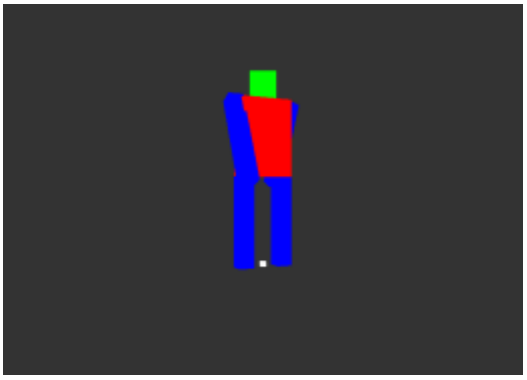
Tutorial 6: Scene Graphs



Summary

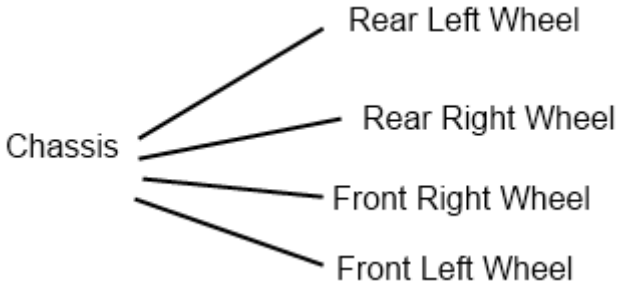
So far you have only been drawing one or two objects in your scenes. What about drawing hundreds of objects - how can you make your graphical renderer scalable? This tutorial will show you how to implement a *Scene Graph*, a way of keeping track of lots of objects, including their transformations.

New Concepts

Scene Graphs, Scene Nodes, local vs. world transformations

Scene Graphs

If you've experimented with rendering multiple objects in your graphical applications, you've probably used something simple like an **array**, or maybe a **vector**, to hold a number of meshes to draw. Maybe you've even grouped a mesh and a transformation together in a **struct**. This will work for simple scenes, but as your scenes get more complicated, this 'linear' method of renderable storage becomes less useful. The classic example for this is a car with a separate mesh for the chassis and wheels - how do you keep track of where the wheels are in relation to the chassis? What if the car also has a driver mesh? Once you start adding in lots of objects, keeping track of the relative positions of everything starts becoming difficult! This is where a *scene graph* comes in handy. Scene graphs consist of a number of *scene nodes*, kept together in a tree-like structure - each node has a *parent* node, and a number of *child* nodes. So, our car example could be represented by the following scene graph:



Each node in the scene graph contains information relating to its graphical representation, so the *chassis* scene node could have a pointer to a car body mesh, while the four *wheel* nodes contain pointers to a single wheel mesh - there's no need to load the same mesh in multiple times! But how does this solve how to keep track of where the wheels are in relation to the chassis? Simple: scene graphs can also store the *spatial* relationship between a parent scene node and its children.

Local Transformations

Each node in a scene graph commonly used in games contains data representing its local transform - its position and orientation in relation to its *parent node*. Generally this is simply a transformation matrix, just like the model matrices you've been using to place objects in your scenes, and so can contain translation, rotation, and scaling information. Instead of translating a mesh from local space to world space, a scene node's transformation matrix transforms its position locally in relation to its parent, instead. This means that all transformation information cascades down the scene graph - including scaling transformations. This means that all scene nodes that are children of a given node with a scale will have their transformation matrices scaled - whether this is desirable or not is context dependent - being able to size up our car to be a monster truck by setting the chassis scale to 5.0 will automatically increase the size of its wheels, due to the parent/child relationship of their transformations - but as you will see in the example, sometimes having a scale in the matrix can cause undesirable artifacts.

Graph Traversal

As well as the *local* transformation of each node in a graph, if we want to render the graph on screen we'll need each node's *world* transformation, to use as the model matrix in a vertex shader. In order to do this, we must *traverse* the graph. Starting from the 'top' or *root* of the scene graph, the world transformation for each node can be calculated by multiplying a node's local transformation matrix with the world transformation of its parent. Any child nodes of that node can then calculate their world position, and so on, down the graph, until *leaf* nodes are reached - that is, a node without any children. This operation is commonly done using either an iterative loop, or a *recursive* function, calling itself on child nodes until it hits a leaf.

Generally, it is better to store both the 'local' and 'world' transformations of an object - by descending the graph from the root, calculating world matrices as we go, we can then always determine the world matrix of an object with a single multiplication - the child's local transformation with the world transformation of its parent. If we *didn't* store the world matrix, we'd have to traverse the tree every time we needed the world matrix of a node - in a deep graph that could be a **lot** of unnecessary multiplications!

Transition and State Nodes

A scene node doesn't necessarily have to contain graphical information, such as a mesh. They may be purely *transitional*, that is, nodes that group together and translate a number of children, but don't render anything themselves. For example, perhaps the axles of our car are a transition node, with the wheels as children. Then, we only have to rotate the axle to rotate both wheels. Or perhaps the node contains only state information, such as a group of subnodes that are rendered using a specific shader, or some other specific rendering option set.

World Scene Graph

It's not just a 'decomposable' object like our car example that can be represented as a scene graph - everything in a game world can be part of one large scene graph. It's common for a game's graphical renderer to have just a single root scene node, containing everything in the current level as children. So maybe our car is on a car transporter (so the car nodes are a child of the car transporter node), which is on a bridge (so is a child thereof), which is itself a child of the overall game 'world' node.

Example Program

To demonstrate scene graphs, we're going to create a simple *SceneNode* class. This class will allow us to do two popular types of node traversal - efficiently generating the correct world transforms for each node in a scene graph, and drawing a whole tree of nodes. To show this, we're going to make a simple robot made of cubes, whose limbs and head are child nodes of its body node, and can be transformed independently. Also, to demonstrate how each node in a scene graph could have shader variables attached to it, we're going to write two new shaders - *SceneVertex* and *SceneFragment*, that will set a vertices colour by a *colour* variable of the *SceneNode* class.

SceneNode Class

Header file

Our *SceneNode* class is quite simple. To create the tree structure of our scene graph, each *SceneNode* has a *parent*, and a **vector** of *children*. Each *SceneNode* has a local transform, a world transform, a colour, a model scale, and pointer to a *Mesh*. It also has **public** accessors for each member variable - note how the accessors for a node's children are **const iterators**, allowing other classes to safely iterate over a node's children. Other than that, there's a pair of **public virtual** functions. *Update* will traverse through a scene graph, building up world transforms and updating member variables in a framerate-independent way, just like the *Camera* class we made a while back, while Draw will actually draw the *SceneNode*, and which takes a **const** reference to the *Renderer* that is currently drawing.

We have a seperate scale to change the mesh size so we can scale the cubes that make up the robots limbs *without* the scaling information affecting the transformation matrices of child nodes - but note, any scaling information actually in the transformation matrix, such as a scale in a a parent's world transform, will still effect the resulting mesh scale.

```
1  #pragma  once
2  #include "Matrix4.h"
3  #include "Vector3.h"
4  #include "Vector4.h"
5  #include "Mesh.h"
6  #include <vector>
7
8  class SceneNode   {
9  public:
10    SceneNode(Mesh*m = NULL, Vector4 colour = Vector4(1,1,1,1));
11    ~SceneNode(void);
12
13    void  SetTransform(const Matrix4 &matrix) { transform = matrix;}
14    const Matrix4& GetTransform()      const  { return transform;}
15    Matrix4        GetWorldTransform() const  { return worldTransform;}
16
17    Vector4        GetColour()         const  {return colour;}
18    void           SetColour(Vector4 c)       {colour = c;}
19
20    Vector3        GetModelScale()           const  {return modelScale;}
21    void           SetModelScale(Vector3 s)         {modelScale = s;}
22
23    Mesh*          GetMesh()           const  {return mesh;}
24    void           SetMesh(Mesh*m)            {mesh = m;}
25
26    void           AddChild(SceneNode* s);
27
28    virtual void   Update(float msec);
29    virtual void   Draw(const OGLRenderer &r);
30
```

```
31
32    std::vector<SceneNode*>::const_iterator GetChildIteratorStart()   {
33                                        return children.begin();}
34
35    std::vector<SceneNode*>::const_iterator GetChildIteratorEnd()     {
36                                        return children.end();}
37 protected:
38    SceneNode*  parent;
39    Mesh*       mesh;
40    Matrix4     worldTransform;
41    Matrix4     transform;
42    Vector3     modelScale;
43    Vector4     colour;
44    std::vector<SceneNode*>    children;
45 };
```

SceneNode.h

Class file

The **constructor** initialises its variables, while the **destructor deletes** all of a node's children. The *AddChild* function adds a *SceneNode* to its *children* vector, and also sets the new child nodes *parent* to itself - neatly keeping the simple tree structure of our scene graph intact. Note that a *SceneNode* does not **delete** its *mesh* variable - we could have lots of nodes all pointing to the same *Mesh* instance. This does mean that we have to handle any *Mesh* **deletion** elsewhere, though.

```
1  #include "SceneNode.h"
2
3  SceneNode::SceneNode(Mesh* mesh, Vector4 colour)   {
4     this->mesh            = mesh;
5     this->colour          = colour;
6     parent                = NULL;
7     modelScale                  = Vector3(1,1,1);
8  }
9
10 SceneNode::~SceneNode(void)   {
11    for(unsigned int i = 0; i < children.size(); ++i) {
12       delete children[i];
13    }
14 }
15
16 void SceneNode::AddChild( SceneNode* s )  {
17    children.push_back(s);
18    s->parent = this;
19 }
```

SceneNode.cpp

Next, we have *Draw*, which unsurprisingly, draws the current SceneNode. We're actually going to let the Renderer handle most of the 'generic' drawing setup (setting the correct model matrix and binding shader variables), so all our default Draw function has to do is call the Mesh class Draw function, if there's a mesh to draw. If a particular SceneNode has to perform more advanced rendering setup (disabling depth testing, or reuploading data) it can be done here. This function has a reference to the renderer that is currently drawing, in case there is anything the SceneNode needs to access from the class.

```
20 void  SceneNode::Draw(const OGLRenderer &r)   {
21    if(mesh) { mesh->Draw()};
22 }
```

SceneNode.cpp

Finally in our simple *SceneNode* class, we have *Update*. This will generate the correct world space transformation for a node and its children by simply multiplying a node's local transform matrix by its parent's world matrix. As the graph traversal takes place in a top-down manner, the parent node of any node performing the *Update* function will have already calculated a correct world space transformation. We'll see shortly how subclasses of the *SceneNode* class may do something more interesting with this *Update* function - remember, it is **virtual**.

```
23 void  SceneNode::Update(float msec)  {
24    if(parent) { //This node has a parent...
25       worldTransform = parent->worldTransform * transform;
26    }
27    else{ //Root node, world transform is local transform!
28       worldTransform = transform;
29    }
30    for(vector<SceneNode*>::iterator i = children.begin();
31                                     i != children.end(); ++i) {
32       (*i)->Update(msec);
33    }
34 }
```

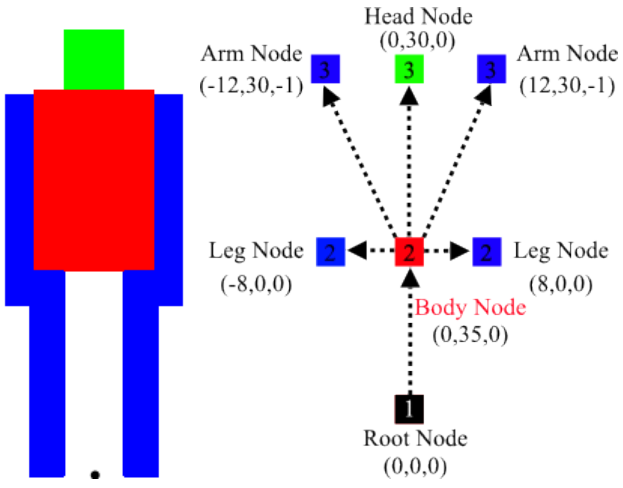
SceneNode.cpp

CubeRobot Class

That's everything we need for a simple *SceneNode* class, but to really show off how flexible the scene graph method of object management is, we're going to extend it, too. This subclass will automatically build a simple robot out of cubes, creating a small *SceneNode* hierarchy.

CubeRobot Scene Nodes

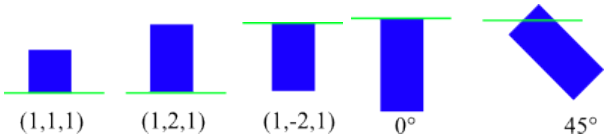
The *CubeRobot* has a *transitional* scene node as its root node. This is to make it easier to place the robot in the world; if we place an instance of the *CubeRobot* at the origin, we really want it to be centered, standing *on* the origin, so the origin is between its legs. This transition node has one child, the robot's body. This body node has five children, the arms, the legs, and the head.



Left: What the CubeRobot should look like and Right: The node hierarchy that makes the CubeRobot, including translations relative to parent node

CubeRobot Mesh

The *CubeRobot* is made entirely out of cubes - ones that have been scaled by varying amounts to produce the various limb shapes. This works as the cube mesh we're going to use is not centered on the origin, but rather 'sits' on it:



Left: Cube scaling and rotation

This allows the cube to effectively be stretched in one direction, and means the mesh's origin works as a pivot point, handy for the rotation of robot limbs!

Header file

There's not too much new for our *CubeRobot* class. We overload the *Update* function, which will handle some simple animation of our cube robot. Also, no matter how many *CubeRobots* we have, we only need one cube *Mesh*, so we're going to keep it as a **static** variable of our class, along with a pair of functions to explicitly create and delete this mesh. Unlike with the triangle and quad we've used in the past, we're going to load the cube up as mesh geometry, from a file format known as Wavefront OBJ. Luckily for you, nclgl comes with a basic OBJ file loader, and a OBJ mesh cube, in the **../Meshes/** folder. Finally, we have some pointers to the *SceneNodes* that make up our robot's limbs, so we can easily animate them.

```
1  #pragma once
2  #include ".\nclgl\scenenode.h"
3  #include ".\nclgl\OBJMesh.h"
4
5  class CubeRobot : public SceneNode  {
6  public:
7     CubeRobot(void);
8     ~CubeRobot(void){};
9     virtual void   Update(float msec);
10
11    static void CreateCube() {
12       OBJMesh*m = new OBJMesh();
13       m->LoadOBJMesh("../Meshes/cube.obj");
14       cube = m;
15    }
16    static void DeleteCube(){  delete cube;}
17
18 protected:
19    static Mesh*   cube;
20    SceneNode*     head;
21    SceneNode*     leftArm;
22    SceneNode*     rightArm;
23 };
```

CubeRobot.h

Class file

The *CubeRobot* **constructor** builds up the robot out of cube *Meshes*, and child *SceneNodes*, using the *SceneNode scale* variable to set the size and shape of the cube, and the local transform to set the position offset of the limbs - you can think of these positions as the joints by which the limbs will be rotated. For good measure, we set colours for each limb, too, in the *SceneNode* **constructors**.

```
1  #include "CubeRobot.h"
2
3  Mesh* CubeRobot::cube = NULL;
4
5  CubeRobot::CubeRobot(void) {
6     //Optional, uncomment if you want a local origin marker!
7     //SetMesh(cube);
8
9     SceneNode*body = new SceneNode(cube,Vector4(1,0,0,1));    //Red!
10    body->SetModelScale(Vector3(10,15,5));
11    body->SetTransform(Matrix4::Translation(Vector3(0,35,0)));
12    AddChild(body);
13
14    head = new SceneNode(cube,Vector4(0,1,0,1));              //Green!
15    head->SetModelScale(Vector3(5,5,5));
16    head->SetTransform(Matrix4::Translation(Vector3(0,30,0)));
17    body->AddChild(head);
18
19    leftArm = new SceneNode(cube,Vector4(0,0,1,1));           //Blue!
20    leftArm->SetModelScale(Vector3(3,-18,3));
21    leftArm->SetTransform(Matrix4::Translation(Vector3(-12,30,-1)));
22    body->AddChild(leftArm);
23
24    rightArm = new SceneNode(cube,Vector4(0,0,1,1));          //Blue!
25    rightArm->SetModelScale(Vector3(3,-18,3));
26    rightArm->SetTransform(Matrix4::Translation(Vector3(12,30,-1)));
27    body->AddChild(rightArm);
28
29    SceneNode* leftLeg = new SceneNode(cube,Vector4(0,0,1,1));  //Blue!
30    leftLeg->SetModelScale(Vector3(3,-17.5,3));
31    leftLeg->SetTransform(Matrix4::Translation(Vector3(-8,0,0)));
32    body->AddChild(leftLeg);
33
34    SceneNode* rightLeg = new SceneNode(cube,Vector4(0,0,1,1)); //Blue!
35    rightLeg->SetModelScale(Vector3(3,-17.5,3));
36    rightLeg->SetTransform(Matrix4::Translation(Vector3(8,0,0)));
37    body->AddChild(rightLeg);
38 }
```

CubeRobot.cpp

Our **overloaded** *Update* function is going to perform some crude animation on our cuboid robot! To start off with, on line 40, we rotate our local transform around the y-axis, so our robot will slowly spin around on the spot. He's also going to rotate his head around in circles, too - why not, he's a robot! Finally, on line 46 and 49, he's going to windmill his arms around in opposite directions on the x-axis. You should be able to see how useful scene graph hierarchies are now; we don't need to care about what the root node's orientation is in order to rotate any of its child nodes, the world transform traversal will handle it all for us automatically. The final thing the function needs to do is call the parent class *Update* function - a handy feature of **virtual** functions. This means that no matter what our **overloaded** *Update* functions do, we can always call the original function code and keep our nodes up to date.

```
39 void CubeRobot::Update(float msec) {
40    transform = transform *
41                Matrix4::Rotation(msec / 10.0f,Vector3(0,1,0));
42
43    head->SetTransform(head->GetTransform() *
44       Matrix4::Rotation(-msec / 10.0f,Vector3(0,1,0)));
```

```
45
46    leftArm->SetTransform(leftArm->GetTransform() *
47       Matrix4::Rotation(-msec / 10.0f,Vector3(1,0,0)));
48
49    rightArm->SetTransform(rightArm->GetTransform() *
50       Matrix4::Rotation(msec / 10.0f,Vector3(1,0,0)));
51
52    SceneNode::Update(msec);
53 }
```

CubeRobot.cpp

Renderer Class

Header file

Finally, we need a *Renderer* class that can handle *SceneNodes*. Our Renderer has a single **pointer** to a *SceneNode*, and a single new function - *DrawNode*, that takes in a *SceneNode*. That's all we need, as we're going to traverse the scene graph, using *DrawNode* on all child nodes of the *root* variable.

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4  #include "./nclgl/Camera.h"
5  #include "./ncl/SceneNode.h"
6  #include "CubeRobot.h"
7
8  class Renderer : public OGLRenderer {
9  public:
10    Renderer(Window &parent);
11    virtual ~Renderer(void);
12
13    virtual void UpdateScene(float msec);
14    virtual void RenderScene();
15
16 protected:
17    void  DrawNode(SceneNode*n);
18
19    SceneNode*  root;
20    Camera*     camera;
21 };
```

Renderer.h

Class file

New in our constructor is the initialisation of the *root* member variable on line 18, and then adding a *CubeRobot* as a child on line 19. Note we also call the **static** *CubeRobot* function *CreateCube* on line 4 - this **must** be called before adding any *CubeRobots*! In our **destructor**, we delete the root node, which will in turn **delete** the *CubeRobot* child node, and call the *DeleteCube* **static** function.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     CubeRobot::CreateCube();          //Important!
5     camera          = new Camera();
6
```

```
7     currentShader  = new Shader("../Shaders/SceneVertex.glsl",
8                                 "../Shaders/SceneFragment.glsl");
9
10    if(!currentShader->LinkProgram()) {
11       return;
12    }
13
14    projMatrix = Matrix4::Perspective(1.0f,10000.0f,
15                          (float)width/(float)height,45.0f);
16
17    camera->SetPosition(Vector3(0,30,175));
18
19    root = new SceneNode();
20    root->AddChild(new CubeRobot());
21
22    glEnable(GL_DEPTH_TEST);
23    init = true;
24 }
25 Renderer::~Renderer(void)  {
26    delete root;
27    CubeRobot::DeleteCube();      //Also important!
28 }
```

Renderer.cpp

UpdateScene has a single new function call in it - we call *Update* on the *root* node, which will update the world transforms for our entire scene graph. In *RenderScene*, we set the rendering up like normal, but this time, instead of doing any rendering directly, we call the *DrawNode* function, passing the *Renderer's root SceneNode* as a parameter.

```
29 void Renderer::UpdateScene(float msec) {
30    camera->UpdateCamera(msec);
31    viewMatrix = camera->BuildViewMatrix();
32    root->Update(msec);
33 }
34
35 void Renderer::RenderScene()  {
36    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
37
38    glUseProgram(currentShader->GetProgram());
39    UpdateShaderMatrices();
40
41    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
42                "diffuseTex"), 1);
43
44    DrawNode(root);
45
46    glUseProgram(0);
47    SwapBuffers();
48 }
```

Renderer.cpp

The last function we need to define is *DrawNode*. If the *SceneNode* passed as a parameter to this function has a non-NULL Mesh instance, we update the *currentShader modelMatrix* with the *worldTransform* of the *SceneNode*, multiplied by its *scale* variable. Then, we send the *SceneNode's colour* to the shader's *nodeColour* **uniform** variable, and *Draw* the mesh.

Whether the node has a *Mesh* or not, we then iterate over a node's children, drawing each of those, too. That's how our *Renderer* class can render an entire scene graph's worth of objects - calling *DrawNode* for the *root* node will recursively call it for all child nodes, too.

```
49 void  Renderer::DrawNode(SceneNode*n)  {
50    if(n->GetMesh()) {
51       Matrix4 transform = n->GetWorldTransform()*
52                           Matrix4::Scale(n->GetModelScale());
53       glUniformMatrix4fv(
54       glGetUniformLocation(currentShader->GetProgram(),
55       "modelMatrix"),   1,false, (float*)&transform);
56
57       glUniform4fv(glGetUniformLocation(currentShader->GetProgram(),
58                    "nodeColour"),1,(float*)&n->GetColour());
59
60       glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
61                   "useTexture"),(int)n->GetMesh()->GetTexture());
62       n->Draw();
63    }
64
65    for(vector<SceneNode*>::const_iterator
66       i = n->GetChildIteratorStart();
67       i != n->GetChildIteratorEnd(); ++i) {
68       DrawNode(*i);
69    }
70 }
```

Renderer.cpp

Vertex Shader

We have a new vertex shader for this tutorial. It is similar to the previous vertex shaders, only this time, instead of settting the vertex colour from the incoming VBO attribute data, we use the *nodeColour* **uniform** variable, set in the *Renderer* class *DrawNode* function.

```
1  #version 150 core
2
3  uniform mat4 modelMatrix;
4  uniform mat4 viewMatrix;
5  uniform mat4 projMatrix;
6  uniform vec4 nodeColour;
7
8  in  vec3 position;
9  in  vec2 texCoord;
10
11 out Vertex  {
12    vec2 texCoord;
13    vec4 colour;
14 } OUT;
15
16 void main(void)   {
17    gl_Position    = (projMatrix * viewMatrix * modelMatrix) *
18                      vec4(position, 1.0);
19    OUT.texCoord   = texCoord;
20    OUT.colour     = nodeColour;
21 }
```

SceneVertex.glsl

Fragment Shader

In the fragment shader we make use of the *useTexture* **uniform** variable. This was set in the *DrawNode* function, and will be non-zero if the currently drawn *Mesh* has a texture, which we use in an if statement to selectively sample and blend the texture map with the current fragment.

```
#version 150 core

uniform sampler2D diffuseTex;
uniform int       useTexture;

in Vertex   {
   vec2 texCoord;
   vec4 colour;
} IN;

out vec4 gl_FragColor;

void main(void)   {
   gl_FragColor = IN.colour;
   if(useTexture > 0) {
      gl_FragColor   *= texture(diffuseTex, IN.texCoord);
   }
}
```

SceneFragment.glsl

Tutorial Summary

Upon running the program, you should see a multi-coloured cuboid robot. He's not a very *good* robot, really - all he does is spin on the spot and wave his arms around. But in doing so, our little cube robot has demonstrated the basics required to draw multiple objects in a simple manner. As his root node spins on the spot, his 'child' node body and limbs are transformed, too. If you were to translate or scale the robot's root node, all of his constituent parts would translate and scale correctly. That's the simple beauty of the scene graph structure. You've also learnt how to traverse a scene graph, which will come in handy in the next tutorial.
You should be starting to get a feel as to how a game's renderer can update and draw hundreds of different types of objects - you could easily pass many different subclasses of *SceneNode* to a *Renderer* class to place as children of it's *root* node, and it'd update, draw, and traverse them all, without requiring any explicit knowledge of what the subclass did.
In the next tutorial, you'll see how to traverse a scene graph in such a way as to partially solve the issue of transparent object draw ordering, how to skip the drawing of nodes outside of the camera's view, and how to take advantage of an optimisation feature of modern graphics hardware - the *early z test*.

Further Work

1) Try adding 10 *CubeRobots* to the scene - do you need to change the *Renderer DrawScene* function at all? How about the *UpdateScene* function? How many *Mesh* class instances would 10 *CubeRobots* require?

2) If you wanted to make a CubeRobot 10 times bigger, how many *SceneNodes* would you have to scale?

3) Make the *CubeRobot* a bit more interesting - try adding a transitional node to act as hips. Which node could it be a child of? What nodes would be children of the hips node?

4) So far you can only add children to a *SceneNode*. Try adding in the ability to *remove* children, too. Handy if the *SceneNodes* are enemies you are defeating during the course of a game. Also, you can currently add a *SceneNode* as a child of itself...what happens then? How would you avoid this?

5) *SceneNodes* currently have separate *Mesh* class instances. Perhaps they could have separate *Shaders*, too? Or even override the current *Mesh's* texture!