

Going from a TriangleSet to the GPU in FreeWRL.

(Background - this short paper was written to help developers migrate CPU-based compute functionality to the GPU, so data and data path details may have changed, but the general scheme will remain unchanged)

A TriangleSet is a simple triangle-based shape, with minimal work required to get it on screen, but it does take us through the same path as with other, more complex, shapes.

Most of the Triangle-based nodes are distilled (compiled) and have equivalent internal representations, and are stored as a “Polyrep”.

1) Rendering phases:

```
src/lib/main/headers.h:#define rendray_TriangleSet render_ray_polyrep
src/lib/main/headers.h:#define collide_TriangleSet collide_genericfaceset
src/lib/main/headers.h:#define make_TriangleSet make_genericfaceset
```

```
src/lib/scenegraph/Component_Rendering.c:void render_TriangleSet (struct
X3D_TriangleSet *node);
```

You'll note that we have render_TriangleSet, but “making”, “colliding”, and “ray tracing” (for mouse clicks) are generic functions, shared amongst other renderable nodes.

2) Rendering on screen:

Lets say we have a simple X3D scene with one renderable node - our TriangleSet.

Through the magic of the FreeWRL scene graph traversal, we'll eventually call the render_TriangleSet node. It consists of very few lines of code:

```
void render_TriangleSet (struct X3D_TriangleSet *node) {
    COMPILER_PLY_IF_REQUIRED(node->coord,
        node->color, node->normal, node->texCoord)
    CULL_FACE(node->solid)
    render_polyrep(node);
}
```

You'll see three elements - “compile Polyrep” if required, cull-face, and then render_polyrep.

The cull_face simply tells the rendering code if the triangles are one-sided or two - so we'll not discuss this element further.

The render_polyrep is a common call amongst almost all visible non-line, non-point geometry, such as IndexedFaceSets, Extrusions, and so on. So, how do we get from a TriangleSet to a Polyrep structure, for common rendering on-screen?

3) Compiling our TriangleSet

The “COMPILE_POLY_IF_REQUIRED” is a key element.

```
#define COMPILE_POLY_IF_REQUIRED(a,b,c,d) \
    if(!node->_intern || node->_change != (node->_intern)- \
    >irep_change) { \
        compileNode ((void *)compile_polyrep, node, a,b,c,d); \
    } \
    if (!node->_intern) return;
```

See the call to “compileNode” there? It expands any PROTOS, then calls

```
nodefn(node, coord, color, normal, texCoord);
```

which really is:

```
compile_polyrep (coord, color, normal, texCoord);
```

We are on our way to taking our TriangleSet, and putting it into a Polyrep structure.

4) Compiling almost all “triangle” nodes - step 1.

Our TriangleSet node is just one node type out of many that throw triangles onto a screen.

Lets go and look at the information required; the first one is a look up table for ALL functions on ALL X3D Node types - the “virtTable”.

ok - so the virtTable, has a few fields; the structure definition is below, and the entry for the field, for nodeType of “NODE_TriangleSet”, is (if you grep through the source code) follows.

```
struct X3D_Virt {
    void (*prep)(void *);
    void (*rend)(void *);
    void (*children)(void *);
    void (*fin)(void *);
    void (*rendray)(void *);
    void (*mkpolyrep)(void *);
    void (*proximity)(void *);
    void (*other)(void *);
    void (*collision)(void *);
    void (*compile)(void *, void *, void *, void *, void *);
};
```

And, our TriangleSet entry in the virtTable is this entry:

```
struct X3D_Virt virt_TriangleSet = {
    NULL,(void *)render_TriangleSet,NULL,NULL,
    (void *)rendray_TriangleSet,(void *)make_TriangleSet,
    NULL,NULL,(void *)collide_TriangleSet,NULL};
```

If you look at the “compile_polyrep” function source code, one of the first things it does is to:

```
struct X3D_Virt *virt;
...
virt = virtTable[node->_nodeType];
```

So, it gets our “virt_TriangleSet” structure, and stores the pointer in the “virt” variable.

compile_polyrep then goes and creates internal data structures, that are common amongst different triangle type nodes.

The last bit of compile_polyrep is:

```
/* make the node by calling the correct method */
virt->mkpolyrep(node);

/* now, put the generic internal structure into OpenGL arrays
   for faster rendering */
/* if there were errors, then rep->ntri should be 0 */
if (polyrep->ntri != 0)
    stream_polyrep(node, coord, color, normal, texCoord);

/* and, tell the rendering process that this shape is now compiled */
polyrep->irep_change = node->_change;
```

Notice the “virt->mkpolyrep(node)” line above. The “mkpolyrep” field of the “X3D_Virt” structure for the “virt_TriangleSet” is “make_TriangleSet”.

So, “make_TriangleSet” is called, then, if there are triangles present, it calls “stream_polyrep”, then it marks the change fields to be equal - the TriangleSet is compiled.

5) “make_TriangleSet”, aka “make_genericfaceset” - Step 2

TriangleSets, along with quite a few other shapes, go through this generic process, as they are just different ways of doing the same thing. The “make_TriangleSet” function does not really exist, but is a #define to “make_genericfaceset”.

This “make_genericfaceset” function sanity checks fields, and grabs fields from each supported node type, for further analysis.

Most nodes have a “coord” field, but the TriangleSet node does not. So, one is created, by a call to:

```
static void checkTriangleSetFields (struct X3D_TriangleSet *node) {
```

which creates a field, “node->_coordIndex”, which helps us later on, as it mimics what an IndexedFaceSet node would have, rendering the same set of Triangles.

After grabbing equivalent fields out of each node type, and setting possible routing in motion via “MARK_EVENT” calls, the resulting Polyrep structure is identical, so whether the original source was from a TriangleSet, IndexedFaceSet, or another equivalent node, the contents of the Polyrep will be identical.

The function

```
void stream_polyrep(void *innode, void *coord, void *color, void *normal,  
struct X3D_TextureCoordinate *texCoordNode) {
```

is then called, and the results are (currently) Vertex Buffer Objects residing on the Graphics Processing Unit.

An example, shown in more detail, follows after the next section.

6) Compiling the Encompassing Shape

Remember the “virtTable” shown above?

```
struct X3D_Virt virt_Shape = { NULL, NULL, (void  
)child_Shape, NULL, NULL, NULL, NULL, NULL, (void *)compile_Shape};
```

So, we have a function called “compile_Shape”, which kind of makes sense, as a visible shape consists of not only the geometry, but the material, and so forth.

If you look at the compile_Shape function, you’ll see that it gathers information about the geometry, whether it has its own shader, whether it is lit or unlit, and so forth.

It creates the rendering shader for this shape. Once this shader is created, we can then go and render the shape’s geometry (compiled into a Polyrep structure), which is still not created yet!

At the end of the compile_Shape, you’ll see a “MARK_NODE_COMPILED” call, which simply sets the “_change” and “_ichange” fields to be identical, making this “compile_Shape” call happen only once.

7) Results of the TriangleSet compile.

Given a simple TriangleSet example:

```
<Shape>  
  <Appearance>  
    <Material/>  
  </Appearance>  
  <TriangleSet solid="FALSE">  
    <Coordinate DEF='COORDINATE'
```

```

        point='-2.1 1.1 1.1, -2.2 -1.2 1.2, 2.3 1.3 1.3,
              2.4 -1.4 1.4, 2.5 1.5 -1.5, 2.6 -1.6 -1.6 '/>
    </TriangleSet>
</Shape>

```

we have three triangles on screen.

Looking at the end of “make_genericfaceset”, at the code:

```
struct X3D_PolyRep *rep_ = node->_intern;
```

and, the structure is:

```

struct X3D_PolyRep {
    int irep_change;
    int ccw;           /* ccw field for single faced structures */
    int ntri;          /* number of triangles */
    int streamed;      /* is this done the streaming pass? */

    /* indices for arrays. OpenGL ES 2.0 - unsigned short for
       the DrawArrays call */
    GLuint *cindex;    /* triples (per triangle) */
    GLuint *colindex;  /* triples (per triangle) */
    GLuint *norindex;
    GLuint *tcindex;   /* triples or null */

    float *actualCoord; /* triples (per point) */
    float *color;        /* triples or null */
    float *normal;       /* triples or null */
    float *GeneratedTexCoords; /* triples (per triangle) of texture coords
                               if there is no texCoord node */
    int tcoordtype;     /* type of texture coord node - is this a
                       NODE_TextureCoordGenerator... */
    int texgentype;     /* if we do have a TextureCoordinateGenerator,
                       what "TCGT_XXX" type is it? */
    GLfloat minVals[3]; /* for collision and default tex coord generation */
    GLfloat maxVals[3]; /* for collision and default text coord generation */
    GLfloat transparency; /* what the transparency value was during compile,
                          put in color array if RGBA colors */
    int isRGBAColorNode; /* color was originally an RGBA, DO NOT re-write
                        if transparency changes */
    GLuint VBO_buffers[VBO_COUNT]; /* VBO indexen */
};

```

printPolyRep at end of make_genericfaceset, items of importance highlighted.

```

irep_change -1292893856
ccw 1
ntri 2
streamed 0

```

```

tcoordtype 0
texgentype 0
isRGBAColorNode 0
tcindex 0xb2550680
      0 1 2 3 4 5
colindex 0xb2551008
      0 0 0 0 0 0
norindex 0xb2551030
      0 1 2 3 4 5
cindex 0xb2550fe0
      0 1 2 3 4 5
actualCoord (nil)
color (nil)
normal 0xb25505a0
      -0.047422 0.045446 0.997841 -0.047422 0.045446 0.997841
GeneratedTexCoords (nil)
minVals 999999.875000 999999.875000 999999.875000
maxVals -999999.875000 -999999.875000 -999999.875000
transparency 0.000000
VBOS Vertex 1, Normal 0, Texture 0, Index 2, Color 0

```

printPolyRep at end of stream_polyrep

```

irep_change -1292893856
ccw 1
ntri 2
streamed 1
tcoordtype 0
texgentype 0
isRGBAColorNode 0
tcindex 0xb254b420
      0 1 2 3 4 5
colindex (nil)
norindex (nil)
cindex 0xb2547420
      0 1 2 3 4 5
actualCoord 0xb254b2b8
      -2.100000 1.100000 1.100000 -2.200000 -1.200000 1.200000
color (nil)
normal (nil)
GeneratedTexCoords 0xb254b3e8
      inf inf -nan -nan inf inf
minVals -2.200000 -1.600000 -1.600000
maxVals 2.600000 1.500000 1.400000
transparency 0.000000

```

VBOS Vertex 1, Normal 3, Texture 4, Index 2, Color 0

Note that, by the end of `stream_polyrep`, we have:

- `boundingBox` defined;
- most indexes sent to VBOS and memory freed;
- VBOs defined.

Looking at the above, there are some errors or omissions that need correcting in the code, so possibly the above may have changed by the time you go through this code.

I hope the above helps!