Indexing Multiple Vertex Arrays

26

Arnaud Masserann

26.1 Introduction

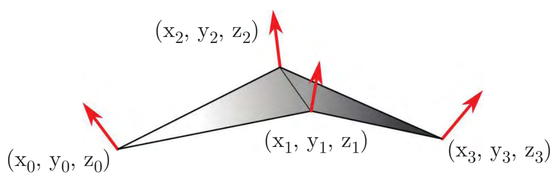
One of OpenGL's features is vertex buffer object (VBO) indexing, which allows developers to reuse a single vertex in several primitives. Since vertex attributes don't need to be duplicated, indexing saves memory and bandwidth. Given that the GPU is often memory-bound, most of the time we can get extra speed with indexing.

Indexing requires having a single index for positions, texture coordinates, normals, and so on. Unfortunately, this is not how many 3D file formats work: for instance, COLLADA has different indices for each vertex attribute. This is problematic in asset pipelines, where models can come from a variety of sources.

This chapter shows a simple algorithm that transforms several attribute buffers, each using different indices, into a format that is directly usable by OpenGL. For applications that do not use indexing, this chapter provides a simple way to improve run-time performance. In practice, speedups of about 1.4 times can be expected, and this format opens possibilities for further optimizations.

26.2 The Problem

With nonindexed VBOs (see Figure 26.1), we need to specify all attributes for each vertex: position, color, and all needed UV coordinates, normals, tangents, bitangents, etc.



Vertex Array Buffer:
(x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2),
(x_2, y_2, z_2), (x_1, y_1, z_1), (x_3, y_3, z_3)

Normal Array Buffer:
(x'_0, y'_0, z'_0), (x'_1, y'_1, z'_1), (x'_2, y'_2, z'_2),
(x'_2, y'_2, z'_2), (x'_1, y'_1, z'_1), (x'_3, y'_3, z'_3)

Figure 26.1. A nonindexed VBO.

Nonindexed VBOs suffer from two performance penalties. First, on most meshes this method uses more memory. For instance, on a sphere with 1000 vertices, all vertices are shared by three triangles. A nonindexed VBO with `GL_FLOAT` attributes for positions, UVs, and normals will take $3 \times 1{,}000 \times (3 \times 4 + 2 \times 4 + 3 \times 4) = 3 \times 1{,}000 \times 32 = 96,000$ bytes. A similar, indexed VBO will take 96,000/3 bytes, plus $3 \times 1{,}000 \times 4 = 12{,}000$ for the index buffer, totaling 44,000 bytes. In this ideal case, the indexed VBO only takes 45% of the size of the nonindexed VBO. Indexing thus reduces both the memory footprint and the PCI-e transfers.

The second performance penalty comes from the difference in cache usage. There are two kinds of vertex caches:

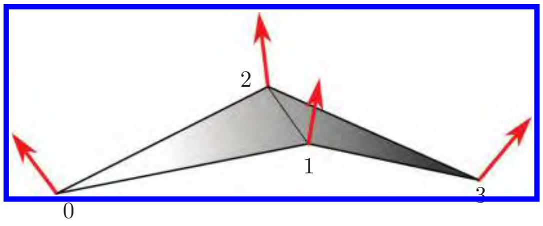
- AMD GPUs have a pretransform vertex cache that contains a part of the raw VBO. This cache is used to feed the vertex shader.

- The post-transform cache is used to store the ouput variables of the vertex shader. This is useful because most of the time, a vertex is used by several triangles. The cache avoids the cost of re-executing the same computations for each vertex shared by several triangles. However, it uses the index of the vertex as a key, so if primitives are drawn without indexing, the cache has no effect.

There are two consequences. First, simple indexing will natively improve performance. Second, the use of both of these caches can be optimized:

- If the element buffer contains indices to vertices that have a good spatial locality, the pretransform cache will make a large number of hits. In other words, indices 0-1-2 are better than 0-50-99.

- If neighboring triangles are drawn consecutively, most of the used vertices will be in the post-transform cache, available for immediate reuse. A number of algorithms can be found in the literature for reorganizing the indices in order to get a better post-transform cache usage. In particular, I recommend `nvTriStrip`, which is slow but ready-to-use, and Tom Forsyth's algorithm [Forsyth 06], which runs in linear time.

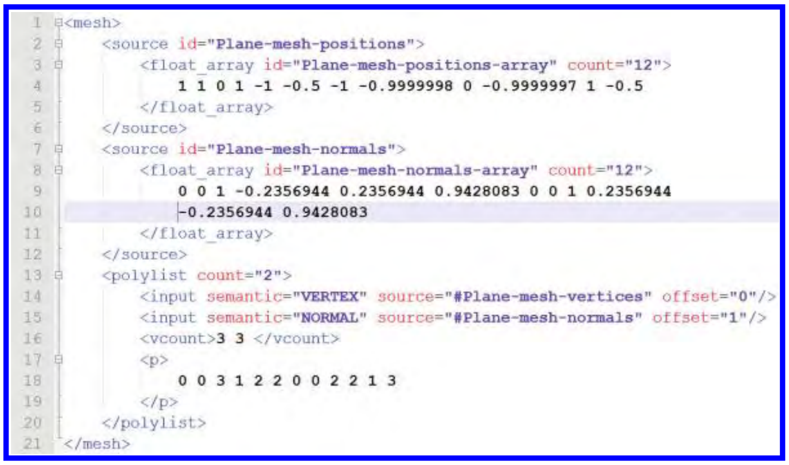


Vertex Array Buffer:	Normal Array Buffer:	Element Array Buffer:
(x_0, y_0, z_0),	(x'_0, y'_0, z'_0),	0 1 2, 2 1 3
(x_1, y_1, z_1),	(x'_1, y'_1, z'_1),	
(x_2, y_2, z_2),	(x'_2, y'_2, z'_2),	
(x_3, y_3, z_3)	(x'_3, y'_3, z'_3)	

Figure 26.2. An indexed VBO.

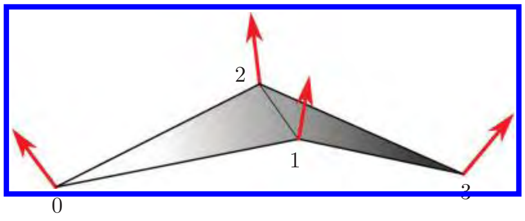
Figure 26.2 shows what an indexed VBO looks like, along with the associated attributes. Note that both coordinates and normals are shared for vertices 1 and 2.

For these reasons, indexing is recommended by all major GPU vendors [NVIDIA 08, Hart 04, Imagination Technologies 09]. However, Figure 26.3 shows an excerpt of the COLLADA export of a similar mesh.



```
<mesh>
    <source id="Plane-mesh-positions">
        <float_array id="Plane-mesh-positions-array" count="12">
            1 1 0 1 -1 -0.5 -1 -0.9999998 0 -0.9999997 1 -0.5
        </float_array>
    </source>
    <source id="Plane-mesh-normals">
        <float_array id="Plane-mesh-normals-array" count="12">
            0 0 1 -0.2356944 0.2356944 0.9428083 0 0 1 0.2356944
            -0.2356944 0.9428083
        </float_array>
    </source>
    <polylist count="2">
        <input semantic="VERTEX" source="#Plane-mesh-vertices" offset="0"/>
        <input semantic="NORMAL" source="#Plane-mesh-normals" offset="1"/>
        <vcount>3 3 </vcount>
        <p>
            0 0 3 1 2 2 0 0 2 2 1 3
        </p>
    </polylist>
</mesh>
```

Figure 26.3. A COLLADA mesh.



Vertices:	Normal:	Indices for Vertices:
(x_0, y_0, z_0),	(x'_0, y'_0, z'_0),	0 3 2, 0 2 1
(x_1, y_1, z_1),	(x'_1, y'_1, z'_1),	
(x_2, y_2, z_2),	(x'_2, y'_2, z'_2),	Indices for Normal:
(x_3, y_3, z_3)	(x'_3, y'_3, z'_3)	0 1 2, 0 2 3

Figure 26.4. COLLADA representation of the mesh.

As shown in Figure 26.4, several index buffers, or element array buffers in OpenGL terminology, are needed—one for each attribute. This is not possible in OpenGL, where all attributes must be indexed by the same element array buffer.

This chapter shows a simple solution to convert nonindexed data into an indexed form, allowing its use in an efficient way with many file formats such as OBJ, X, VRML, and COLLADA.

26.3 An Algorithm

The trick is to only reuse a vertex if all of its attributes match. We can simply iterate through all input vertices and append them to the output buffer when it doesn't already contain a matching vertex.

In Listing 26.1, `in_vertices` and `out_vertices` are arrays. It is important that `getSimilarVertexIndex` is as fast as possible since it is called once for each input vertex. This can be done using a data structure like a `std::map`, which limits the complexity of the algorithm to $O(n \log n)$, or a `std::hash_map`, with a theoretical complexity of $O(n)$.

An important detail is that this version of the code assumes that `in_vertices` and `out_vertices` contain packed vertices, so comparing two vertices takes all attributes into account.

```
// Reserve space in output vectors
out_indices.reserve(in_vertices.size());
// ...

std::map<PackedVertex,unsigned short> VertexToOutIndex;

// For each input vertex
for (unsigned int i = 0; i < in_vertices.size(); i++ )
{
  PackedVertex packed(in_vertices[i], in_uvs[i], in_normals[i]);

  unsigned short index;
  bool found = getSimilarVertexIndex(packed, VertexToOutIndex , index);

  if (found)
  {
    // A matching vertex is already in the VBO, use it instead.
    out_indices.push_back(index);
  }
  else
  {
    // If not, it needs to be added in the output data.
    out_vertices.push_back(in_vertices[i]);
    out_uvs.push_back(in_uvs[i]);
    out_normals.push_back(in_normals[i]);
    unsigned short newindex = (unsigned short)out_vertices.size() - 1;
    out_indices.push_back(newindex);
    VertexToOutIndex[packed] = newindex;
  }
}

// Downsize the output vectors to the exact needed size
std::vector<unsigned short>(out_indices).swap(out_indices);
// ...
```

Listing 26.1. The indexing algorithm.

26.4 Vertex Comparison Methods

The containers need a comparison function in order to create their internal tree. This function does not need to have a real meaning; the only requirement is that for two equal vertices, `v1` and `v2`, `compare(v1,v2) == false` and `compare(v2,v1) == false` (no vertex is greater than the other).

26.4.1 If/Then/Else Version

The comparison function can be implemented as shown in Listing 26.2. This is the most generic version, and it will work on any platform. What's more, we can tweak `isEqual` depending on our needs. If we know that similar vertices will have exactly the same coordinates, we can implement `isEqual` with the `==` operator. This is usually the case, because the coordinates are not modified with floating-point operations during the export and import phases. On the other hand, we may want to weld vertices with slight differences in their normals to reduce the size of the VBO or to smooth out the rendering. This can be done by using an epsilon in `isEqual`.

```
if (isEqual(v1.x,v2.x))
{
  // Can't sort on this criterion, try another
  if (isEqual(v1.y,v2.y))
  {
    if (isEqual(v1.z,v2.z))
    {
      // Same for UVs, normals, ...
      // Vertices are equal
      return false;
    }
    else
    {
      return v1.z > v2.z;
    }
  }
  else
  {
    return v1.y > v2.y; // Can't sort on x, but y is ok
  }
}
else
{
  return v1.x > v2.x; // x is already discriminant, sort on this axis
}
```

Listing 26.2. Comparison function 1.

26.4.2 memcmp() Version

If the vertices are packed and we only want to weld vertices with perfectly equal coordinates, this function can be greatly simplified by using `memcmp` instead, as shown in Listing 26.3.

```
return memcmp((void*)this, (void*)&that, sizeof(PackedVertex))>0;
```

Listing 26.3. Comparison function 2.

This will work if the structure is tightly packed (aligned on 8 bits, which may not be a good idea since the driver will probably realign it on 32 bits internally), or if all unused zones are always set to the same value. This can be done by zeroing the whole structure with `memset`, for instance in the constructor, as shown in Listing 26.4.

```
memset((void*)this, 0, sizeof(PackedVertex));
```

Listing 26.4. Dealing with alignment.

26.4.3 Hashing Function

We can also implement the algorithm using a `Dictionary`, or `std::hash_map` in C++, instead. Such a container requires two functions: a hash function, which converts our vertex into an integer and an equality function. The equality function is straightforward: all attributes must be equal. The hash function can be implemented in a variety of ways; the main constraint is that if two vertices are considered equal, their hash must be equal. This actually heavily restricts our possibilities of using an epsilon-based equality function.

Listing 26.5 shows a simplistic implementation. It groups the vertices in a uniform grid of 0.01 units and computes the hash by multiplying each new coordinate by a prime number, which avoids clustering vertices in common planes. Finally, the hash is modulated by 2^{16}, which creates 65,536 bins in the hashmap. Other attributes are not used, because position is usually the most separating criterion, and they will be taken into account by the equality function.

For a more detailed analysis of hashing functions for 3D vertices, see, for instance [Hrádek and Skala 03].

```
class hash<PackedVertex>
{
  public size_t operator()(const PackedVertex & v)
  {
    size_t x = size_t(v.position.x) * 100;
    size_t y = size_t(v.position.y) * 100;
    size_t z = size_t(v.position.z) * 100;
    return (3 * x + 5 * y + 7 * z) % (1 << 16);
  }
}
```

Listing 26.5. Hashing function.

26.5 Performance

Table 26.1 and Figure 26.5 give indexing times (in milliseconds) for models of various complexities. A standard `std::map` is used, with the `memcmp` version of the

Model	# vertices	# triangles	Indexing time (ms)
Suzanne	500	1,000	0.7
Plane	10,000	20,000	14
Sponza	153,000	279,000	820

Table 26.1. Indexing times for various models.

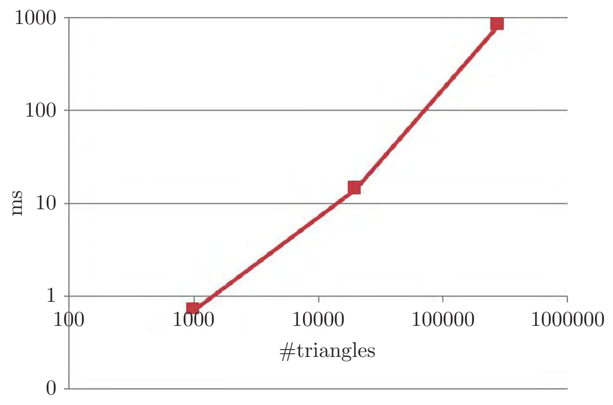


Figure 26.5. Indexing time w.r.t. triangles count.

comparison operator. The vertices are packed and have floating-point UVs and normals. Times are given for an Intel i5 2.8GHz CPU.

Table 26.2 gives rendering speeds (in milliseconds) for the same three models. Each model is rendered 100 times per frame in different positions, with a vertex shader that outputs five varyings and a Blinn-Phong fragment shader with one texture fetch.

Indexed versions are at least as fast, and up to 1.8 times faster than their non-indexed equivalents. This number is mostly valid for meshes with a topology that is already cache-friendly; other models will usually require a post-transform cache-optimization pass, as mentioned above. For instance, a Standford Bunny model

Model		ms/frame			
		Indexed interleaved	Indexed noninterleaved	Nonindexed interleaved	Nonindexed noninterleaved
NVIDIA GTX 470	Suzanne	1.6	1.6	1.6	1.6
	Plane	6.2	6.3	11.5	11.5
	Sponza	154	154	161	161
AMD HD 6570	Suzanne	4.8	4.8	5.1	5.1
	Plane	25	25	31	32
	Sponza	281	277	282	304
Intel GMA 3000	Suzanne	16	15	16	16
	Plane	77	74	128	129
	Sponza	1170	1166	1228	1229

Table 26.2. Rendering performance.

Method	Rendering time (ms/frame)
Nonindexed interleaved	0.70
Indexed interleaved	0.58
Indexed interleaved and optimized	0.50

Table 26.3. Rendering performance for an optimized model.

with 35,000 vertices is rendered 1.4 times faster when indexed and optimized with `nvTriStrip`, as shown in Table 26.3.

26.6 Conclusion

The proposed algorithm has some key advantages:

- it is simple to implement;
- it is cross-platform, and works with any CPU, OpenGL version, programming language, and OS;
- it is simple to customize and to integrate into existing code;
- it can either generate one interleaved array or separate arrays, depending on our needs;
- it can be integrated directly into our asset pipeline so that we have no runtime performance penalty;
- it opens possibilities for further performance gains through pre- and post-transform cache optimizations;
- most importantly, it can get us extra milliseconds for free.

An example implementation using interleaved arrays and an `std::map` with the `memcmp` comparison function can be found on the OpenGL Insights website, www.openglinsights.com.

Bibliography

[Forsyth 06] Tom Forsyth. "Linear-Speed Vertex Cache Optimisation." http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html, September 28, 2006.

[Hart 04] Even Hart. "OpenGL Performance Tuning." http://developer.amd.com/media/gpu_assets/PerformanceTuning.pdf, 2004.

[Hrádek and Skala 03] Jan Hrádek and Václav Skala. “Hash Function and Triangular mesh Reconstruction.” *Computers & Geosciences* 29:6 (2003), 741–751.

[Imagination Technologies 09] Imagination Technologies. “PowerVR Application Development Recommendations.” http://www.imgtec.com/powervr/insider/sdk/KhronosOpenGLES2xSGX.asp, 2009.

[NVIDIA 08] NVIDIA. *NVIDIA GPU Programming Guide*, 2008.