

Fast Geometry Processing with and



Drew Card
DCard@ati.com

Jason L. Mitchell
JasonM@ati.com

Introduction

One of the major OpenGL application performance bottlenecks in systems with transform-capable graphics chips is the transfer of vertex and index data to the graphics processor. Traditional OpenGL vertex arrays exist in client memory, requiring them to be copied into server-side memory every time they are used. The `ATI_vertex_array_object` and `ATI_element_array` extensions give the OpenGL application the ability to allocate space for vertex and index arrays in persistent (hardware accessible) server-side memory, thus drastically reducing the overhead of data transfer to the graphics processor.

Object Buffers and Array Objects

The `ATI_vertex_array_object` and `ATI_element_array` extensions allow the application to allocate **object buffers** and **array objects**. An object buffer is merely a buffer of persistent server-side memory. At some point later, it will contain array objects. This is why it is called an object buffer—it is a buffer of objects.

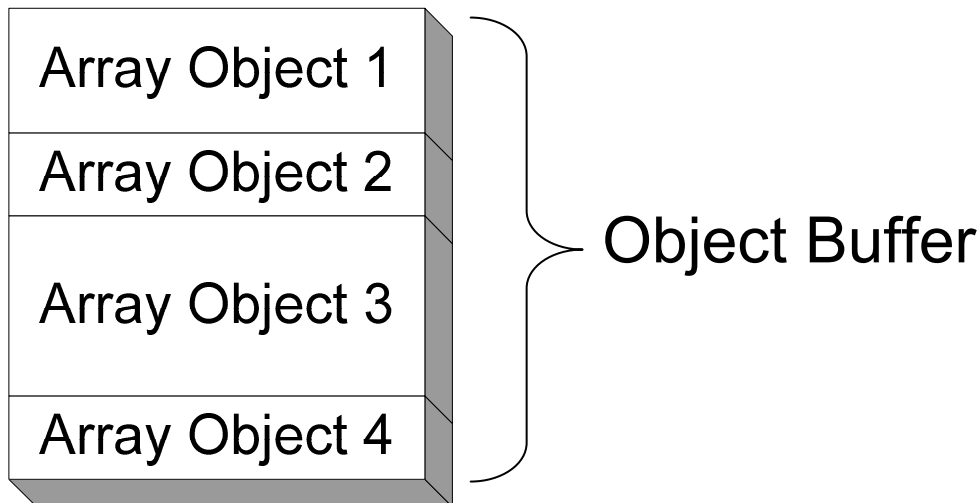


Figure 1 – Array Objects in an Object Buffer

The array objects may be of various types such as `GL_NORMAL_ARRAY`, `GL_VERTEX_ARRAY` or any of the other types supported by `glEnableClientState()`. If `ATI_element_array` is also supported,

array objects may also be of type `GL_ELEMENT_ARRAY_ATI`. Array objects of type `GL_ELEMENT_ARRAY_ATI` contain indices which define primitives drawn through `glDrawElements()`.

Allocating Object Buffers

Object buffers are allocated with the `glNewObjectBufferATI()` entrypoint, which returns the name of the object buffer created. This name is used on subsequent calls to the `glArrayObjectATI()` entrypoint which defines arrays objects inside the object buffer. If `glNewObjectBufferATI()` returns 0, it means that the object buffer could not be allocated.

When allocating an object buffer with `glNewObjectBufferATI()` the application has the option of specifying a pointer to data in client-side memory to be copied into the object buffer. It is important to note that the client-side pointer must point to a contiguous chunk of memory that can fill the entire object buffer. Passing `NULL` as the *pointer* parameter to `glNewObjectBufferATI()` will create an empty object buffer that can be filled by subsequent calls to `glUpdateObjectBufferATI()`. If the data to be inserted into an object buffer is fragmented in client-side memory, it is recommended that the application pass a `NULL` pointer to `glNewObjectBufferATI()` and use subsequent calls to `glUpdateObjectBufferATI()` to copy data into the object buffer.

Static and Dynamic Object Buffers

Some applications may require the ability to update a given array object within an object buffer during application execution. If this is necessary, the application should set the *usage* parameter of `glArrayObjectATI()` to `GL_DYNAMIC_ATI` when allocating the object buffer, otherwise the usage parameter should be set to `GL_STATIC_ATI`. If the application specifies an object buffer as static, its contents may still be updated, but this may result in reduced performance. It is a good idea to allocate array objects that an application intends to update in dynamic object buffers while keeping static data in separate *static* object buffers.

Object Buffer Verification

To determine if a handle references a valid object buffer, the application must use the `IsObjectBufferATI(uint buffer)` entrypoint.

Updating a Vertex Array Object (optional)

The data stored in a vertex array object can be updated after it is initially created. This is done via the `glUpdateObjectBuffer(uint buffer, uint offset, size_t size, const void *pointer, enum preserve)`. If a vertex array is going to be updated frequently, it should be declared to be *dynamic* when it is first created.

Referencing a Vertex Array Object

Now that the data is stored in server-side memory, it can be referenced via the handle returned by `glNewObjectBufferATI()`. For each array stored in the vertex array object, a call to `glArrayObjectATI()` will tell OpenGL where (the offset from start of the object) and how (type, size and stride) the array is stored. `glArrayObjectATI()` can accommodate all of the standard OpenGL array types (vertex, normal, texture coordinate, color, index and edge flag) as well as the extended types (e.g. weight array used in `ATI_vertex_blend` extension). The stride parameter of `glArrayObjectATI()` allows array object data to be stored in an interleaved fashion. When using array

objects, `glArrayObjectATI()` replaces the `gl*Pointer()` family of calls (e.g. `glVertexPointer()`, `glTexCoordPointer()`, `glNormalPointer()`, etc).

Rendering a Vertex Array Object

Rendering with a vertex array object is done exactly the same way as with standard OpenGL vertex arrays. The recommended path is via `glDrawElements()` but calls to `glDrawArrays()` and `glArrayElement()` are also available. Compiled vertex arrays (using `glLockArraysEXT()`, `glUnlockArraysEXT()`) have no effect on vertex array objects. They will, however, still increase performance for any standard OpenGL vertex arrays that are being used. If vertex array objects are being used in conjunction with standard vertex arrays, it is recommended that the application use compiled vertex arrays.

Freeing a Vertex Array Object

Freeing the resources associated with a vertex array object is done via `glFreeObjectBufferATI(uint object)`. Calling this function will free all of the vertex array object's resources.

Element Arrays

The `ATI_element_array` OpenGL extension provides a mechanism for using a vertex array object as indices into other vertex arrays objects for rendering via `glDrawElementArrayATI()`, which is designed to behave like the standard `glDrawElements()`. Storing this index in server-side memory eliminates the need to transfer it from client-side memory for every draw call, thus increasing performance. Element arrays are created and managed using the `ATI_vertex_array_object` extension. The presence of the `ATI_element_array` extension indicates that this type of vertex array object may be created. Creating and freeing element arrays is done exactly as previously described for vertex array objects.

Referencing an Element Array Object

Referencing an element array object is done the same way as referencing any vertex array object. The application must enable the client state for element arrays and let OpenGL know how and where the array is stored via `glArrayObjectATI()` specifying the array type as `GL_ELEMENT_ARRAY_ATI`.

Rendering an Element Array Object

Rendering with an element array object involves replacing the standard `glDrawElements()` call with `glDrawElementArrayATI()` or `glDrawElementArrayRangeATI()`. `glDrawElementArrayATI(enum mode, int count)` will render from the beginning of the element array up to the specified *count*. `glDrawElementArrayRangeATI(enum mode, int start, int end, count)` is used to render a subset of an element array that does not start at the beginning of the array.

Object Buffer Memory Layout

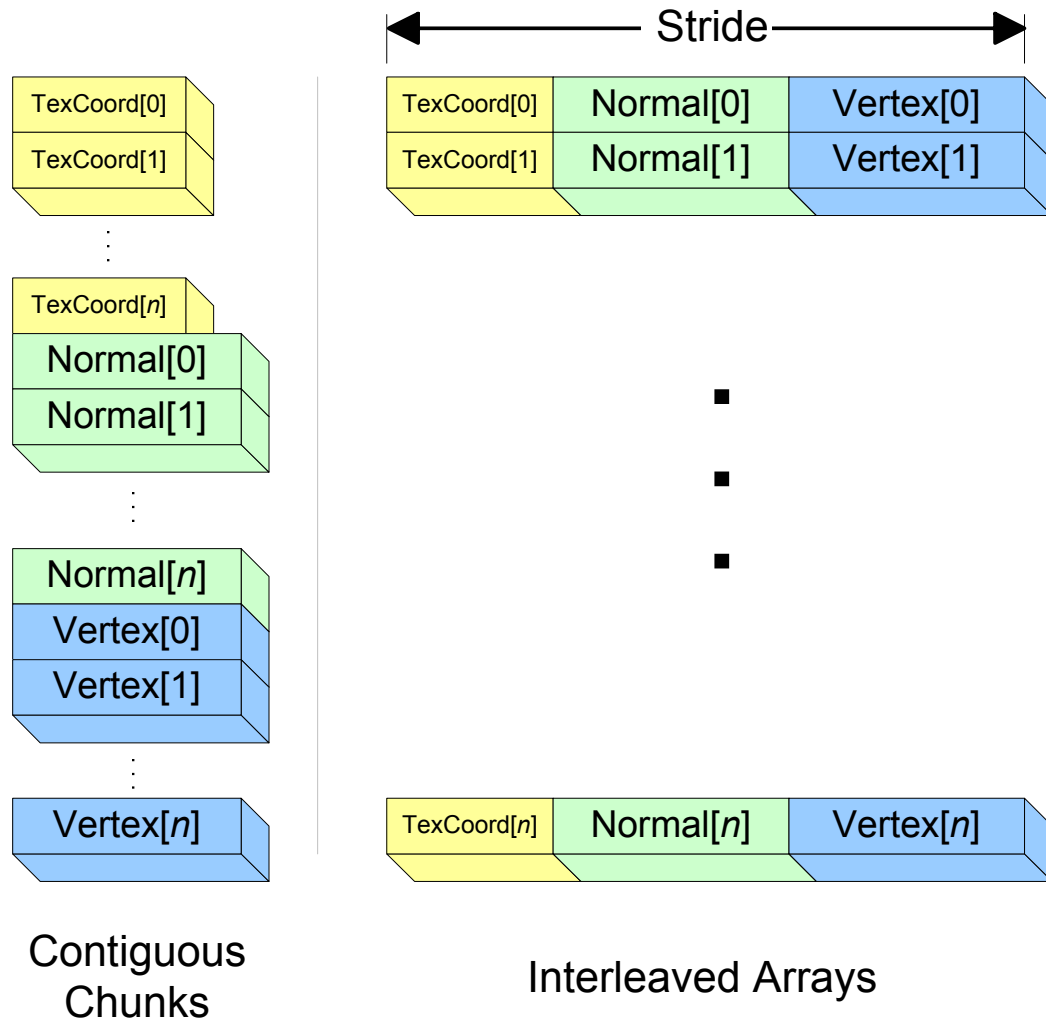


Figure 2 - Object Buffer Layouts

Figure 2 shows two examples of vertex array object layouts. The left side of the figure shows a contiguous chunk layout where arrays are tightly packed and separated from one another. The contiguous chunk example could be broken into separate vertex array objects. In keeping with OpenGL's convention, tightly packed arrays are said to have a stride of zero. The application can specify a stride of zero for tightly packed data or specify the actual array stride. In the contiguous chunk example above, the texture coordinate array would have an offset of zero and a stride of zero. The normal array would have an offset of `sizeof(TexCoord) * numTexCoord` and have a stride of zero. The vertex array would have an offset of `sizeof(TexCoord) * numTexCoord + sizeof(Normal) * numNormals` and have a stride of zero.

The right side of Figure 2 shows an interleaved array layout. This layout would be found when an array of data structures is used. The stride of the interleaved arrays example is equal to the size of the data structure that is used. In the interleaved arrays case, the texture coordinate array would have an offset of zero and a stride of `sizeof(TexCoord) + sizeof(Normal) + sizeof(Vertex)`. The normal array would have an offset of `sizeof(TexCoord)` and a stride of `sizeof(TexCoord) + sizeof(Normal) + sizeof(Vertex)`. The vertex array array would have an offset of `sizeof(TexCoord) + sizeof(Normal)` and a stride of `sizeof(TexCoord) + sizeof(Normal) + sizeof(Vertex)`. In the interleaved array case, the following might be a data structure used:

```
typedef struct vertex_s
{
    float    texCoord[2];
    float    normal[3];
    float    vertex[3];
} vertex_t;
```

These are just two examples of how to store data in a vertex array object. Naturally, an application should use the format that most conveniently works with its own dataset. The contiguous chunk method would be convenient for an application that stores its data as separate arrays. The interleaved arrays method would be good for an application that uses a custom vertex structure. It is important to note that structure packing may need to be specified in order to guarantee that the compiler does not pad structures for performance reasons. Server-side memory should be laid out thoughtfully; each element of an array should start on a 32-bit boundary. For example an array of unsigned character three-tuples (e.g. array of 24-bit RGB color values) should add an extra byte of padding to each element to ensure that each element starts on a 32-bit boundary.

Unlike vertex array objects, element array objects cannot be stored in an interleaved fashion. Element array chunks within a vertex buffer must be tightly packed and, while element arrays can be stored in the same object buffer with other vertex array objects, it is generally better to separate element arrays into their own object buffers.

Code Sample

First, we will declare a vertex structure with a 2D texture coordinate, a 3D normal and a 3D vertex position. This code uses the interleaved memory layout described above.

```
typedef struct vertex_s
{
    float    tc[2];
    float    normal[3];
    float    vertex[3];
} vertex_t;
```

The next task is to create the object buffer and fill it with data. This is done in one call to `glNewObjectBufferATI()` in the following routine:

```
void CreateVertexArrayObject(vertex_t *verts, int vertCount)
{
    int    objectSize;

    // compute the size of the object
    objectSize = sizeof(vertex_t) * vertCount;

    // create the object and pass a pointer to the object data
    g_uiVertexObject = glNewObjectBufferATI(objectSize, verts, GL_STATIC_ATI);
}
```

Rendering with the data in this object buffer is done in a very similar manner to rendering with generic vertex arrays in unextended OpenGL. Note, however, that the elements are stored in a client-side array (triangles):

```
void RenderObject( )
{
    // enable the appropriate client states
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);

    // define the array pointers using the vertex array object
    glArrayObjectATI(GL_TEXTURE_COORD_ARRAY, 2, GL_FLOAT, sizeof(vertex_t),
                    g_uiVertexObject, 0);
    glArrayObjectATI(GL_NORMAL_ARRAY, 3, GL_FLOAT, sizeof(vertex_t),
                    g_uiVertexObject, sizeof(float) * 2);
    glArrayObjectATI(GL_VERTEX_ARRAY, 3, GL_FLOAT, sizeof(vertex_t),
                    g_uiVertexObject, sizeof(float) * 5);

    // use standard glDrawElements
    glDrawElements(GL_TRIANGLES, triangleCount * 3, GL_UNSIGNED_INT, triangles);

    // disable the appropriate client states
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
}
```

In order to render this data using a server-side hardware accessible element array, we create an additional object buffer which contains the element data:

```
void CreateElementArrayObject(int *tris, int triCount)
{
    int    objectSize;

    // compute the size of the element object
    objectSize = sizeof(int) * 3 * triCount;

    // create the element object and pass a pointer to the element data
    g_uiElementObject = glNewObjectBufferATI(objectSize, tris, GL_STATIC_ATI);
}
```

In the following code fragment, we enable an additional array object of type `GL_ELEMENT_ARRAY_ATI` which contains the element data. In this case, `glDrawElementArrayATI()` is used to render the triangles:

```
void RenderObjectWithElementArrays ( )
{
    // enable the appropriate client states
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);

    glEnableClientState(GL_ELEMENT_ARRAY_ATI);

    // define the array pointers using the vertex array object
    glArrayObjectATI(GL_TEXTURE_COORD_ARRAY, 2, GL_FLOAT, sizeof(vertex_t),
                    g_uiVertexObject, 0);
    glArrayObjectATI(GL_NORMAL_ARRAY, 3, GL_FLOAT, sizeof(vertex_t),
                    g_uiVertexObject, sizeof(float) * 2);
    glArrayObjectATI(GL_VERTEX_ARRAY, 3, GL_FLOAT, sizeof(vertex_t),
                    g_uiVertexObject, sizeof(float) * 5);
    glArrayObjectATI(GL_ELEMENT_ARRAY_ATI, 1, GL_UNSIGNED_INT, 0,
                    g_uiElementObject, 0);

    // use glDrawElementArrayATI to render using the element array
    glDrawElementArrayATI(GL_TRIANGLES, g_nNumTriangles * 3);

    // disable the appropriate client states
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_ELEMENT_ARRAY_ATI);
}
```

Hardware Support

Boards based on ATI's RADEON™ and RADEON™ 8500 graphics processors support the `ATI_vertex_array_object` and `ATI_element_array` extensions. This includes the original RADEON™, All-In-Wonder™ RADEON, RADEON™ 8500, FireGL™ 8800 and All-In-Wonder RADEON™ 8500 DV boards.

Online References

[ATI_vertex_array_object](#) and [ATI_element_array](#) specs and sample applications, including [SimpleVAO](#) and [RADEON8500PointLightShader](#) are available online.