control point (*x*, *y*, *z*, and *j*) into the beginning of each corner surface vertex.

Corner Initialization

Now we need to compute the second partial derivatives described above for each corner of the surface. Fortunately, the second partial in *u* and the second partial in *v* at each

LISTING 1. *Formats for data storage in DMEM. The* j *fields are unused, we include them for data alignment.*

```
struct ControlPoint {
        s15     x, y, z, j;
};

struct SurfaceVertex {
        s15     qx, qy, qz, qj;
        s15     quux, quuy, quuz, quuj;
        s15     qvvx, qvvy, qvvz, qvvj;
        s15     quuvvx, quuvvy, quuvvz, quuvvj;
};
```

LISTING 2. *Pseudocode for computing* $Q_{uu}(o,o)$ *and* $Q_{vv}(o,o)$ *using vector processing.*

```
# Load the vector registers with point data
vload     vectora, P[0,0], P[0,0]       # = x y z j, x y z j
vload     vectorb, P[1,0], P[0,1]
vload     vectorc, P[2,0], P[0,2]

# Do vector computations to simultaneously compute quu and qvv.
vadd              vectord, vectora, vectorc     # D = A+C
vmul              vectore, vectorb, vconst[5]   # E = B*(-2)
vadd              vinter, vectord, vectore      # inter = A-2B+C
vmul              v00, vinter, vconst[3]        # v00 = 6*(A-2B+C)
vstore2           v00, v00uu, v00vv             # Store results to uu and vv fields
```

corner control point can be computed with similar equations that use different points. Here are the equations to perform at control point (0, 0):

$$Q_{uu}(0,0) = 6\,(P_{00} - 2P_{10} + P_{20})$$

$$Q_{vv}(0,0) = 6\,(P_{00} - 2P_{01} + P_{02})$$

We need to do this computation in *x*, *y*, and *z* for each equation. This is a great place to take advantage of vector processing. We'll do this operation in parallel, computing both equations for *x*, *y*, and *z* simultaneously. First, we load both sets of control point positions into the vectors, as shown in pseudocode in Listing 2. Then just a few vector computations are performed and all coordinates are simultaneously calculated.

Note that we have the constants –2 and 6 stored in a vector constants (`vconst`) register, which makes it easy to multiply each slice in another vector by each scalar. Using vector processing we've reduced two additions and two multiplies for each of six coordinates to just two additions and two multiplies total.

We can perform this same process to compute Q_{uuvv}. But we'll have to pair up the operations. We have four control points, the corner points, which we need in order to calculate Q_{uuvv}. We can compute two separate control points simultaneously by jamming them into the same vector and doing vector operations. So we'll perform this process twice in order to compute Q_{uuvv} for all four points.

Surface Subdivision

For code simplicity, we're going to subdivide the surface iteratively, not recursively. We're going to subdivide many times, so let's make a function out of it. What do we need to pass to this function? Well, we'll need the data for the endpoints of the curve we're splitting, and a *du* value which is the distance in parametric space from the midpoint to the endpoint. This is 0.5 for our first subdivision. For simplicity, we'll pass in the value $(du)^2/2$ so we don't have to compute the square and multiply by one-half each time we use the function. We'll also stuff this value in a vector slice so that we can use it in vector computations. We'll call the vector which contains this value `vecdusqhalf`. Our function ends up looking like this (there will be one for u-curve splits, and one for v-curves):

```
void tesselSubdivide[U|V](Vertex v0, Vertex v1, Vector vecdusqhalf)
```

The microcode for the `tesselSubdivideU` function appears in Listing 3. The function `tesselSubdivideV` will be very similar.

The first thing you'll notice in this code is that we block together the Q_{uu} and Q_{uuvv} computations. We also block together the Q and Q_{vv} computations. That's because both of these are very similar computations. For Q_{uu} and Q_{uuvv} we're doing this:

$$Q_{uu}(u_{mid}) = \frac{Q_{uu}(u_0) + Q_{uu}(u_1)}{2}$$

$$Q_{uuvv}(u_{mid}) = \frac{Q_{uuvv}(u_0) + Q_{uuvv}(u_1)}{2}$$

The computations for Q and Q_{vv} depend on the prior computations, so we do them second. They look like this:

$$Q(u_{mid}) = \frac{Q(u_0) + Q(u_1)}{2} - \left(dusqhalf * Q_{uu}(u_{mid})\right)$$

$$Q_{vv}(u_{mid}) = \frac{Q_{vv}(u_0) + Q_{vv}(u_1)}{2} - \left(dusqhalf * Q_{uuvv}(u_{mid})\right)$$

Most of the opcodes you see in the microcode listing make intuitive sense. But one that bears explaining is `vmudm`. The RSP provides many multiplication operations. They vary depending on the sign of the operands and whether the operands are fractions or integers. The `vmudm` operation performs multiplication of signed integers by unsigned fractions. The resulting integer part of each vector slice computation is stored in the destination vector register (first operand), and the 32-bit integer/fraction results are stored in the accumulator slices.

This microcode currently is not optimized for dual processing, nor for accumulator storage of intermediate results. Vector loads and stores are scalar operations, so we could easily tighten this microcode up by executing loads and stores in parallel with vector operations.

Performance Figures

Calculating a regular grid of points on the Bézier surface using the standard double sum equation is a not a very efficient way to tessellate. Using floating-point arithmetic on the CPU, this process took 272,500 CPU cycles. While central differencing has a large performance cost at initialization, the subdivision step is very fast. Implemented on the CPU, it takes 70,400 CPU cycles to tessellate our surface.

When we moved this algorithm to the RSP, we made some sacrifices. We used 16-bit fixed-point arithmetic and took a hit for DMA-ing data to and from the RSP. But our algorithm, including RSP load and save time, runs in just 16,600 CPU cycles. And the CPU itself is free during this process to do other computations.

N64 Optimizations Keep the Platform Fresh

We've examined how Bézier surfaces can be implemented on the N64 using a central difference tessellation algorithm in microcode. The payback we got for choosing a more efficient surface tessellation algorithm and pushing it onto the RSP was substantial.

Technically, the N64 is still a powerhouse. Programming the microcode and taking advantage of vector processing gives developers the ability to implement algorithms that aren't feasible on much bigger and faster CPUs. In addition, learning to program the N64 now will give developers a big advantage when it comes to next-generation console development (including Nintendo's upcoming system, currently known as Dolphin), many of which use vector processing. If you're interested in becoming an N64 developer, or if you are an N64 developer and you want to know about microcode development kits, please contact Nintendo by sending e-mail to support@noa.com. ■

FOR FURTHER INFO

Books

Watt, Alan, and Watt, Mark. *Advanced Animation and Rendering Techniques: Theory and Practice*. New York: ACM Press, 1992.

Periodicals

Clark, J. H. "A Fast Scan-Line Algorithm for Rendering Parametric Surfaces." *Computer Graphics* Vol. 13 No. 2: pp. 289–299.

Game Developer

Sharp, Brian. "Implementing Curved Surface Geometry" (June 1999) and "Optimizing Curved Surface Geometry" (July 1999).

Gamasutra

For a detailed derivation and comparison of Bézier surface algorithms, see the expanded version of this article at http://www.gamasutra.com.

Bézier Surface Microcode Source

If you're a Nintendo 64 developer, log on to Nintendo's developer web site at https://www.warioworld.com.

LISTING 3. *Microcode for the* tesselSubdivideU *routine.*

```
##########################################################
# tesselSubdivideU
#
# Subdivide this curve in the U direction
#

# Surface Vertex structure offsets
.symbol   VERTEX_POS,                0
.symbol   VERTEX_UU,         8
.symbol   VERTEX_VV,         16
.symbol   VERTEX_UUVV,       24

# Register aliases
.name     pnew,              $10     # Position of output surface vertex (u)
.name     pminus,            $11     # Position of input left surface vertex (u-du)
.name     pplus,             $12     # Position of input right surface vertex (u+du)
.name     vecdusqhalf,       $v6     # Vector which contains 0.5*du*du in slice 0
.name     vecminus,          $v7     # Vector for storing left surface vertex info
.name     vecplus,           $v8     # Vector for storing right surface vertex info
.name     vecuus,            $v9     # Temp vector for UU and UUVV computation
.name     vecuushalf,        $v10    # Final results of UU and UUVV computation
.name     vecposvvsinter,    $v11    # Temp vector for POS and VV computation
.name     vecposvvshalf,     $v12    # Temp vector for POS and VV computation
.name     vecmulleduus,      $v13    # Temp vector for POS and VV computation
.name     vecposvvs,         $v14    # Final results of POS and VV computation

          .ent      tesselSubdivideU
tesselSubdivideU:

          # Do quu and quuvv computations together
          ldv       vecminus[0], VERTEX_UU(pminus)
          ldv       vecminus[8], VERTEX_UUVV(pminus)
          ldv       vecplus[0],  VERTEX_UU(pplus)
          ldv       vecplus[8],  VERTEX_UUVV(pplus)
          vadd      vecuus, vecminus, vecplus                    # Add endpoints
          vmudm     vecuushalf, vecuus, vecconst[1]              # Mul by one-half

          # Do qpos and qvv computations together
          ldv       vecminus[0], VERTEX_POS(pminus)
          ldv       vecminus[8], VERTEX_VV(pminus)
          ldv       vecplus[0],  VERTEX_POS(pplus)
          ldv       vecplus[8],  VERTEX_VV(pplus)
          vadd      vecposvvsinter, vecminus, vecplus            # Add endpoints
          vmudm     vecposvvshalf, vecposvvsinter, vecconst[1]   # Mul by one-half
          vmudm     vecmulleduus, vecuushalf, vecdusqhalf[0]     # uus/2 *(du^2)/2
          vsub      vecposvvs, vecposvvshalf, vecmulleduus       # Subtract...

          # Store everything
          sdv       vecuushalf[0], VERTEX_UU(pnew)
          sdv       vecuushalf[8], VERTEX_UUVV(pnew)
          sdv       vecposvvs[0],  VERTEX_POS(pnew)
          jr        return
          sdv       vecposvvs[8],  VERTEX_VV(pnew)
          .end      tesselSubdivideU
```