

The Cube

a Flexporter tutorial
Version 1.0

Urban legend : the cube is the easiest possible object to test an exporter.

Right ? Wrong. Actually, while it is the simplest object geometrically speaking, it might very well be the most complex one as far as exporting is concerned.

Here's why.

HARDWARE RULES

MAX native data is not hardware-friendly. It just isn't. "Hardware-friendly" means an object is available as a unique list of vertices, or better, *indexed* vertices. From now on we'll only speak about indexed vertices, since indexed primitives are usually the fastest ones to render.

Now what's a *vertex* ? In this context of hardware-friendliness, a vertex is actually a *fat vertex*, or *flexible vertex* in D3D parlance. That is, it's not just a position - it's a position and all its attributes (diffuse color, normal vector, etc). From now on, we'll say "point" for a simple geometric position, and "vertex" for a point and its attributes. For example, here's how a vertex structure could look like :

```
struct Vertex
{
    Point    mPos;           // The 3D point
    udword   mDiffuseColor;  // The point's diffuse color
    Point    mNormal;       // The point's normal vector
};
```

So, two different vertices can actually have the same point, but with different normals.

Just to make sure we're using the same vocabulary, let's define :

- the *geometry*, as a list of points (or vertices, see below)
- the *topology*, as a list of indices referencing points from the geometry

You must be familiar with this concept, since it's widely used in both OpenGL or D3D, for example by *glDrawElements* (GL) and *DrawIndexedPrimitive* (D3D) methods.

MAX native data is not hardware-friendly, because it uses *different* geometries and topologies to describe a mesh. You read that well. MAX internally has a (Geometry, Topology) couple for *each* vertex attribute. To handle a position, a single set of mapping coordinates and vertex colors, you end up with 3 geometries (a list of points, a list of UVs, a list of colors) and 3 independent topologies indexing those 3 different pools (a list of faces, a list of texture-faces, a list of color-faces). The only requirement is that all topologies have the same number of faces. On the other hand, the geometries can have an arbitrary number of elements. This is a very flexible way

of storing a mesh, but it doesn't help when it comes to stuffing all of this in a *vertex buffer*.

We said above that the geometry was defined as a list of points or a list of vertices. That exactly where the hiatus lies :

- MAX defines it as a list of points (3D positions)
- GL and D3D need it as a list of vertices (3D position + attributes)

THE CANONICAL CUBE

If you export the canonical MAX cube with Flexporter's ASCII exporter, you get something like that :

```
MAX native mesh data:
12 faces
8 vertices
12 mapping coordinates
0 vertex colors
Parity is true.
```

Let's have a closer look at this.....

12 faces is easy to understand, since we're only dealing with triangles. The cube has 6 *quad-faces*, each quad is 2 triangles, that's $6 * 2 = 12$ triangles, no problem.

8 vertices is even more obvious. Cube. 8 vertices. Obvious ? Yes, but inaccurate with our new wordlist. It's 8 *points*, actually. 8 geometric points. In the end, in your vertex buffer, you'll later see that a cube has actually 24 vertices. But we'll handle that later. For now, our 8 points are listed as :

```
Vertices:
-39.999992 0.000000 -39.999989
39.999992 0.000000 -39.999989
-39.999992 0.000000 39.999989
39.999992 0.000000 39.999989
-39.999992 79.999977 -39.999989
39.999992 79.999977 -39.999989
-39.999992 79.999977 39.999989
39.999992 79.999977 39.999989
```

The following 12 mapping coordinates don't sound very obvious, and indeed, they aren't. This is confusing, but that's just what MAX uses internally, and what you get out of the MAX SDK. We'll have to deal with it. The list is :

```
Mapping coordinates:
0.000000 0.000000 0.000000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
1.000000 1.000000 0.000000
0.000000 0.000000 0.000000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
1.000000 1.000000 0.000000
```

```
0.000000 0.000000 0.000000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
1.000000 1.000000 0.000000
```

As you see, there are redundant entries here. The number of mapping coordinates (12) is also different from the number of geometric points (8). This is what we were saying in the first part : in MAX, the different geometries for each vertex attribute can have different numbers of elements.

This is exactly what happens for the cube, and that's one of the reason why it is not that simple object people are usually thinking about. You just *can't* simply copy the MAX data into a vertex buffer. There's a compulsory cleaning pass somewhere that must be performed.

But wait, that's only the beginning. There's worse.

SMOOTHING GROUPS AND CONSEQUENCES

This is where the cube really becomes a painful object to deal with, as opposed to, say a sphere.

What about vertex normals ? They aren't exported as part of the MAX native mesh, so we'll have to compute them ourselves, using the object's *smoothing groups*. Smoothing groups are exported as a 32-bits mask for each face. Each bit corresponds to a given group.

In our exported text file, smoothing groups are stored along with topologies :

```
Faces:
(vref0 vref1 vref2  tref0 tref1 tref2  cref0 cref1 cref2  MatID Smg EdgeVisCode)
0 2 3 9 11 10 -1 -1 -1 0 2 0
3 1 0 10 8 9 -1 -1 -1 0 2 0
4 5 7 8 9 11 -1 -1 -1 0 4 0
7 6 4 11 10 8 -1 -1 -1 0 4 0
0 1 5 4 5 7 -1 -1 -1 0 8 0
5 4 0 7 6 4 -1 -1 -1 0 8 0
1 3 7 0 1 3 -1 -1 -1 0 16 0
7 5 1 3 2 0 -1 -1 -1 0 16 0
3 2 6 4 5 7 -1 -1 -1 0 32 0
6 7 3 7 6 4 -1 -1 -1 0 32 0
2 0 4 0 1 3 -1 -1 -1 0 64 0
4 6 2 3 2 0 -1 -1 -1 0 64 0
```

The first line is here as a helper, to describe exported data. We see the 3 different topologies introduced at the start of the document :

- one for points (vref0, vref1, vref1)
- one for UVs (tref0, tref1, tref2)
- one for colors (cref0, cref1, cref2).

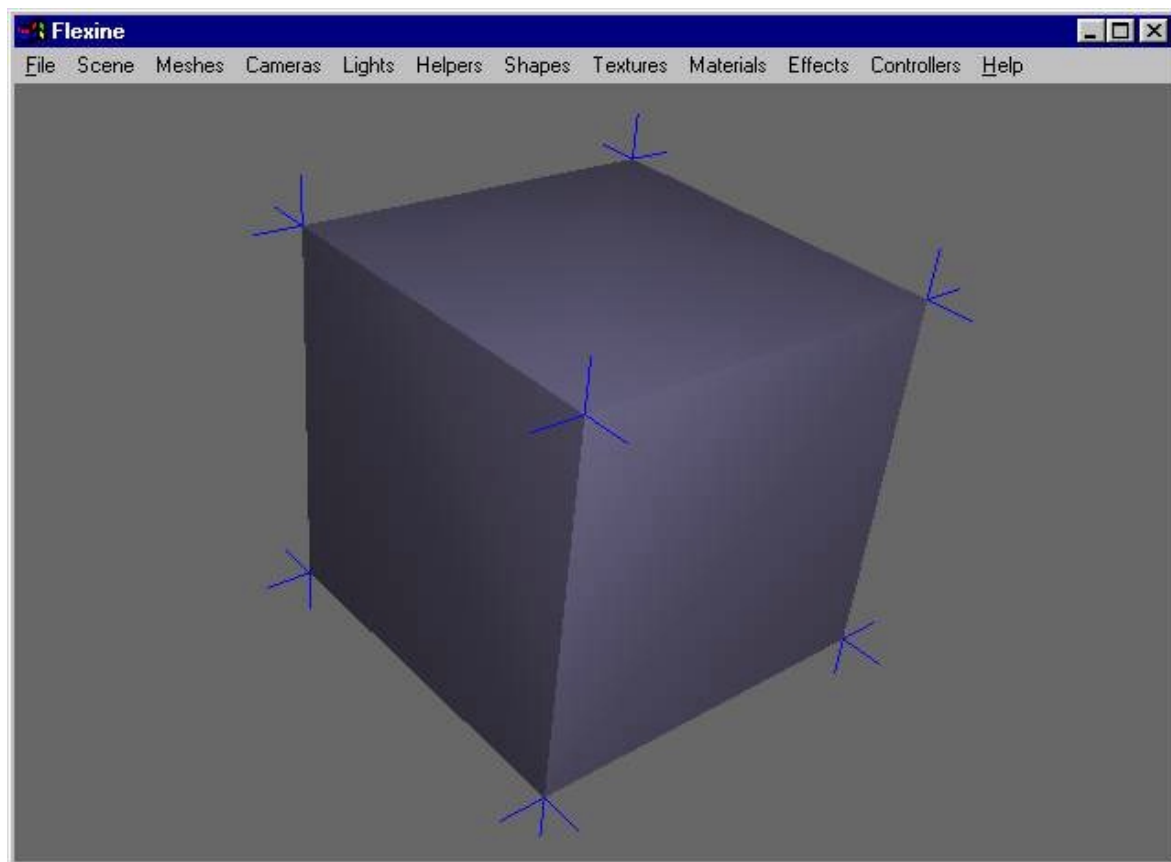
Each line then describes a triangle in the mesh. The first one, for example, tells us that :

- the first triangle references vertices (0, 2, 3) in the vertex pool (the previous list of 8 vertices)
- the first triangle references mapping coordinates (9, 11, 10) in the UV pool (the previous list of 12 mapping coordinates)
- the first triangle has no vertex colors (-1, -1, -1)
- the first triangle uses material 0
- the first triangle uses smoothing groups 2
- the first triangle has edge visibility code 0

As you see, the cube isn't our friend since it actually uses different smoothing groups for each quad-face. Since each point of the cube belongs to 3 triangles, this means that each point will actually have 3 different normals, depending on which triangle the point belongs to. So, in the end, we'll have to use $3 \times 8 = 24$ vertices in our final vertex buffer. For example the first point P0 could be *replicated* 3 times in the vertex buffer :

```
Point P0;
Point FirstNormal;
Point P0;
Point SecondNormal;
Point P0;
Point ThirdNormal;
```

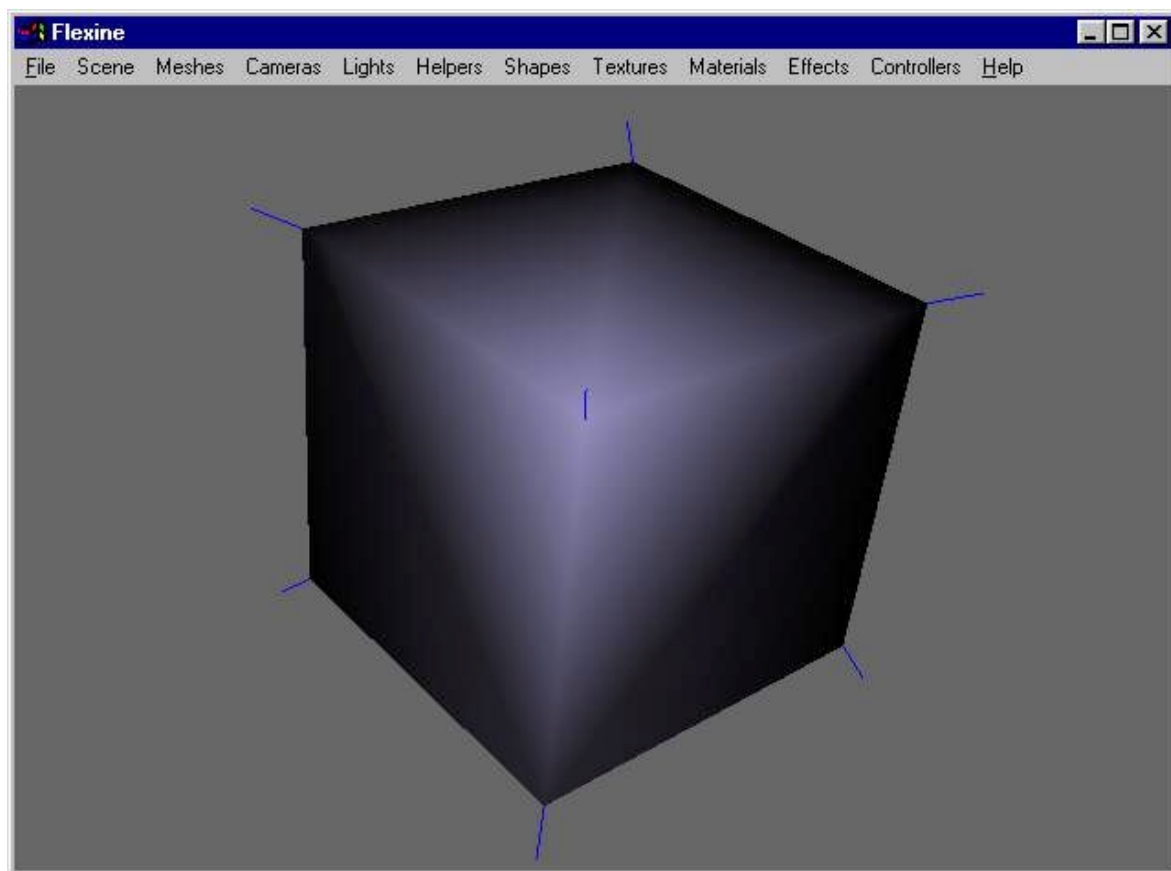
It may seem like a waste, but that's still the most efficient way to organize things.



Picture 1 : standard cube, standard smoothing groups

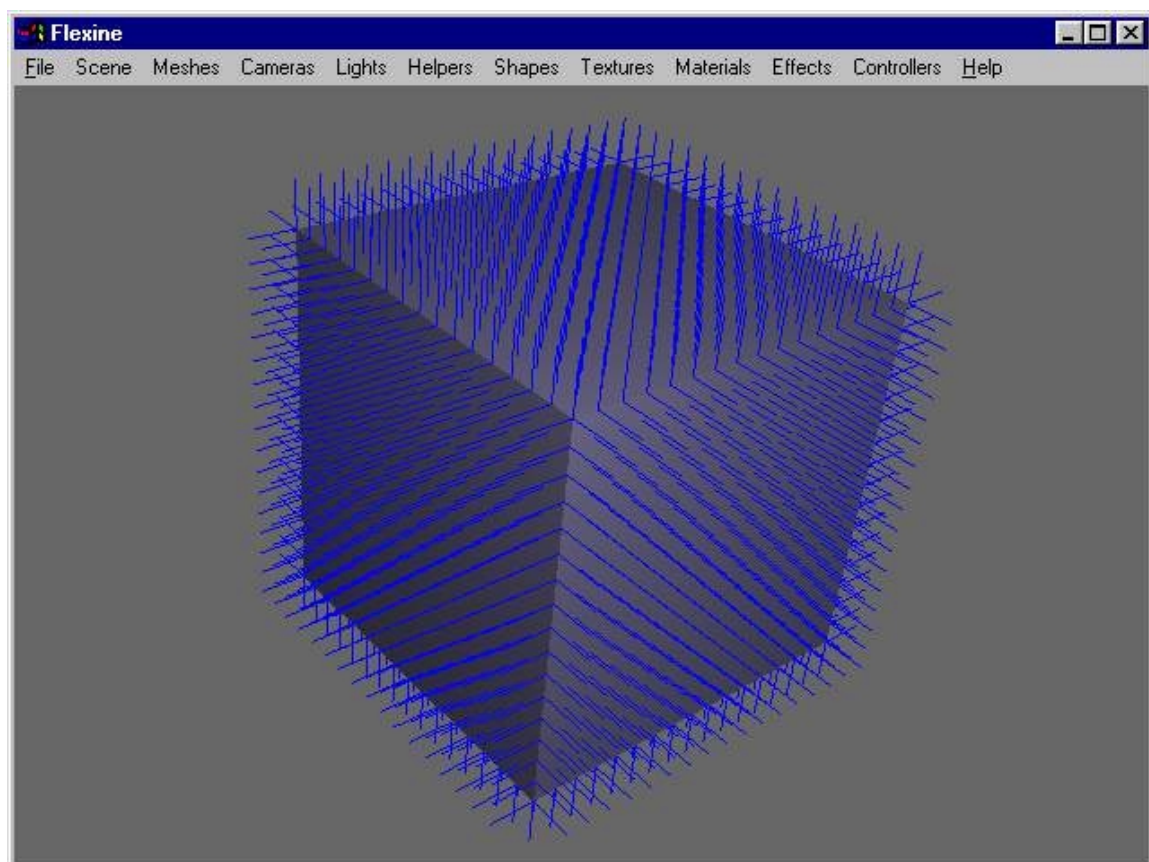
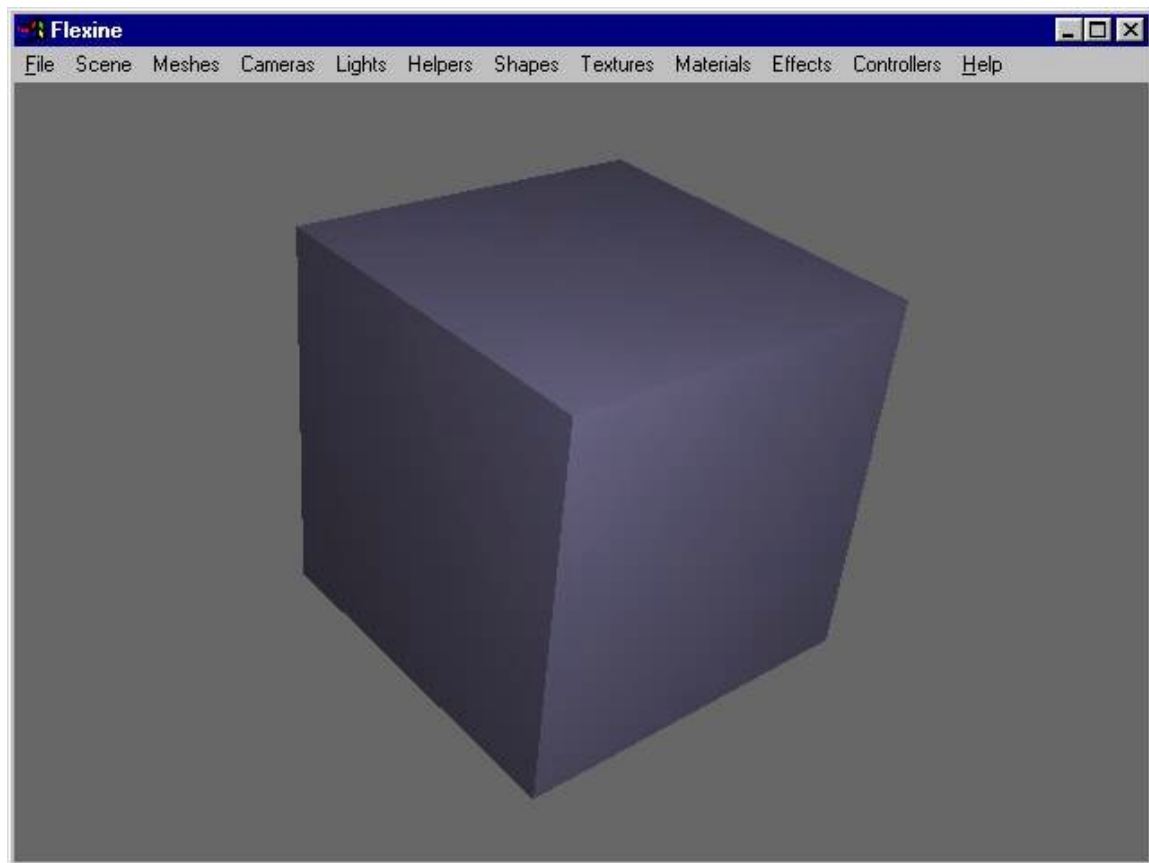
In case those 3 different normals for each cube point doesn't sound obvious to you, let's have some pictures. In picture 1 you can see the standard cube (12 faces) with default smoothing groups. Each little blue segment is a normal vector. Each of the 8 cube points in this picture has 3 normals, as expected.

Notice the effect of those different normals on shading : there's a clear cut between each quad-face (each edge is clearly visible). This happens because each quad-face belongs to different smoothing groups. Without smoothing groups, using a single normal at each point, our cube would look like the one in picture 2. Not really what we want (i.e. it doesn't look like in MAX)

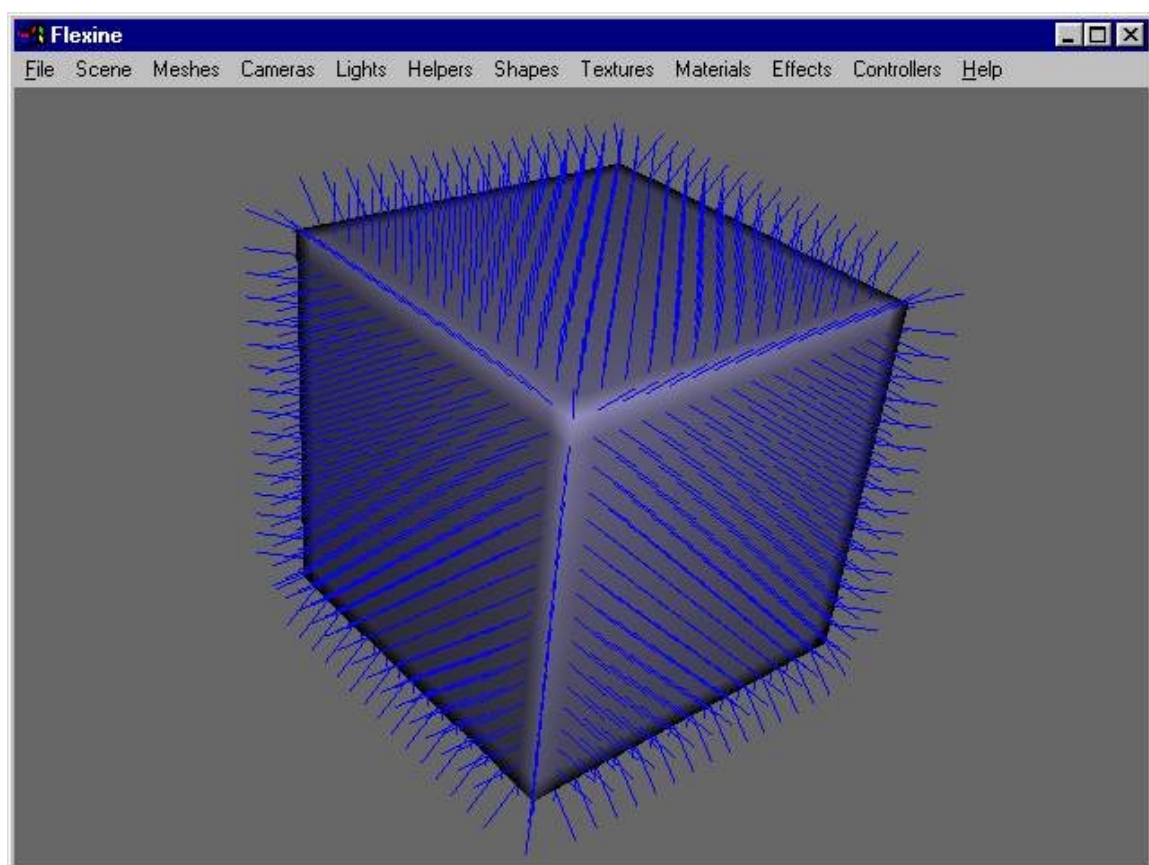
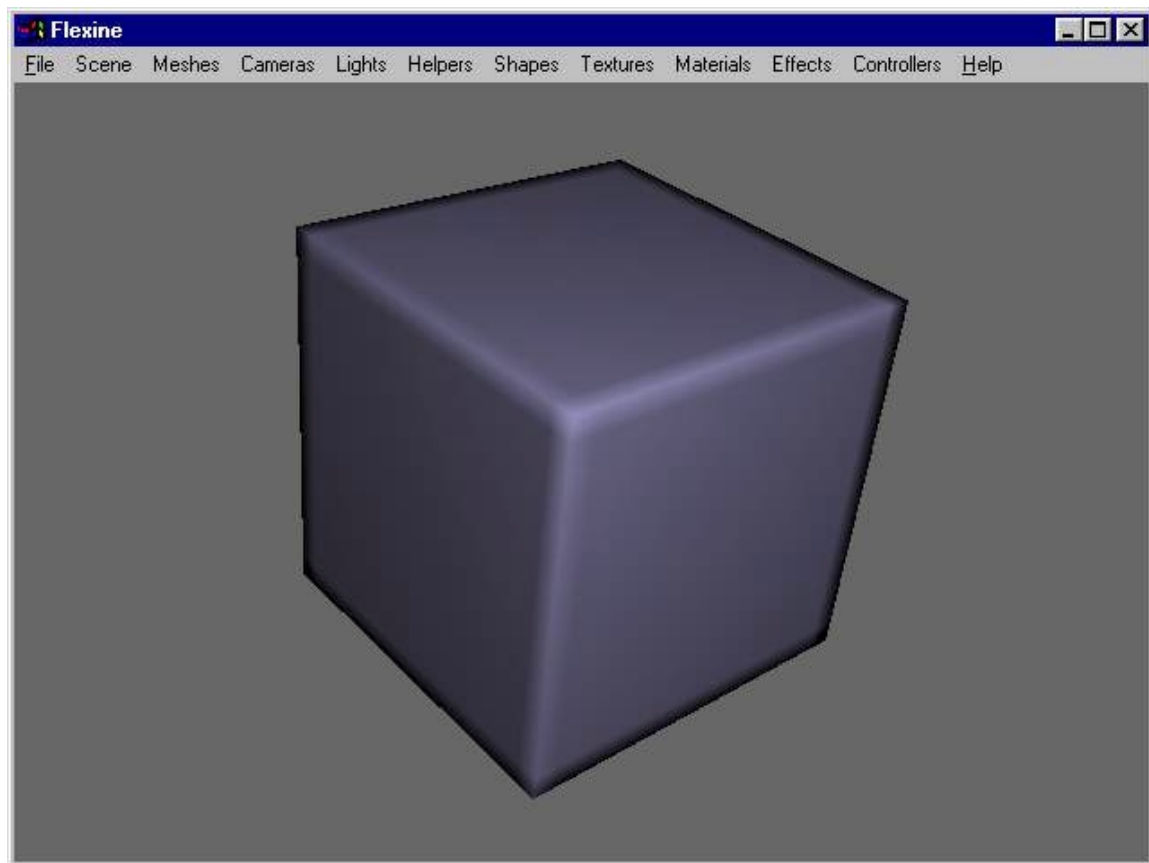


Picture 2 : standard cube, unique smoothing group

As a side note however, a smooth shading can produce interesting effects when tessellation is high. In picture 3, our cube has been tessellated a lot, and uses default smoothing groups. As you see, there's no visible difference between this and the previous non-tessellated version. That's because all vertex normals are aligned with face normals, (picture 4) hence tessellation has no influence on shading.



Picture 3 and 4 : tessellated cube, standard smoothing groups



Picture 5 and 6 : tessellated cube, unique smoothing group

Now, the same tessellated cube using a unique smoothing group produces a very different result, as seen in picture 5. Notice how vertex normals are now curved at each corner point (picture 6).

So, to summarize, our "simple" cube :

- doesn't expose a one-to-one mapping between the native 8 points and the native 12 mapping coordinates
- uses different topologies for points and mapping coordinates
- uses 3 different normals for each native point

In other words, the cube captures all possible troubles into a single mesh. Not bad for the "most simple object" !

How do we deal with that mess ? The solution is called *consolidation*.

CONSOLIDATION TO THE RESCUE

Consolidation is the process by which MAX data is transformed into a hardware-friendly representation, where a single topology references a single list of fat vertices. Each fat vertex, as described before, has a geometric position, a normal vector, and possibly more attributes like diffuse or specular colors.

Flexporter does that for you, you just have to select "Consolidate mesh" in the *Options Panel*. Then, the ASCII file contains consolidated data. For our cube it goes like this :

GEOMETRY

Nb GeomPts: 8
Nb Verts: 24
Nb TVerts: 4

"Nb GeomPts" is the number of geometric points in the original vertex cloud.

"Nb Verts" is the number of vertices after some points have been replicated by the consolidation process. As expected, we end up with 24 vertices.

"Nb TVerts" is the number of texture vertices. Why 4 ? This is just how Flexporter reduced the original MAX list of 12 texture-coordinates, down to 4 (we'll see the new list below).

Next we have a list of indices :

Indexed geometry:
0 3 2 1 4 7 5 6 0 5 1 4 1 7 3 5 3 6 2 7 2 4 0 6

Here's a hint : when you're facing a list of indices and you're not sure what pool it's actually indexing, just check the highest index, and look for a pool containing (highest_index + 1) entries...

In this case, we have a list of 24 indices into the list of geometric points. You can check that the highest index is 7, so the pool of points should contain 8 entries. Those are our 8 original geometric points, exposed that way in the consolidated mesh :

```
Vertices: (8)
-39.999992 0.000000 -39.999989
39.999992 0.000000 -39.999989
-39.999992 0.000000 39.999989
39.999992 0.000000 39.999989
-39.999992 79.999977 -39.999989
39.999992 79.999977 -39.999989
-39.999992 79.999977 39.999989
39.999992 79.999977 39.999989
```

This list may differ from the one already exposed in the native MAX data, even if for the cube it doesn't. Sometimes indeed, MAX has redundant or unused vertices in its meshes. The consolidation process gets rid of them.

The same goes for mapping coordinates :

```
Indexed UVWs:
1 2 3 0 0 3 1 2 0 3 1 2 0 3 1 2 0 3 1 2 0 3 1 2
```

This is a list of 24 indices as well. Max index is 3, so the pool of UV coords should contain 4 entries. This is our mysterious "Nb TVerts = 4" from above, and next we find the obvious optimized list :

```
UVW mappings: (4)
0.000000 0.000000
1.000000 0.000000
0.000000 1.000000
1.000000 1.000000
```

This is followed by the list of vertex normals. That one might be confusing : it is not indexed like points and UVs ! Why ? Simply because the normal computation takes another code path in Flexporter, and that code path is incomplete. Nothing should prevent normals to be indexed otherwise. But so far that's a general rule out of Flexporter : vertex normals are not indexed.

```
Vertex normals: (24)
0.000000 -1.000000 0.000000
0.000000 -1.000000 0.000000
0.000000 -1.000000 0.000000
0.000000 -1.000000 0.000000

0.000000 1.000000 0.000000
0.000000 1.000000 0.000000
0.000000 1.000000 0.000000
0.000000 1.000000 0.000000

0.000000 0.000000 -1.000000
0.000000 0.000000 -1.000000
0.000000 0.000000 -1.000000
0.000000 0.000000 -1.000000
```

```

1.000000 0.000000 0.000000
1.000000 0.000000 0.000000
1.000000 0.000000 0.000000
1.000000 0.000000 0.000000

0.000000 0.000000 1.000000
0.000000 0.000000 1.000000
0.000000 0.000000 1.000000
0.000000 0.000000 1.000000

-1.000000 0.000000 0.000000
-1.000000 0.000000 0.000000
-1.000000 0.000000 0.000000
-1.000000 0.000000 0.000000

```

If you examine this list carefully, you'll quickly discover the 6 obvious parts, artificially separated by a blank line above. Each of them obviously corresponds to a quad-face of the cube. As you could have guessed for this object, all vertex normals of a same quad-face are the same, aligned with one of the axis of the object's local frame.

Now you might wonder why all of this is better than the native MAX data. After all, we still have a list of (8) vertices, a list of (4) mapping coordinates, and a list of (24) normals. Right. But now we can access them through a single topology :

TOPOLOGY

```

Nb faces:    12
Nb submeshes: 6

```

We get back our 12 original triangles, no surprise. However they've been cut into 6 *submeshes*. A submesh here, is a bunch of triangles referencing hardware-friendly (fat) vertices. Different materials lead to different submeshes. So do different smoothing groups. That's why we have 6 submeshes here, even if all of them use the same material. Without looking at the actual data, you can guess that each submesh corresponds to a quad-face of the cube. 6 quads, 6 submeshes, sounds correct.

In our ASCII file, each submesh then contains the following information :

```

Submesh #0
Material ID:  0
SmGrps:      2
NbFaces:     2
NbVerts:     4
NbSubstrips: 0

```

For our cube, each submesh has 2 faces and 4 vertices, which makes a total of 12 faces and 24 vertices, as expected. The actual indices referencing those 24 vertices are further exported for each submesh, from :

```

Submesh #0 (2 faces)
0 1 2
1 0 3

```

To :

```
Submesh #5 (2 faces)
20 21 22
21 20 23
```

That's it, that's our single topology. The trick is that it doesn't simply reference a list of fat vertices, it actually references our previous lists of *indices*, which *then* map lists of vertices. Warning, there's an extra level of indirection here ! It's usually a bit confusing at first. We did it that way for a very obvious reason that should sound familiar now : *flexibility*. Different people have different goals, and some users might not be interested in the hardware-friendly way of organizing things, with replicated vertices, etc. By using indices all the way, we let the user have the last word, and organize things the way he wants. Indices were the obvious way to go, because it's easy to go from this representation to the one replicating vertices. Going the other way round is much more difficult.

Now, let's get back to our cube. To make things completely clear, let's examine the first face of the first submesh, reported as (0, 1, 2), and see how to use the data.

For points, (0,1,2) maps the three first entries of :

```
Indexed geometry:
0 3 2 1 4 7 5 6 0 5 1 4 1 7 3 5 3 6 2 7 2 4 0 6
```

Which then map entries of :

```
Vertices: (8)
-39.999992 0.000000 -39.999989
39.999992 0.000000 -39.999989
-39.999992 0.000000 39.999989
39.999992 0.000000 39.999989
-39.999992 79.999977 -39.999989
39.999992 79.999977 -39.999989
-39.999992 79.999977 39.999989
39.999992 79.999977 39.999989
```

For mapping coordinates, (0,1,2) maps the first three entries of :

```
Indexed UVWs:
1 2 3 0 0 3 1 2 0 3 1 2 0 3 1 2 0 3 1 2 0 3 1 2
```

Which then map entries of :

```
UVW mappings: (4)
0.000000 0.000000
1.000000 0.000000
0.000000 1.000000
1.000000 1.000000
```

For normals, (0,1,2) directly maps the non-indexed list of normals :

```
Vertex normals: (24)
```

0.000000 -1.000000 0.000000
0.000000 -1.000000 0.000000
0.000000 -1.000000 0.000000
 0.000000 -1.000000 0.000000
 (etc)

In summary, the three fat vertices for our first triangle are :

(x, y, z)	-39.999992	0.000000	-39.999989
(u,v)	1.000000	0.000000	
(Nx, Ny, Nz)	0.000000	-1.000000	0.000000

(x, y, z)	-39.999992	0.000000	39.999989
(u,v)	0.000000	1.000000	
(Nx, Ny, Nz)	0.000000	-1.000000	0.000000

(x, y, z)	39.999992	0.000000	39.999989
(u,v)	1.000000	1.000000	
(Nx, Ny, Nz)	0.000000	-1.000000	0.000000

This is what we should copy in our vertex buffer.

CONCLUSION

That was a lot to write for a simple cube !

Contrary to what popular wisdom says, the cube is one of the most painful object to get right. It touches almost all the problems you'll have when exporting meshes. I got a lot of mails from people getting very frustrated not to be able to handle the "simple cube" with a snap. Well, now you know - *the cube is a bitch*.

Consolidation is here to help you bringing MAX meshes to your engine with less pain. It might look confusing at first - and to some extent it is - but one way or another you'll have to do it as well if you want decent performances out of your engine. So, even if you later recode your own one, it might be a good idea to start from Flexporter's built-in version, just to get things started, and have something to feed your engine with.

But the choice, as always, is yours.

Pierre Terdiman
 April 13 2003, Sunday