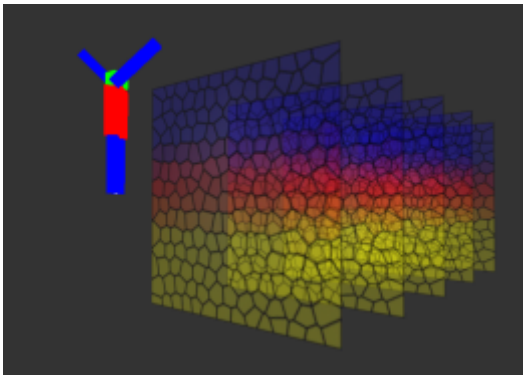
Tutorial 7: Scene Management



Summary

Using a scene graph gets you well on your way to the high-performance rendering of lots of objects. However, we still need to sort out the issue of of transparent objects requiring a specific render order. Also, it is common to skip the drawing of geometry outside of the view - an operation called frustum culling. In this tutorial, you'll see how to perform both frustum culling and geometry ordering, as well as as how to use this ordering to improve rendering performance.

New Concepts

Node Separation, Node Sorting, Frustum Culling

Introduction

Node Ordering

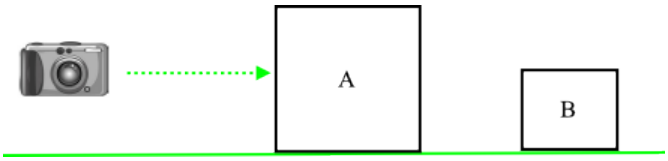
Transparency

In the depth & transparency tutorial, you saw how when it comes to rendering geometry with alpha blending, the order of rendering *matters*. A basic example is that of two objects - one transparent, one opaque, with the transparent object physically in front of the other. If the opaque object is drawn first, alpha blending works correctly, but if the *transparent* object is drawn first, alpha blending fails. Ideally, transparent geometry should *always* be drawn after other geometry, and should be drawn in back to front order. That is, the transparent mesh furthest away from the camera is drawn first, then the next closest, and so on, until finally the closest transparent mesh is drawn. This will allow alpha blending to take place, and hopefully render correct transparent geometry. Even this isn't perfect, as transparent meshes could still intersect, either with other meshes or even themselves - the only way to really render transparent geometry is to do so on a per triangle basis! But for games, splitting transparent objects off and drawing them from back to front generally suffices.

Overdraw

It turns out, that sorting opaque geometry has a benefit, too. Under normal rendering conditions, as mesh geometry is passed through the graphics hardware pipeline, the fragment shader will be ran on each fragment the geometry covers. Only *after* the fragment shader has ran will the depth test be performed, with the result of the fragment shader dropped if the fragment fails the depth test. If an

expensive fragment shader is being used, this can be very wasteful. This is part of a more general problem known as *overdraw* - with randomly ordered input geometry, overlapping objects will cause many fragments to be processed for each final on-screen pixel. Fortunately, both of these problems can be alleviated by sorting the geometry before rendering, just like with transparent objects. However, rather than sorting the objects from back to front (something which would actually create a *worst-case scenario* for overdraw!), the objects are drawn from front to back. For example, take this simple scene:



If object B is draw before object A, the colour and depth writes caused by the rendering of object B will be wasted, as object A entirely obstructs object B. If object A is drawn first, the depth and colour writes of object B will be skipped entirely, saving bandwidth. Rendering front to back will still perform the fragment shader multiple times per pixel, but reduces memory bandwidth - as a well-behaved, ordered scene will more likely fail the depth test more often, resulting in less writes to graphics memory.

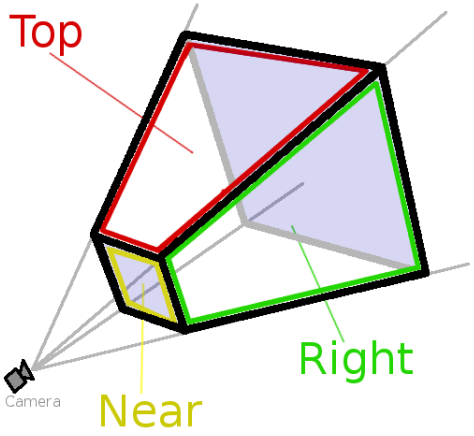
Early Z Test

It's not *quite* true that the fragment shader will always run on fragments that will never be seen due to depth test failure. Modern graphics hardware has an *early z* stage in its pipeline. This is a depth test component that is actually placed *before* the fragment shader in the pipeline - so we can avoid those expensive fragment shader operations after all! As long as certain conditions are met, potential fragments can be dropped before they hit the fragment shader, and sorting objects from front to back massively increases the chances of this early z taking place. The exact conditions are hardware dependent, but generally disabling the stencil buffer and certain alpha capabilities will allow the early-z to take place. Being a purely hardware component, there's no API state to specifically enable or disable this early-z test, it will just automatically happen under the right circumstances. Unfortunately, this also means it's hard to test whether early-z is actually taking place, but as rendering front-to-back has overdraw benefits anyway, there's no harm in trying to utilize the early-z, too.

Frustum Culling

When rendering meshes, each individual triangle is clipped against the screen, so that only those triangles that are actually on screen pass to the fragment shader. However, this means that that vertex and geometry shaders will still run for every triangle of an object, even if it's not on screen at all! Fortunately, there is a solution to this - *Frustum Culling*.

A *frustum* is the 6 sided shape formed that fully encloses everything that can be seen. The shape formed depends on the type of projection applied to the scene - an orthographic projection will produce a cuboid frustum, and a perspective projection will form a pyramid shape, due to the vertical and horizontal field of vision. In either case, the near and far values used in the projection matrix will form the front and back of the shape.



Each of these 6 frustum sides can be represented using a *plane* - an infinitely large boundary that splits space into two *half spaces*, the space behind it, and the space in front of it. If a mesh is entirely behind *any* of the 6 frustum planes, rendering it can be skipped, as it will not be seen on screen, saving costly vertex processing.

Bounding Volumes

In order to efficiently test whether an object is within the viewing frustum or not, it is common to test the *bounding volume* of its geometry. This is a simple shape, such as a cube or a sphere, that completely envelops an object - much as the frustum completely envelops everything the camera can see. So, instead of testing whether every vertex is inside a viewing frustum, a small number of simple calculations can be performed.

An example game character and two example bounding volumes, a sphere and a box

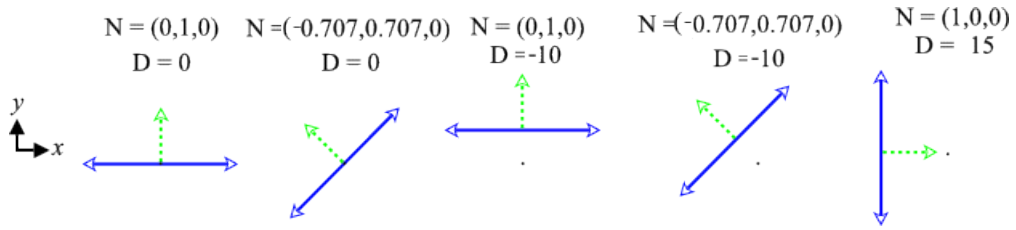
It is unlikely that these shapes will conform exactly to the objects you try to draw in your graphical scenes, but as long as these volumes completely surround an object, they will serve their purpose in frustum culling. Although it possible an object will sometimes be make it to the vertex processing stage when it is not *really* on screen (There's a lot of empty space below the monster's arms in that example above...), performing frustum culling generally results in an overall increase in rendering performance.

Culling Equations

As mentioned earlier, the view frustum can be thought of as 6 planes. The classic equation for a plane is as follows:

$$ax + by + cz + d = 0$$

Where (a, b, c) is the *normal* of the plane, d is the plane's distance from the origin along the normal, and (x, y, z) is a reference vector. We'll cover normals in further detail later in the tutorial series, but for now, if you don't know already, a normal is a *direction vector*, *orthogonal* to a surface. For example, the normal of your desk faces straight up, and for a plane, points away from the plane:



Example planes (blue) and their normals (green), and their relation to the origin (black dot)

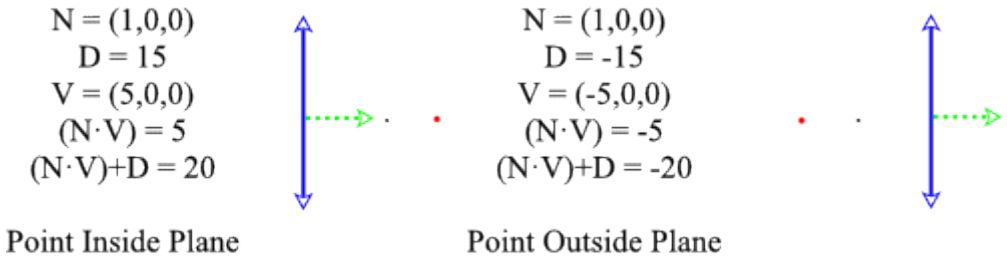
We can rewrite the equation slightly, to the more simple:

$$(N \cdot V) + d = 0$$

Where N is the normal vector, V is the reference vector, and the dot represents the *dot product* of two vectors:

$$\text{dot(Vector3 a, Vector3 b)} = a_x b_x + a_y b_y + a_z b_z$$

which as you can see, is equivelent to the original equation. What this equation means is that a plane is defined as every reference point in space where the above equation holds true. More generally, the equation results in the distance of the reference vector from the plane - if the distance is negative, then the point is *behind* the plane, if it is *positive* it is in front of the plane. This equation can easily be extended out to test whether a sphere is inside the plane, by checking whether distance is less than the negated radius of the sphere.



Examples of red point being inside and outside a plane

So, instead of performing vertex shader operations on potentially huge meshes that may not even contribute to the final visible scene, we can cull a mesh simply by comparing the result of a *dot product* and an *add*, for each plane, and skipping the entire draw process for the mesh if it fails.

Deriving a Frustum

OK, so that's what a frustum is, what culling does, and how it does it, but how to actually generate a viewing frustum? To generate the 6 sides of a viewing frustum, we can derive them from a matrix - the one formed by multiplying the current *projection* matrix by the current *view* matrix. The view matrix is required to set up the position and orientation in Cartesian space, while the projection matrix is required to set the near and far plane positions, and the angles of the others, according to the projection matrices vertical field of view. Once we have this matrix, we can extract the axis for each of the planes from each row, with the distance from the origin being the fourth value in the row

$$\text{Normal axis} \quad \left[\begin{array}{ccc:c} 1 & 0 & 0 & x \\ \hdashline 0 & 1 & 0 & y \\ \hdashline 0 & 0 & 1 & z \\ \hdashline 0 & 0 & 0 & w \end{array}\right] \quad \text{distance}$$

To account for any perspective added by the projection matrix, each axis must be either added to, or subtracted from, the w axis, and the axis distance added to or subtracted from the w distance:

Right Plane *Normal* = w axis - x axis *Distance* = w distance - x distance
Left Plane *Normal* = w axis + x axis *Distance* = w distance + x distance

You can think of this as *tilting* the normal to match the field of view. Finally, before these values can be used, they must be *normalised* - both the direction vector and distance must be divided by the *length* of the direction vector. The length of a vector can be determined as follows:

$$|V| = \sqrt{Vx^2 + Vy^2 + Vz^2}$$

Which you should quickly notice is the same as the square root of the *dot product* of a vector and itself.

$$|V| = \sqrt{(V \cdot V)}$$

Doing so will make the plane normal unit length, and make the distance how far the plane is from the origin, making the values suitable for use in the plane equation earlier.

Example Program

To demonstrate these concepts, we're going to create a variation on last tutorial's program. The *CubeRobot* is still going to stand at the origin, but is going to have a number of transparent panes of glass surrounding him. This relatively simple scene would break using the *Renderer* we've been using so far. Depending on their position in the scene graph, the panes of glass could be drawn either before or after the robot, which would not play nicely with their transparent effect - neither would which way through the panes of glass the camera was looking.

Using what we've learnt in this tutorial, we can solve these problems, by traversing the scene node and splitting the nodes into those which are opaque, and those which are transparent. They can then be sorted according to how far away they are from the view origin. We can then draw the opaque nodes from front to back to take advantage of the *early z test*, and then draw the transparent objects from back to front, producing correct alpha influenced colours. For good measure, we're going to add the capability to frustum cull SceneNodes, using a simple sphere radius check - this isn't perfect for the cubes in our scene, but should demonstrate the frustum culling ability well enough. We're going to need to make two new classes in the nclgl project - A *Frustum* class, and a *Plane* class. We'll keep using the same old Main file from Tutorial 2, though!

SceneNode Class

Our SceneNode class needs a few little changes. We have a couple of new member variables, along with accessor functions. To perform frustum culling we need a something to test against - we're going to keep things simple and have a bounding sphere for each object, represented by a single radius **float**. To sort the *SceneNodes*, we need to store their distance from the camera, again as a **float**. We also need a **static function** that can perform a comparison against this float - you'll see how this works soon.

Header file

```
...
public:
   float    GetBoundingRadius() const     {return boundingRadius;}
   void     SetBoundingRadius(float f)    {boundingRadius = f;}

   float    GetCameraDistance() const     {return distanceFromCamera;}
   void     SetCameraDistance(float f)    {distanceFromCamera = f;}

   static bool CompareByCameraDistance(SceneNode*a,SceneNode*b) {
      return (a->distanceFromCamera <
              b->distanceFromCamera) ? true : false;
   }
...
protected:
...
   float    distanceFromCamera;
   float    boundingRadius;
...
```

SceneNode.h

Class file

Of course, if our *SceneNode* class has new member variables, they must be **initialised** in the **constructor**:

```
SceneNode::SceneNode(Mesh*mesh, Vector4 colour) {
...
   boundingRadius       = 1.0f;
   distanceFromCamera   = 0.0f;
...
}
```

SceneNode.cpp

Plane Class

Header file

Our *Frustum* class is built up from 6 *Planes*. As described earlier, these have 4 values - a 3-component *vector*, and a *distance* value. Our simple *Plane* class has accessors for these values, and to ease the creation of *Planes*, has a **constructor** that sets these values directly. This **constructor** will also perform the normalisation described earlier to produce the correct values. We also require functions to determine which side of a plane a sphere is.

```
#pragma once
#include "vector3.h"

class Plane {
public:
   Plane(void){};
   Plane(const Vector3 &normal, float distance, bool normalise=false);
   ~Plane(void){};

   void     SetNormal(const Vector3 &normal)  {this->normal = normal;}
   Vector3  GetNormal() const                 {return normal;}

   void     SetDistance(float dist)           {distance = dist;}
   float    GetDistance() const               {return distance;}

   bool SphereInPlane(const Vector3 &position, float radius) const;

protected:
   Vector3  normal;
   float    distance;
};
```

Plane.h

Class file

The **constructor** takes three values - a normal, a distance from the origin, and a **bool**, which determines whether or not the other values should be normalised or not. If so, we work out the length of the vector by taking the square root of the dot product of it and itself, and then dividing the normal and distance by the result.

```cpp
#include "Plane.h"

Plane::Plane(const Vector3 &normal, float distance, bool normalise) {
   if(normalise) {
      float length    =sqrt( Vector3::Dot(normal,normal));

      this->normal    = normal      / length;
      this->distance  = distance    / length;
   }
   else{
      this->normal   = normal;
      this->distance = distance;
   }
}
```

Plane.cpp

We covered Sphere / Plane checks earlier, so the code in the *SphereInPlane* function shouldn't be too difficult to understand - it's just the classic plane equation, with an added allowance for the radius of our bounding sphere.

```cpp
bool Plane::SphereInPlane(const Vector3 &position,
                          float radius) const {
   if(Vector3::Dot(position,normal)+distance <= -radius) {
      return false;
   }
   return true;
}
```

Plane.cpp

Frustum Class

Header file

Our Frustum class is not very complex. It contains 6 *Planes*, one for each side of the pyramid shape the view frustum forms. We need a **public** function to generate a view frustum from a matrix, and a **public** function to determine whether one of our *SceneNodes* is within the frustum.

```cpp
#pragma once

#include "Plane.h"
#include "Matrix4.h"
#include "SceneNode.h"
class Matrix4; //Compile the Mat4 class first, please!

class Frustum  {
public:
   Frustum(void)  {};
   ~Frustum(void) {};
   void           FromMatrix(const Matrix4 &mvp);
   bool           InsideFrustum(SceneNode&n);
protected:
   Plane planes[6];
};
```

Frustum.h

Class file

To determine whether a *SceneNode* is inside the view frustum we use the function *InsideFrustum*. In our simple *SceneNode* system, we're going to assume everything is contained within spheres, so we can simply do a sphere / plane check against each of the 6 *Planes* that make up the frustum. If the bounding sphere of the *SceneNode* is outside any of the frustum planes, it cannot be seen, and so should be discarded. Only if it is inside all of the planes should it be drawn.

```
#include "Frustum.h"
bool Frustum::InsideFrustum(SceneNode&n)  {
   for(int p = 0; p < 6; ++p) {
      if(!planes[p].SphereInPlane(n.GetWorldTransform().
          GetPositionVector(),n.GetBoundingRadius())) {
         return false;  //scenenode is outside this plane!
      }
   }
   return true;         //Scenenode is inside every plane...
}
```

Frustum.cpp

That's simple enough, but how to actually determine the frustum planes to test against? As we saw earlier, we can build up these planes using a matrix - one formed by multiplying the current projection and view matrices together. The *FromMatrix* performs the task of generating the corrent planes, using the *Plane* **constructor** to perform the normalisation. We extract each axis from the matarix, and then add or subtract them as necessary to tilt the normals to account for the field of vision of the projection matrix. The *FromMatrix* function assumes that the mat parameter is already the correct projection · view matrix.

```
void Frustum::FromMatrix(const Matrix4 &mat) {
   Vector3 xaxis =Vector3(mat.values[0],mat.values[4],mat.values[8]);
   Vector3 yaxis =Vector3(mat.values[1],mat.values[5],mat.values[9]);
   Vector3 zaxis =Vector3(mat.values[2],mat.values[6],mat.values[10]);
   Vector3 waxis =Vector3(mat.values[3],mat.values[7],mat.values[11]);

   //RIGHT
   planes[0] = Plane(waxis - xaxis,
                     (mat.values[15] - mat.values[12]), true);
   //LEFT
   planes[1] = Plane(waxis + xaxis,
                     (mat.values[15] + mat.values[12]), true);
   //BOTTOM
   planes[2] = Plane(waxis + yaxis,
                     (mat.values[15] + mat.values[13]), true);
   //TOP
   planes[3] = Plane(waxis - yaxis,
                     (mat.values[15] - mat.values[13]), true);
   //FAR
   planes[4] = Plane(waxis - zaxis,
                     (mat.values[15] - mat.values[14]), true);
   //NEAR
   planes[5] = Plane(waxis + zaxis,
                     (mat.values[15] + mat.values[14]), true);
}
```

Frustum.cpp

CubeRobot Class

As we now have a bounding sphere radius for each node in our scene graph, we must set these values as appropriate for our *CubeRobot's* various limbs. This is performed simply, by using the new *SceneNode* function *SetBoundingRadius* in the *CubeRobot* **constructor**:

```
1  CubeRobot::CubeRobot(void) {
2  ...
3     body->SetBoundingRadius(15.0f);
4     head->SetBoundingRadius(5.0f);
5
6     leftArm->SetBoundingRadius(18.0f);
7     rightArm->SetBoundingRadius(18.0f);
8
9     leftLeg->SetBoundingRadius(18.0f);
10    rightLeg->SetBoundingRadius(18.0f);
11 ...
12 }
```

CubeRobot.cpp

Renderer Class

In order to accommodate frustum culling and separate node lists, we need to change our *Renderer* class a bit. We need three new **protected** member variables - a variable to keep the current view frustum, and two std::vectors, to store the opaque and transparent scene nodes. Along with the *DrawNode* function we created last tutorial, we need 4 new functions, all of them related to the lists of *SceneNodes*. Every frame, we're going to perform the following cycle of operations:

1) Update the scene graph via the root node's *Update* function.

2) Cull nodes outside the view frustum and put those inside into their correct vector, using the function *BuildNodeLists*.

3) Sort the two node vectors by their distance from the camera, in the function *SortNodeLists*.

4) Draw the opaque nodes, followed by the transparent nodes, via the *DrawNodes* function.

5) Clear the frame's vectors, using *ClearNodeLists*.

Sounds like a big change, but each of those functions is quite small, as you'll see shortly.

Header file

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4  #include "./nclgl/Camera.h"
5  #include "./nclgl/SceneNode.h"
6  #include "./nclgl/Frustum.h"
7  #include "CubeRobot.h"
8  #include <algorithm>            //For std::sort...
9
10 class Renderer : public OGLRenderer {
11 public:
12    Renderer(Window &parent);
```

```
13    virtual ~Renderer(void);
14
15    virtual void UpdateScene(float msec);
16    virtual void RenderScene();
17
18 protected:
19    void        BuildNodeLists(SceneNode* from);
20    void        SortNodeLists();
21    void        ClearNodeLists();
22    void        DrawNodes();
23    void        DrawNode(SceneNode*n);
24
25    SceneNode*  root;
26    Camera*     camera;
27    Mesh*       quad;
28
29    Frustum     frameFrustum;
30
31    vector<SceneNode*> transparentNodeList;
32    vector<SceneNode*> nodeList;
33 };
```

Renderer.h

Class file

As usual, our *Renderer* class begins with its **constructor**. We keep things the same as last tutorial up until line 14, where we initialise the quad variable, and set its texture to be the stained glass texture we used in the depth and transparency tutorial.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     CubeRobot::CreateCube();
5     projMatrix     = Matrix4::Perspective(1.0f,10000.0f,
6                      (float)width/(float)height,45.0f);
7
8     camera         = new Camera();
9     camera->SetPosition(Vector3(0,100,750.0f));
10
11    currentShader  = new Shader(SHADERDIR"SceneVertex.glsl",
12                                SHADERDIR"SceneFragment.glsl");
13
14    quad           = Mesh::GenerateQuad();
15    quad->SetTexture(SOIL_load_OGL_texture(
16 TEXTUREDIR"stainedglass.tga",SOIL_LOAD_AUTO,SOIL_CREATE_NEW_ID, 0));
17
18    if(!currentShader->LinkProgram() || !quad->GetTexture()) {
19       return;
20    }
```

Renderer.cpp

Now for the scene generation. We initialise a new root *SceneNode*, and add a number of quads to it as children - these will represent large panes of glass. As these nodes have an alpha value less than 1.0, they'll be placed in the transparent node list. Then, we add a *CubeRobot* to the root node. In the old rendering method, this wouldn't work, as the robot would be drawn *after* the transparent quads, and result in incorrect alpha blending.

```
21    root = new SceneNode();
22
23    for(int i = 0; i < 5; ++i) {
24       SceneNode * s = new SceneNode();
25       s->SetColour(Vector4(1.0f,1.0f,1.0f,0.5f));
26       s->SetTransform(Matrix4::Translation(
27                       Vector3(0,100.0f,-300.0f + 100.0f + 100 * i)));
28       s->SetModelScale(Vector3(100.0f, 100.0f, 100.0f));
29       s->SetBoundingRadius(100.0f);
30       s->SetMesh(quad);
31       root->AddChild(s);
32    }
33
34    root->AddChild(new CubeRobot());
```

Renderer.cpp

To finish the **constructor**, we enable depth testing and alpha blending, and set *init* to **true**.

```
35    glEnable(GL_DEPTH_TEST);
36    glEnable(GL_BLEND);
37    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
38    init = true;
39 }
```

Renderer.cpp

In the **destructor**, we **delete** the scene graph, along with the quad and cube *Meshes*.

```
40 Renderer::~Renderer(void)  {
41    delete root;
42    delete quad;
43    delete camera;
44    CubeRobot::DeleteCube();
45 }
```

Renderer.cpp

In *RenderScene*, we build up the current frame's view frustum, and, like last tutorial, update the scene graph via a call to the *Update* function of *root*.

```
46 void Renderer::UpdateScene(float msec) {
47    camera->UpdateCamera(msec);
48    viewMatrix = camera->BuildViewMatrix();
49    frameFrustum.FromMatrix(projMatrix*viewMatrix);
50
51    root->Update(msec);
52 }
```

Renderer.cpp

For every frame's rendering, we build up two lists of *SceneNodes* - those that are opaque, and those that are transparent. We do this via the recursive function *BuildNodeLists*, starting from the scene graph root. For each node, we first check whether it is inside the frustum, and if so we calculate the node's distance from the camera, and place the node in the correct vector. Then, each of the node's children call *BuildNodeLists* in turn, building up two complete vectors, which between them, contain every visible *SceneNode*.

There are two things to note here. First off, we're using the distance *squared*. When used to sort the scene nodes, this will preserve their *relative* ordering, but avoids having to perform a square root for every object in the graph to determine their exact distance. Second, we're cheating a *little bit* in regards to what counts as a *transparent* node. For the purposes of this tutorial, we're going to test against the *SceneNode colour* variable. Technically, the *Mesh* of a *SceneNode* could have an alpha-mapped texture applied to it, but correctly rendering such nodes is left to further work!

```
53 void  Renderer::BuildNodeLists(SceneNode* from) {
54    if(frameFrustum.InsideFrustum(*from)) {
55       Vector3 dir = from->GetWorldTransform().GetPositionVector() -
56                           camera->GetPosition();
57       from->SetCameraDistance(Vector3::Dot(dir,dir));
58
59       if(from->GetColour().w < 1.0f) {
60          transparentNodeList.push_back(from);
61       }
62       else{
63          nodeList.push_back(from);
64       }
65    }
66
67    for(vector<SceneNode*>::const_iterator i =
68     from->GetChildIteratorStart();
69     i != from->GetChildIteratorEnd(); ++i) {
70       BuildNodeLists((*i));
71    }
72 }
```

Renderer.cpp

Once *BuildNodeLists* has iterated through every node in the scene graph, we can *sort* them. To do so we use **std::sort**. This takes two iterator objects, and a *function pointer*. It then iterates through all the objects between the start and end iterator object, sorting them by the result of the function pointer. The function we're going to use is the *CompareByCameraDistance* function of the *SceneNode* class. This will sort the *SceneNodes* so that the first node in the vector is the closest node to the camera, and the last the farthest away node. We do this for both opaque and transparent nodes.

```
73 void  Renderer::SortNodeLists()  {
74    std::sort(transparentNodeList.begin(),
75              transparentNodeList.end(),
76              SceneNode::CompareByCameraDistance);
77    std::sort(nodeList.begin(),
78              nodeList.end(),
79              SceneNode::CompareByCameraDistance);
80 }
```

Renderer.cpp

To draw our scene graph, we use the *DrawNodes* function. it's simply two **for** loops - we draw the opaque nodes in the *nodeList* first, and then the *transparentNodeList*. Note how the transparent nodes are draw in **reverse** order, simply by using the stl **reverse iterator** functionality.

```
82 void  Renderer::DrawNodes()    {
83    for(vector<SceneNode*>::const_iterator i = nodeList.begin();
84     i != nodeList.end(); ++i ) {
85       DrawNode((*i));
86    }
```

```
87      for(vector<SceneNode*>::const_reverse_iterator i =
88       transparentNodeList.rbegin();
89       i != transparentNodeList.rend(); ++i ) {
90         DrawNode((*i));
91      }
92  }
```

Renderer.cpp

We need to modify our *DrawNode* function slightly. In the last tutorial, we could draw the entire scenegraph using one *DrawNode* call, as the function was recursive, and went through the entire scene graph. As we have lists of nodes this time around, we don't need to do this, so we remove the **for** loop.

```
93  void  Renderer::DrawNode(SceneNode*n)  {
94      if(n->GetMesh()) {
95        glUniformMatrix4fv(   glGetUniformLocation(
96        currentShader->GetProgram(), "modelMatrix"),  1,false,
97        (float*)&(n->GetWorldTransform()*Matrix4::Scale(n->GetModelScale())));
98
99         glUniform4fv(glGetUniformLocation(currentShader->GetProgram(),
100        "nodeColour"),1,(float*)&n->GetColour());
101
102        glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
103        "useTexture"),(int)n->GetMesh()->GetTexture());
104
105        n->Draw(*this);
106     }
107 }
```

Renderer.cpp

Finally, we have *RenderScene*. Instead of calling *DrawNode* on the root, as in the last tutorial, this time we're going to call *DrawNodes*, drawing our two lists of nodes. Also, now we're done with the contents of the two *SceneNode* vectors, we must empty them - otherwise they'd get filled with duplicate nodes every frame!

```
108 void Renderer::RenderScene()  {
109     BuildNodeLists(root);
110     SortNodeLists();
111
112     glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
113
114     glUseProgram(currentShader->GetProgram());
115     UpdateShaderMatrices();
116
117     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
118                 "diffuseTex"), 0);
119     DrawNodes();
120
121     glUseProgram(0);
122     SwapBuffers();
123     ClearNodeLists();
124 }
125 void  Renderer::ClearNodeLists() {
126     transparentNodeList.clear();
127     nodeList.clear();
128 }
```

Renderer.cpp

Tutorial Summary

Once the program has been compiled and ran, you should see the CubeRobot again, this time through some stained glass windows. A fairly abstract scene, but one which introduces some important concepts. You've solidified your knowledge of scene node traversal, this time to split nodes into those that are opaque, and those that are transparent. You've learnt how to take advantage of the *early z pass* by drawing opaque nodes from front to back, and how to *mostly* solve the issue of transparent objects by drawing such nodes in back to front order, *after* other nodes. Finally, you've seen how to perform simple frustum culling, using spheres. In the next two tutorials, you'll see how to use the new concepts of indices and transformation hierarchies to implement skeletal animation - a modern graphical technique for generating high-quality animations.

Further Work

1) As you've probably realised, plane / sphere checking, although very fast is not ideal for every type of shape - there's a lot of wasted space in the bounding sphere for the stained glass windows, for example. Investigate bounding boxes, specifically Object Oriented Bounding Boxes, and Axis Aligned Bounding Boxes. Perhaps there should be an abstract *BoundingVolume* class, from which a *BoundingSphere* class derives from...

2) A popular way of increasing rendering performance is to make a node's bounding area large enough to envelope all of its children, allowing all child nodes of a node outside of a frustum to be quickly culled. How would you implement this? Scene graph traversals can go upwards from a *leaf* to the *root*...

3) The SceneNode class currently relies on its alpha value in order to be correctly sorted by transparency. What about *SceneNodes* with alpha-mapped *Mesh* textures? How to correctly sort such *Meshes*? Investigate the OpenGL function **glGetTexLevelParameter**, and how it could be used to determine whether a texture has an alpha component...