CHAPTER 4

Graphics Programming -- Getting Stuff To Screen

Core Elements Of Graphics Engine 2
- Rendering Context 2
- Geometry Primitive 2
- Texture 3
- Shader 3

Foreword For The Tutorials 4
Application Framework For The Tutorials 5
Tutorial: Drawing Two Textured Triangles 7
- Using Smart Pointers To Manage Resources Automatically 9
- Preparing the List of Geometry Primitives 12
- Texture And Shader Setup 17
- Rendering a Frame 18

Tutorial: Using Transformations 20
- Transformation Spaces 21
- Transformation With Rotation 21
- Representing Transformations With Matrices 23
- Using Transformations In Practice 26

Tutorial: Introduction To Shaders 32
Power of Hierarchies -- Scene Graphs 37
- Nodes 38
- Meshes And Other Visuals 39
- Lights 41
- Camera 45
- The rest of the scene graph 46

Tutorial: Scene Graph Animation Playback 47
Tutorial: Transformation Hierarchies And Parenting 53
Glow Post-Processing Effect Implementation Details 56
Tutorial: Character Animation 59
- Separate upper body animation, aiming and shooting 70

Tutorial: Rigid Body Physics Simulation 73
- Simulation basics 73
- Text rendering (debug info and in-game HUD) 76
- Simulation in practice 78
- Inside ODEWorld::step 82

Particle Effects 84
- About particles, emitters and systems 85
- Particle simulation 86
- Concept of domains 87
- Particle properties 88
- Emitter properties 94
- System properties 95
- About particle effect examples 95
- Example particle effect: Fire 96

Fire system properties..... 97
Fire emission properties..... 97
Fire particle properties..... 98
Fire bitmap properties.....100
Playing particle effects in the 3D engine.....101
Some Good Read.....102

Core Elements Of Graphics Engine

Programming graphics is fun with today's hardware. You can just say *'Hey, I want 100,000 polygons down here!'* and 3D-graphics accelerator does the job. But to draw 100,000 polygons you need to learn to draw one first... Before we go to actual tutorials, let's take a look at what a graphics engine is made of.

Rendering Context

Before you draw anything you need to have some destination where you draw the stuff. Also if you think about 3D-graphics rendering and accelerators, it's easy to realize that you need some abstraction, which represents the graphics device in your computer. Rendering context serves this purpose. Rendering context manages rendering device state and rendering device dependent resources like textures and vertex buffers (we'll get back to them later). Rendering state defines for example how each vertex or pixel is processed before it's written to the back buffer. Main responsibilities of rendering context are that you can *create* other objects with it and flip rendered back buffer to screen.

Geometry Primitive

To get something visible on screen, you need to have also something to draw. *Geometry primitive* represents unit of this renderable geometry. Geometry primitive can be triangle, line or point. And as with modern hardware everything is about *batching*, we don't feed the graphics hardware with units of single primitives but we use *primitive lists*. In it's simplest form a primitive list is just a list of vertex triplets,

each of which defines a triangle to be rendered. It can be also a set of polygons, lines or points. It can be either directly defined by vertices (as for example three vertices form a triangle), or indirectly specified by index list, which allows re-use of same vertices in multiple polygons or lines to save storage and provide more efficient rendering. But having just a primitive list isn't enough, since primitives define only the geometry to be rendered. You also need somehow define how you want the geometry of a primitive to be drawn, or to cover the geometry with, however you want to think about it. Here a *texture* comes into the picture.

Texture

Having just plain geometry without any surface details is a bit boring, and spending several millions of polygons on surface details alone (to make the details) is out of question even by today's advanced hardware. Good compromise is to use a *texture* to give the surface the details it needs. You can for example give a surface appearance of rusted metal, grass or skin, by mapping an image of such material over the geometry. Texture mapping simply stretches an area from the bitmap over your surface on coordinates you have specified. But as you can imagine, impressiveness of a simple image over your geometry doesn't carry that far. For example it would be nice to get lighting to affect the colors of the surface, and that's why you need *shaders*.

Shader

Shader, or *material*, describes how a surface and texture should be rendered to screen. Vertex shader takes input of lights and geometry and maybe some shader specific constants, and gives output to the pixel shader. Pixel shader takes input from vertex shader and textures as input from application and outputs actual result on screen. Figure 4-1 illustrates this data flow.

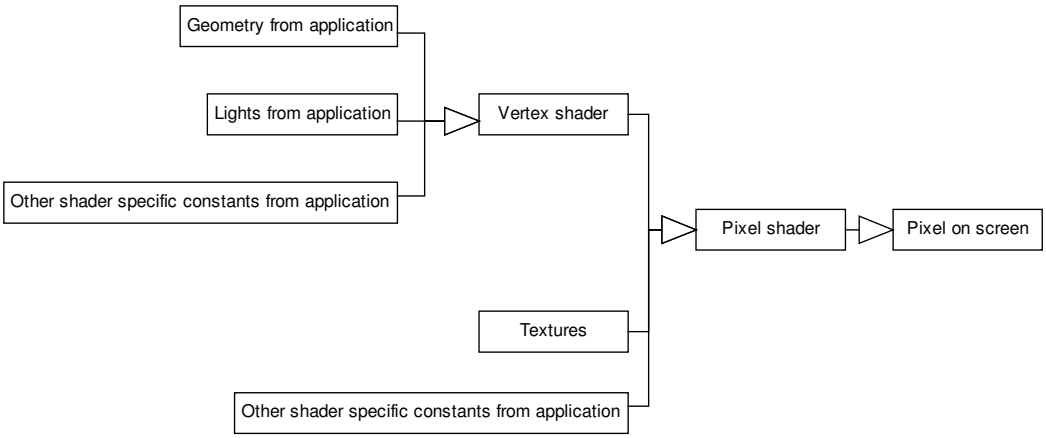

Figure 4-1. Shader data flow.

A shader might for example tell us that all geometry using this shader should have colored shiny highlight, like metals. Or shader might specify that the surfaces are rendered using one texture map and maybe shaded by one point light.

In other words a shader defines the look of the surface on screen. It requires the application to give to shader the parameters it needs. Shaders might have almost any parameters you can imagine. For example plain diffuse shader might require only the color of the material and position of a light source. And on the other hand some complex marble shader might require very specific information about the curvature the marble type currently being rendered. If this explanation of shaders really didn't quite open to you, don't worry, it will all become clear later. For now you can just think shaders as something, which draw the textures to the screen and combines the result with lighting. (And actually, that's pretty much exactly what they do)

Foreword For The Tutorials

So that's it. You need rendering context, primitive, texture and a shader. That's what Direct3D and OpenGL provide already as is, right? So, you might be wondering why there is all this fuzz about 3D-engines? The thing is that you quickly start to realize that you need a lot more than just the bare bones 3D-graphics core to make a game.

For example instead of moving points of every object to move them around on screen it would be nice to be able to move *camera*. And when object moves you can't just move the points (as it would quickly explode due to inaccuracies!), but you need to give them *transformation frame of reference.* And you need *much* more to get a single animated 3D character on screen: You need to have file format support for some 3D-modeling application so that your team's artists can create the characters. And you need to be able to process the geometry to be able to render it efficiently. Or render it at all, since for example Pixel Shader 1.1 hardware generation shaders require that you have only maximum of about 30 bones, and usually characters have more. And there's a *lot* more to do which I won't go in here.

The point is just that at if you're not careful you'll end up making *only* the engine surrounding the game, and you'll never make it to the point where you can work on your game at all. Which is fine of course if you *love* the 3D-engine development, as I do, but if want to program games then making an own engine isn't probably a good way to go. This doesn't mean that the (relatively speaking) 'low level' stuff is not useful to learn, on the contrary. Efficient usage of *any* 3D engine requires that you understand what's going on under the hood.

Application Framework For The Tutorials

This all tutorials we'll be using a simple application framework, which frees us from very repeating (and uninteresting) routine housekeeping tasks. The framework handles following responsibilities:

The framework proposes default settings for the application by calling application implemented function configure() with a reference to a framework::App::Configuration structure as parameter. Application can modify these settings before the framework actually performs the initialization. Usually setting just

the name of the application is enough and other defaults are ok. But if you need, following settings can be configured:

- Name of the application main window (config.name)
- Width of the back buffer (config.width)
- Height of the back buffer (config.height)
- Back buffer pixel format, bits per pixel (config.bits)
- Is full-screen or desktop window used? (config.fullscreen)
- Is stencil-buffer required for the application? (config.stencilbuffer)
- After making sure the configuration is ok, the framework performs the initialization by first creating the application main window and then the rendering context.

For better maintenance and resource usage you really want to keep your shaders and particles in one place neatly organized into properly named subdirectories by usage category. For this reason the framework handles this already for you, it loads default set of shaders and particles from the directories pointed by SHADERS and PARTICLES environment variables, respectively (you can set environment variables in Windows XP by selecting 'My Computer', 'Advanced', and then 'Environment Variables'). For actual game you'd of course load the shaders from some local data directory (all data loading functions allow you to specify shader, particle and texture directories separately, it's just more convenient to handle it by environment variables during the development process of the game).

After main window and rendering context initialization and loading the default set of shaders and particles, the framework asks user application to create it's application instance by calling init() function implemented by the application. init() must return an instance implementing App -interface. By minimum, your class derived from App

should implement update(dt,context) function and nothing else. init() function gets in the rendering context as parameter, so you can already create your textures, shaders and geometry in your application, say MyApp, constructor. Normally the user supplied init() function just returns its application instance created with operator new, so you should perform all your initialization in the application class constructor.

After init(), the framework enters in a main update/render loop. During the main loop, the framework measures elapsed time since last rendered frame, and asks the application to update itself and render a frame by calling App::update. App::update gets in delta time (in seconds) since last frame and rendering context, so you can for example update your animations based on elapsed time. In general, it's a good idea to base your game update to elapsed time instead of frame rate, since frame rate can vary so much on PC due to various (uninteresting and rather obvious) reasons, which are not dependent on your application.

In addition to the main loop, the framework calls application when for example the main window receives a key press. In this case, App::keyDown and App::keyUp virtual functions are called. In addition to these functions, you can also use App::isKeyDown to check for key state, which is handy for continuous input reading like movement.

When the application is closed down, your application instance's destructor is called as you would expect.

Tutorial: Drawing Two Textured Triangles

So before drawing million polygons to screen we need to learn to draw one. In this tutorial, and all following the ones, we'll be using the simple framework introduced in the previous section. In this tutorial you will learn how to:

- Use smart pointers to make resource management less error prone

- Prepare a geometry primitive list ready for rending
- Initialize a base texture map and shader
- Render the texture mapped primitive list to back buffer
- Show back buffer to screen

At first, this might seem just like a basic example to get you started and really not anything useful in real world. But this intuition isn't exactly correct, since, as you will see later, the techniques introduced in this chapter will be used *directly* in a bit more complicated example later, which renders whole HUD which you might need in a game. Note that rendering in screen space this way used because using 3D primitives instead of 2D surfaces is more effective usage of the hardware, and in architecture level gives nice uniform approach to the rendering, which is nice for us as the users.

But without longer speeches, let's jump to the code. Open now header file DrawTriangleApp.h from tutorials/draw_triangle/ directory:

```
#include <framework/App.h>
#include <gr/all.h>
#include <lang/all.h>

class DrawTriangleApp : public framework::App
{
public:
    DrawTriangleApp( gr::Context* context );
    void update( float dt, gr::Context* context );
private:
    P(gr::Primitive) m_prim;
};
```

The header file is relatively self-explanatory. First, our simple application framework headers are included by including <framework/App.h>. Then all 3D-graphics library headers and C++ language support library (lang) headers are included. If you have used Java before then you might find this library quite familiar looking, as many of the classes have designs based on java.lang.* classes.

This header file doesn't have any include guards (#ifdef _MYAPP_H ... #define _MYAPP_H ... #endif), just because we have only a one header and one cpp file in our simple application.

In the class declaration, main application class, DrawTriangleApp, is derived from framework::App and it's constructor and update() function declarations are based on the framework conventions as well -- both take rendering context as parameters. The application has only one member variable, geometry primitive (class gr::Primitive). Note that the variable is neither regular pointer or object, but it is a *smart pointer*, a *reference counting smart pointer* to be exact. Next we look at how these smart pointers are used. If you're familiar with the concept, you can just browse the next section through for the main points.

Using Smart Pointers To Manage Resources Automatically

As you're familiar with C++ programming, you know that sometimes dealing with pointers can be a hassle, especially if you have multiple pointers to the same object. Also it's easy to forget to call operator delete or to call it for a wrong object. There is numerous errors you can make easily, even if you're an experienced programmer. Smart pointers make a life easier by keeping count in the object how many times it has been referenced. So when a pointer is assigned to smart pointer, smart pointer increments the object's reference count, and each time smart pointer loses the pointer, it decreases the object's reference count and the object is deleted if the reference count reaches 0. This means in practice that you don't need to call delete manually almost *ever*, which releases you from most of the problems associated with pointers.

Beware cyclic references though. Cyclic reference appears if A has pointer to B and B has pointer to A. For example if GameLevel has pointer to GameCharacter and GameCharacter has again pointer to the game level, which is quite common scenario.

If you directly convert these pointers to smart pointers you end up with cyclical reference, which might not get freed automatically, since even if you release pointers to both GameLevel and GameCharacter, since both of them have still pointers to each other. There are a couple of techniques to solve the problem:

- Use escalation to move responsibility to higher level. Use smart pointer only in the higher level 'owning' class, in this case GameLevel. This usually works very well since for example all objects in game level will be released at the same time as the level too.
- Move responsibility to lower level, in other words *demote* responsibility lower. So example instead of having two classes dependent on each other, separate the shared component to new class and make *both* classes dependent on the new one. Voila! You got rid of cyclic dependency.
- Have a separate destroy() function, which sets smart pointers to 0, releasing objects referenced by them.

Luckily cyclic references are usually very straightforward and easy to solve using techniques (1) and (2), but sometimes you can end up accidentally having a pointer *indirectly* without realizing it, and then you have to debug the memory allocations to find the leak. Debug memory allocation functions, that is debug routines that let you know the source file name and line number where the allocation was made, can help you a lot tracing back to the source where the leak chain started.

One thing to keep in mind with cyclic references is that if you end up with them, there *might* be also some problems in your software architecture design. In general, it is bad to have classes depending on each other, because then you have hard time testing or modifying the classes individually. Any chance you do can potentially

affect both of the classes. Pretty tricky, so best to do is to avoid cyclic dependencies in the first place, both in software architecture design and in implementation.

So how to use smart pointers correctly?

1. First, you use smart pointer when you *store* a pointer to an object.
2. Second, you use smart pointers when you hold *potentially* the last pointer to the object, for example when you allocate object in main().
3. Third, you *don't* need to use smart pointers when you give parameters to functions, because functions can assume that the objects are valid until the execution of the function anyway, since *caller* has a reference to the object anyway.

So ok, they are good. But how to use them, in practice, programming wise, in your own applications?

You include <lang/Object.h> and derive your own class (say, MyClass) from it using public inheritance, in other words 'class MyClass : public lang::Object'.

You declare a smart pointer P(MyClass) and DO NOT call delete on it, ever!

When you declare functions or methods which accept MyClass as parameter, you just use MyClass*, for efficiency, since the caller already has a valid reference to it, but when you *store* a pointer then you use smart pointer declaration P(MyClass).

For all not-so-gritty details, look at source code and header files <lang/Ptr.h> and <lang/Object.h>.

This all can sound a bit confusing, but it all becomes clear when you look at the code and get used to using smart pointers a bit. After getting used to them, you can't leave without them. I can hardly remember when was the last time I made an error while using smart pointers, but they make my life easier all the time as they release allocated resources safely. And smart pointers do it efficiently too, as resources are

deallocated as soon as they are not referenced anymore. Ok, but now back to the rendering!

Preparing the List of Geometry Primitives

In 3D rendering, performance equals to *batching*. Batching means that you give as many triangles to be rendered in a single function call as possible to get best performance. You also have to remember that 3D-accelerators work in parallel to CPU. This has implication that 3D-accelerators need their own memory to be able to process the stuff independently of the CPU memory bus, and that the memory has to be allocated efficiently. 'Efficiently' in turn is the same as not-during-rendering. Both of these issues lead to conclusion that to be able to render most efficiently, you need to allocate geometry primitives on 3D-device's own memory, and prepare the memory ready for rendering beforehand. This section shows how to prepare a list which in our example contains two triangles forming a textured quad.

Vertex format describes internal layout of the vertex memory on the 3D-device. First in the application DrawTriangleApp constructor, we need to set up this vertex format for the geometry primitive list:

```
VertexFormat vf;
vf.addTransformedPosition();
vf.addDiffuse ();
vf.addTextureCoordinate( VertexFormat::DF_V2_32 );
```

In this case, we specified that we want to use a) screen space, that is transformed, vertices b) vertex colors and c) 32-bit float 2-vectors as data format for texture coordinates. You can also specify exact data formats for both transformed vertices and diffuse colors if you want, but we just accepted the defaults. Note that not all platforms support all possible vertex formats, so even tho the graphics rendering library (gr) interface let's you *suggest* exact format for the vertices' bit layout, the platform specific implementation can modify it if needed, and choose another

compatible format for example for performance reasons. The implementation can also store the vertices either as stride data or as continuous data, so beware counting on some exact bit layout when you deal with vertex data. Luckily you can avoid this bit fiddling easily by using setVertexPositions(...) type of functions present in Primitive class.

Next, we create the actual primitive list object, which serves as interface to the storage on actual 3D-device memory:

```
const int VERTS = 4;
const int INDICES = 6;
m_prim = context->createPrimitive( Primitive::PRIM_TRI, vf, VERTS, INDICES );
```

Since this primitive list is 3D-device dependent object, it is managed by the rendering context and we need to create it by calling one of the createXXX functions in gr::Context class. To this function we specify that we want a list of triangle primitives (Primitive::PRIM_TRI) and we want 4 vertices and an index list of 6 indices. You can either use indexed or non-indexed primitives, but usually indexed primitives are better since they allow you to save memory on data storage. You might be wondering how since indices themselves take memory too, but the trick is that using indices take only a very little memory when compared to vertices, and usually many triangles share vertices so indices helps 3D-device to take advantage of that. So in addition to memory usage, using indices is the way to go since they can be more effectively rendered by the 3D-devices. (Note*: On Playstation 2 you really need to pre-process the indexed triangle lists and use *triangle stripes* instead, but we don't worry about it now) Note also that we store the returned object to a smart pointer declared in DrawTriangleApp class, since we need the object later on when doing the rendering.

After creating the primitive list object, it might contain some random garbage data which was present in the 3D-device memory, so we need to write all the needed data

to the primitive list memory before starting to use the object. But before we write the data, we need to *lock* the object for writing. Remember that the device objects have their own memory, so to be sure the memory is not in use when we access it, we need to lock it before using it. We could do this locking by calling familiar lock() and unlock() functions on Primitive object, but instead we initialize a auto/stack-object of type Primitive::Lock:

```
Primitive::Lock lock( m_prim, Primitive::LOCK_WRITE );
```

This lock-handle class works like a smart pointer to the lock, it makes locking/unlocking automatic process. The lock is automatically freed at the end of the scope, so you don't need to call unlock() or anything else to release the lock. This has even more help when you start thinking about error situations, since the code using this lock class is *exception-safe* and lock()/unlock() functions aren't if there is even a single peace of code between them which might throw an exception. And remember that even though *your* code doesn't throw an exception someone else's code might, so better to be always exception safe instead of relying on something not to throw them.

After locking we can write the data to our primitive list, starting from vertex position data:

```
const int VERTS = 4;
float4 vertpos[VERTS] = {
    float4(10,10,0,1),
    float4(300,10,0,1),
    float4(300,300,0,1),
    float4 (10,300,0,1) };
m_prim->setVertexTransformedPositions( 0, vertpos, VERTS );
```

The first parameter to the function specifies the first vertex index to set. In this case we want to set all vertices. The coordinates passed to the function form a quadrangle, starting from top left clockwise and ending at bottom left. Note that the coordinate system: Coordinates are in *device coordinate system*, which is to say in

screen pixels. Device coordinate system origin (0,0) is in top left corner and (width-1,height-1) coordinate is in bottom right corner.

As you already know, texture coordinates are normally used to map bitmap image to a polygon surface. After vertex positions we write these texture coordinates to the primitive list for each vertex:

```
float4 vertuv[VERTS] = {
    float4 (0,0,0,0),
    float4 (1,0,0,0),
    float4 (1,1,0,0),
    float4 (0,1,0,0) };
m_prim->setVertexTextureCoordinates( 0, 0, vertuv, VERTS );
```

Note that now, in addition to the first vertex index, the function also gets in another argument specifying texture coordinate layer to be written to. Since we added only one texture coordinate layer (vf.addTextureCoordinate(VertexFormat::DF_V2_32); in vertex format creation) we must set this parameter as 0.

Texture coordinate axis are the same as device coordinates, so that X-axis (or 'U-axis', as it is more commonly referred when speaking about texture coordinates) grows right and Y-axis (or 'V-axis') grows down. Texture coordinates however, are *relative* instead of absolute, so bottom right corner is coordinate (1,1) instead of pixel count. This is very convenient for us, since different 3D-hardware accelerators might have different requirements for the textures, and the platform might need to scale texture bitmaps to different size in what they were originally. Having relative coordinates makes our geometry setup independent of the texture sizes.

Note that our simple texture coordinate usage doesn't result in *pixel perfect* rendering of the texture image to the screen, since texture coordinates specify *center* of the texture pixel in the texture, and we actually have specified top-left corner of the pixels. To get exactly each texture pixel to correspond pixels on device back buffer

we'd need to offset texture coordinates by half of a relative texture pixel, which is to say 1/texture width and 1/texture height, but for the purposes of this tutorial we just simply ignore the 0.5 pixel error. (even tho it might be very visible, especially due to bi-linear filtering the 3D-accelerator might do while sampling the texture in drawing process!)

After texture texture coordinates, we can relax a bit by settings vertex colors to the primitive:

```
float4 vertcolor[VERTS] = {
    float4 (255,255,255,255),
    float4 (255,255,255,255),
    float4 (255,255,255,255),
    float4 (255,255,255,255) };
m_prim->setVertexDiffuseColors( 0, vertcolor, VERTS );
```

In this we set vertex colors as bright white as we just want to get the textured quad without any color modulations. Note that not all shaders require vertex colors, and you should take care that the primitive's vertex format matches the declaration used in the shader file.

Last component to write to the primitive is the index data:

```
int indices[INDICES] = {
    0,1,2,
    0,2,3 };
m_prim->setIndices( 0, indices, INDICES );
```

Indices tell the primitive which vertices form which triangle. In this case our four vertices form two triangles: The first triangle consists of vertices (0,1,2) and the second one consists of vertices (0,2,3). Note that we already benefited from sharing data, since we were able to re-use two vertices from the first triangle in the second one!

In general, less data, more bandwidth, faster rendering speed. Of course in this particular example the benefit doesn't make any difference as the rendering setup cost is far greater than rendering two triangles. In practice you should try to batch your

triangle rendering so that you render always at least 2000 triangles in a single primitive list to gain maximum rendering speed (at the time of writing this text, so the number might be even higher when your read this!). HUD and other '2D' rendering makes an exception tho, usually the number of triangles rendered in a single primitive list are quite small, but on the other hand it doesn't make much difference as we're talking about so small number of triangles and primitive lists anyway.

Texture And Shader Setup

Textures are 3D-device resources as well as primitives, so they need to be created through the rendering context:

```
String datapath = "../../data/images";
P(Shader) tex = context->createTexture( PathName(datapath,"rgb_text4b.bmp").toString() );
```

In this particular example case we of course knew already the exact filename of the texture beforehand so we could have directly use the string "../../data/images rgb_text4b.bmp", but usually this is not the case so I decided to give a small example of pathname processing. Texture file names might for example be specified in some scene file format or (for example HUD-related) Lua script, and on the other hand we probably want to load textures from a separate directory and not always specify the directory name in the script for flexibility. Using PathName class we can easily manipulate path names however we need, for example ask which is the parent directory of given file name, which is the basename (filename without extension), etc. without error prone direct manipulation of filename character strings.

Shader setup is only one line more code than texture setup:

```
P(Shader) fx = context->createShader( "sprite-copy" );
fx->setTexture( "BASEMAP", tex );
m_prim->setShader( fx );
```

First, we initialize 'sprite-copy' shader. The shader is very simple, it just copies the texture pixels to screen and modules the color by interpolated vertex colors. The

shader file itself can be found from directory shaders/sprite/sprite-copy.fx if you want to take a look already now, but you don't really need to since we take a look into the shader programming later. 'Sprite' shaders are good for us in this case since we are only using pre-transformed (screen space) vertex positions in this example and sprite shaders don't require to set up so many parameters as shaders normally do.

Note that we don't need to specify any path for the shader, since the application framework has loaded the default set of shaders for us to be used as templates to create new ones. In addition to nice convenience, this works as an abstraction since on all platforms separate shaders might not directly match to files at all. This is be the case for example on Playstation 2, which doesn't have programmable pixel shader hardware, but it does have something similar to vertex shaders, namely it's programmable Vector Unit 1. (Playstation has also Vector Unit 0, but it's used normally more as CPU co-processor to help with vector math).

Anyway, after loading the shader we set the texture which the shader should use and then set the shader to the geometry primitive list.

Note again the use of smart pointers: Only gr::Primitive (smart) pointer was stored to the application class, resources of gr::Texture and gr::Shader are managed automatically as the primitive has a reference to the shader and the shader has a reference to the texture. When primitive is released (at the destructor of DrawTriangleApp class) the (last!) reference to the primitive is released, and when the primitive is released the shader is released, and when the shader is released the texture is released as well... Elegant, convenient and before all, safe programming.

Rendering a Frame

Since all our 3D-device objects have been setup already beforehand, there is not much left to do in actual main loop App::update(float dt, gr::Context* context) method,

which is called by the framework in every frame after it has flushed Win32 message queue. Note that the main loop needs to be abstracted away this way, since there is no such concept of 'flushing messages' for example on Playstation 2 or Symbian platforms.

First, we start rendering a scene by creating Context::RenderScene object:

```
Context::RenderScene rs( context );
```

In general, you should have a single RenderScene object in your main loop. This *exception safe* code ensures that the scene rendering is ended at the end of scope even if the loop would be exited by thrown exception.

After beginning the scene, we inform the shader of the primitive that we want to render objects using it:

```
Shader* shader = m_prim->shader();
int passes = shader->begin();
```

The 'passes' value returned by the shader begin() functions tells us how many rendering passes are required to render the shader effect on current hardware platform. You should then call shader's beginPass(index), primitive's render() and then shader's endPass() functions as many times before ending the rendering with shader->end() call:

```
for ( int i = 0 ; i < passes ; ++i )
{
    shader->beginPass( i );
    m_prim->render();
    shader->endPass();
}
shader->end();
```

Note that there is even a better way of calling begin()..end() and beginPass(i)...endPass() functions: You can use Shader::Begin and Shader::Pass helper classes, which call the methods in their constructor and destructor. These classes have the advantage that they provide *exception safety*, which ensures that

corresponding endXXX() method is always called properly even if exception is thrown.

Now from our point of view, the primitive has been rendered to the back buffer (in practice it might take a while before 3D-device actually decides to start rendering, since it works parallel to the CPU). Now we end the update by swapping the back buffer to screen:

```
context->present();
```

Now that's almost it. Now we have only two 'housekeeping' functions to go:

```
void framework::configure( App::Configuration& config )
{
        config.name = "Low Level Rendering Example";
}

App* framework::init( gr::Context* context )
{
        return new DrawTriangleApp( context );
}
```

The first function, framework::configure, modifies the default App::Configuration only by changing application main window title as 'Low Level Rendering Example', and the framework::init function creates our DrawTriangleApp to be managed by the application framework. And that's it, now we have a textured triangle on the screen and hopefully a bit of information about what is happening behind the scenes.

Tutorial: Using Transformations

Now that we've covered basic drawing, we move on to transformations. In this tutorial, you will learn to:

- Learn to understand 3D transformation spaces
- Use hierarchical (combined) and inverse transformations
- Setup and transformations in rendering
- Apply transformations to shaders

- Animate transformations over time

Before getting into the actual code, let's go through some background on transformation spaces, matrices and what exactly they're made of.

Transformation Spaces

Let's say you have a camera at position (X=0, Y=0, Z=-10), that is, 10 units (pick your favorite distance unit here) behind world space origin. Don't worry about rotation yet. Now let's assume that you want to know where a cube is located with relation to camera when cube located at the world origin.

Not much to compute of course: If the camera is 10 units behind origin, then origin (and the cube) is of course 10 units in front of camera. After formalizing this intuitive result we get:

$P' = P_{obj} - P_{cam}$ (object position in camera space)

What you exactly did here is *inverse transform* (only translation but still), you transformed something from world space to camera space. Ok, let's flip it other way around: Cube is in position P_{obj} and it has vertex position V_0 in cube's own local space. Now we want to know what's cube's vertex location in world, so we calculate:

$V_{world} = P_{obj} + V_{model}$ (vertex position in world space)

So far we have assumed that all objects have identity rotation. That is, objects (and camera) are facing forward, their Y-axis up, X-axis pointing right (in left-handed coordinate system). How much would our equations change if this is not the case?

Transformation With Rotation

From basic trigonometry we remember that cosine of an angle represents X-axis position on a unit circle, and sine is position on Y-axis.

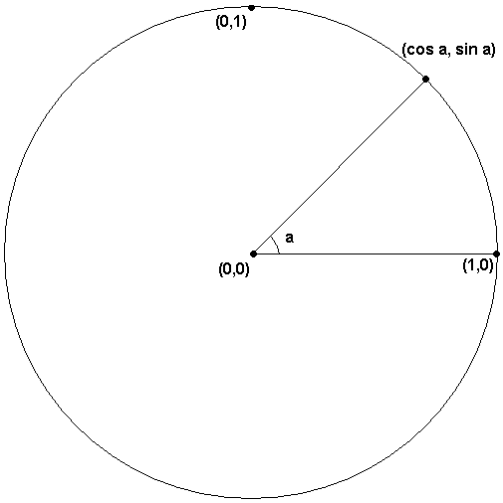

Figure 4-2. Unit circle.

Let's assume we want to rotate our vertices on XZ-plane instead of XY of the classic unit circle. Of course this doesn't change almost a thing, now we have just Y=0 and Z=sin(angle) instead other way around.

Now we make an interesting observation: To know where the vertices of an object are in world space, we don't need to rotate the vertices individually by this angle, but instead we can rotate the model space axes of the object to the world space. So model space is the space where vertex positions are defined originally before applying any transformation to the object. This way we can keep the vertex positions unmodified and compute world vertex positions directly from the axes and the model vertex positions.

Let's say we have rotated our cube about Y-axis (so on XZ-plane) 'angle' degrees. Now the axes of the cube in world space are:

X' = (cos(angle), 0, sin(angle))

Y' = (0, 1, 0)

Z' = (cos(90+angle), 0, sin(90+angle)) = (-sin(angle), 0, cos(angle))

'+90' degrees comes from the fact that when we're rotating only about Y-axis, Z-axis moves exactly the same route as X-axis, but it's just 90 degrees ahead (think

about unit circle!). Simplification is just basic trigonometry, since both sine and cosine curves have the same shape, they are just in different *phase* of the curve with the relation cos(angle) = sin(angle+90). Note the usage of angle units: I just keep the angles here in degrees for clarity. When programming, it's best to keep angles *always* in radians to avoid mixups.

Now that we know the axes of the object in world space, we can combine this rotation with the translation and transform model vertices to world space:

$V_{world} = X' * V(x)_{model} + Y' * V(y)_{model} + Z' * V(z)_{model} + P_{obj}$

(model-world -transform)

So we multiply rotated X-axis with X-coordinate of the vertex and do the same thing with Y and Z-axes. This makes sense since we're moving model coordinates *along* the axes and at the end translating with (world space) position of the object. If rotation is identity (X=(1,0,0), etc.) you get the equation we started with, namely

$V_{world} = P_{obj} + V_{model}$, since

$X' * V(x)_{model} + Y' * V(y)_{model} + Z' * V(z)_{model} = V_{model}$

when X'=(1,0,0), Y'=(0,1,0) and Z'=(0,0,1). (try this out and see!)

Representing Transformations With Matrices

When you start to think about the stuff you need to do with transformations in a typical game, you quickly see why the transform equation of the previous section doesn't fly as is. Both in graphics and games in general, you frequently need to compute inverse transforms (transform to/from some space), combine transforms ('first transform vertices to the space of this parent object, THEN transform the vertices to the world space...'), and all this comes quite error prone and even slow too, since after all, the combined transform is still a transform and you don't care about the

intermediate results which you're forced to calculate with the previous approach. To overcome these problems we represent the transforms with matrices.

Here is the previous vertex position represented as homogeneous 4-column vector and the transformation as 4x4 matrix:

$$\begin{bmatrix} X'(x) & Y'(x) & Z'(x) & P_{obj}(x) \\ X'(y) & Y'(y) & Z'(y) & P_{obj}(y) \\ X'(z) & Y'(z) & Z'(z) & P_{obj}(z) \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} V(x) \\ V(y) \\ V(z) \\ 1 \end{bmatrix}$$

You multiply matrices row by column, so that the result column vector is exactly the same as with the previous transform equation. The last element will be always 1 with these affine transformations, since we don't apply any perspective projection (yet) before transforming the points to screen.

Using matrix to represent this single transform doesn't pay us much, but the benefits become clear when we express some more complex transform operations using matrices. Below we represent matrices with M and we start with an inverse transform:

$$M_{inverse} = M^{-1}$$

So when we have a transform matrix M, we get inverse transform by taking inverse of the matrix. Now, let's say that we want to first transform vertices to parent object space, then parent's parent and then to world. We can do this with matrices by first constructing matrices from each object's rotation and then multiplying the matrices together:

$$M_{total} = M_{parent's\ parent} * M_{parent} * M_{obj}$$

What's even better is that this combined transformation can be used as is, and we can use it instead of doing *three* transformations for vertices. The benefit of all this

comes crystal clear when you think about rendering pipeline and how it transforms vertices:

Vertices are transformed from object's space to parent space, from parent space to parent's parent's space, until they are in the world space

From world space the vertices are transformed to the camera/view space. Vertex in view space is like transforming the world so that camera is in the origin.

View space vertex is transformed to projection space

This all can be done with a single matrix-vector multiplication $M_{total} * V_{model}$ as all the matrices can be combined:

$$M_{total} = M_{proj} * (M_{camera})^{-1} * M_{model\text{-}to\text{-}world}$$

Pretty convenient, uh.

In the 3D-engine's <math/...> classes all operations work as presented here, so you'd just write the previous equation like this:

```
float4x4 total = proj * camera.inverse() * world;
```

And the vertex would be transformed as follows:

```
float4 v1 = total * float4 (x,y,z,1);
```

Or more conveniently

```
float3 v1 = total.transform( float3(x,y,z) );
```

You also don't need to construct rotation matrices as tediously as presented in previous section from sine and cosine, but you can just say which axis you want to rotate about, and how much. Following example constructs Y-rotation and (1,2,3) translation combined:

```
float3x3 rotation( float3(0,1,0), Math::toRadians(30.f) );
float3 translation( float3 (1,2,3) );
float4x4 tm( rotation, translation );
// now tm is ready to transform vertices about Y-axis and then translate them by (1,2,3)
```

Note that matrix classes are in math library (<math/float4x4.h>, <math/float3x3.h>, <math/float3.h>, <math/float4.h>, ...) and core math functions are directly in language support library <lang/Math.h>. float4x4 class also contains

function for setting up perspective transform used in the example of combining transformations:

```
float4x4 proj;
float horzfov = Math::toRadians( 90.f );
float frontplane = 0.10f;
float backplane = 10000.f;
float aspectratio = 800.f / 600.f;
proj.setPerspectiveProjection( horzfov, frontplane, backplane, aspectratio );
```

This would set up perspective projection with 90 degree horizontal field-of-view, 0.1 unit (meters) front plane distance and 10000 unit back plane distance and aspect ratio of 1.33 (most common one). When you use this transformation to calculate actual device coordinates (pixels) of a camera space vertex position, you still need to scale the transformed vector with reciprocal homogeneous w after projection transform:

```
v1 = proj * v0;
v1 *= 1.f / v1.w;
```

Now v1 is in relative screen space [-1,+1]. To get actual pixel positions on screen you need to scale the coordinate with half of viewport size (width/2,height/2) and translate the coordinate origin to viewport center (viewport (x1+x0)/2, viewport (y1+y0)/2). Of course you can formulate this as transform matrix as well if you want!

Using Transformations In Practice

Now that you know the basic math, so let's move into the source code example. In directory tutorials/using_transforms/ you will find TransformApp.cpp, which we take a closer look next. The application is really simple; it animates a square (two triangles) moving and rotating triangles across the screen. Still, it is still useful for us as it contains all the basic transformations needed to get 3D graphics to screen.

First the application setups up texture and shader. Texture setup is similar to our first tutorial; it creates the texture from image file, 16-color palettized bmp-file to be exact. Shader to be used in this example is diffuse shader with texture. For now, you

can think of the shader as something, which applies lighting to the surface and draws surface to screen. We get into the shaders in detail in the next chapter. To the shader application sets up its diffuse map texture, which modulates diffuse lighting received by the surface and diffuse and ambient colors of the surface:

```
// texture setup
P(Texture) tex = context->createTexture( "data/rgb.bmp" );

// shader setup
P(Shader) fx = context->createShader( "diff-tex" );
fx->setTexture( "BASEMAP", tex );
fx->setVector( "AMBIENTC", float4(0.1f,0.1f,0.1f,0.1f) );
fx->setVector( "DIFFUSEC", float4(1.f,1.f,1.f,1.f) );
```

Next the tutorial sets up the quad (two triangles) used for rendering. The setup looks a bit similar than in the previous tutorial, but this time we're not specifying screen space pixels, but *model space* coordinates. First we need to define the data:

```
// geometry primitive data
const int VERTS = 4;
const int INDICES = 6;

float4 vertpos[VERTS] = {
        float4(-100,100,0,1),
        float4(100,100,0,1),
        float4(100,-100,0,1),
        float4(-100,-100,0,1) };

float4 vertnorm[VERTS] = {
        float4(0,0,-1,0),
        float4(0,0,-1,0),
        float4(0,0,-1,0),
        float4(0,0,-1,0) };

float4 vertuv[VERTS] = {
        float4(0,0,0,0),
        float4(1,0,0,0),
        float4(1,1,0,0),
        float4(0,1,0,0) };

int indices[INDICES] = {
        0,1,2,
        0,2,3 };
```

So we have three components in the vertex format, vertex *position*, vertex *normal* and *texture coordinate* (uv). Index list defines how triangles are formed from the vertices. Now we create the device object using the format we're interested in:

```
// create primitive with our vertex format
VertexFormat vf;
vf.addPosition();
vf.addNormal();
vf.addTextureCoordinate( VertexFormat::DF_V2_32 );
m_prim = context->createPrimitive( Primitive::PRIM_TRI, vf, VERTS, INDICES );
```

First we add position and normal components, then texture coordinates and then ask the primitive using the vertex format from the rendering context. Now we have un-initialized device object so we need to lock it and copy our data to it:

```
// lock primitive & set data
Primitive::Lock lock( m_prim, Primitive::LOCK_WRITE );
m_prim->setVertexPositions( 0, vertpos, VERTS );
m_prim->setVertexNormals( 0, vertnorm, VERTS );
m_prim->setVertexTextureCoordinates( 0, 0, vertuv, VERTS );
m_prim->setIndices( 0, indices, INDICES );
m_prim->setShader( fx );
```

Now we're done with the pre-work, and move to the more interesting part of the application, TransformApp::update() function. In this function, we'll setup all the transforms ready for rendering and render the created primitive using the transforms.

TransformApp::update() starts by setting up model to world transform. Remember that the benefit of this transform is that by keeping vertices in model space we can move the object freely in the world without painfully applying movement to each vertex of the object. This has also other benefits, since during time the modified vertices would drift away from each other because of floating point computational inaccuracy. Anyway, the model-to-world transform setup code first updates on-going time and computes animated angle from angle speed (180 degrees, pi, per second):

```
m_time += dt;
float angle = m_time * Math::PI;
```

Then it computes world transform by first setting world transformation matrix translation part on sinus curve based on the animated angle:

```
float3x4 worldtm( 1.f );
float movedistance = 100.f;
float x = Math::sin(angle) * movedistance;
worldtm.setTranslation( float3(x,0,0) );
```

Over the time, the animated position looks like oscillating movement on X-axis, which repeats every two seconds (remember, angle speed was 180 degrees/second, so full circle takes two seconds!). In actual application, you'd use some 3D modeling/animating package to make the animations you need, but this kind of 'debug animations' are quite frequently useful when you just need to get something moving on screen. Finally, for the world transform, we set up rotation part by rotating object about Z-axis:

```
// setup rotation in world space
worldtm.setRotation( float3x3( float3(0,0,1), angle ) );
```

The principle is almost the same as with position animation; we make rotation matrix (3x3) about Z-axis, rotated by animated angle.

Now we have transformation, which moves from our quadrangle vertices from model space to world space, wherever the object happen to be. But since we're doing rendering, we usually need things in *camera space*, so we setup the transformation of the camera in world space as well:

```
// setup camera world space transform
float3 camerapos = float3(100,200,-400);
float3x3 camerarot( float3(1,0,0), Math::toRadians(30.f) );
float3x4 cameraworldtm( camerarot, camerapos );
```

So we first set camera position to be at coordinate (100,200,-400) and rotation to be a bit tilted down towards the object moving along X-axis. Remember *trigonometric circle* from high school math when you think about rotations: If you look along Z-axis, and X-axis points right and Y-axis points up, then positive rotation

about Z-axis is *counter-clockwise* from your point of view. After we have setup camera transformation in world space we compute how to transform our now-in-world-space object to the view (=camera) space:

```
// setup view transform, which transforms
// world space to camera space
float3x4 viewtm = cameraworldtm.inverse();
```

Notice how handy matrices are in this context. From *any* camera position and rotation we might have, we can get easily transformation *to* camera space by inverting the matrix formed from the position and the rotation of the camera in the world space. And by basic matrix algebra, we get transformation from object model space to the camera view space:

```
// calculate transformation from model space to camera space
float3x4 modelview = viewtm * worldtm;
```

At least in this point you should have fallen in love with this elegant and beautiful matrix math! Now that we have our stuff in view space, we would like to view them in *perspective*. To do this, we set up a perspective projection matrix to transform the vertices to homogenous projection space. This space allows us to perform for example polygon clipping without calculating actual pixel coordinates (which require division by homogenous w-coordinate, so worth avoiding as long as the rendering hardware can):

```
// setup view->projection transform
float horzfov = Math::PI/180.f * 90.f;
float farz = 10000.f;
float nearz = 0.10f;
float4x4 projtm;
projtm.setPerspectiveProjection( horzfov, nearz, farz, context->aspect() );
```

In this particular example, we selected 90 degrees as *horizontal field-of-view*, which specifies the angle how much is visible on the horizontal direction on screen. And as we *don't* want our objects to be squeezed or stretched, we use *aspect ratio* (=ratio of our screen viewport width and height) to define what exactly should the

vertical field-of-view be. Near clipping plane defines the closest (in camera space Z-distance units) point that can be displayed on screen, and far plane defines the opposite, the most far away point before it's not displayed at all. From 0.1 to 10000 is pretty wide range so we should be clear. You should note however, that selecting *too* wide range has negative side effect: As Z-buffer has limited and linear accuracy (for example 16-bit or 32-bit), this means that the part close to camera can get *very* inaccurate if you select arbitrary large far plane distance or very small near plane distance. But this is in general very highly dependent on application (are you doing space simulation or ant action game?) so best is to try more closely ranged values if visual artifacts occur.

Finally we combine all transformations to *total transformation*, which transforms model vertices to homogenous projection space:

```
// setup total (model->screen) transform
// (note: 'screen space' in this context is platform dependent)
float4x4 totaltm = context->screenTransform() * projtm * modelview;
```

There is one trick, as the comment suggested: We need to apply additional rendering context specific 'screen' space transformation too before feeding the total transformation to the shader. This is caused by the fact that different rendering implementations on different platforms might expect different coordinates as input. For example pixel pipeline in Direct3D expects *normalized device coordinates*, that is coordinates between -1 and -1, but Playstation 2 shaders might expect coordinates expressed directly in pixels from user's total transform, so we need to give platform a chance to express it's needs about the projection space transform conversion to screen coordinates as well.

Now that we are done with the transformation calculations, we can just set the transformations to the shader and render the primitive using the shader, and end rendering by flipping back buffer to screen:

```
// render frame
{
	Context::RenderScene rs( context );
	Shader* fx = m_prim->shader();
	Shader::Begin begin( fx );

	// setup dynamic shader parameters
	fx->setMatrix( Shader::PARAM_TOTALTM, totaltm );
	fx->setMatrix( Shader::PARAM_WORLDTM, worldtm );
	fx->setMatrix( Shader::PARAM_VIEWTM, viewtm );
	fx->setVector( Shader::PARAM_LIGHTP0, float4(camerapos,1.f) );
	fx->setVector( Shader::PARAM_LIGHTC0, float4(.7f, 1, .7f, 1) );

	for ( int i = 0 ; i < begin.passes() ; ++i )
	{
		Shader::Pass pass( fx, i );
		m_prim->render();
	}
}
// flip back buffer
context->present();
```

If the shader stuff feels a bit confusing, don't worry, since we get into them just next!

Tutorial: Introduction To Shaders

What exactly happened in the last tutorial after we applied transformations to the 'diff-tex' shader and rendered the primitive using the shader? In this tutorial we get into the details, and learn about other shaders as well that you have at your usage in the 3D engine.

From the dataflow perspective, we need to separate two different kinds of shaders. *Vertex shaders* take vertices as input from the application, and vertex shaders produce output that is consumed by *pixel shaders*, which produce final pixel color (among other things on the latest hardware) to screen. See figure 4-1 for data flow diagram. Both shaders have in common that they consume and produce values interpolated by the hardware. For example texture coordinates are interpolated between triangle

vertices, between pixels on screen, and so forth. In ideal world, we're interested only in calculating values by pixel on screen, so we'd need only pixel shaders. After all, we could move the whole rendering equation to the pixel shader. But in practice we need to use vertex shaders to prepare data for more convenient form for the pixel shader to process. For example it would be grossly inefficient to compute vertex world position for each pixel, when we can just compute the positions for corners and interpolate them. So a good way to see vertex shaders is to look at them as pre-processor for pixel shaders -- they calculate *per vertex* data values, which can be calculated *per pixel* by interpolating between extremes defined by the vertices.

Now let's see what the 'diff-tex' shader used in the previous example exactly did. From file C:/ka3d/data/shaders/level/diff-tex.fx you can find Direct3D effect file, which describes the shaders. For now, you can ignore numerous #ifdef pragmas in the shader file, they are meant there for re-use of the same shader implementation in other shaders. Now, find function named vshader from the file:

```
PS_IN vshader( const VS_IN IN )
{
....
}
```

From C:/ka3d/data/shaders/vertexformats/NormalVertex.fxi you can find parameter VS_IN taken in by the shader. As you can see, NormalVertex.fxi/VS_IN consists of position, normal and texture coordinate, as in our example:

```
struct VS_IN
{
	float3 pos : POSITION;
	float3 normal : NORMAL;
	float2 uv0 : TEXCOORD0;
};
```

There is also optional data used in skinning, but we're not interested in that now. The point is that vertex shader assumes some specific vertex layout, and the application needs to conform to it if it wants to use the shader. Now let's get back to

the vshader function. From the previously introduced declaration we can see that it returns PS_IN structure as return value:

```
struct PS_IN
{
	float4 pos		: POSITION;
	float4 dif		: COLOR0;
	float2 uv0		: TEXCOORD0;
};
```

This is the data format used by the pixel shader. But since it's only more like internal data transfer between the vertex shader and pixel shader (both part of 3D accelerator), it's not that big deal for us, since we're not directly manipulating any data accepted by the pixel shader in this example. Actual vshader function body is very short, first we copy texture coordinates and do the transformations:

```
PS_IN vshader( const VS_IN IN )
{
	PS_IN o;

	o.uv0 = IN.uv0;

	float3 worldpos, worldnormal;
	transformNormalVertex( IN, o.pos, worldpos, worldnormal );
```

First we just copy texture coordinates from the application forward to the pixel shader, or more specifically to the 3D accelerator hardware interpolator, which gives interpolated texture coordinate values to the pixel shader, then we compute homogenous projection space vertex position (revisit the previous tutorial!) and world space normal and position coordinates as well. transformNormalVertex() abstracts away details of vertex transformation. You might be wondering why this is needed since it's only a simple matrix multiplication, but the catch is that we might want to use this shader with animated skinned characters as well, so isolate the transformation code to separate function and provide alternative implementations for it, both skinned version and non-skinned (this we're using now) version.

Lighting up the vertex is done next:

```
    ...
    float3 worldlight = normalize( LIGHTP0 - worldpos );
    float LdotN = saturate( dot(worldlight,worldnormal) );
    o.dif = float4( DIFFUSEC.xyz * LIGHTC0.xyz * LdotN, 1 );
    return o;
```

In this shader code, we first take the world position of the vertex, calculated previously, and compute vector to light position (LIGHTP0, as set in the application) from it. Then we compute cosine angle between this (normalized) light vector and vertex normal. This gives the effect that the vertex receives more lighting when the light source is directly above the vertex. This value needs to be *saturated* between [0,1] so that if light source is *behind* polygon using the vertex, the vertex doesn't get lit incorrectly as plain LdotN would be negative. Finally we compute light diffuse color component by combining this saturated value with light color (LIGHTC0) set by application and material diffuse color (DIFFUSEC) set by application as well. This color is fed to the pixel shader.

Now see function pshader in the same fx file:

```
float4 pshader( PS_IN IN ) : COLOR
{
    float4 diftex = tex2D( basetex, IN.uv0 );
    float3 dif = diftex.xyz*AMBIENTC.xyz + diftex.xyz*IN.dif.xyz;
    return float4( dif, 1 );
}
```

This is a very light-weight function: First it samples color from 2D texture map (our rgb.bmp image) from interpolated coordinates passed in by the vertex shader, then it scales the texture color by ambient color and diffuse color (computed by vertex shader) and adds them together as final output color. Notice that the light source color (LIGHTC0) is a bit greenish, so the polygon is green shaded as well because of the way diffuse light color is combined in the vertex shader.

At the end of fx files there is a list of *techniques* (or just one technique), which provides multiple different ways to render the same material. Techniques describe

which vertex and pixel shader to use, and possibly other parameters needed by the rendering device before actual rendering is done. At runtime, application might evaluate the set of techniques and find appropriate one for current hardware. This provides flexibility needed to cope with wide scope of different hardware generations out in the market.

In addition to this diffuse texture shader used in the example, the 3D engine provides a wide array of ready-to-use surface shaders for your use (see data/shaders/*.fx). Even though they are not introduced here, their usage is pretty much the same as this basic diff-tex shader, only vertex format changes according to needs of the vertex shaders.

There are three different vertex formats currently used by the shaders: Lit vertex, Normal vertex and Tangent vertex, sorted in order of increasing complexity. Lit vertex (data/shaders/ vertexformats/LitVertex.fxi) contains vertex color, provided directly by the application. Normal vertex (data/shaders/vertexformats/NormalVertex.fxi) provides vertex normal so that vertex shader is be able to do lighting calculation in the vertex shader, and Tangent vertex (data/shaders/vertexformats/TangentVertex.fxi) provides surface tangent space as well. Surface tangent space is used to do *per pixel lighting* in surface space, for example normal mapping is example of applying this technique. Various shaders (*.fx) use different vertex formats, so you need to make sure your data is in correct format when you initialize the primitives and select shaders to them. If you run into problems, make sure you have Direct3D Debug Runtime selected from Control Panel, and then run the application inside debugger to see Direct3D debug output, or use external debug output viewer such as DebugView utility from http://www.sysinternals.com/ntw2k/freeware/debugview.shtml. Direct3D debug

runtime gives you information if something when wrong with the vertex format setups.

Normally, thought, you don't need to worry that much about data formats, as usually your game's geometry data comes from some modeling package like 3DS Max or Maya. The data from those packages is *exported* to the file format used by the engine. This conversion process includes converting geometry to more hardware friendly format, creating correct vertex formats, and so forth. These tools do the vertex format conversions already for you, and you can just load the scene in the 3D engine and render it without knowing specifics which vertex format is in use at what point. But vertex format abstraction is only a scratch what it comes to the benefits of having more high-level view of the rendering engine, so next we'll introduce some abstraction to the rendering by introducing transformation hierarchies and scene graphs.

Power of Hierarchies -- Scene Graphs

Matrices are very convenient when it comes to dealing with transformations, but when you start thinking more about real-world situations related to transformations you quickly see how tedious task manual transformation management can end up to be, even with matrices, since world is full of various (and complicated!) transformation hierarchies. Earth moves around Sun, Moon moves around Earth, train moves on ground, in a train mother pushes her baby around in a baby carriage. It would be really troublesome for the baby if he wants to stand up and he would have to start considering his absolute movement starting from the position of the sun. For the same reasons hierarchy is your friend when it comes to hiding complexity.

Nodes

So hierarchies represent object relationships. But what kind of objects we have in 3D-scene? We have already mentioned a few: Camera, various geometry primitives, lights and of course particle systems, which we'll get into a bit later in detail. But when we start looking into object properties, we notice that they have quite a bit in common. More specifically, all of them have:

- Position
- Rotation
- Scale
- 'Parent' object, which defines the frame of reference in which object's transformation is defined. For example the baby's parent is technically the baby carriage (not the mother).

We'll encapsulate this data into *Node*, which serves as base class for the rest. This is convenient, since by using node abstraction, we can for example request a world space transformation of an object deep in the hierarchy, which would be very error prone to calculate otherwise. Now we can calculate the carriage-running baby's absolute position in the universe simply by calling baby->worldTransform().translation():

```
float3x4 Node::worldTransform() const
{
    float3x4 worldtm = m_modeltm;
    for ( Node* parent = m_parent ;
        parent != 0 ;
        parent = parent->m_parent )
    {
        worldtm = parent->m_modeltm * worldtm;
    }
    return worldtm;
}
```

Note that this function is iterative implementation that could be (more clearly) recursively expressed as:

```
float3x4 Node::worldTransform() const
{
    if ( m_parent != 0 )
        return m_parent->worldTransform() * m_modeltm;
    else
        return m_modeltm;
}
```

So we calculate world space transformation by applying successive parent frame of reference transformations until no more parents left. Note the order of multiplications: To transform our object's local transformation to parent's space, we need to multiply the parent's transformation matrix *before* the object's local transformation matrix. This might seem a bit counter-intuitive at first, but remember that matrix multiplication is not commutative (that is, a*b != b*a, in general), and if we want to transform something from A to B to C, then we indeed need to multiply the matrices in order T(C)*T(B)*T(A) to get correct combined transformation (writing the matrix multiplications open is left as an exercise to the reader).

So far so good, Node abstracts transformation hierarchy very nicely. Next we move on to rendering -- how does the *primitives* introduced earlier fit into our scene graph?

Meshes And Other Visuals

At first, it would seem straightforward design solution to make a geometry primitive list to be also a node. But this approach has a couple of problems:

Each primitive list has only a single shader associated with, so to represent for example character, we'd need to have multiple primitives to represent it, a number of primitives exactly the same as number of different materials in the character. For example hair would probably require own shader. Why not? The problem is that if we

want to move the object, we'd need to change the transformation of number of primitives. This problem, however, could be solved also by using transformation hierarchy functionality provided by Node.

The bigger problem is data sharing. Think about the situation where we have 1000 soldiers marching forward on the road. Each is individual, but we most definitely don't want duplicate soldier geometry 1000 times! This would be a requirement if each primitive can have only one transformation associated with it.

The solution to both issues is to leave transformation *out* of the primitive altogether. Instead, we provide a container node with transformation, we call it *Mesh*, that contains a set of geometry primitive lists. This way each geometry primitive can be in multiples Meshes, and be rendered multiple times during the same rendered scene. This also gives nice separation between rendered geometry and their transformation in the world in general. In addition to being a container of primitives, Mesh provides functionality for setting up dynamic shader parameters, which need to be reset in every frame. These kinds of parameters include for example camera and mesh transformation matrices and light parameters. By setting them automatically, the user can just render meshes without manually setting rendering constants for each shader.

Mesh class is a big help for rendering standard primitives, but some visual objects fit poorly to the container concept provided by Mesh. For example *particle systems* have usually only a single primitive, but they need constantly update in every frame. They could be described as *dynamic* primitives with transformation. Still, they have also some things in common with mesh, namely that

- They can be rendered
- They have bounding volume in the world

- They have shader associated with them

When rendering, we definitely won't want to handle particle systems differently that meshes. For this purpose, we give them a shared base class, *Visual*, that contains the interface to the shared functionality. In addition to particle systems, other Visuals include for example a line list (class Lines), very useful for debugging, and Console, which is a class for text rendering.

Lights

Light is another class derived from Node, which contains common light properties like color and range, in addition to node transform and parenting properties. In a scene graph lights are a bit lame group of objects when you think about modern rendering architecture based on shaders: Alone themselves, the lights do not even cause scene to be lit! To be precise, they are just input data for the shader, which either can use them or ignore them totally.

Complex shaders, like bump mapping, usually don't support many light sources, for performance and pixel shader instruction count limit reasons. In this case, only *key light* is used for the object, and other lights are either approximated or ignored altogether.

Key light can be selected by various different logics, very dependent on application being developed. Simple way is to find the closest light source to the object, but some times more proper better approximations can be used:

In outdoor daylight environment, sun probably makes very good candidate for key light source. Even on cloudy day, sun most likely dominates object lighting.

In the evening or night you might have for example street lights as light sources. In this case choosing the closest light source might work well for static objects ,but for moving objects it probably doesn't work as is, since in that case you get really

sudden change of light source when you walk under one street light to the next. Much better approximation for this kind of situation is to *weight* light sources, their positions, colors and other properties, based on their inverse squared distance. C++ source code for calculating position and color of this 'imaginary' light source would be:

```
// setup character lighting
LightSorter* lightsorter = m_level->lightSorter();
Array<Light*>& lights = lightsorter->getLightsByDistance( position() );

float ltweight = 0.f;
float3 ltpos(0,0,0);
float3 ltcolor(0,0,0);

for ( int i = 0 ; i < lights.size() ; ++i )
{
	Light* lt0 = lights[i];
	float3 ltpos0 = lt0->worldTransform().translation();
	float3 ltcolor0 = lt0->color();

	// weight light effect with distance attenuation
	float3 worldlight = ltpos0 - pos;
	float lensqr = worldlight.lengthSquared();
	lensqr *= lensqr;
	if ( lensqr > Float::MIN_VALUE )
		worldlight *= 1.f / lensqr;
	float ltweight0 = worldlight.length();

	if ( ltweight0 > 0.f )
	{
		ltpos += ltpos0 * ltweight0;
		ltcolor += ltcolor0 * ltweight0;
		ltweight += ltweight0;
	}
}

if ( ltweight > Float::MIN_VALUE )
{
	float ltscale = 1.f/ltweight;
	ltpos *= ltscale;
	ltpos *= ltscale;
	m_light->setPosition( ltpos );
}
```

The code above calculated weighted positions and colors of real light sources based on inverse distance, and normalized these to new key light for the object. The effect in practice is that you get smooth transitions between different key lights, and this is what we wanted in the first place. The same technique is used in the game demo provided on the CD to light dynamic objects like the player character *Zax*.

Choosing simply the closest light source is not very good for static game level lighting either. First of all, many meshes and primitives in levels are big, so picking 'closest' light source has no meaning, which would make sense, since there are multiple light sources close to each level primitive. For example one primitive in a level might be whole room, or even bigger area! Level objects have also nice property that they are *static* as mentioned, and our closest light source selection doesn't take advantage of that.

Instead, we can compute the base light solution *off-line*, and only add dynamic lighting as extra component to the level geometry. This way we can for example use a lot of computer time for example to calculate soft shadows in non-real-time, and use pre-computed solution in real time to provide realistic look. Exactly how we calculate the solution and how we use it is a bit application dependent, but using *radiosity* or other global illumination methods for calculating static lighting are generally good candidates. Radiosity takes into account not only direct lighting, but indirect lighting as well, which gives very realistic look for the scenes lit with this technique.

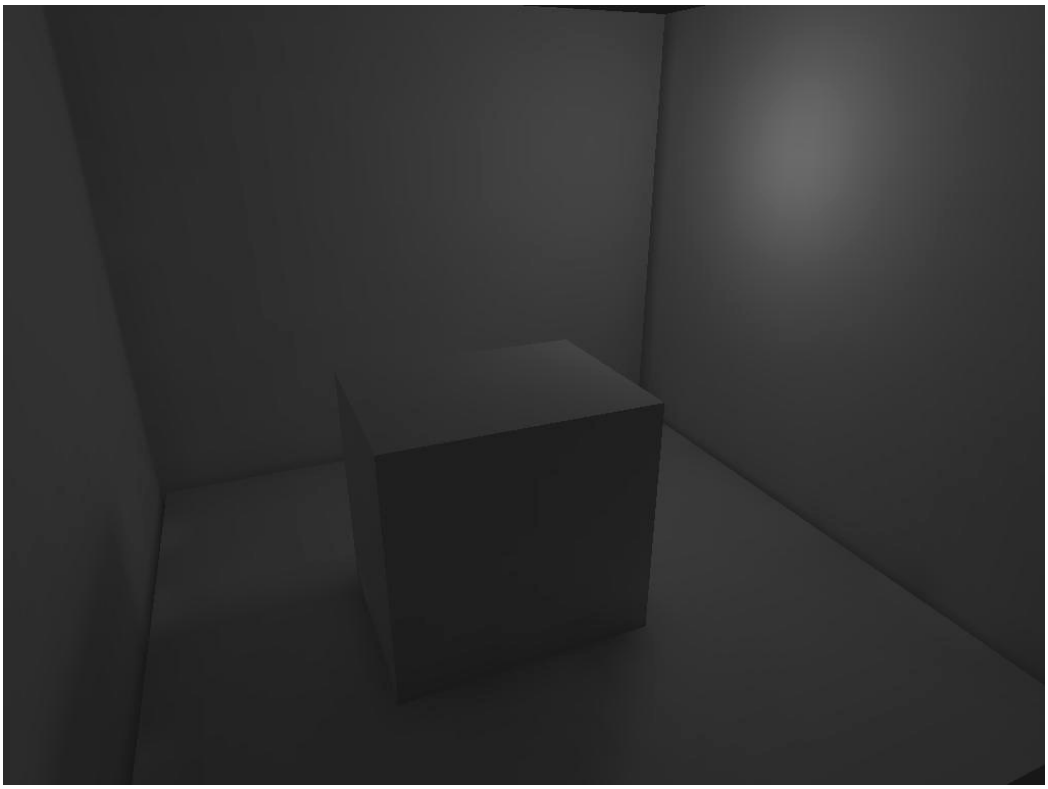

Figure 4-3. Screenshot of real-time scene utilizing lightmaps computed with radiosity global illumination.

In Figure 4-3 is a screenshot using radiosity rendering with *lightmaps*. Lightmaps are a technique where light values are stored *per texel* (texture map pixel) in the scene. This can provide very realistic lighting as almost infinite amount of computer time can be spend to calculate lighting solution. For example Max Payne used this technique for level lighting. In the screenshot, notice how the back of the cube is lit as well, even though there is only one light source. This is due to mentioned radiance transfer. In real-time rendering, we can just combine the calculated light map with texture pixel color by modulation. Pixel shader for doing this could be for example:

```
float4 lightMapPixelShader( PS_IN IN ) : COLOR
{
        float4 texmapvalue = tex2D( basetex, IN.uv0 );
        float4 lightmapvalue = tex2D( lighttex, IN.uv1 );
        return texmapvalue * lightmapvalue;
```

```
}
```

Radiosity and other off-line global illumination methods can be used also on vertex level. This has the benefit that less data is needed, only light color (4 bytes) per vertex, but the negative side is that lighting is much less accurate, as it has only polygon accuracy, when using lightmaps the accuracy is per texel. Still, these two methods of using pre-computed lighting are not mutually exclusive. Indeed, many games use *both* methods: Vertex based lighting on objects which do not have hard shadow edges, and texture based lightmaps when more accurate shadows are needed.

Nowadays dynamic lighting has become more and more useful approach for level lighting too. Still, pre-computed lighting has many advantages, for example it can simulate radiance transfer effects much more believably, and it can be combined with dynamic lighting too, so you can safely expect pre-computed lighting to be used in games for many years to come.

Camera

We're almost done with classes derived from base Node. However, we haven't touched *Camera* at all, yet. In the previous tutorials, we set up perspective and view transforms manually. Normally, though, this is very routine operation and would be benefit from extra abstraction too. Also camera can benefit from parenting: For example in a racing game, camera could be parented to cockpit in the car, providing inside view. For performance reasons, camera is also a natural place for render-time temporary buffers and optimizations, since camera is an only object, which is directly being involved with each other object in the scene being rendered. For example our Camera class provides functionality for *view frustum culling*, which quickly rejects objects outside screen area.

The rest of the scene graph

Now we have gone through the most important classes of the scene graph used in the engine. The rest of them are listed below:

Console is used for rendering text and HUD-graphics like bitmaps to screen. Console has familiar printf style function for text rendering, and drawImage to draw textures as regular bitmap to screen. Basically Console could be used to make a 2D game, which uses 3D-accelerator hardware! Still, Console is part of the scene graph, because it can be rendered as part of 3D-scene as well.

Dummy class is an invisible box object, which is used frequently to for example mark area based trigger positions in the game.

Lines class is a list of line primitives. Lines are used usually for debug purposes, but they could be used in a game as well, for example as visual effect of laser gunshot or rain.

ParticleSystem is a particle system, usually created from script description. We get into particle system details later.

Scene is a class for loading scenes, a hierarchy of nodes, from resource files, usually exported from 3dsmax. Scene contains also some global (shader) parameters common for all nodes in the scene, like fog.

Class hierarchy of the scene graph and its relationship to primitives, shaders and textures is given in figure 4-4.

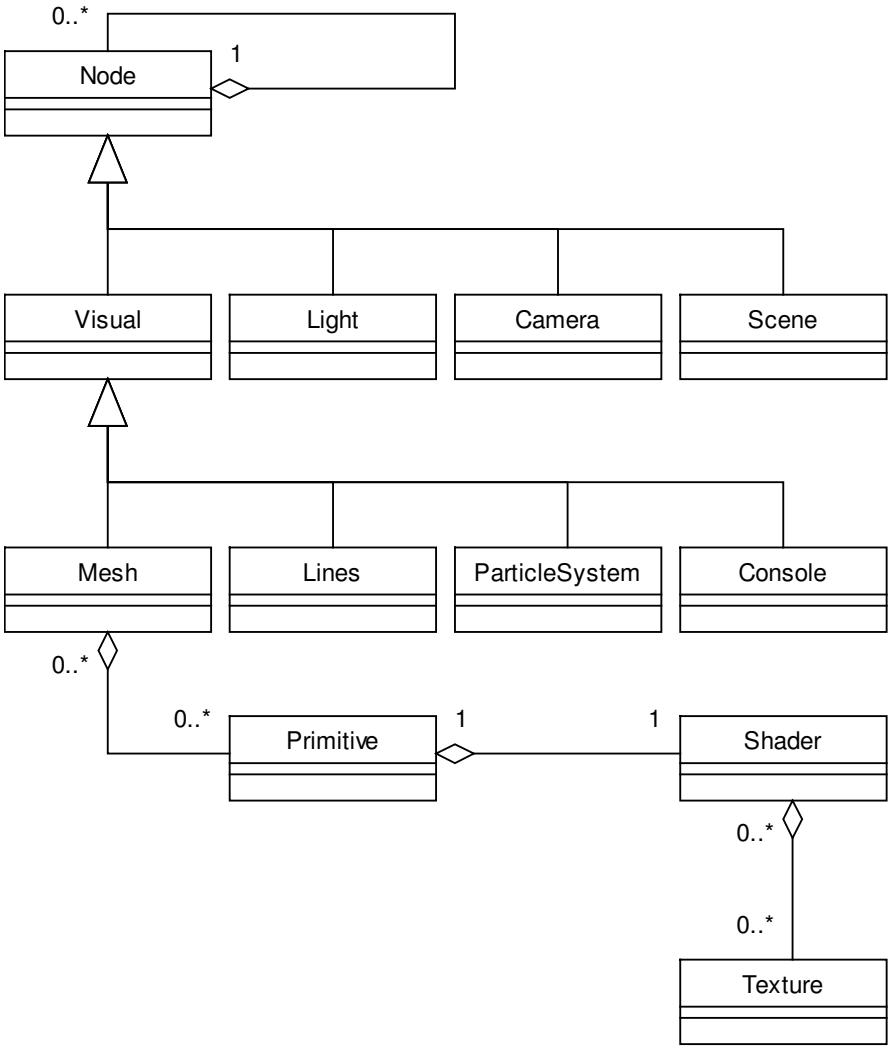

Figure 4-4. Class hierarchy of the scene graph (hgr and gr libraries)

Tutorial: Scene Graph Animation Playback

In file tutorials/scene_animation/SceneAnimApp.cpp is a simple application, which loads a scene graph exported from animated 3dsmax scene and then plays back the animation on screen real-time. The application constructor SceneAnimApp is very simple:

```
SceneAnimApp::SceneAnimApp( Context* context ) :
        m_time( 0 )
{
        // load scene file and textures from the same directory
        m_scene = new Scene( context, "data/zax.hgr" );
```

```
        m_camera = m_scene->camera();

        // print object names in hierarchy to debug output
        m_scene->printHierarchy();
}
```

First the constructor loads scene from data/zax.hgr file (hgr stands for 'hierarchical graphics') by creating a Scene class instance with the scene file name. The Scene class constructor accepts also texture path, shader path and particle path as parameter, but we can accept the defaults (empty path strings) in this case, since the textures are in the same directory as the scene file, shaders are pre-loaded by the framework from 3D-engine default shader directory and particles are not used in this example.

After loading Scene, the SceneAnimApp constructor requests a camera from the loaded scene. Usually scenes contain cameras already in the first place (to provide some meaningful view to the scene), but if the scene wouldn't have camera, Scene::camera() method would add a default camera to the scene origin, so it's safe to assume that the method returns valid camera.

Finally, SceneAnimApp constructor prints loaded scene graph hierarchy to the debug output. Debug output can be viewed either in debugger or with some external debug output viewing utility. Here's a copy of the printed hierarchy:

```
data/zax.hgr
  Camera01.Target
  Bip01 Spine
    Bip01 R Thigh
      Bip01 R Calf
        Bip01 R Foot
          Bip01 R Toe0
            Bip01 R Toe0Nub
    Bip01 L Thigh
      Bip01 L Calf
        Bip01 L Foot
          Bip01 L Toe0
            Bip01 L Toe0Nub
    Bip01 Spine1
      Bip01 Spine2
        Bip01 Neck
```

- Bip01 R Clavicle
  - Bip01 R UpperArm
    - Bip01 R ForeArm
      - Bip01 R Hand
        - Bip01 R Finger3
          - Bip01 R Finger31
            - Bip01 R Finger3Nub
        - Bip01 R Finger2
          - Bip01 R Finger21
            - Bip01 R Finger2Nub
        - Bip01 R Finger1
          - Bip01 R Finger11
            - Bip01 R Finger1Nub
        - Bip01 R Finger0
          - Bip01 R Finger01
            - Bip01 R Finger0Nub
- Bip01 L Clavicle
  - Bip01 L UpperArm
    - Bip01 L ForeArm
      - Bip01 L Hand
        - Bip01 L Finger3
          - Bip01 L Finger31
            - Bip01 L Finger3Nub
        - Bip01 L Finger2
          - Bip01 L Finger21
            - Bip01 L Finger2Nub
        - Bip01 L Finger1
          - Bip01 L Finger11
            - Bip01 L Finger1Nub
        - Bip01 L Finger0
          - Bip01 L Finger01
            - Bip01 L Finger0Nub
- Bip01 Head
  - Bip01 HeadNub

Bip01
- Bip01 Pelvis
- Bip01 Footsteps

Omni01

Camera01

Zax

Most of the objects in the hierarchy are character mesh bones (=all objects starting with 'Bip01...'). Character mesh in the scene is animated by animating the character skeleton bones and then skinning the mesh over animated bones. Character has a set of bones, which each of them influence to some subset of vertices in the mesh.

Multiple bones can influence to the same vertex, in which case transformation is defined by weighting influence of the bones. This provides crack-free bending of for example arms and legs, which would be otherwise impossible to animate with rigid objects. In practice skinning is implemented by so that the mesh has a list of bones, and each vertex has a few (2-4) bone indices to the list, and weights of each of the indexed bone. Notice how deep the hierarchy of the character skeleton is, for example finger nubs have 10 parent nodes each! It would be practically impossible to manage such complexity without having proper support of the scene graph.

In addition to mesh bones, the hierarchy contains a point light source ('Omni01'), a camera ('Camera01') and the actual mesh ('Zax'), and the Scene root node named after scene file, 'data/zax.hgr'.

SceneAnimApp::update(float dt, Context* context) function is almost as simple:

```
void SceneAnimApp::update( float dt, Context* context )
{
	// update animations
	m_time += dt;
	m_scene->applyAnimations( m_time, dt );

	// render frame
	{
		Context::RenderScene rs( context );
		m_camera->render( context );
	}

	// flip back buffer
	context->present();
}
```

First we update the animation time by adding delta time (seconds). Then we set scene key-framed animations to the specified time. Note that Scene::applyAnimations functions takes in both absolute time and delta time, since some animations might not be based on absolute timing, like particle effects animations. Let's take a closer look at Scene::applyAnimations():

```
void Scene::applyAnimations( float time, float dt )
{
	// update key frame animations
	if ( m_transformAnims != 0 )
	{
		float3x4 tm;
		for ( Node* node = next() ; node != 0 ; node = node->next() )
		{
			String name = node->name();
			TransformAnimation* tmanim = m_transformAnims->get( name );
			if ( tmanim != 0 )
			{
				tmanim->eval( time, &tm );
				node->setTransform( tm );
			}
		}
	}

	// update particles
	for ( Node* node = next() ; node != 0 ; node = node->next() )
	{
		ParticleSystem* ps = dynamic_cast<ParticleSystem*>( node );
		if ( ps )
			ps->update( dt );
	}
}
```

First, the function iterates through the scene graph without recursion by using Node::next() function, which returns the 'next' node in the hierarchy in children-first-then-siblings -order. For each node, the function requests TransformAnimation from TransformAnimationSet, a hash table containing all animations in the scene file. You might be wondering why animations are kept separate from nodes. The reason is simple: There is no simple relationship between animations and nodes. For example you might have a single scene file, which contains the character mesh, but multiple animation scene files which to load animations from. By keeping animations and nodes totally separate, we can keep memory usage close to optimal by loading *only* animations from the animation scene files.

After getting TransformAnimation for the node by name, its value, a transformation matrix, is evaluated at given time by calling TransformAnimation::eval with time and output matrix as arguments. Finally, the transform animation is set to the node by calling Node::setTransform, and so we're done with key-framed animation playback implementation as done by Scene::applyAnimations.

After key-framed animation playback, Scene::applyAnimations also updates particle systems, whose updates are based on delta time, but we don't get into particle systems update here since we have entire section devoted to them later.

After animation and time update, SceneAnimApp::update continues by rendering the frame to the rendering context. This is done by instancing Context::RenderScene class and calling Camera::render(context) function. In more complex example, we might have multiple rendering *pipes*, which perform multi-pass rendering. For example the game demo supplied on the CD uses five rendering passes:

1. Scene is rendered normally
2. Scene is rendered by blending in fog color
3. Scene *glow* effect is rendered to the second render target, glow is blurred and blur added to the top of the fogged scene
4. Particle systems and other sprites are rendered
5. HUD is rendered on top

Multi-pass rendering is achieved by using simple classes derived Pipe base class and their common shared setup class PipeSetup. But at the core of these classes, Camera::render is still used to do the actual work.

After scene has been rendered and rendering ended (at the destructor of Context::RenderScene object), the results are shown to screen at the end of

SceneAnimApp::update() by calling Context::present() function. This either flips or copies back buffer to screen, depending are we in desktop window mode or full-screen window mode.

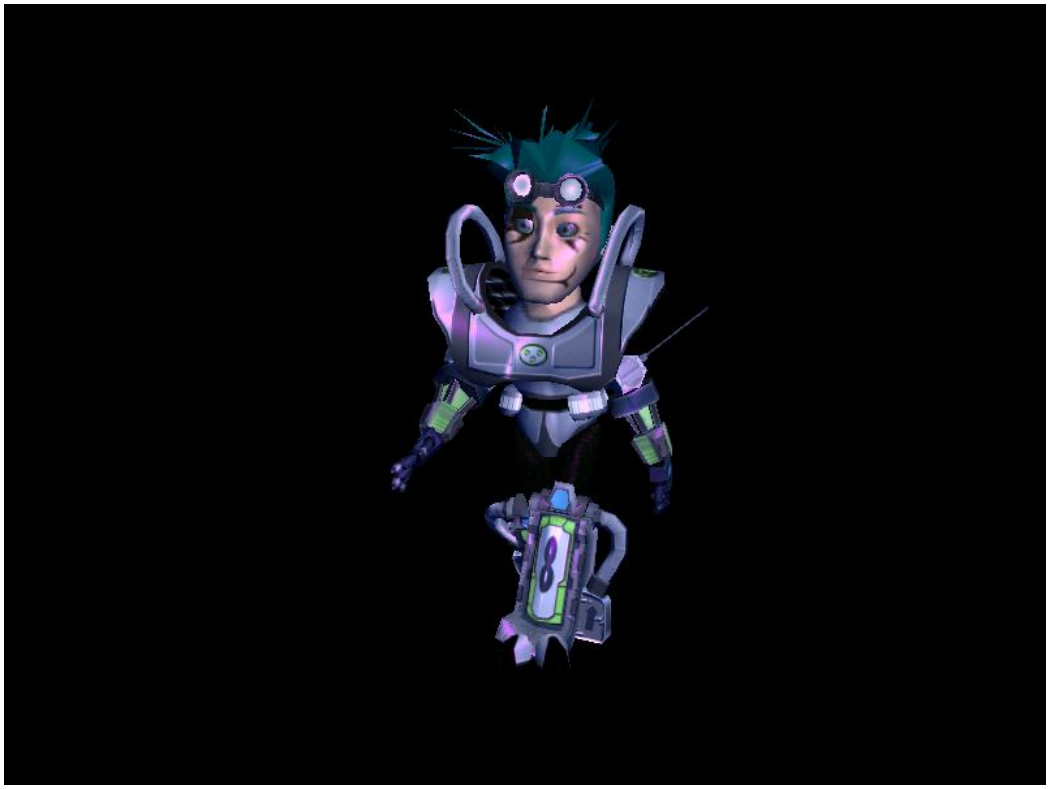

Figure 4-5. Scene graph animation playback tutorial

Tutorial: Transformation Hierarchies And Parenting

In this tutorial we load objects from one scene file, but we construct hierarchy and animate the objects manually. This is closer to actual in-game scenario, since normally you most likely do a lot more than just play a one big animation (with exception of cut-scenes, maybe).

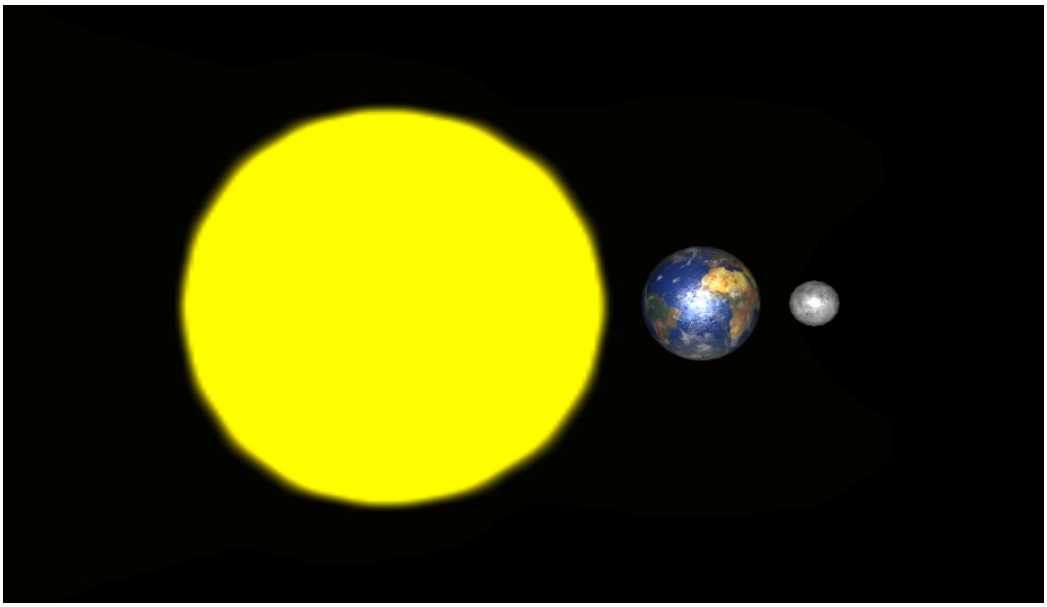

Figure 4-6. Earth is parented to Sun and Moon is parented to Earth.

First in the application constructor `HierarchyApp`, we load up the resources we'll be using for the rest of the tutorial:

```
HierarchyApp::HierarchyApp( Context* context ) :
        m_time( 0 )
{
        // get resources
        m_scene = new Scene( context, "data/space.hgr" );
        m_camera = m_scene->camera();
        m_sun = m_scene->getNodeByName( "Sun" );
        m_earth = m_scene->getNodeByName( "Earth" );
        m_moon = m_scene->getNodeByName( "Moon" );
```

So we first load the scene, and then find Sun, Earth and Moon objects from the scene by name. After this, we'll set up the hierarchies in natural manner:

```
        // set up some hierarchies (note: this hierarchy could also
        // be made already in 3dsmax, but we do it here just to show how its done)
        m_sun->linkTo( m_scene );
        m_earth->linkTo( m_sun );
        m_moon->linkTo( m_earth );
```

So Earth is child of Sun and Moon is child of Earth. We also want Sun to actually light up the other objects, so we'll add a light source to the scene as well and parent it to the sun:

```
        // parent simple point light to sun
```

```
P(Light) lt = new Light;
lt->setColor( float3(1,1,0) );
lt->linkTo( m_sun );
```

Now we're almost done with the scene setup, but we need still to initialize camera a bit further away from the action:

```
// put camera looking at sun
// (don't try this at home)
m_camera->setPosition( float3(2,5,-16) );
m_camera->lookAt( m_sun );
```

In 3dsmax, we have applied bump-glow.fx shader to the Sun, so we need to setup the rendering pipeline, which supports this effect as well. Normally calling Camera::render is like shortcut for using DefaultPipe rendering pipeline, but now since we need other pipe as well, we create this DefaultPipe explicitly along with GlowPipe and PipeSetup. PipeSetup contains the shared rendering setup code (frustum culling, etc.), which needs to be done only once for all pipes to be used in rendering. Actual setup code is very simple:

```
// setup extra rendering pipeline for glow
m_pipeSetup = new PipeSetup( context );
m_defaultPipe = new DefaultPipe( m_pipeSetup );
m_glowPipe = new GlowPipe( m_pipeSetup );
```

HierarchyApp::update(), called every frame during the sample is active, starts by updating time as the previous samples. Then it sets the transformations of the Sun, Earth and Moon -objects:

```
// put sun to center and rotate it about Y-axis
float sunangle = m_time * Math::toRadians(45.f);
m_sun->setPosition( float3(0,0,0) );
m_sun->setRotation( float3x3(float3(0,1,0), sunangle) );

// put earth 25 units away from sun and rotate it twice as fast
float earthangle = m_time * Math::toRadians(90.f);
m_earth->setPosition( float3(25,0,0) );
m_earth->setRotation( float3x3(float3(0,1,0), earthangle) );

// put moon 10 units away from earth and rotate it twice as fast
float moonangle = m_time * Math::toRadians(180.f);
m_earth->setPosition( float3(10,0,0) );
m_earth->setRotation( float3x3(float3(0,1,0), moonangle) );
```

Notice that each object is set to *constant* position, but due to the rotation of their parents, the objects move in world space. Very convenient for us, as this way we can make complex animations with simple code.

Finally, we render the pipelines to default render target (which is rendering context) and flip back buffer to screen:

```
// render frame
{
	Context::RenderScene rs( context );
	m_pipeSetup->setup( m_camera );
	m_defaultPipe->render( 0, context, m_scene, m_camera );
	m_glowPipe->render( 0, context, m_scene, m_camera );
}

// flip back buffer
context->present();
```

You should experiment a bit here. You can for example try to disable glow effect by commenting out // the line 81, m_glowPipe->render(), which actually renders the glow effect.

Glow Post-Processing Effect Implementation Details

How exactly the glow effect is done, then? It's actually relative simple, see GlowPipe::render() implementation in hgr/GlowPipe.cpp:

First the scene is rendered to texture using GLOW technique as defined by the shaders used in the scene:

```
// render scene to texture
context->setRenderTarget( m_rtt );
context->clear();
setup->setTechnique( "GLOW" );
camera->render( context, 0, 100, setup->visuals, setup->priorities, &setup->lights );
```

This technique is opaque black for non-glowing objects (so that non-glowing objects block the glow from objects further away), and the glow color scaled by glow strength for glowing objects. Glow color comes from glow texture and glow strength comes from alpha channel of normal map. Note that alpha channel of base map

cannot be used since it's already in-use for controlling 'glossiness' of the specular highlight in bump-glow.fx shader:

```
half4 psGlow( PS_IN IN ) : COLOR
{
        half4 colorMap = tex2D(colorTex, IN.uv0);
        half4 N = tex2D(normalTex, IN.uv1);
        half3 c = colorMap.xyz * N.w;
        return half4( c, 1 );
}
```

Now that we have the glow rendered to a render-target texture, we use it as source texture for blurring. Actual blurring is done by two pixel shaders, which sample four neighboring texels each and blend them by weights to the final color:

```
half4 psBlur0( PS_IN p ) : COLOR
{
        half3
        c =  tex2D( basetex, p.uv0 ).xyz * BLUR_BRIGHTNESS * 0.165256;
        c += tex2D( basetex, p.uv1 ).xyz * BLUR_BRIGHTNESS * 0.160336;
        c += tex2D( basetex, p.uv2 ).xyz * BLUR_BRIGHTNESS * 0.15093;
        c += tex2D( basetex, p.uv3 ).xyz * BLUR_BRIGHTNESS * 0.137845;
        return float4( c, 1 );
}

half4 psBlur1( PS_IN p ) : COLOR
{
        half3
        c =  tex2D( basetex, p.uv0 ).xyz * BLUR_BRIGHTNESS * 0.122128;
        c += tex2D( basetex, p.uv1 ).xyz * BLUR_BRIGHTNESS * 0.10499;
        c += tex2D( basetex, p.uv2 ).xyz * BLUR_BRIGHTNESS * 0.0875721;
        c += tex2D( basetex, p.uv3 ).xyz * BLUR_BRIGHTNESS * 0.0708685;
        return float4( c, 1 );
}
```

Weights have been calculated off-line from Gauss curve to provide smooth shape for the blurring. Due to Pixel Shader 1.1 hardware limitations (we want to stay compatible with GF3 and GF4 generation 3D-accelerators!), we need to do the blurring separately in horizontal and vertical directions, and both twice since we want to use 8 samples per direction and we can use only 4 texture samplers in one pass. This causes the blur rendering to be done at the end 8 times! Luckily this is relatively

light-weight operation, since we're only rendering a single overlay quad from texture to another:

```
// horizontal blur (to texture)
context->setRenderTarget( m_rttBlurH );
renderOverlay( m_overlay, m_blurH, m_rtt );

// vertical blur (to texture)
context->setRenderTarget( m_rttNewBlur );
renderOverlay( m_overlay, m_blurV, m_rttBlurH );
```

After the glow has been blurred to both directions, blur from the previous rendered frame is combined with the current one. Combining is done by 'trail' pixel shader, which multiplies both colors of the new blurred glow and the glow of the previous frame by constant coefficients and adds them together:

```
// combine faded old blur to the new blur (to texture)
context->setRenderTarget( m_rttBlur );
m_trail->setTexture( "BASEMAP1", m_rttNewBlur );
m_trail->setTexture( "BASEMAP2", m_rttOldBlur );
renderOverlay( m_overlay, m_trail );
```

The trail pixel shader code goes as follows:

```
half4 pshader( PS_IN p ) : COLOR
{
	half4 c1 = tex2D( baseTex1, p.uv0 );
	half4 c2 = tex2D( baseTex2, p.uv1 );
	return c1*LEVEL1 + c2*LEVEL2;
}
```

This causes the nice motion blur effect that old glow slowly fades away over multiple rendered frames.

Finally, the combined blur is combined with the normally rendered:

```
// add blur to actual target
context->setRenderTarget( target );
renderOverlay( m_overlay, m_add, m_rttBlur );
```

The combining is done simple by returning texture color from the pixel shader:

```
half4 pshader( PS_IN p ) : COLOR
{
	return tex2D( basetex, p.uv0 );
}
```

And enabling additive blending in the technique definition at the end of pp-add.fx file:

```
technique Default
{
	pass P0x
	{
		VertexShader = compile vs_1_1 vshader();
		PixelShader = compile ps_1_1 pshader();
		CullMode = NONE;

		SrcBlend = ONE;
		DestBlend = ONE;
		AlphaBlendEnable = TRUE;

		ZWriteEnable = FALSE;
		ZEnable = FALSE;
	}
}
```

At the end of GlowPipe::render() function, new blur texture is swapped with the old one so the new blur will become old blur in the next frame's blur rendering.

Tutorial: Character Animation

In this a bit more advanced tutorial, we take a look at character animation system. What is character animation system, exactly? We have already shown you have to play animations, right? Well, when you have a character with multiple animations the issue is a bit trickier. Consider for example simple walking forward movement:

1. First, character is standing still.
2. Character starts walking forward, so we need animation *transition* from standing to walking.
3. While character is walking, we need to move the character position in the game world. However, we *cannot* simply set linear velocity for the movement, because most movements like walking are not exactly linear, so using constant velocity would cause slipping of the feet, which in turn

looks bad. To cure this problem we'd need to *extract movement speed* from the animation position key frames.

4. When the character stops, we need again transition to standing animation. Also this transition needs to take into account that the speed of the character is slowing down, so we need actually make a transition for the speed as well.

This example of walking should convince you that it's not exactly walk in the park to implement even walking with animated character. So how do we tackle this problem in more general case?

First we need to think about the *states* where the character can be. These states might match to the animation list, but not necessarily. For example you might use a single *idle* state and have multiple animations associated with it. In our case, however, we decide to map our character animations directly to states and define to CharacterAnimationApp class declaration enum State:

```
enum State
{
        STATE_STAND,
        STATE_WALK_FORWARD,
        STATE_STEP_LEFT,
        STATE_STEP_RIGHT,
        STATE_JUMP_UP,
        STATE_JUMP_OUT,
        STATE_IDLE1,
        STATE_IDLE2,
        STATE_COUNT
};
```

As we have direct 1-1 mapping between states and animations, we have direct array indexing between State and TransformAnimationSet associated with each State. We initialize the character mesh and the animations in CharacterAnimationApp constructor:

```
CharacterAnimationApp::CharacterAnimationApp( Context* context ) :
        m_time( 0 ),
```

```
        m_state( STATE_STAND ),
        m_transitionState( TRANSITION_NONE ),
        m_transitionTarget( STATE_STAND ),
        m_transitionTime( 0 ),
        m_transitionStart( 0 ),
        m_transitionLength( 0 ),
        m_idleAnimIndex( 1 )
{
        // setup scene, key light and camera
        m_scene = new Scene;
        P(Light) lt = new Light;
        lt->linkTo( m_scene );
        lt->setPosition( float3(0,3,0) );
        m_camera = new Camera;
        m_camera->linkTo( m_scene );

        // setup character mesh
        m_character = new Scene( context, "data/mesh/zax_mesh.hgr" );
        m_character->linkTo( m_scene );

        // load character animations
        m_anims[STATE_STAND] =
        loadCharacterAnimation( context, "data/anim/idle_stand.hgr" );

        m_anims[STATE_WALK_FORWARD] =
        loadCharacterAnimation( context, "data/anim/walk_frwd.hgr" );

        m_anims[STATE_JUMP_UP] =
        loadCharacterAnimation( context, "data/anim/jump_up.hgr",
                TransformAnimation::BEHAVIOUR_STOP );

        m_anims[STATE_JUMP_OUT] =
        loadCharacterAnimation( context, "data/anim/jump_out.hgr",
                TransformAnimation::BEHAVIOUR_STOP );

        m_anims[STATE_STEP_LEFT] =
        loadCharacterAnimation( context, "data/anim/sidestep_left.hgr",
                TransformAnimation::BEHAVIOUR_STOP );

        m_anims[STATE_STEP_RIGHT] =
        loadCharacterAnimation( context, "data/anim/sidestep_right.hgr",
                TransformAnimation::BEHAVIOUR_STOP );

        m_anims[STATE_IDLE1] =
        loadCharacterAnimation( context, "data/anim/idle_look_around.hgr" );

        m_anims[STATE_IDLE2] =
```

```
        loadCharacterAnimation( context, "data/anim/idle_stretch.hgr" );

        // setup debug line grid
        P(Lines) linegrid = createXZGrid( context, 30.f, 1.f );
        linegrid->linkTo( m_scene );

        // setup rendering pipelines
        m_pipeSetup = new PipeSetup( context );
        m_defaultPipe = new DefaultPipe( m_pipeSetup );
        m_glowPipe = new GlowPipe( m_pipeSetup );
}
```

In loadCharacterAnimation, we first load the animation file, then make sure its aligned correctly to the character, and finally scale inches to meters as inches are the most natural unit to work with characters in 3dsmax, but we want to keep everything in metric system to keep it consistent. Alignment check is done by checking the direction of Bip01 bone, which is used in 3dsmax as base node of the character skeleton. This alignment check is of course not a requirement if you have uniform animation data:

```
P(TransformAnimationSet) CharacterAnimationApp::loadCharacterAnimation(
        Context* context, const String& filename,
        TransformAnimation::BehaviourType endbehaviour )
{
        // load animation set
        P(Scene) scene = new Scene( context, filename );
        P(TransformAnimationSet) animset = scene->transformAnimations();
        assert( animset != 0 );

        // make sure root bone Bip01 points to forward
        TransformAnimation* anim = animset->get( "Bip01" );
        assert( anim );
        float3x4 tm;
        anim->eval( 0.f, &tm );
        if ( tm.rotation().getColumn(0).z > 0.f )
                animset->rotate( m_character, float3x3(float3(0,1,0), Math::PI) );

        // character animations have been created in inch units
        // so scale them to metric system
        const float INCH_M = 0.02540000508f;
        animset->scale( m_character, float3(1,1,1)*INCH_M );

        animset->setEndBehaviour( endbehaviour );
```

```
        return animset;
}
```

Default end behavior for an animation is 'repeat'. The default is good for example to animations like walking and running, which definitely need to repeat after ending. On the other hand some animations (jump, sidestep) specifically require that they are not repeated automatically, so we need to let animation system know this by setting TransformationAnimationSet end behavior explicitly.

So far so good; we have now character states defined and a correctly aligned animation associated with each state. Next we need to think about animation and state transitions.

We define two transition modes, namely *none* and *blending*. *None* is used when only one animation is playing and *blending* is used when two animations are playing. In general case, we might have n animations blending at the same time, since the blend target state might not be achieved before the next blend starts (for example user might be tapping left and right -buttons quickly), but we can exclude this case by agreeing that no new transitions can start when before old transition is done. This might seem restricting, but in practice blending n arbitrary animations would lead to bad visual artifacts anyway, since for example we couldn't be sure that hand doesn't intersect with body when we are animating the hand movement using multiple source animations. Transition mode is used to select active animations in the start of CharacterAnimationApp::animateCharacter function:

```
void CharacterAnimationApp::animateCharacter( float dt )
{
        m_time += dt;
        m_transitionTime += dt;

        // select active animations
        TransformAnimationSet* animlist[2] = {0,0};
        float animweights[2] = {0,0};
        float animtimes[2] = {0,0};
        int animsets = 0;
```

```
switch ( m_transitionState )
{
case TRANSITION_NONE:
        animlist[animsets] = getAnimation( m_state );
        animweights[animsets] = 1.f;
        animtimes[animsets] = m_time;
        animsets = 1;
        break;

case TRANSITION_BLEND:
        animlist[animsets+1] = getAnimation( m_transitionTarget );
        animlist[animsets] = getAnimation( m_state );

        animweights[animsets+1] = (m_transitionTime - m_transitionStart)
                / m_transitionLength ;
        animweights[animsets] = 1.f - animweights[animsets+1];

        animtimes[animsets+1] = m_transitionTime;
         animtimes[animsets] = m_time;

        animsets = 2;

        if ( m_transitionTime >= m_transitionStart+m_transitionLength )
        {
                Debug::printf( "Transition done: %s (t=%g)\n",
                        toString(m_transitionTarget), m_transitionTime );

                m_state = m_transitionTarget;
                m_time = m_transitionTime;
                m_transitionState = TRANSITION_NONE;
        }
        break;
}
```

The TRANSITION_NONE case is easy; we just call getAnimation(state), which returns animation directly from the array indexed by character state. We could have also used m_anims array directly, but getAnimation() has some error checking functionality (namely, it checks that the animation has been defined), so we use the function instead. In TRANSITION_NONE state, the animation weight is naturally 1 (there is only a single animation), and it's time is the time how long state has been active.

In TRANSITION_BLEND the situation is not much more complicated: We blend using linear interpolation between two animations, based on how long the transition is and what part of the transition is still left. When a transition starts, old animation is fully in effect and weight of the new animation (m_transitionTarget) is 0, and when transition reaches its end, m_transitionTarget animation weight has reached 1. When the transition has reached end, transition target is set as the new active state (including state time), and transition mode is reverted back to TRANSITION_NONE, ready for waiting for the next transition.

After we know the active animation sets and their weights, we can apply them to the nodes:

```
// update node transforms from animations
for ( Node* node = m_character ; node != 0 ; node = node->next() )
        TransformAnimationSet::blend( animlist, animtimes,
                animweights, animsets, node  );
```

TransformAnimationSet::blend takes in a list of animation sets, their animation times, blending weights, number of animation sets in the lists and the node which to apply the blended output. Note that this function performs n animation blending, so it is not limited to two animation sets used here, if we for some reason would feel need to blend more animations at some point.

Next we calculate how much we should actually move the character in the world based on the animation. This is done by calculating Bip01 bone movement velocity from the distance moved between successive key frames. We need to weight this value as well, since we might have multiple animations sets in a transition:

```
// calculate character speed from animation of Bip01 bone
float3 localvelocity(0,0,0);
Node* bipnode = m_character->getNodeByName( "Bip01" );
assert( bipnode );
for ( int i = 0 ; i < animsets ; ++i )
{
        TransformAnimation* a = (*animlist[i])[ bipnode->name() ];
        if ( a != 0 )
```

```
        {
            float weight = animweights[i];
            localvelocity +=
                a->getLinearVelocity( animtimes[i] ) * weight;
        }
    }
```

Note that without this movement speed weighting character feet would slip due to incorrect velocity during character state transitions.

Now that we know the velocity, we need to negate the effect of the moving character in mesh by moving all top-level nodes in character scene with negative offset calculated from Bip01 position:

```
// compensate movement extraction from Bip01 node
// by translating top level nodes
const float biplevel = bipnode->position().y + .01f;
float3 pos = -bipnode->position();
pos.y += biplevel;
for ( Node* n = m_character->firstChild() ; n != 0 ;
        n = m_character->getNextChild(n) )
{
        n->setPosition( n->position() + pos );
}
```

This has the cause that character mesh effectively stands still when we eventually apply the movement in *game world*. If we wouldn't do this then the character would be moved *twice*, and we really need to keep the mesh in center of the game world object anyway, since otherwise we would have hard time keeping track of correct game object positions for example while doing collision checking.

Next, since we don't have any collision checking in use in this example, we keep the character object on ground if he's not jumping:

```
// make sure we're on ground if not jumping
if ( !isJump(m_state) && !isJump(m_transitionTarget) )
{
        float3 pos = m_character->position();
        pos.y = 0.f;
        m_character->setPosition( pos );
}
```

When we later in the actual game demo have collision checking working, things don't get much more complicated since we have kept the character mesh in the origin of the game object local space. This way we can just add character-world collision checking using capped cylinder in place of the character, and map that cylinder position directly to the character mesh position.

Next, we'll rotate the character 90 degrees/second based on user input:

```
// update character rotation
float3x3 rot = m_character->rotation();
float angle = Math::PI * dt * .5f;
if ( isKeyDown(KEY_LEFT) )
        rot = rot * float3x3(float3(0,1,0),-angle);
if ( isKeyDown(KEY_RIGHT) )
        rot = rot * float3x3(float3(0,1,0),angle);
```

In actual game, we'd probably add turning left/right as animation states as well, but here we simply rotate the character rotation.

Next we make sure the character doesn't fall to his nose while jumping and walking around, in other words character rotation Y-axis points up:

```
// orthonormalize rotation so that character doesn't tilt
float3 xaxis = rot.getColumn(0);
float3 yaxis = rot.getColumn(1);
xaxis.y = 0.f;
yaxis.x = 0.f;
yaxis.z = 0.f;
rot.setColumn( 0, xaxis );
rot.setColumn( 1, yaxis );
rot = rot.orthonormalize();
m_character->setRotation( rot );
```

This is very important not to forget, since even though the character would be perfectly aligned upwards in the animations, calculations can drift pretty quickly due to floating point inaccuracy issues.

Then we update character position in the world based on the velocity we calculated from the animation data earlier:

```
// update character position by world space linear velocity
float3 worldvelocity = rot.rotate( localvelocity );
```

```
worldvelocity.y = 0.f;
m_character->setPosition( m_character->position() + worldvelocity*dt );
```

Next we evaluate possible new target state, and trigger transition blending if the target has changed:

```
// update character state
if ( TRANSITION_NONE == m_transitionState )
{
	evaluateTransitionTarget();

	// start transition if changed
	if ( m_transitionTarget != m_state )
	{
		Debug::printf( "%s -> %s (t=%g)\n", toString(m_state),
			toString(m_transitionTarget), m_time );

		m_transitionStart = m_time;
		if ( needsTimeReset(m_transitionTarget) )
			m_transitionStart = 0.f;

		m_transitionLength = 0.30f;
		m_transitionTime = m_transitionStart;
		m_transitionState = TRANSITION_BLEND;
	}
}
```

Notice that some target states require time reset, for example when you step left the stepping animation needs to always start from the beginning. On the other hand it's equally important that the time is *not* reset for some states, like when you would transition between walking and running, it would look very weird if the feet would go to default pose every time transition is done.

Actual target state selection is done by evaluateTransitionTarget(). The function uses current character state and user input to select desired target character state in current situation:

```
void CharacterAnimationApp::evaluateTransitionTarget()
{
	m_transitionTarget = m_state;

	if ( STATE_WALK_FORWARD == m_state )
	{
		// walk state update
```

```
                if ( !isKeyDown(KEY_UP) && !isKeyDown(KEY_W) )
                        m_transitionTarget = STATE_STAND;
                if ( isKeyDown(KEY_SPACE) )
                        m_transitionTarget = STATE_JUMP_OUT;
                if ( isKeyDown(KEY_A) )
                        m_transitionTarget = STATE_STEP_LEFT;
                if ( isKeyDown(KEY_D) )
                        m_transitionTarget = STATE_STEP_RIGHT;
        }
        else if ( STATE_STAND == m_state || isIdle(m_state) )
        {
                // trigger new idle animation if more than 4 seconds elapsed
                if ( m_transitionTime-m_transitionStart > 4.f &&
                        isAnimationEnd(0) )
                {
                        m_transitionTarget = selectIdle();
                }

                // idle state update
                if ( isKeyDown(KEY_UP) || isKeyDown(KEY_W) )
                        m_transitionTarget = STATE_WALK_FORWARD;
                if ( isKeyDown(KEY_SPACE) )
                        m_transitionTarget = STATE_JUMP_UP;
                if ( isKeyDown(KEY_A) )
                        m_transitionTarget = STATE_STEP_LEFT;
                if ( isKeyDown(KEY_D) )
                        m_transitionTarget = STATE_STEP_RIGHT;
        }
        else if ( isJump(m_state) )
        {
                // jump state update
                if ( isAnimationEnd(0.1f) ) //  state re-evaluation needed?
                        m_transitionTarget = STATE_STAND;
        }
        else if ( isStep(m_state) )
        {
                // step state update
                if ( isAnimationEnd(0.3f) ) //  state re-evaluation needed?
                        m_transitionTarget = STATE_STAND;
        }
}
```

Now that we're done animating the character, let's take a look at how to animate the camera in animateCamera() function:

```
void CharacterAnimationApp::animateCamera( float dt )
{
        // update camera position based on character transform
```

```
float3x4 targettm = m_character->transform();
float3 cameradistv( 0.f, 2.5f, 3.f );
m_camera->setPosition( targettm.transform(cameradistv) );
m_camera->lookAt( targettm.translation() + float3(0,1.5f,0) );
```

This provides the basic following movement for the camera. So first camera is set to a position (0,2.5,3) relative to character transformation, then camera is tilted so that it looks to character head, which is about 1.5m above feet. Next we handle the idle state special camera angle and we're done:

```
// set idle camera position
if ( isIdle(m_state) && m_state != STATE_STAND )
{
        m_camera->setPosition( targettm.transform(float3(0,0.2f,-2)) );
        m_camera->lookAt( targettm.transform(float3(0,1.5f,0)) );
}
```

That's it! There are a couple of more helper functions, which we haven't taken a look at, but we can ignore them here since their function is relatively self-evident and they don't have much to do with character animation anyway, for example createXZGrid creates the debug line grid shown below character's feet. Now you should start playing around with the example application. When you feel more comfortable with the source code and how it works, you could for example try to add new animations to the application -- there is plenty of them on the CD shipped with this book.

Separate upper body animation, aiming and shooting

In the game demo shipped with this book, we'll add a couple of more features to the character animation system presented here: Most importantly, we add separate *upper body animation*, which is quite like the body animation state, but applies only to the upper body of the character. We need this kind of separate upper body state, because for example aiming and shooting require that the character can perform other activities at the same time, for example character can walk and run while shooting.

Notice that you cannot escape this requirement by using complete 'shooting and walking' complete animation, because you cannot know when the player wants to shoot or aim beforehand, and the cycle of that animation does not match to the walking animation.

Luckily having this complication does not make things much more difficult. We just need to keep track of the upper body state, and apply related animations only to it like this:

```
// apply upper body animation
Node* bipspine1 = m_root->getNodeByName( "Bip01 Spine1" );
for ( Node* node = m_root ; node != 0 ; node = node->next() )
{
        if ( node->hasParent(bipspine1) )
        {
                TransformAnimationSet::blend( anims, upperanimtimes,
                        upperanimweights, animsets, node );
        }
}
```

Code above applies specified animation set only to the bones, which are children or grand children of 'Bip01 Spine1'. Notice that we detect is specific bone part of upper body by comparing it to the root of the upper body, usually 'Bip01 Spine1' in 3dsmax Character Studio models.

There is one additional complexity with upper body animation implementation: We need to take into account that the character can potentially be aiming/shooting upwards or downwards. This can be handled by keeping separate sets of up/front/down animations, and then blending between them accordingly. First, we calculate the blend between front and up/down animation from the character aim pitch angle (in radians, limited to range [-1,1] to prevent extreme tilting):

```
// find out blend between up/front/down
float pitch = clamp( m_pitch, -1.f, 1.f );
float u1 = Math::abs(pitch);
float u0 = 1.f - u1;
```

Now u0 is the weight of the 'front' animation and u1 is the weight of the up/down animation. Now, let's assume that we are performing shooting animation. Next we calculate the blend between animation sets:

```
float upperanimweights[4];
float upperanimtimes[4];
TransformAnimationSet* anims[4];
int animsets = 0;

anims[0] = m_upperBodyAnims[ UPPERBODY_SHOOT ][VERT_FRONT];
upperanimweights[0] = u0;
upperanimtimes[0] = animtime;
++animsets;

if ( m_pitch < 0 )
        anims[1] = m_upperBodyAnims[ UPPERBODY_SHOOT ][VERT_UP];
else
        anims[1] = m_upperBodyAnims[ UPPERBODY_SHOOT ][VERT_DOWN];
++animsets;
```

This way character is aiming/shooting up/down according to his pitch angle. This naturally isn't too accurate what it comes to aiming with actual weapon, so we could improve the aiming still by aligning weapon to the aim target.

```
float3x4 tm = handtm;
float3 aimz = normalize ( target - handtm.translation() ) );
float3 weaponz = -handtm.getColumn(0);
float angle = Math::acos( dot (weaponz,aimz) );
if ( Math::abs(angle) > 0.001f )
{
        float3x3 rot(normalize(cross(weaponz,aimz)), angle );
        tm.setRotation( rot * tm.rotation() );
}
```

In the above code example, it is assumed that the hand's (holding weapon) transformation negated X-axis points to the direction the weapon is pointing. This may sound weird assumption, but you cannot always control what kind of data is coming from the modeling program, so in practice you need to conform to this kind of assumptions a bit. Actual alignment above is done by computing angle between ideal aiming direction (aimz) and actual weapon direction (weaponz), and then rotating weapon by that angle towards the right direction.

Tutorial: Rigid Body Physics Simulation

You might be wondering what physics simulation is doing as graphics tutorial. Well, one way to look at physics is as a tool for animation -- people majored in physics probably view the issue other way around.

Also when doing especially simulations, it is common issue that everything doesn't go as expected. In that case it's very useful to be able to put some run-time text on screen to find out what's going on, so we also take a look at the methods available in the engine for that purpose.

Simulation basics

Simulation *world* is a container for rigid *bodies* and *joints* (or *constrains*). Rigid body is the actual object being simulated and *joint* limits relative movement between bodies. Rigid bodies have following properties, which change over time:

- Position (3-vector)
- Linear velocity (3-vector)
- Orientation (3x3 rotation matrix)
- Angular velocity (3-vector)

And following constant properties:

- Mass
- Center of mass (in body's local coordinate space)
- Inertia matrix (distribution of mass in the body, in local space)

Note that rigid bodies do *not* have geometry. Instead, the simulator asks for *contacts* from the collision system. Orthogonal concept to the simulation world is the collision *space*, which contains a set of *geometry* objects. Geometry objects do not know anything about the simulation. Geometry objects' only functionality is to

resolve collisions and contacts between each other. This approach allows independence of the simulator from the used collision system. We have only one kind of body, but we might have any number of geometry object types, for example:

- Sphere
- Box
- Capped cylinder (=capsule)
- Cylinder
- Plane
- Ray
- Triangle mesh

In this case the generality is only available for the user -- since any geometry can potentially collide with any geometry user defines, the collision checking library would need to implement 6! = 720 different types of collision checks with the geometry object types listed above!

In addition to rigid bodies, the simulation world has *joints*. Joint is a relationship between two or more bodies -- it *constraints* movement between the bodies. Simulator works by trying to satisfy these constraints. There can be various constraints, for example:

- Ball joint connects two bodies so that the other one works as 'socket' and the other one as ball.
- Hinge joint connects two bodies as door is connected to wall (expect that in this case the wall isn't necessarily stationary).
- Slider joint connects two bodies so that they can slide along a common slider axis shared by the bodies.

- Contact joint prevents two bodies of penetrating each other. In practice this works so that the contact joints try to keep bodies having velocity direction only away from the surfaces they are in contact with.

Ok, now we have defined world, body, joint, space and geometry. The actual simulation setup with 3D rendering is done in following way:

- Load (or otherwise construct) a visual 3D scene
- Create dynamics world
- Create rigid body geometry objects in the world from the visual scene, for example by creating rigid bodies with box geometry from meshes' bounding boxes (you probably want to use some tag to indicate which mesh to use as source for the rigid body, since not *all* visual objects are usually simulated)
- Set state (position, rotation, etc.) of the rigid bodies from the visual mesh transformations.
- Create and attach joints (if any) to the bodies
- Set the parameters of all joints
- Create a joint group to hold (temporary) contact joints

And the mainloop goes like this:

1. Apply forces to the bodies (if any)
2. Adjust joints parameters (if any)
3. Do collision detection
4. Collect contacts from collision detection system to the contact joint group
5. Do a simulation step
6. Remove all contacts from the contact joint group
7. Update visual meshes' transformations from rigid body transformations

8. Render visual meshes

Text rendering (debug info and in-game HUD)

Everyone is familiar with printf function usage. Its not much use in graphics applications, but luckily we have hgr::Console class in the 3D engine and its function Console::printf, which is almost as simple to use. Only difference to the standard printf is that with Console you need to flush the text to get output visible by rendering the console.

Let's have a look at the source code tutorials/physics_simulation directory. Before initializing the simulation, we initialize a text rendering console in PhysicsApp constructor:

```
// create text output console (using Comic Sans MS font)
m_console = new Console( context,
        context->createTexture("data/comic_sans_ms_20x23.png"), 20, 23 );
```

The code creates hgr::Console object, using specified comic_sans_ms_20x23.png texture. Texture is itself fairly simple, it's 16x16 table of ASCII characters. The constructor takes in also individual character width and height so that it can perform the calculations where each character bitmap can be found. For example ASCII code of 'A' is 0x41, which means that the character bitmap is second from the left in the fourth row. This code might look bad to you from internationalization viewpoint, but in fact the Console supports full range of Unicode characters: You can add to Console any number of this kind of *character pages*, which define a 16x16 set of characters. If you need for example Unicode code points in range 0x1000-0x1100, you could add fonts of character page number 32, which would contain the specified 16x16 set. Strings themselves are treated as UTF-8 encoded, so full Unicode code point range is at your usage!

Text rendering is of course very useful for example to create a game UI, but it is very useful for debug purposes as well. For example to keep your simulations performance-friendly, you should always try to keep number of enabled bodies (*enabled* means that they are participating in the simulation) as low as possible. So its very useful to know how many active bodies you have running all the time. This kind of information doesn't really fit to some debug log, since log files and outputs are usually viewed only after something has gone wrong. And you most definitely don't want to swamp your log file by dumping information in every frame anyway! Having debug output screen directly for example over the game HUD is much more useful.

Actual text rendering is done in PhysicsApp::renderDebugInfo() function. First text console position is set to top-left corner of the screen, and then the text lines are printed using familiar-looking printf() syntax statements:

```
void PhysicsApp::renderDebugInfo( Context* context, float fps )
{
	// set text console origin to top-left corner
	m_console->setPosition( float3(0,0,0) );

	// show UI
	m_console->printf( "F5 resets\n" );

	// show number of active bodies in simulation
	m_console->printf( "active bodies: %d\n",
		m_world->enabledBodies() );
```

Notice that you can change the origin of the console also later, since the text itself is not rendered to screen until Console::render is called. But before that, we print out some function performance profiling statistics (function name, % of used execution time and call count):

```
	// render info about profiled blocks
	for ( int i = 0 ; i < Profile::blocks() ; ++i )
	{
		m_console->printf( "%-16s = %5.1f %% (x%d)\n", Profile::getName(i),
			Profile::getPercent(i), Profile::getCount(i) );
	}
```

```
Profile::reset();
```

By adding PROFILE(functionName) macro to start of a function (see for example PhysicsApp::simulate() implementation below), we get performance statistics of that function recorded by our profiler. The macro records number of times the block was executed and the time spend executing it. Profile::getPercent() is scaled by the time spend in whole frame, which is to say time spend between Profile::beginFrame() and Profile::endFrame() calls. You probably still want to use some external full profiling tool like Intel VTune or AMD CodeAnalyst every now and then (for example to get a call graph), but this simple and very light-weight profiler can give you useful run-time information about your code performance without causing practically any overhead.

Simulation in practice

Now let's move to the actual simulation code. The tutorial shows you how to make a simple simulation from a scene created in 3dsmax as outlined in the *Simulation basics* section.

PhysicsApp constructors calls restart(). The same function is used also when the user pressed F5 to reset simulation. This is convenient since by sharing the init and reset code, we can make sure that *both* features work properly. The function itself is very simple:

```
void PhysicsApp::restart()
{
	// (re)start time
	m_time = 0.f;

	// (re)load scene to be simulated
	m_scene = new Scene( m_context, "data/scene.hgr" );
	m_camera = m_scene->camera();

	// (re)start simulation
	initSim();
}
```

First time is reset, then scene is loaded and finally simulation world is initialized. initSim() starts by removing old objects (if the application was already initialized) and create the simulation world and space:

```
void PhysicsApp::initSim()
{
	m_objects.clear();
	m_world = new ODEWorld;
```

ODEWorld is a simple wrapper that combines simulation world and main collision space. ODEWorld constructor sets also most commonly needed default values for our simulation, for example gravity is set as -9.8 (m/s^2) along world space Y-axis. You can change gravity with ODEWorld::setGravity() later if you want, for example if your game is happening on the Moon surface your gravity would need to be less than the default.

After adjusting the parameters we loop through the loaded scene nodes and find meshes:

```
	// create simulation objects from meshes
	for ( Node* node = m_scene ; node != 0 ; node = node->next() )
	{
		Mesh* mesh = dynamic_cast<Mesh*>( node );
		if ( mesh )
		{
```

Now we should take a break and think about simulation a bit. Do we want to simulate all meshes we find from the scene? Most definitely not, since in real game most geometry would actually be *level geometry*, which can be considered to have infinite high mass that causes it to be immovable. On the other hand in the simulator we don't want to create this kind of ultra high mass bodies for numerical accuracy reasons. Instead, we create *only* collision geometry for static level geometry, and leave it without actual simulated rigid body. This way level geometry participates into the collisions of other simulated objects, but doesn't move by itself. Convenient.

Next we parse simulation options from the user property string of the mesh. User property string is object specific text field, which can contain for example settings that don't make necessarily any difference to graphics rendering code:

```
// parse properties (from 'User Defined Properties' text field)
String props = m_scene->userProperties()->get( mesh->name() );
ODEObject::GeomType geomtype;
ODEObject::MassType masstype;
float mass;
ODEObject::parseProperties( mesh, props,
        &geomtype, &masstype, &mass );
```

First, GeomType is parsed from Physics=<x> property, where <x> is either box, sphere or trimesh. Then MassType is parsed: It receives value `MASS_INFINITE` if mass=<x> and density=<x> properties were missing, otherwise either MASS_DENSITY or MASS_TOTAL, depending whether total mass was set, by using Mass=<x>, or relative mass by using Density=<x> property. This kind of text string user property usage may look a bit of a hack, but this way new properties can easily be introduced without making any changes in the interfaces or braking backward-compatibility. Besides, a lot of values we want to cover are by nature scriptable-like, so user property parsing is fine for that kind of purpose.

After parsing rigid body properties, we make sure that it is no longer animated:

```
// remove from animation set
m_scene->transformAnimations()->remove( mesh->name() );
```

Finally, we create ODEObject, a wrapper for both body and its geometry, using parsed properties and add it to the list of all simulated objects:

```
// create rigid body with these properties
P(ODEObject) obj = new ODEObject( m_world->world(),
        m_world->space(), mesh, geomtype, masstype, mass );
m_objects.add( obj );
```

ODEObject class uses meshes bounding volume and triangle information to calculate requested rigid body objects for the simulation. ODEObject also takes care of mapping transformations between visual meshes and simulated rigid bodies.

Now we're done with the initialization and we can move on to the update(). First we start measuring elapsing time of the update, and then update accumulative time:

```
void PhysicsApp::update( float dt, Context* context )
{
	Profile::beginFrame();

	// update time
	float fps = 1.f / dt;
	float fixdt = 1.f / 100.f; // simulate physics at 100Hz
	if ( isKeyDown(KEY_F4) )
	{
		dt *= .2f;
		fixdt *= .2f;
	}
	m_timeToUpdate += dt;
```

Next, we'll update all key-framed animations in the scene (in this 3dsmax scene we happen to have none), update simulation by using fixed update time, render frame to back buffer, swap back buffer to screen and end frame timing:

```
	m_scene->applyAnimations( m_timeToUpdate, dt );
	simulate( fixdt );
	render( context, fps );
	swapBackBuffer( context );

	Profile::endFrame();
}
```

Now let's look at the details. First the simulate():

```
void PhysicsApp::simulate( float dt )
{
	PROFILE(simulation);

	// update simulation at fixed interval
	for ( ; m_timeToUpdate > dt ; m_timeToUpdate -= dt )
		m_world->step( dt, 0 );

	// get visual object positions from the rigid bodies
	for ( int i = 0 ; i < m_objects.size() ; ++i )
		m_objects[i]->updateVisualTransform();
}
```

First we stepped the rigid body simulation by gived fixed update interval (0.01 seconds) and then updated visual object transformations based on simulated objects'

transformations -- quite straightforward. Notice, however, that we cannot simply copy the transform between rigid body and the visual body, since the visual mesh might have for example scaling or non-center-of-mass pivot point. Of course you can always *avoid* such geometry in the modeling application in the first place, but updateVisualTransform() handles transformation conversions already without special cases or restricting the artists.

Notice that you should almost always modify transformation of the simulated objects by applying forces and impulses to the rigid bodies and not by modifying rigid body positions and velocities directly, since it might have undesirable and unexpected effects in the stability of the simulation.

Inside ODEWorld::step

What exactly happens inside ODEWorld::step()? Let's find out. In ODEWorld.cpp you can find the function:

```
void ODEWorld::step( float dt, ODECollisionInterface* checker )
{
	// use default collision checker if needed
	ODEDefaultCollisionChecker defaultcollisionchecker;
	if ( !checker )
		checker = &defaultcollisionchecker;

	// collect contacts
	dJointGroupEmpty( m_contacts );
	CollisionCallbackProxyData data;
	data.world = this;
	data.checker = checker;
	dSpaceCollide( m_space, &data, collisionCallbackProxy );

	// simulate step
	dWorldQuickStep( m_world, dt );
}
```

Note that all functions starting with small 'd' are Open Dynamics Engine (ODE) functions. ODE is used by the ODEWorld and ODEObject implementation.

First, as we passed null ODECollisionInterface* to the step function, we select ODEDefaultCollisionChecker as active collision interface.

Then we collect all contact joints from the collision space by clearing old contacts and calling dSpaceCollide. Space collide finds out which geometry objects are close to each other and calls (indirectly) our callback interface ODECollisionInterface* which we defaulted to ODEDefaultCollisionChecker. ODEDefaultCollisionChecker has only one function, namely checkCollisions:

```
int ODEDefaultCollisionChecker::checkCollisions( dGeomID o1, dGeomID o2,
        dContact* contacts, int maxcontacts )
{
        // get geometric properties of contacts
        int numc = dCollide( o1, o2, maxcontacts,
                &contacts[0].geom, sizeof(contacts[0]) );

        // set contact non-geometric parameters
        for ( int i = 0; i < numc ; ++i )
        {
                contacts[i].surface.mode = dContactBounce | dContactSoftCFM;
                contacts[i].surface.mu = dInfinity;
                contacts[i].surface.mu2 = 0.f;
                contacts[i].surface.bounce = 0.1f;
                contacts[i].surface.bounce_vel = 0.1f;
                contacts[i].surface.soft_cfm = 0.0001f;
        }

        return numc;
}
```

checkCollisions() gets in both geometry objects that potentially can collide and a buffer to receive the contacts/collisions if there is any. Actual geometry object type dependent contact check is done with a call to dCollide, After finding out the contacts, we process them by setting the surface properties, for example we set friction (mu field) to infinity, and specify that soft constraint force mixing should be used. If you run into stability problems (suddenly some object jumps to sky!) with your simulation, the first thing you should try is to increase CFM coefficient.

Now back to ODEWorld::step(): After all contacts have been collected, the simulation step is taken by call to dWorldQuickStep(), and we're done.

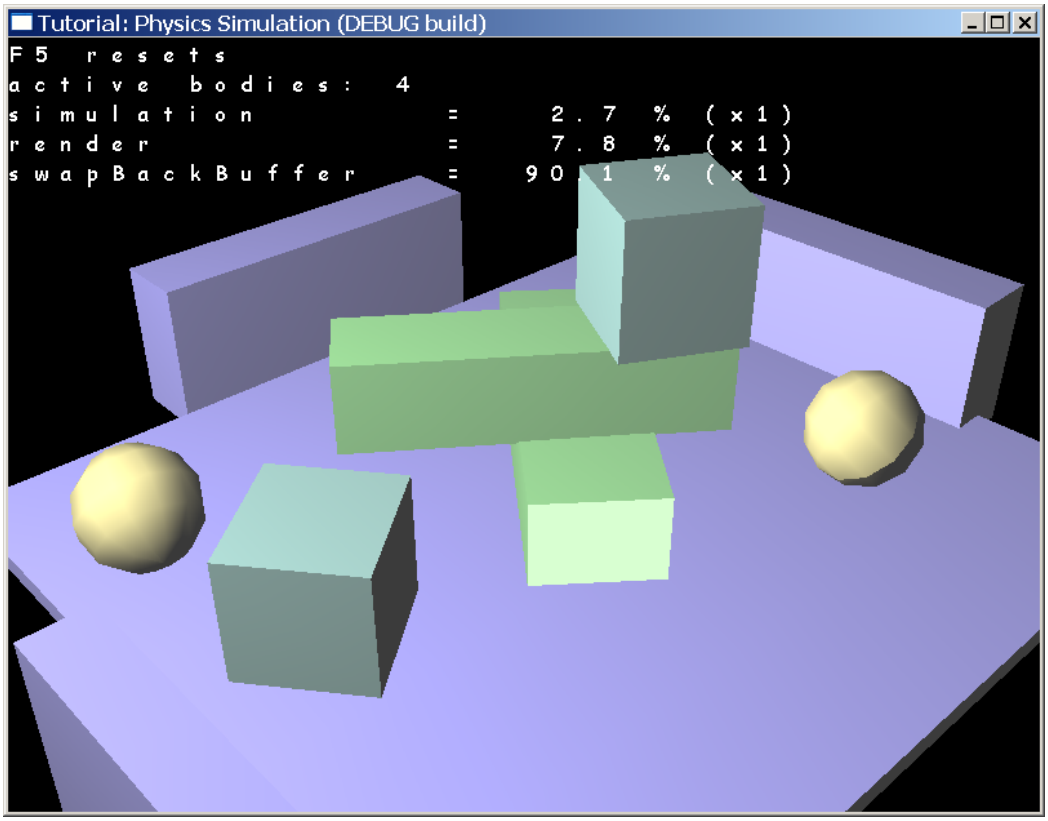

Figure 4-7. Rigid body physics simulation of 3dsmax scene and performance profiling information

Particle Effects

Particle effects are old trick but they haven't lost their edge. Sometimes it's even funny how relatively simple particle effects can still cause *'Whoaa!'* feeling even in experienced viewers. One 3rd person action game I worked on a few years ago was heavily hyped about game play, etc. already at a point where all we had was a couple of cool particle effects with slow motion effect. That's a good example how far they can take you!

First we'll take a quick look at basic Newton mechanics, which form groundwork for the particle simulations. Then concept of *domain* is introduced, which is used to define random variable distributions for various values used in simulations. After the basics are covered, we get into the actual properties of particles, emitters and systems. Last, but not least, we step through a few particle systems to show how the particle systems are built in practice.

About particles, emitters and systems

Before we start with the simulations, it's good to clarify what we exactly mean by particles, emitters and systems.

Particle is the smallest unit of simulation, a point like object. Particle has position, velocity, image, color, transparency, and other properties, which we'll get into later in this section.

Emitter is a set of particles. Emitters have their own properties too, like emission rate, maximum particle count and emission life time.

And as you might have guessed, *system* is a set of emitters. System has properties like for example number of emissions per second. Now you may be wondering why to have this 'system' level in particle simulations at all. Couldn't we just simply use emitters directly, and maybe launch multiple particle emitters from application if needed? We of course could, and many games do just that, but then we'd lose a lot of expression power from our particle systems engine. Neat thing with having the system level is that with system, you can coordinate more accurately how you want multiple seemingly independent emissions to behave. To make this more concrete, think about rain splashes on the ground. You can of course look at them as independent effects, but when you look at the system as whole you can specify for example the rate of splashes to match the rate of rain drops falling down. Very convenient and helps you

to build more impressive systems more easily. But now we move on to the simulation basics.

Particle simulation

What makes particles so nice objects to simulate is that they are so simple. Particles are just points in space after all. The simulation loop goes like this:

1. Create new particles by elapsed time
2. Assign initial properties to new particles
3. Remove old (dead) particles
4. Simulate all particles, update their velocities and positions.
5. Transform and render particles to screen

From high school physics classes we know the Newton's laws of motion:

1. Every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it.
2. The relationship between an object's mass m, its acceleration a, and the applied force F is F = ma.
3. For every action there is an equal and opposite reaction.

The first law hints us how the simulation should work -- by applying forces to particles. The second law in turn gives us a starting point for the simulation. First, we'll pick a force that we want to apply for particles. In this case we pick gravity, G=mg, since it's pretty hard to try to avoid it. Constant g is the familiar 9.8 m/s^2. Replacing gravity G equation in the place of F in the equation gives us:

mg = ma

Now, since the mass is constant larger than 0, we can divide the equation by m and get equation for acceleration a=g. This is not a surprise, but it's nice result since it

says us that acceleration is not dependent on mass when we simulate gravity, so we can drop the mass from our particle simulation altogether. By integrating the acceleration over time we get familiar equation of velocity:

$v = a * t + v_0$

Now we make another observation: We don't need to be interested in arbitrary point of simulation in time, since we update the simulation in each frame. What we are have been doing implicitly is in fact integrating the state of the simulation over time each time when we update a frame in the game (or whatever real time rendering application we have). This gives us nice update equation when we know that time since last update is dt:

$v = a * dt \quad => \quad v = v_0 + a * dt$

And position is updated naturally with the velocity with following familiar equation:

$p = v * dt \quad => \quad p = p_0 + v * dt$

So now we're done with the particle simulation. If we would have forces affecting particle which do not depend on mass, then we'd take the masses of particles into account. But it wouldn't make the simulation much more complex either, since we can get the acceleration from force by dividing the force by particle mass.

Note that all these equations had only scalars in them. But the equations are precisely the same when vectors are involved, so you can just replace F with vector F, a with vector a and v with vector V to get vector equations.

Concept of domains

The particle system engine is heavily based on concept of *domains*. It is a bit abstract concept, but understanding it is essential for effective usage of the particle systems. On the benefit side once you understand it, using the particle systems becomes very

intuitive and orthogonal, as nearly all properties of particle systems are simply defined by setting appropriate value domains for each property.

Domains are source value ranges for various random values needed by particle systems. Domains can be sources for scalar values, 2D and 3D vectors used in the simulation. Let's make this more concrete:

For example by saying that some particle effect has particle initial position domain set as 'Sphere with radius 4' means that the initial positions of new particles are randomized inside sphere which has radius 4. If in turn the particle initial velocity domain is set as Point (-50,0,0) then all particles get initial velocity of -50 along negative X-axis. Position domain is very straightforward, since it's literally the position in space where the particles are born. Velocity domain means the space where tip of velocity vectors will be when their start point is in origin. For example unit sphere velocity domain means that particles get random velocity between length 0 and 1 to random direction.

With various domains it is easy to make pretty much any kind of particle effect. For example rain could be done by setting flat 100x100 position box to 50 meters above ground with Box(-100,50,-100, 100,50,100) and then set velocity as Point(0,-20,0) and Gravity=-9.8. In this case the particles would be born on that flat 100x100 box 50m above ground and their initial velocity would be 20m/s downwards and acceleration 9.8m/s^2. In addition, particle sprite elasticity could be set above 0 (say, to 0.1) to cause particle bitmap to stretch along movement speed, which works also very well with effects like sparks.

Particle properties

When particles are created, they get assigned properties from various different domains, which is to say random value distributions. Below is a description of each of

these properties and how they are used in the particle effect system. The most obvious from these are particle position and velocity, so we start with those:

Particle start position specifies the volume in space where the intial particle positions are randomized. Note that this position is specified in local space, which means that it might be affected also by the parent object position if particle system is parented to some object in 3D scene (as they usually are). For example gun fire flame might be parented to the barrel of the gun, so it's positions are defined relative to the barrel transsformation frame of reference. Most common particle position domains are sphere and box volumes, which are very simple and intuitive to use, but don't limit your imagination to these two, since for example using triangle or rectangle position sources might be used to create effects like energy shields with complex shapes.

Particle start velocity 3D domain describes direction and length of initial particle velocities. For example rain might have constant velocity downwards (like Point(0,-20,0) domain) and gun blast might have velocity sphere or cylinder somewhere further away from origin, which would create directed blast with some minor variation as well. This vector is also specified in local space, so the direction in world space might be affected by the parent object if any.

Particle life time is a 1D domain that specifies how long the particle exists in the simulation before it is removed. Usually Range(a,b) domain is used for this purpose as normally we don't want particles to die always at the same time to make it more appear more natural. You can also use this value as to optimize simulation, since for example there is no point to simulate rain drops after they have gone through the ground plane.

Particle start size specifies the initial radius (half of width/height) of particles when they are created to the simulation. The engine doesn't force any specific units,

but you can consider it to be for example meters or inches if you want. Range domain types can be used to introduce variation to the simulation, for example Range(1,2) means that particle sizes are randomized between 1 and 2 meters size.

Particle end size is the same units as particle start size, and its normally used to simulate particles to get smaller when end of life time approaches. Of course no one prohibits you to use it for more funky purposes, for example gas particles might get larger (and fade out) when they grow older.

Particle start alpha domain defines initial constant alpha of the particles when they are created. Exact usage of the alpha is shader dependent, but normally (both with additive and alpha shaders) alpha means constant transparency, so that when alpha=0.5 then the particle is half transparent, and when alpha=1 then the particle is fully opaque. Note that this constant transparency is combined with particle texture (animation) transparency, so in the end particles might be more transparent than what the constant transparency level specifies alone.

Particle end alpha is same kind of 1D domain as particle start alpha, but for time at the end of particle life. Normally this property is used to fade particles to black at the end of simulations, but try other way around too, you might be able to create interesting effects that way! In general, the best way to learn how to use the particle simulation parameters is to play with the particle effects and boldly try combinations, which no man has tried before.

Particle start color is (R,G,B) domain that defines initial color of the particle which is multiplicatively -- at least by default particle sprite shaders -- combined with particle texture color. For example if texture color is white (1,1,1) and particle color is red (1,1,1), then you get red (1,0,0) texture, since (1*1, 1*0, 1*0) = (1,0,0). But beware, since for example having red (1,0,0) texture and green (0,1,0) particle color

results totally black end result because of combining! Be sure to check for this if you don't get anything to screen when you're playing with the effects.

Particle end color (R,G,B) says color just before the particle system is removed from the simulation. More conventional usage includes for example fire, which might have more blue tone at the end of life of each particle, but more red/yellow as the start color. But of course if you want to get wild you can create pretty hallucinogenic effects by randomizing start and end colors.

Particle sprite elasticity requires a bit longer explanation. Sometimes when particle moves really fast, it needs to be stretched in direction of movement to create a realistic looking effect. This 1D domain can be used just for that, since it specifies how much particle velocity affects to the shape of the particle. Obvious usage for this property is when you make sparks, rain and other similar velocity oriented effects. Normally values between 0 and 0.2 are ok, but out-of-range values could be used in some special cases. Trial and error is the best way to find out as usual. Note that particle elasticity is mutually exclusive property with particle sprite rotation, introduced next.

Particle sprite start rotation is the angle how the particle is initially rotated on screen. For example having particle sprite rotation domain set as Range(-90,90) says that particles can be rotated anything between +-90 degrees to either direction. If sprite rotation speed parameter (see next) is set to Constant(0) then the particle will also stay in that pose.

Particle sprite start rotation speed can be used to create for example various good looking gas, smoke and haze type of effects when combined with big semi-transparent, multiplicative sprite bitmaps. Units are degrees per second, so setting

sprite start rotation as Constant(360) means that the particle sprite will be rotating on screen one full round in a second.

Particle sprite end rotation speed is orthogonal value to particle end alpha and end color in a sense that it says the particle rotation speed at the end of life of a particle. Normally used to slow down particles towards the end.

Texture name is bitmap file name to use to render particles with. Texture is combined with possible alpha and color values multiplicatively, so that if for example texture alpha channel is 50% and constant alpha is set to 50%, then end result alpha will be 25%. Same combining is used with colors. This behavior can be changed though by using custom shader to render particles. Basically the shader is not bound how it uses the parameters, it's just that the default sprite shaders use the parameters as described here. And of course if you don't use the parameters in intuitive way it might be confusing for the user to apply the shader.

Texture frame count specifies how many images are 'embedded' to the bitmap using n*m grid. For example if you have a texture of size 256x256 and you specify texture frame count as 16 then you should have 4x4 textures of size 64x64 each in the texture image. This is basically the same as having multiple images as sequence img0001.bmp, img0002.bmp, ..., but just more effective as no rendering state changes are required when rendering the particle images.

Texture animation type specifies what is the logic of selecting each frame used to render each particle. Available options in the engine are selection by simple playback loop, selection by life phase and random frame selection. Simple playback loop means that texture animation frames are use one after another using constant frame change rate (described below), and when the last frame is reached then the next one will be the first texture frame again. Selection by life phase means that when the particle is

born it uses the first texture frame, and just before it's removed it will be using last frame, and during it's life naturally all the frames between those. Selection by random selects the frame to be used randomly. This might not seem too useful at first, but it can create for example nice fire effects when you use random frames from a set of flame images. Output can be quite impressive when you have a lot of flames overlapping each other as used for example in games like Max Payne.

Texture frame rate is in units of number of frames per second and says how often the image used to render each particle is changed. In some frame selection logics the frame rate might not affect at all, for example if the frame is picked by particle's life phase, then the displayed frame will be independent of frame rate. Note also that each particle is unique and their images change independently of each other.

Texture view type describes how the particle sprites are rendered with relation to camera. To be more precise, the parameter selects which direction is considered 'up' by the particles. This property might feel a bit confusing, but this example should clarify it a bit: If particles consider 'up' direction to be the same as camera up direction, then particles bitmap's direction is *always* the same as camera up direction, independent of camera rotation. This is generally ok behaviour, but for slowly moving big particles it might look funny as the particle seem to rotate along with the camera Z (look-at) axis. The other option is to select particle 'up' direction from world space Y-axis direction. In that case particle rotation is more independent from camera's rotation, but problems arise if you look particles more or less along world space Y-axis. In that case particles seem to rotate along their axis very rapidly since world Y-axis direction can change very rapidly in camera space. But don't worry if you don't completely understand the description above, since both of the situations are more or less special cases which rarely occur in normal game environment.

Shader specifies exactly how the various particle parameters are used to render the particles. For example particles might be rendered using additive blending (sprite-add shader) or alpha blending (sprite-alpha shader). In general, additive blending creates better 'halo' type of over-lighting effects and alpha blending is more suitable for smoke and haze kind of transparent effects.

Emitter properties

Emission property domains resemble a lot particle properties -- which is good so that we don't need to repeat the lengthly list here. But a few differences are good to point out:

Emission position domain specifies pivot point for an emission. Normally this is origin Point(0,0,0), but occasionally it's useful to use some other range. For example you can create multiple rain splash particles by creating new emissions inside random box which covers the area that is under rain.

Emission max particles sets limit of how large number of particles there can be in a single emission. Be careful with this parameter if you have multiple simultaneous emissions, since the number of particles can quickly become large!

Emission rate describes as the rate of particles/second how often new particles are added to the emission. If maximum count of particles is reached then existing particles are removed by the logic of kill mode as described next.

Emission kill limit specifies how particles are removed from the system when maximum limit is reached. Options in the engine are none, oldest and random. None means that when maximum number of particles is reached, then no more particles are added but also that no existing particles are removed before they are removed naturally (by dying old age). Using oldest as kill logic means that when a new particle

needs to be added, then oldest one is removed. Random kill removes just some random existing particle when adding a new one.

Emission stop time have also same kind of life time as individual particles, but *stop time* is a bit different: It describes when the emission stops emitting new particles. This is useful because otherwise you might experience sudden vanishing of particles at the end of emission life time, because individual particles belonging to an emission are removed along with the emission.

System properties

System wide properties are used to configure how emissions behave in the system. Emissions' relationship with system is more or less the same as the particles have with their emission source. There are only a few system properties:

System rate specifies number of new emissions per second that is added to the system.

System stop time sets the time when no new emissions are created. The behaviour and reasons for it's usage are similar than the emission stop time described in *Emission properties*.

System max emissions is maximum number of simultaneous emissions in the system.

System kill limit specifies how emission count limit is maintained. Emissions can be either left alone (so that no new emissions are created when maximum count is reached), removed by age or removed randomly.

About particle effect examples

Now we're pretty much done with particle effect properties. The list of adjustable properties might have felt a bit overwhelming at first, but luckily you don't need to immediately learn every possible parameter to make cool looking effects. The best

way to get started is to take some existing (and working) effect as base and start by modifying it. Change only one or two parameters at a time so you see clearly how your changes affect the simulation. It's also good to use some 'real' context as for example game level as background when you design your effect. This way you have better changes of fitting the effect into the mood of the level. After all, none of the effects exist in isolation from the rest of the game (hopefully!). Next we'll design a few effects and see how they work.

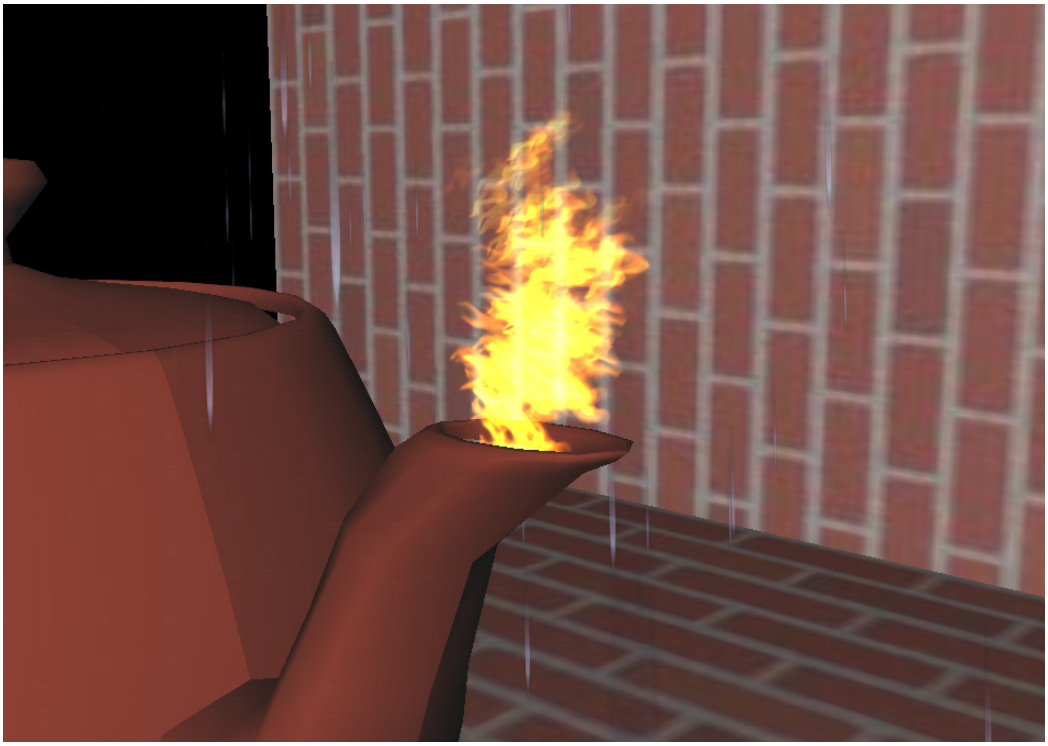

Figure 4-8. Fire and rain particle effects.

Example particle effect: Fire

This example shows only the particle system prt file and the comments. Actual playback code can be found in *Playing particle effects in the 3D engine* -section. Anyway, now into the fire! The particle system files start with the documentation and helper functions, so we skip directly to the *System properties* on line 94:

Fire system properties

```
SystemMaxEmissions = 1
```

This sets maximum number of simultaneous emissions. Now, since we're modeling fire, we have only one emission, but for example if we'd be modeling rain splashes, we could have up to 100 emissions active at the same time (as rain drops hit the ground, each would require a new emission).

```
SystemLimitKill = "NONE"
```

SystemLimitKill describes how emission limit is maintained:

NONE says that when the limit is reached, no old emissions are removed before free space becomes available by natural removal

OLDEST would say that oldest emission should be removed to make room for a new emission.

RANDOM removes random emission if maximum limit is reached

```
SystemRate = Constant(100)
```

Number of new emissions / second specifies born rate. Even though we have only one simultaneous emission, we still put this high number because we want the emission to be lit up fast.

```
SystemStopTime = Infinity()
```

This describes when the whole system should stop starting new emissions. Infinity() keeps it alive as long as the object exists in the application.

```
SystemLifeTime = Infinity()
```

This is the time limit when all remaining emissions are removed from the system and the system deactivates itself.

Fire emission properties

Next we move on to properties of a single emission. Remember that system consists of emissions, which in turn consists of particles. We find familiar looking parameters:

```
-- Maximum number of particles simultanously
EmissionMaxParticles = 150
```

```
-- How max particle limit is maintained: kill NONE, OLDEST, RANDOM
EmissionLimitKill = "NONE"

-- Emission rate, particles/second
EmissionRate = Range(250,300)

-- Time after no more new particles are emitted
EmissionStopTime = Infinity()

-- Lifetime of particle emitter
EmissionLifeTime = Infinity()
```

Note that emission rate is set quite high so that the fire looks aggressive as new particles are frequently born. Otherwise the parameters are very similar to system parameters, this time just they affect particles instead of emissions. However, the last parameter is different, it describes the position of individual emission in the system:

```
-- Pivot point for the emissions
EmissionPosition = Point(0,0,0)
```

For the fire we use the system origin, but for example if we'd be making rain, we could say that emissions are born on flat 'sky' box above ground, like `Box(-50,30,-50,50,30,50)`, which would cause rain drop emissions to cover larger area when they drop all over the 100x100 area from 30 units above the ground.

Fire particle properties

Finally we move to properties of individual particles. First we define particle life time quick short, as individual fire flames don't exist that long:

```
-- Particle life time in seconds
ParticleLifeTime = Range(0.5,2)
```

Then we set that the particle system doesn't need to be updated if it's not visible for a long time:

```
-- If true then particles are updated even though they are not visible.
-- Note: Normally this should be set to false
ParticleUpdateAlways = false
```

Then we give particles a random position inside 2 unit radius sphere, 2 units above ground, to simulate effect that some flames actually don't get born in the very center of the fire:

```
-- Volume in which the particles are born
ParticleStartPosition = Sphere( 0,2,0, 0,2 )
```

As we're still aiming for aggressive fire, we set initial velocity for individual particles quite high speed up Y-axis:

```
ParticleStartVelocity = Sphere( 0,22,0, 3,5 )
```

So particles initial velocity vectors get randomized inside 3-5 meter (hollow) sphere 22 units above ground. This gives almost direct upward movement, but with some variation to keep the flames more interesting.

Next, we define ranges for particle sprite world space size, color and transparency over time:

```
-- Start size of particle
ParticleStartSize = Range(3,7)

-- Life time end size of particle
ParticleEndSize = Range(3,7)

-- Start color (R,G,B in range 0-1) of particle
ParticleStartColor = Box(1,1,1, 1,1,1)

-- End color (R,G,B in range 0-1) of particle
ParticleEndColor = Box(1,1,1, 1,1,1)

-- Start opacity of particle
ParticleStartAlpha = Constant(1)

-- End opacity of particle
ParticleEndAlpha = Constant(0)
```

And this time we don't use sprite elasticity property, but give some initial spin, which fades away over time:

```
-- Particle sprite elasticity wrt particle velocity, set Constant(0) to disable
-- (note that elasticity and rotation are mutually exclusive)
ParticleSpriteElasticity = Constant(0)

-- Particle sprite initial rotation (degrees)
```

```
-- (note that elasticity and rotation are mutually exclusive)
ParticleSpriteRotation = Range(0,360)

-- Particle sprite initial rotation speed (degrees/sec)
ParticleSpriteStartRotationSpeed = Range(-360,360)

-- Particle sprite end rotation speed (degrees/sec)
ParticleSpriteEndRotationSpeed = Constant(0)
```

Note that 'elasticity' means how much individual particle's current velocity affects its shape. Elasticity is very handy for modeling for example sparks and rain-drops or other fast-moving particles, which need to be affected by their movement direction.

Fire bitmap properties

We animate fire bitmap over time, so that each of the 16 frames embedded in a single bitmap is used when the particle is at certain point of its age. This gives us effect that older particles can be recognized by appearance:

```
-- Texture bitmap
Texture = "textures/fire1.bmp"

-- Number of frames embedded to texture (nxn grid)
TextureFrames = 16

-- Method of selecting which frame to display:
-- LOOP plays back animation in Constant frame rate (TextureFramerate)
-- LIFE starts from first frame and gradually animates until end-of-life
-- RANDOM selects random frame in interval defined by TextureFramerate
TextureAnimation = "LIFE"
```

Finally, we select the shader, which should be used to render the particles. In this case 'sprite-add' shader works best, since we want fire to cause some 'brightness' effect as well, and additive shader fakes that nicely. This happens because the pixel colors of the fire bitmap are *added* by the shader over the old pixel colors in frame buffer.

Playing particle effects in the 3D engine

Playing particle effects in the engine is nearly trivial. You just need to supply ParticleSystem constructor with the rendering context and the filename of the particle system, and then call ParticleSystem::update(dt) function in every frame to keep the particles moving.

Note that you can for example link particle systems to a node hierarchy to animate them along other scene. For example you could attach a particle to the head of a torch, and when player carries the torch around the particle effect moves along with it.

Below (from tutorials/particle_rendering/ParticleApp.cpp) is a fully working example of rendering a single particle system on screen:

```
ParticleApp::ParticleApp( Context* context )
{
	// scene setup
	m_scene = new Scene;

	// particle system setup
	m_particleSystem = new ParticleSystem( context, "data/fire.prt" );
	m_particleSystem->linkTo( m_scene );

	// camera setup
	m_camera = new Camera;
	m_camera->linkTo( m_scene );
	m_camera->setPosition( float3(0,3,-20) );
	m_camera->lookAt( m_particleSystem );
}

void ParticleApp::update( float dt, Context* context )
{
	// update particles
	m_particleSystem->update( dt );

	// render frame
	{
		Context::RenderScene rs( context );
		m_camera->render( context );
	}

	// flip back buffer
	context->present();
```

}

Some Good Read

- A solid understanding of basics in (linear) algebra and trigonometry. So dig up those high-school, college and university-level books again if you have let yourself to become a bit rusty on the topics.
- Alan H. Watt: *3D Computer Graphics, 3rd Ed.*, Addison-Wesley, 1999. (A modern introduction of a wide array of topics related to computer graphics. A very good read.)
- James D. Foley, et al: *Computer Graphics: Principles and Practice in C, 2nd Edition*, Addison-Wesley, 1995. (A classic, but still I prefer Watt's book since it focuses on 3D.)
- Alan Watt and M. Watt: *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992. (A very good book, but you probably don't need both if you have already *3D Computer Graphics* by the same author, or other way around)
- Tomas Akenine-Möller and Eric Haines: *Real-time Rendering, 2nd Ed.*, AK Peters, 2002. (Another good book about 3D computer graphics. Very up-to-date and lots of topics covered, including collision detection too. See also the book's website www.realtimerendering.com for a nice link collection about various topics. Very useful even if you don't have the book!)
- Wolfgang F. Engel (editor): *ShaderX2: Introductions and Tutorials with DirectX 9.0*, Wordware Publishing, 2003. (I haven't actually read this book, so take my recommendation with a reservation, but since there are only so few

books using HLSL in shader programming yet, I decided to include this one in the list as well)

- Cormen, et al: *Introduction to Algorithms, 2nd Ed.*, MIT Press, 2001. (If you don't know what's the difference between O(n) and $O(n^2)$ then you're in trouble. Basic knowledge of algorithms is essential for anyone working on computer graphics programming, and this book gives a nice introduction to the topic. Heavy read though, 1184 pages, but at least you get bang for your buck)