Using GL_NV_vertex_array_range and GL_NV_fence on GeForce Products and Beyond

John Spitzer
john.spitzer@nvidia.com
NVIDIA Corporation

Cass Everitt
cass@nvidia.com
NVIDIA Corporation

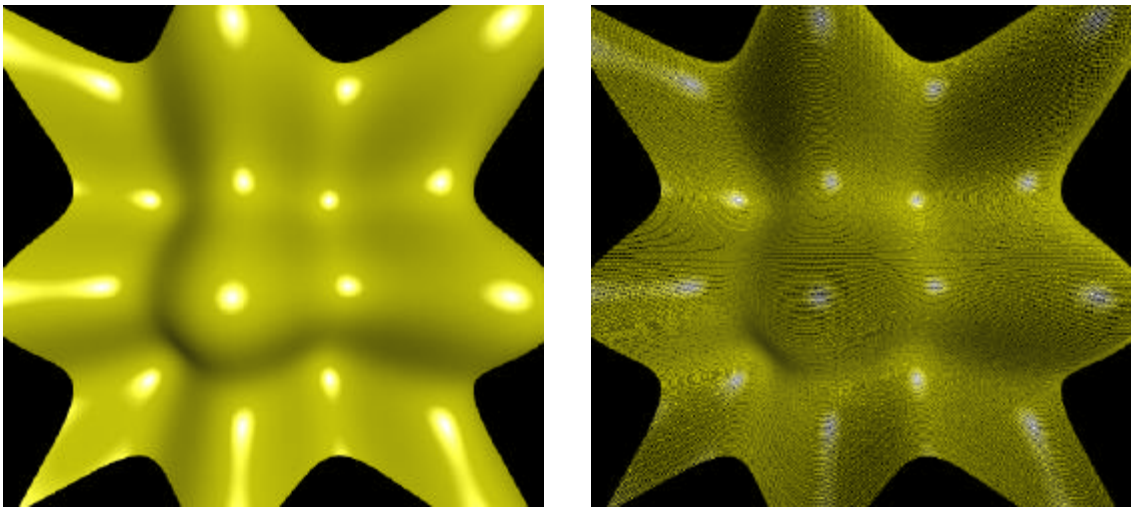

Figure 1. This dynamic, undulating mesh of 180,000 triangles (shown in wireframe on the right) renders at approx. 39fps (7 Mtri/sec) using NV_vertex_array_range and NV_fence. Without NV_vertex_array_range, the performance drops to approx 25 fps (4.3 Mtri/sec). This data was collected from a Pentium III 700 MHz with a GeForce2 GTS and AGP2X.

Overview

The OpenGL extension, GL_NV_vertex_array_range, offers the absolute highest performance available for transferring geometry from the application to the GPU. By allowing the application to allocate and access memory that's usually only accessible by the driver, a developer can minimize data copying and thus maximize performance. The memories accessible through this extension are in limited supply, and whole frames will often not fit completely. The GL_NV_fence extension provides for a very fine-grained synchronization to enable the best possible pipelining of CPU and GPU processing. These extensions are both available in the Detonator2 5.32 (and subsequent) drivers.

Figure 1 shows an example program that uses these extensions to achieve very high polygon throughput for *dynamic* geometry. It may be difficult to tell, but the image on the right is actually a wireframe view of the image on the left.

General Process

1. Allocate memory
2. Allocate fences*
3. Enable memory
4. Define vertex arrays
5. Finish fence(s)*
6. Copy data into arrays
7. Draw indexed primitives
8. Set fence(s)*
9. Repeat steps 4, 5, 6, 7 and 8 as necessary

Allocate Memory

This extension allows one to allocate either video memory (that which is resident on the card) or AGP memory (system memory that is easily accessed by the bus/card). Video memory, though fast, is a precious resource on any system, and should be used sparingly, and only for static geometry. AGP memory is typically much more plentiful (often many megabytes of memory can be allocated), and it may offer similar if not identical performance to video memory (depending upon the AGP bus speed).

To allocate AGP memory or video memory, one **must** supply the wglAllocateMemoryNV call with the appropriate arguments:

Memory Allocated	ReadFrequency	WriteFrequency	Priority
AGP Memory	**[0, .25)**	**[0, .25)**	**(.25, .75]**
Video Memory	**[0, .25)**	**[0, .25)**	**(.75, 1]**

All arguments outside these ranges will yield poor results.

For best results, only allocate memory once. If multiple buffers are needed, sum the sizes and allocate one large buffer, then partition it after the allocation. Switching between multiple (separately allocated) buffers is expensive and unnecessary. There is no proxy mechanism to determine the maximize size buffer that can be allocated. If an allocation request cannot be satisfied, wglAllocateMemoryNV will return NULL.

* Fences are only required in a memory-limited situation, and can be ignored if all arrays for a frame will fit into the buffer allocated.

Allocate Fences *[optional]*

Fences are allocated using the same semantics as those of texture objects. A typical allocation is as follows:

```
GLuint fence;
glGenFencesNV(1, &fence);
```

Enable Memory

Call glVertexArrayRangeNV on the entire allocated memory buffer. Calling it upon a subset of the memory buffer will result in an invalid range, and will not work. Once the range has been defined, enable the extension by calling:

```
glEnableClientState(GL_VERTEX_ARRAY_RANGE_NV);
```

Define Vertex Arrays

At this point, one must define the vertex array pointers, just like normal. Use one of the glInterleavedArrays formats, or use glVertexPointer, glNormalPointer, etc. Performance between these two methods will be similar, if not identical. The following restrictions apply to all defined arrays:

- strides for defined arrays must be less than 256
- strides must be multiples of 4
- pointers must be 4-byte aligned
- must be enabled
- must exist entirely within enabled buffer

Furthermore, each defined array must fall into one of the following formats:

Array	**Size**	**Type**	**Stride**	**Pointer Alignment**
Color	3	GL_FLOAT		
Color	4	GL_FLOAT		
Color	3	GL_UNSIGNED_BYTE	≠ 0	
Color	4	GL_UNSIGNED_BYTE		
Normal	-	GL_FLOAT		
Normal	-	GL_SHORT	Multiple of 8, ≠ 0	8-byte
TexCoord	1	GL_SHORT	≠ 0	
TexCoord	2	GL_SHORT		
TexCoord	3	GL_SHORT	Multiple of 8, ≠ 0	8-byte
TexCoord	4	GL_SHORT		8-byte
TexCoord	1,2,3,4	GL_FLOAT		
Vertex	2	GL_SHORT		
Vertex	3	GL_SHORT	Multiple of 8, ≠ 0	8-byte
Vertex	4	GL_SHORT		8-byte
Vertex	2,3,4	GL_FLOAT		
VertexWeight	1	GL_FLOAT		

The Color formats apply to both glColorPointer and glSecondaryColorPointerEXT. Note that glEdgeFlagPointer, glFogCoordPointerEXT and glIndexPointer are not supported under this extension, and their use will disable it.

Finish Fence(s) *[optional]*

In the memory-limited condition, it will be necessary to recycle AGP buffer space. Ideally, we only want to make sure that the GPU has finished up to some point in the OpenGL command stream before the CPU begins filling that memory. The GL_NV_fence extension provides just this sort of fine-grained synchronization. Before reusing buffer space, the fence that was set following its last use must be *finished*:

```
glFinishFenceNV(fence);
```

Copy Data into Arrays

When writing to video memory, only systems that support Fast Writes will get good write performance. When using AGP memory, it is absolutely essential that data be written to the buffer sequentially to maximize memory bandwidth. This is because AGP memory is uncached, and writing sequentially will take full advantage of the write combiners. Additional performance can be obtained by creating two or more partitions within the buffer, then alternately writing between them. This allows the CPU to fill one partition while the other is simultaneously drained by the GPU. If using a single partition (or a small number), ensure that any previous rendering calls are finished with the partition before refilling it. This can be accomplished by using `GL_NV_fence` as described in this paper. The synchronization provided by the GL_NV_fence extension is significantly finer-grained than that of the `glFlushVertexArrayRangeNV()` function, and should be preferred.

Draw Indexed Primitives

For drawing primitives, it's best, though not essential, to use indexed formats (i.e. glDrawElements, glDrawRangeElements) over non-indexed ones (i.e. glDrawArrays), and to avoid glArrayElement altogether (and do not mix glArrayElement with immediate mode calls – this will certainly yield poor results). Indexed elements not only have the advantage of minimizing bandwidth (only a single index needs to be transferred, not all of a vertex's data), but the driver can also quickly and easily detect shared vertices. Upon encountering a vertex the first time, it will need to be transformed and potentially lighted. The second time it's encountered, the index may show up in the vertex cache as being already processed. GeForce products have a vertex cache of 16 entries, although it's effectively only 10 entries due to pipelining.

Cache usage can be maximized by spatially sorting geometry. Long strips are fairly efficient even without the vertex cache, so great pains need not be taken to insure good performance. However, when triangle/quad lists are processed, it is definitely best to render adjacent primitives at the same time, if possible.

GeForce products require indices to not exceed 65535. If more indices are necessary, break the object into smaller parts.

Set Fence(s) *[optional]*

As soon as a buffer will no longer be referenced by subsequent rendering calls, set a fence. When memory must be reclaimed, these fences can be *finish*ed in the order they were set. The following command is used to set a fence:

```
glSetFenceNV(fence, GL_ALL_COMPLETED_NV);
```

Important Caveats for `NV_vertex_array_range`

When using vertex array range, avoid two-sided lighting, clip planes, logic op or bordered textures, because (depending on the specific hardware) these may disable the extension. The vertex arrays will still reside in uncached memory, yielding much lower performance than had they resided in regular system memory. General avoidance of these modes is advised, since they all trigger some amount of software acceleration.

Because the memory allocated by the vertex array range extension is uncached, reading from the buffer will be very, very slow. If an application needs to read the vertex array data (e.g. for collision detection, deformation or animation), it should maintain a second copy of the data in cached system memory. Once the modification is completed, the data should be copied to the vertex array range buffer in large, contiguous blocks. Failure to do this will result in extremely poor performance. Finally, it's best to tightly pack one's data in the vertex array range buffer. In other words, don't include application-specific data - only the vertices, normals, colors, texture coordinates and/or weights that are needed by OpenGL. This not only optimizes memory usage, but also minimizes cache misses when the data is fetched by the GPU.