CS 1122

Programming Games for PC & Xbox

David Beirne

0705140

Programming Report

Real-Time 3d Shadow Effects in DirectX

14/12/2007

Contents

1 Abstract

This paper seeks to investigate, implement and evaluate common approaches, both basic and advanced used in creating real-time shadow effects within a 3D game environment under the DirectX API.

David Beirne
BSc(Hons) Computer Games Programming
University of Abertay Dundee

2 Introduction

Few natural lighting phenomenon are as important in the intelligibility of 3D environments as shadowing. The world we see consists of a complex interplay between light and shadows. Shadows help us to understand the relative position and scale of objects within a scene, they also help us to understand the complexity of geometry in both receivers and occludes. Shadows can even be used to supply visual cues about the scene outside the cameras field of view – such as the shadow of a plane passing over the viewer or a character hidden around a corner whose shadow is cast onto the ground.

This paper seeks to investigate two popular shadowing effects used in computer games today: Planer Shadows and Shadow Mapping. Initially the paper provides a detailed overview of the bespoke engine being used for the attached sample applications in order for the reader to understand how it works and why the shadowing effects were implemented in the way they are.

Next the paper looks at Planer shadows, a relatively early shadow effect which represents shadows as the projection of geometry onto a plane along a light vector. The paper will give an overview of the planer shadows concept and a detailed explanation of how this effect has been implemented in the example application, concluding with an evaluation of the effect and its usefulness within complex 3D scenes.

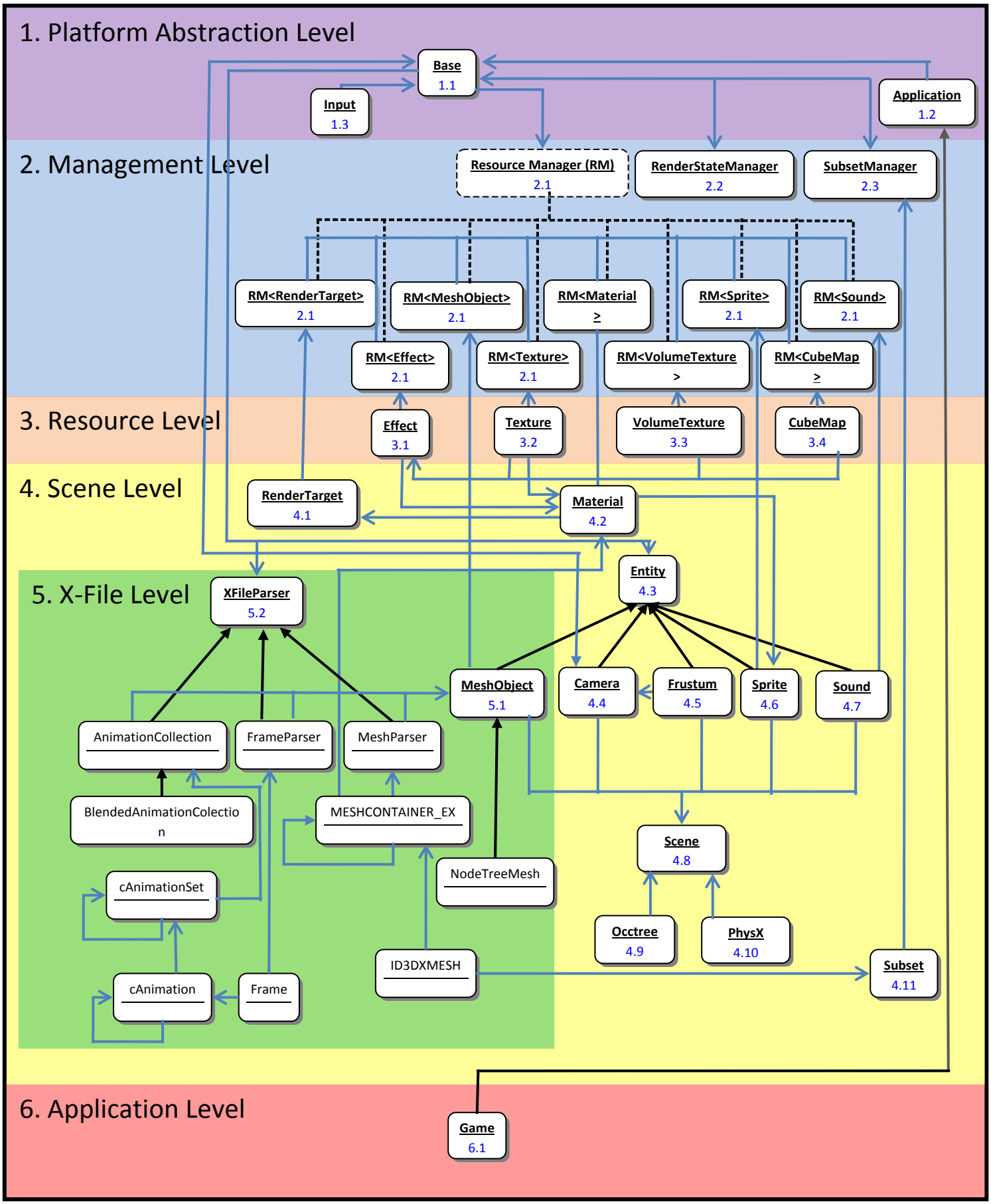
Finally the paper will tackle one of the most widely adopted advanced approaches to creating shadows in complex 3D scenes – Shadow Mapping. The concept of shadow mapping will be explained in detail and the reader will be taken through an implementation step-by-step. Finally, the paper will evaluate the shadow mapping application and critically assess the effect and its impact on the overall visuals and performance of the sample application.

3 Cyclone Game Engine

3.1 Overview

Cyclone is the name of a library of useful wrapper classes and functions I have written over the last few years to help in creating, managing and rendering DirectX based 3d environments and gaming applications for the PC and Xbox games console. I started Cyclone in 2005 and since then, it has gone through a number of iterations. At present the engine contains classes for handling most of the important features of a modern game engine including input, 3d and 2d Sound, Sprites, Materials, Effects, Geometry, Cameras, Scene management, Resource Management, File IO, Logging and Physics Simulation.

A Simplified UML Class Diagram of Cyclone would look something like this:


1. Platform Abstraction Level
2. Management Level
3. Resource Level
4. Scene Level
5. X-File Level
6. Application Level
Base 1.1
Input 1.3
Application 1.2
Resource Manager (RM) 2.1
RenderStateManager 2.2
SubsetManager 2.3
RM<RenderTarget> 2.1
RM<MeshObject> 2.1
RM<Material> 2.1
RM<Sprite> 2.1
RM<Sound> 2.1
RM<Effect> 2.1
RM<Texture> 2.1
RM<VolumeTexture> 2.1
RM<CubeMap> 2.1
Effect 3.1
Texture 3.2
VolumeTexture 3.3
CubeMap 3.4
RenderTarget 4.1
Material 4.2
Entity 4.3
XFileParser 5.2
MeshObject 5.1
Camera 4.4
Frustum 4.5
Sprite 4.6
Sound 4.7
AnimationCollection
FrameParser
MeshParser
BlendedAnimationColection
MESHCONTAINER_EX
NodeTreeMesh
Scene 4.8
cAnimationSet
Occtree 4.9
PhysX 4.10
ID3DXMESH
Subset 4.11
cAnimation
Frame
Game 6.1

3.2 Engine Details

This section will take a more detailed look at each of the classes in the Engine. Each sub-section relates directly to the diagram on the previous page which is constructed of the level number and a unique number. Each sub-section is then prefaced by the '3.2' of this section i.e. the Subset class on the diagram is labelled "4.11" so its description can be found at "3.2.4.11". If you are reading this electronically, clicking on each class label in the diagram will link directly to the appropriate description.

3.2.1 Platform Abstraction Level

The platform abstraction layer provides the fundamental classes and functions used by Cyclone to create a cross-platform environment.

3.2.1.1 Base Class

The core of the engine is the Base Class. The aim of the base class is to encapsulate the core elements of the DirectX API which must be shared by other classes within the engine, giving access to shared objects such as the graphics, input and sound devices. The Base also provides a single point of reference for obtaining other shared resources such as textures, meshes, sounds, sprites and render-targets. Finally the Base class contains platform independent methods for writing text to the screen, logging errors and posting message boxes.

3.2.1.2 Application Class

The Application class is a virtual implementation of a cross platform application. Programs written to use Cyclone will normally derive from this class and provide a concrete implementation by overriding the virtual functions. The Application class contains functions to handle the overheads involved in creating a generic application for the windows and Xbox platforms including windows message handling, Icons, windows titles, application names, window scaling and positioning and exiting cleanly. The Application is also responsible for creating and cleaning up the single Base instance. In order to allow the user to create their own derivations of Application, the application class does not contain the Windows winMain() or Xbox equivalent Main() application entry point functions – these should be included at the furthest extension of the Application class.

3.2.1.3 Input Class

The input class in the diagram actually represents two classes, one for the PC and one for the XBOX, which class is loaded is determined by pre-processor code as the application compiles. The PC version of the input class provides a wrapper around a DirectInput keyboard, mouse and joystick device whereas the Xbox version provides a wrapper for up to four XInput devices (these are Xbox controllers, joysticks, dance mats, light-guns and the like). Both the PC and Xbox classes provide a common interface, based mainly on the Xbox interface. The PC implementation also provides

more low-level access to the keyboard and mouse which break the cross-platform compatibility of the application if used, but are useful if developing specifically for PC.

3.2.2 Management Level

The next level in the Engine is the resource management level.

3.2.2.1 Resource Manager

All resources, be they Meshes, Textures, Sprites, Sounds or RenderTargets are stored in a resource manager. The Resource Manager is written using the template design pattern, which means that a resource manager can be created for any type of object provided it has a FileName string (which acts as a unique identifier). The ResourceManager's internal data structure is a std::Map which is private and only accessible by using the supplied methods. The methods for adding to a Resource Manager are as follows:

```
U* addInstance(U* newItem);       // Adds an instance to the Resource Manager.
U* addCopy(U* newItem);           // Adds a copy to the Resource Manger.
```

Where U* is a pointer to an instance of the typename specified on creating the ResourceManager. For Example:

```
ResourceManager<Texture>* textureManager;
textureManager = new ResourceManager<Texture>();
Texture*someTexture = textureManager->addInstance(new Texture("C:\Wall.dds"));
```

This code creates a ResourceManager which stores textures then adds a new texture to the resource manager (loaded from the file "Wall.dds"). In this instance, the texture will be added to the internal map and its address will be returned to the someTexture pointer. If the third line of code was repeated i.e.:

```
ResourceManager<Texture>* textureManager;
textureManager = new ResourceManager<Texture>();
Texture*someTexture = textureManager->addInstance(new Texture("C:\Wall.dds"));
Texture*otherTexture = textureManager->addInstance(new Texture("C:\Wall.dds"));
```

The ResourceManager will find that a texture with the file name "C:\Wall.dds" is already mapped so will delete the replication and return the pointer to the existing instance. In this case, the 'otherTexture' pointer will be the same memory address as the 'someTexture' pointer. This is useful when, for example, one texture file is used often or used in multiple meshes. Of course there are some instances where the user will not want only one shared instance of a particular resource, if for example they intend to edit the texture in one particular place and not in others. In this case the addCopy() function is used:

```
ResourceManager<Texture>* textureManager;
textureManager = new ResourceManager<Texture>();
```

```
Texture*someTexture = textureManager->addCopy(new Texture("C:\Wall.dds"));
Texture*otherTexture = textureManager->addCopy(new Texture("C:\Wall.dds"));
```

Here a ResourceManager of type Texture is created and two of the same texture are added using the addCopy() function. In this case, the two texture pointers, 'someTexture' and 'otherTexture' point to the unique memory addresses for the new texture created within the addCopy method and mapped within the ResourceManager. The benefit of using the ResourceManagers when adding a copy is that once mapped, the resource will be automatically cleaned up when the application closes. As a deliberate design decision, Cyclone does not inforce the use of ResourceManagers, it is perfectly acceptable for an application to manage its own resources and as such resources are not constrained by or reliant on the existance of a corrispondant ResourceManager.

3.2.2.2 Render State Manager

The RenderStateManager is a singleton whos isntance is stored in the single Base instance. The aim of the RenderStateManager is to keep a cache of calls to the Device->SetRenderState() function and to esure taht a render state is not set multiple times. In some applications, this can greatly improve performance as setRenderState calls are relatively expensive but are often used every frame. The RenderStateManager can be used in a similar way to setting a normal render state in DirectX, only instead of calling on the device directly, the RenderStateManager is called:

```
base->renderStateManager->SetRenderState(D3DRS_FILLMODE,D3DFILL_WIREFRAME);
```

As with the ResrouceManager, the RenderStateManager is not imposed on the user of the engine and is entirely optional. The D3DDevice is still accessible directly from the Base instance so setting the render state directly is entierly possible.

3.2.2.3 Subset Manager

The subset manager is a class designed to render a set of subsets as quickly and efficiently as possible. A Subset (see Subsets section) is essentially a pointer to an ID3DXMesh, a pointer to a Material (See Material section) and a World Matrix – that is, a piece of geometry and a description of how and where to render that geometry. The main costs in rendering with DirectX come from the multiple state changes (such as texture stage state, render state, effect and material state) involved in changing the material. For this reason the SubsetManager is used to maintain a list of all subsets to be rendered ordered by material.

When each MeshObject (see MeshObject section) is told to render, each of its subsets are passed into the SubsetManager. When the SubsetMager is told to Render, it loops

through all visible materials and renders the corresponding subsets. The SubsetManager attempts to draw the subsets as accurately as possible, that is, if a subsets material has an effect which is beyond the capability of the device on which it is running, the SubsetManager will automatically draw the subset using the fixed function pipeline.

By maintaining a list, the subsets stored in the subset manager can be drawn multiple times, when required, without resorting to calling the render function of each object again, this can save a lot of computation and many calls on the call stack. Another useful feature of the SubsetManager is its ability to render all subsets within its list using a global effect instance. This can be very useful in effects that require the scene to be rendered multiple times or effects that require data such as depth from the scene such as ShadowMapping (see Shadow Mapping Section).

3.2.3 Resource Level

The resource level provides a set of classes which encapsulate the common resources used by 3d applications. At present the resource classses are as follows:

3.2.3.1 Effect

This class provides a wrapper for the ID3DXEffect interface, encapsulating the initialisation and updating of effects loaded from DXSAS compliant effect (.fx) files.

3.2.3.1.1.1 Semantics

With the new set of semantics, parameters can be labelled by an agreed word which can be understood by multiple programs. By comparing the semantic to each of the agreed words and applying the agreed value from the program effects can be treating in the same way. For example:

```
Float 4x4 worldMatrix : WORLDVIEWPROJECTION;
```

Cyclone would read this semantic in the Initialise function of the Effect class and attribute it to the a matrix concatenated from the current world, view and projection matrix at the time the Effects Update() is called. DXSAS specifies most of the useful matrices, the current camera position, the current time and also some lighting and colouring definitions.

3.2.3.1.1.2 Annotations:

“Annotations are not used by the DirectX framework but can be used by the application. An example of an annotation might be to hint to associate a file with a texture type, which is defined in a string...

```
<string filename = “Colormap.dds”;>”
```

(Wolfgane Engel (2004), Programming Pixel and Vertex Shaders) This annotation

combined with the TEXTURE semantic on the declaration of a texture would tell the Effect instance to load a texture into the Texture ResourceManager and pass it back into the constant of the effect:

```
texture SkyColour_Tex : TEXTURE
< string ResourceName = "Shaders\\Skybox\\SkyColours.dds"; >;
```

In the above example, the texture in the file "Shaders\\Skybox\\SkyColours.dds" would be loaded. It should be noted that Cyclone uses the base->getMedia() function to load media specified in an effect from a relative media path rather then forcing an absolute path.

3.2.3.2 Texture

The Texture class provides a wrapper around an LPD3DXTexture instance which allows a texture to be loaded from a specific file path and also adds the functionality to initialise a new texture in memory as a RenderTarget. Each Texture is identified by either unique file name or a unique string stored in the FileName attribute.

3.2.3.3 VolumeTexture

The VolumeTexture class provides a wrapper around an LPD3DXVolumeTexture, instance which allows a texture to be loaded from a specific file path. Each VolumeTexture is identified by either unique file name or a unique string stored in the FileName attribute. A volume texture consists of a texture containing layers of images stacked on top of each other, also known as a 3D texture.

3.2.3.4 CubeMap

The CubeMap class provides a wrapper around an LPD3DXCubeMapTexture, instance which allows a texture to be loaded from a specific file path. Each CubeMap is identified by either unique file name or a unique string stored in the FileName attribute. A cube map consists of a texture which represents the six faces of a unit cube.

3.2.4 Scene Level

The Scene Level represents the set of classes which encapsulate objects within a 3d Scene and the data structures used to handle them.

3.2.4.1 RenderTarget

the RenderTarget class encapsulates a Texture instance (which has been initialised as renderable) along with a renderable surface and a Sprite instance. The aim of the render target is to create a screen-aligned-quad with a texture onto which other geometry can be drawn. This enables multiple render passes to occur on different textures (as opposed to the devices back-buffer). RenderTargets are used for many effects, as defined in the DXSAS texture semantic "RenderColorTarget". RenderTargets are generally used for

effects where either multiple views of a scene are required, i.e. a mirror, shadow map or a multiplayer game. RenderTargets are also used where effects require post-processing – that is where a pixel shader is run on every visible texture in clip space such as light blooming, screen-space edge detection and colour filters.

3.2.4.2 Material

A material is used to describe the surface of a subset (see subset section). Each subsets' material can be described with the minimum of a single vertex colour, which is either imported from the DirectX (.x) format mesh. The vertex colour is stored within a MATERIAL instance, which is a pointer to a platform independent DirectX material (ID3DXMATERIAL) interface. A Material may also have a texture associated with it which is multiplied with the vertex colour within the fixed function pipeline. A material can have an optional Effect instance associated with it, this will describe how the vertices and pixels of the mesh are rendered by the programmable pipeline.

While a material can describe a very simple or very complex subset surface, the final representation of the surface is decided by the SubsetManager to suit the specific hardware as best as possible. Also because the materials are stored, by default, within the Base Material ResourceManager and only referenced within each subset or any given loaded mesh, it is possible to replace materials in real-time by simply switching the reference within the mesh subset to another material in the bases Material ResourceManager.

3.2.4.3 Entity

The Entity class represents any object with a unique transformation in 3d space. The Entity encapsulates the translation, rotation and scaling of objects within 3d space along with the matrices on which the transformation operations are carried out, and vectors to describe the current position, scale and rotation of the Entity. Each entity also contains a pointer to the single instance of the Base class which provided it and any derived class with access to the functions and references within the Base.

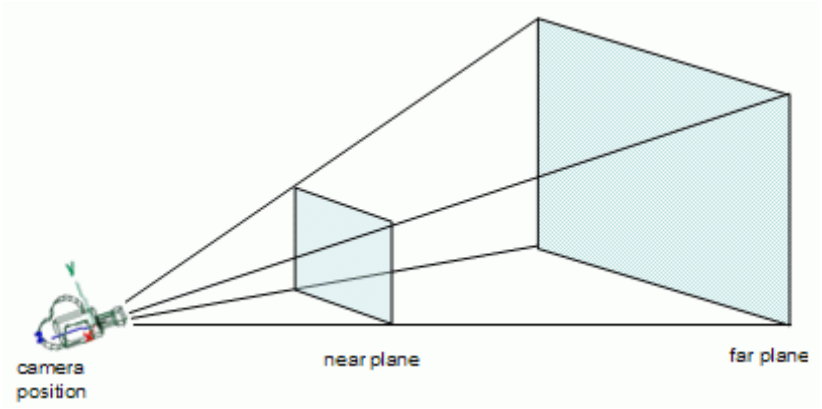
3.2.4.4 Camera

A Camera in Cyclone represents an Entity through with the 3d scene can be viewed. For convenience, the engine creates a default camera when the Base is initialised (see Base section). The analogy of a camera is carried through in the ability of the camera to be configured and manipulated in the same way as a physical film camera, with the ability to set the perspective, the near and far clipping distances (or range), and the ability to pan, dolly, roll, pitch and yaw. It is possible to have multiple cameras within an environment. However it is encouraged that each camera be set as the Base camera before it is rendered so that any other elements of the scene have access to it if they require it – i.e. an Effect may need to access the projection matrix or view position of the current camera to render correctly or a NodeTree mesh (see NodeTree section) will need to access the current cameras view frustum (see frustum section). This design decision was made to keep the

camera system open-ended so the programmer has the choice of how they want to use the camera.

3.2.4.5 Frustum

Each camera contains a viewing frustum. "In 3D computer graphics, the viewing frustum or view frustum is the region of space in the modelled world that may appear on the screen; it is the field of view of the notional camera. The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid." *(Wikipedia (2004) Viewing Frustum Definition)* The Frustum class constructs a rectangular pyramid from 6 planes representing the near, far, left, right, top and bottom clipping planes of the field of vision in front of the camera:



(Image from http://www.lighthouse3d.com/opengl/viewfrustum/images/vf.gif)

The Frustum class contains methods to check if a given point, sphere, cube or cuboid (called a rectangle to avoid confusion) is within the space of the frustum. This is calculated using the D3DXPlainDotCoord function to check the dot product of a point (or the points describing a shape) with the six sides of the frustum. If the point is inside, or any points of the shape are inside then the method returns true, otherwise it returns false. These methods are used by the Scene, NodeTree and Occtree classes (see later sections) for culling. However these methods could also be used in other situations, such as AI - for this reason, the Frustum Class is kept separate from the camera.

3.2.4.6 Sprite

The term sprite refers to a 2D element rendered within a 3D environment. Sprites can be used for on screen displays, menus or even to make 2d games. In Cyclone, the sprite class constitutes an Entity with an ID3DXSprite instance. As Entities, Sprites can be positioned, scaled and rotated in screen space using the same matrices and vectors as the Entity class. Sprites can be loaded as a single image, or a sequence of frames from a single "Sprite-Sheet" image. The Sprite Class is used in the Base instance to render text to the screen (this uses a sprite sheet of ASCII characters). Sprites are also used within the RenderTarget class (see RenderTarget section), where a sprite with a render-to texture is used to create a screen-aligned quad.

3.2.4.7 Sound

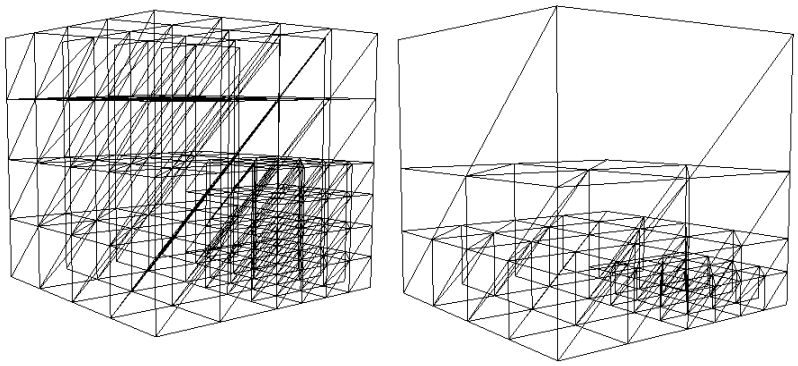
Cyclone provides a common interface to sound on Xbox and PC. A 3d sound is a sound which has a listener and emitter position. The volume of the sound is then calculated by the distance between the emitter and the receiver. To play 2d sounds, the emitter is placed in the same position as the receiver. Using DirectSound, the Sound class encapsulates the loading an playback of sounds from the Wav, Midi and wma file formats.

3.2.4.8 Scene

The Scene Class is currently being re-written but at this time loads and saves a 3d scene from two text files, one containing a list of all entities possible within the scene and the other containing specific instances of those entities with positions and other values. The Scene also uses the Tokamak Physics SDK (which I am moving away from in favour of PhysX) to load dynamic objects into the scene. Finally the scene performs some visibility testing and sends all visible meshes to the SubsetManager each frame. As with most of the managers, the Scene is not imposed on the user and is entirely optional.

3.2.4.9 Occtree

The Occtree class is currently being written with the aim of partly replacing the old Scene class. Whereas the Scene class allowed some nodes to be parents and children of each other to reduce the number of visibility tests, it performed no special partitioning (unless using a NodeTree which has special partitioning programmed into it). The Occtree recursively splits a given area into 8 or 4 (depending on whether a quadtree or occtree is required) equal subsets. The tree stores each subset within its parent so visibility testing can be performed very quickly by traversing the tree i.e. if the parent is not visible, none of its children are.



OccTree & QuadTree Rendered as wireframes in Cyclone

3.2.4.10 PhysX

I am currently working on integrating the PhysX physics simulation API into the Cyclone Game Engine, for this reason PhysX is described in the diagram as a monolithic blob-class but it does belong at the scene level.

3.2.4.11 Subset

A Subset is a set of polygons within a particular mesh which share the same material. A Subset consists of a pointer to the mesh which the subset represents, a pointer to the material which is to be applied and the subsets final world matrix. The world matrix must be supplied so that it can either be set for each subset before it is rendered on the fixed function pipeline, or passed into the effect of subsets' material if present.

3.2.5 X File Level

3.2.5.1 MeshObject

The MeshObject class allows static and animated meshes to be loaded from a DirectX .x format mesh. The class is composed of a reference to an instance of the MeshParser, FrameParser, and BlendedAnimationCollection classes (see following sections), along with methods involved in loading, updating and rendering a DirectX mesh and also some collision detection and intersection methods. The MeshObject class essentially acts as an interface between the programmer and a single DirectX file whether it is animated or static.

3.2.5.2 XFileParser

The XFileParser class is an abstract virtual class which encapsulates the methods required to parse through a file written in the DirectX X-File format.

3.2.6 Application Level

3.2.6.1 Game

The Game class is used in these examples to create a game containing a reference to a MeshObject instance and an Effect, called mainObject and mainEffect, the game also creates a simple interface to the mainEffect instance which allows the attributes of whatever effect is pointed to by mainEffect to be edited in real-time. The Game class also contains the common code used in each demo for controlling the camera and camera collision detection with the scene. Each demo inherits from the game class.

4 Basic Demo

4.1 Planer Shadows

4.1.1 Concept

The Concept of planer shadows is a fairly simple one. The shadow of an object on a flat surface represents the projection of the shadow casting object onto the plane of the surface.

4.1.2 Implementation

In order to implement planer shadows in Cyclone, it was more convenient to write an Effect then use standard DirectX functions.

4.1.2.1 Creating the Shadow Matrix

First the shadow effect is loaded in, then the D3DXMatrixShadow function is used to create a shadow matrix which is then passed into the shadow effect.

```
//Load in Flat Shadows Effect
mainEffect = base->effectManager->addInstance(
                        new Effect(base->getMedia("Materials\\FlatShadow.fx")));

//Create Shadow Matrix
D3DXVECTOR4 lightRay(0.0f, 120.0f, -90.0f, 1.0f);
D3DXPLANE basePlane (0.0f, 1.0f, 0.0f, 0.0f);

D3DXMatrixShadow(&S, &lightRay,     &basePlane);

S._42 += 0.002f;
S._22 += 1.0f;

//Set Shadow Matrix in the Effect
mainEffect->setValue("matShadow",&S,sizeof(float)*16);
```

4.1.2.2 Flat Shadow Effect

The flat shadow effect is very simple, the vertex shader takes the shadow matrix created in the previous step and transforms each vertex by its regular world matrix, followed by the shadow matrix, followed by the view and projection matrix. The pixel shader simply returns black. One important thing to note is that the render states involved in drawing the shadow with the stencil buffer are encapsulated in the effect file.

Before the shadows are rendered, the stencil buffer is enabled, allowing the shadow to burn into the stencil buffer. Source and destination blending are set and the z-buffer is disabled to prevent z-fighting. After the shadow is rendered, the render states are returned to defaults:

```
// ------------------------------------------------------------
// vertex shader function (input channels)
// ------------------------------------------------------------
  VS_OUTPUT VS_MAIN(float4 Pos : POSITION)
  {
      VS_OUTPUT Out = (VS_OUTPUT)0;

      Out.Pos = mul(Pos,matWorld);        // Transform by usual World matrix.
      Out.Pos = mul(Out.Pos,matShadow);   // Offset by Shadow Matrix.
      Out.Pos = mul(Out.Pos,matViewProj); // Transform by View & Projection matrix.

     return Out;
  }

// ------------------------------------------------------------
// Pixel Shader (input channels):output channel
// ------------------------------------------------------------
  float4 PS_MAIN() : COLOR
  {
      return float4(0,0,0,1.0f);
  }


// ------------------------------------------------------------
// Technique
// ------------------------------------------------------------
technique FlatShadows
{
    pass P0
    {
      stencilenable=true;
      srcBlend=srcColor;
      destblend=invsrcAlpha;
      zenable=false;


       // compile shaders

        VertexShader = compile vs_1_0 VS_MAIN();
        PixelShader  = compile ps_2_0 PS_MAIN();


        zenable=true;
        srcBlend=srcColor;
        destblend =zero;
        stencilenable=false;

    }
```

4.1.2.3 Rendering The Scene

The final step is rendering the scene, first the non-shadowed objects are drawn, then the SubsetManager is cleared and the shadows are drawn using the shadow effect as a global effect.

```
//Begin Drawing to Back Buffer
      if(SUCCEEDED(base->D3dDevice->BeginScene()))
      {

            base->D3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER ,
                  D3DCOLOR_COLORVALUE(0.0f,0.0f,0.0f,1.0f), 1.0f, 0L);


            //Add any non-shadowed objects to subset manager
            sky->Render();
            mainObject->Render();
            skinnedMesh->Render();

            //Render Non-Shadow Objects
            base->subsetManager->Render();
            base->subsetManager->clear();

            //Render Shadow Objects using flat shadow Effect
            skinnedMesh->Render();
            base->subsetManager->Render(mainEffect);
            base->subsetManager->clear();

            base->D3dDevice->EndScene();
            base->D3dDevice->Present(0, 0, 0, 0);


      }
```

4.1.3 Evaluation

While very simple to implement, planer shadows have severe drawbacks, if the scene were complex and there were multiple flat surfaces, a shadow matrix would have to be calculated for each plane and the scene would have to be rendered again. Also being limited only to planes, this shadowing method is virtually useless in most scenarios for this reason, more advanced shadowing techniques are required.

5 Advanced Demo

5.1 Shadow Mapping

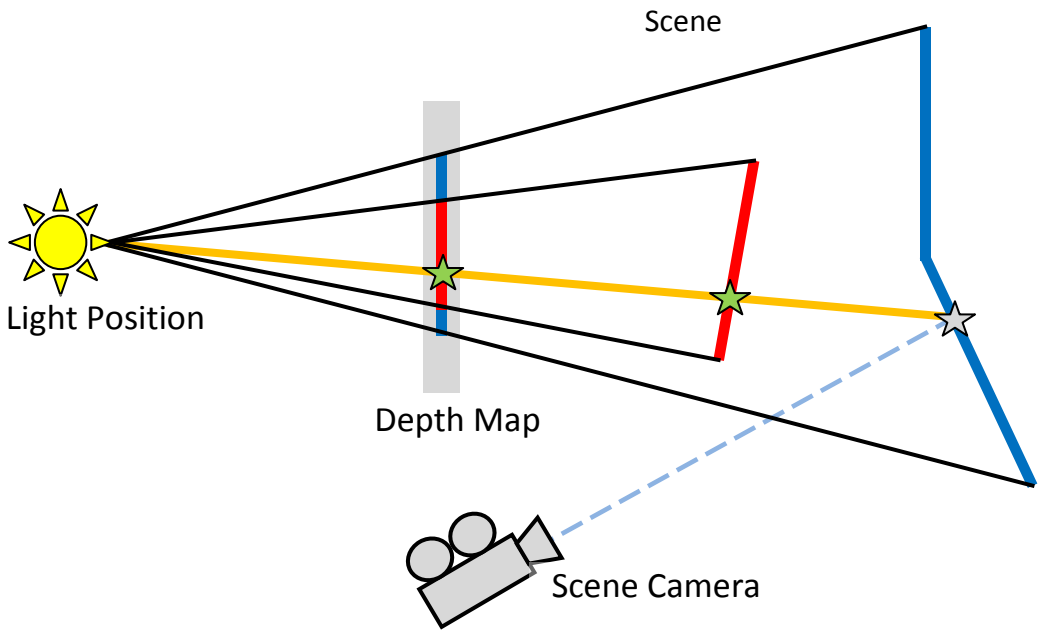
5.1.1 Concept

Shadow Mapping is one of the most commonly used techniques for creating realistic shadows in real-time 3d computer graphics. The concept was introduced by prominent graphics researcher Lance Williams in his SIGRAPH 1978 paper entitled "Casting curved shadows on curved surfaces". Since then, shadow mapping has been used in both pre-rendered scenes, real-time 3D applications, and more recently Computer Games. Shadow mapping is the technique used in Disney Pixar's RenderMan software and has subsequently been used in many feature Cg Films including the incredibly successful Toy Story and Shrek series.

Shadow mapping can be described by the following algorithm:

1. A view of the scene is constructed from the point of view of the light source. Only the Z-depth values are computed and stored.

2. A view of the scene is then constructed from the observers eye. A linier transformation exists which maps (projects) X,Y,Z points from the cameras view into X,Y,Z points in the light source view. For each pixel viewed by the camera, compare distance to the light ☆ with the depth stored in the shadow depth map ★ if the pixels depth is greater than the shadow depth then the pixel is visible to the camera and not visible to the light, therefore is in shadow.



5.1.2 Implementation

Shadow mapping was implemented through the use of an HLSL effect and multiple render targets (MRT) approach. Two render targets were used along with the default back-buffer, the first render target was used to store the depth map and the second to store the shadow information as a shadow map in screen space, finally the shadow map is blended in screen space with the scene rendered on the back-buffer using additive blending. This ensured that the scene could be shaded in any way required and then the shadows are added onto the top of the rendered scene.

5.1.2.1 Creating The Scene

The Scene used in the Shadow Mapping demo is a collection of MeshObjects (see earlier section). Each MeshObject is created in a 3d modelling program where animations, materials and effects are applied to subsets before it is exported in the X-File (.x) format. The Scene contains a room and an animated skinned mesh.

5.1.2.1.1 The Room

The Room is a fairly simple static mesh of a temple containing pillars, a pool of water and a model of the classical Greek statue of the goddess (of youth) Hebe. The MeshObject's intersection methods are used with rays cast from the camera to facilitate collision detection between the camera and the room.

5.1.2.1.2 The Skinned Mesh

The Skinned mesh walking around in the room is another MeshObject used to add complexity to the scene and make it more interesting. The Skinned mesh was modelled and animated in 3ds Max, before being exported to the X-file format using the Panda DirectX exporter. The Skinned Mesh walks around the room casting an interesting animated shadow as he goes.

5.1.2.2 Rendering the Scene

As mentioned earlier, the scene is rendered in multiple passes to different render targets before being blended onto the back buffer. Because Cyclone contains a SubsetManager which effectively contains a snapshot of the geometry at each frame, it is only necessary to tell each MeshObject in the scene to render once. Once the MeshObjects subsets have been sent into the SubsetManager, they can be rendered multiple times, either as normal or with a global effects applied to them, this saves a lot of computation and calls on the call stack when rendering a scene multiple times.

5.1.2.3 Rendering To The Depth Map

Once the SubsetManager is filled, it its necessary to render the scene from the point of view of the light to a depth map. The Depth Map render target is specified in the ShadowMappin.fx effect file which is used throughout the shadow mapping demo and contains the two techniques and render targets used for creating a shadow map:

```
//Shadow Map Render Target
texture2D depth_map: RenderColorTarget
<
      float2 ViewportRatio = {1.0f,1.0f};
      int MIPLEVELS = 0;
      string format = "D3DFMT_R16F";
>;

//Shadow Map Sampler State
sampler DepthRenderTarget = sampler_state
{
      Texture = (depth_map);
      ADDRESSU = CLAMP;
      ADDRESSV = CLAMP;
      MAGFILTER =linear;
      MINFILTER =linear;
      MIPFILTER =none;
};
```

This section of the ShadowMapping.fx file specifies the RenderTarget used for rendering the z-depth of the scene from the point of view of the light (the depth buffer). The texture2D part specifies that it is a 2D texture (as opposed to a cubemap or 3d texture), “depth_map” is the name of the texture. The “RenderColorTarget” semantic specifies that this texture should be renderable – in this case it specifies that Cyclone create a RenderTarget instance with attributes corresponding to those specified between the angled brackets: ViewportRatio specifies that the size of the texture should be 1x the with and height of the DirectX Device back buffer. There are 0 mipmapping levels (as generating mipmaps every frame would be very slow). The format of the Depth Map is very important. D3DFMT_R16F represents a 16bit floating point red channel with no other channels. For numerical accuracy, and in order to prevent texture tearing in shadow mapping it is important to use a floating point texture which can hold a greater range of values then the default 8-bit colour channel. It is also important to maintain as low a graphics card buss load as possible so since the texture is only being used to store a single value, it only needs one colour channel (as opposed to the normal 4; ARGB). The Sampler state specifies that the deptMaps texture coordinates are clamped to the vertices which define it and is filtered using a linear filter (which will help prevent excessive aliasing), Since there are no mipmaps generated by the texture, there is no need for a mipfilter.

A pointer to the RenderTarget created by the shadow mapping Effect instance is obtained during the initialisation method of the Demo class:

```
depthMap = mainEffect->getRenderTarget("depth_map");
```

Once the depthMap RenderTarget is obtained, it is possible to render the scene to the depth map texture.

However if you recall the description of the shadow mapping algorithm, it is important that the depth map is rendered from the point of view of the light and not the camera, and also only the z-depth is rendered to the depth buffer not the normal colours associated with the scene.

5.1.2.3.1 Rendering from the Lights Point of View

In order to render from the lights point of view, an array of "shadowMapLight" structs is created with one instance representing each light. The ShadowMapLight is defined as follows in the "Advanced.h" header file:

```
struct shadowMapLight
{
    Camera* lightCam;
    RenderTarget* shadowMap;
    D3DXVECTOR4 lightColour;
    bool changed;
};
```

This simple struct is composed of a reference to a Camera instance which represents the light itself, a RenderTarget (the shadow map of the particular light) a vector containing a colour value which can be passed into the light effect to colour the spotlight which can be changed by the user and a bool which keeps track of whether the light has been changed in any way (in which case it needs to be updated). For every shadow casting light in the scene there is a corresponding shadowMapLight instance:

```
//Render one shadow map per light (share the same depth map)
for(int i = 0; i < numLights; i++)
{
	if(FAILED(mainEffect->setValue("lightViewProjection",
				&lightCams[i]->lightCam->WorldMatrix,sizeof(float)*16)))
	{
		base->messageBox(base->hWnd,"Could not set light matrix");
	}

	//Draw Scene to depth map Render Target
	if(SUCCEEDED(depthMap->Begin()))
	{

		base->D3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
			D3DCOLOR_COLORVALUE(1.0f,1.0f,1.0f,0.0f), 1.0f, 0L);

		base->camera = lightCams[i]->lightCam;	//Set View/Projection to Light
		base->camera->Render();			//Update camera matrices

		//Render Scene using DepthMap Technique (for all subsets)
		base->subsetManager->Render(mainEffect,"DepthMap");

		depthMap->End();
	}
	else
	{
		base->messageBox(base->hWnd,"Failed To Render To Shadow Map");
}

	//Render To Shadow Map Render Target
	if(SUCCEEDED(lightCams[i]->shadowMap->Begin()))
	{
		base->D3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
				D3DCOLOR_COLORVALUE(1.0f,1.0f,1.0f,0.0f), 1.0f, 0L);

		base->camera = defaultCam;		// Set Camera back to default
		// Make sure the projection matrix is set back to default
		base->camera->setPerspective(D3DX_PI/4);
		base->camera->Render();		// Update camera matrices

		// Render Scene Using Receive Technique (for all subsets)
		base->subsetManager->Render(mainEffect,"Receive");

		lightCams[i]->shadowMap->End();
	}
	else
	{
		base->messageBox(base->hWnd,"Failed To Render To Shadow Map");

	}
}
```

When rendering the scene, the program must loop through each shadowMapLight instance, render the scenes z-depth from the point of view of the light to the DepthMap RenderTarget, and then use the depth stored in the DepthMap to calculate the shadows stored in the shadowMapLights shadowMap RenderTarget. In order to do this, the current

shadowMapLights cameras world matrix is passed into the "lightViewProjection" constant of the shadow mapping Effect, to provide a view and projection of the scene from the point of view of the current light:

```
if(FAILED(mainEffect->setValue("lightViewProjection",
                        &lightCams[i]->lightCam->WorldMatrix,sizeof(float)*16)))
{
      base->messageBox(base->hWnd,"Could not set light matrix");
}
```

5.1.2.3.2 Rendering the Z-Depth to the Depth Buffer

Once the lightViewProjection matrix has been passed to the ShadowMapping Effect, the scene must be drawn to the depthMap RenderTarget:

```
//Draw Scene to depth map Render Target
if(SUCCEEDED(depthMap->Begin()))
{

      base->D3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
            D3DCOLOR_COLORVALUE(1.0f,1.0f,1.0f,0.0f), 1.0f, 0L);

      base->camera = lightCams[i]->lightCam;    //Set View/Projection to Light
      base->camera->Render();                   //Update camera matrices

      //Render Scene using DepthMap Technique (for all subsets)
      base->subsetManager->Render(mainEffect,"DepthMap");

      depthMap->End();
}
else
{
      base->messageBox(base->hWnd,"Failed To Render To Shadow Map");
}
```

The first part of the code deals with beginning and clearing the depthMap render Target. Notice that the target is cleared to a white colour with no alpha. This is important so that the depth map blends correctly later. Next the camera of the Base instance (the main camera) is set to point to the current lights camera and the camera is updated, this ensures that any matrices required by the subsequent effect instances use the view and projection matrix of the current lights camera, that is, the scene is rendered from the point of view of the light.

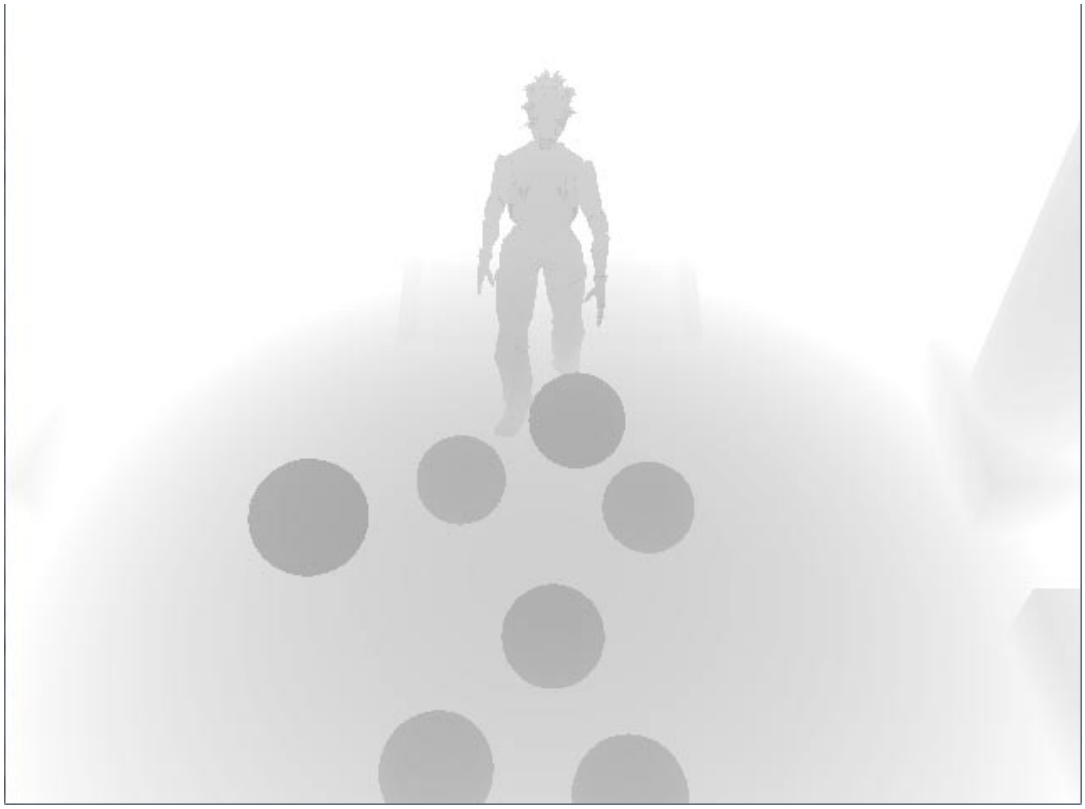
Finally, and most importantly, the scene is rendered via the SubsetManager (which contains a list of all subsets in the scene) using the mainEffect Effect instance (which is the shadow mapping effect), specifying the technique "DepthMap". Essentially every subset in the SubsetManager is rendered to the depthMap RenderTarget using the "DepthMap" technique of the shadow mapping Effect:

```
//Section for Rendering Depth Map
```

```
	float4x4 matWorldViewProjection     : WORLDVIEWPROJECTION;
	float4 lightPos                     : CAMERAPOSITION;
	float distanceScale = 0.000148f;
	float4x4 lightViewProjection;

	//Depth Map Render Target
	texture2D depth_map: RenderColorTarget
	<
		float2 ViewportRatio = {1.0f,1.0f};
		int MIPLEVELS = 0;
		string format = "D3DFMT_R16F";
	>;
	//Depth Map Sampler State
	sampler DepthRenderTarget = sampler_state
	{
		Texture = (depth_map);
		ADDRESSU = CLAMP;
		ADDRESSV = CLAMP;
		MAGFILTER =linear;
		MINFILTER =linear;
		MIPFILTER =none;
	};

	// Shadow Map Vertex Shader Output
	struct VS_OUT
	{
		float4 POS : POSITION;
		float3 LightVec : TEXCOORD0;
	};

	////////////////////////////////////////////////
	//    Shadow Map Vertex Shader
	////////////////////////////////////////////////////
	VS_OUT vs_main(float4 inPos : POSITION)
	{
		VS_OUT Out;
		//Project the object into the lights space
		Out.POS  = mul(inPos,matWorldViewProjection);
		//Output the distance from the light
		Out.LightVec = distanceScale * Out.POS;

		return Out;

	}

	////////////////////////////////////////////////////
	//    Shadow Map Pixel Shader
	////////////////////////////////////////////////////
	float4 ps_main(float3 lightVec : TEXCOORD0) : COLOR
	{
		//Return the distance from the light
		float4 colour = length(lightVec);
		return colour;
	}
//--------------------------------------------------------------//
// Technique Section for Depth Map
//--------------------------------------------------------------//
technique DepthMap
```

```
{
   pass pDepth
   {
         VertexShader = compile vs_2_0 vs_main();
         PixelShader = compile ps_2_0 ps_main();
   }
}
```

As described in the Shadow Mapping overview, the depth map pass of the shadow mapping Effect simply renders the Z-Depth of the scene from the lights point of view. The vertex shader transforms each vertex into its final world position (remembering that the current view and projection matrix are that of the light and not the camera). The LightVec output is simply the position of the transformed vertex. The pixel shader returns the length of the lightVec vector for each vertex. As described earlier, this value is a 16-bit float stored in the Red channel of the DepthMap RenderTarget texture.



5.1.2.4 Rendering The Shadow Map

Now that the Z-depth of each pixel from the lights point of view is stored in the DepthMap RenderTargets texture, It can be used to determine whether each pixel from the cameras point of view is in shadow or not. As mentioned earlier, we want to keep the shadows on a separate RenderTarget from the standard scene colours. This way they can

be blended as a post-process instead of limiting all shading to a single pixel and vertex shader.

The game code responsible for rendering the shadow map is very similar to that used to render to the depth map:

```
	//Render To Shadow Map Render Target
	if(SUCCEEDED(lightCams[i]->shadowMap->Begin()))
	{
		base->D3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
					D3DCOLOR_COLORVALUE(1.0f,1.0f,1.0f,0.0f), 1.0f, 0L);

		base->camera = defaultCam;		// Set Camera back to default
		// Make sure the projection matrix is set back to default
		base->camera->setPerspective(D3DX_PI/4);
		base->camera->Render();		// Update camera matrices

		// Render Scene Using Receive Technique (for all subsets)
		base->subsetManager->Render(mainEffect,"ShadowMap");

		lightCams[i]->shadowMap->End();
	}
	else
	{
		base->messageBox(base->hWnd,"Failed To Render To Shadow Map");
	}
```

First the shadow map texture of the current light is cleared (again to white with no alpha for blending purposes). Next the Base instance (default) camera is set back to the normal default camera (as opposed to the lights camera used in rendering to the depth map) and the camera is updated by calling its Render() function. Finally the SubsetManater (list of all visible subsets) is rendered using the ShadowMapping effect, this time specifying the “ShadowMap” technique:

```
//SECTION FOR RENDERING SHADOW MAP


	float4x4 matViewProjection: VIEWPROJECTION;
	float4 view_position : CAMERAPOSITION;
	float4x4 matWorld: WORLD;


	float4  shadowColour  = float4(0.2f,0.2f,0.2f,1.0f);
	float4  lightColour  = float4(1.0f,1.0f,1.0f,1.0f);


	//Shadow map Texture
	texture2D shadow_map: RenderColorTarget
	<
		float2 ViewportRatio = {1.0f, 1.0f};
		int MIPLEVELS = 0;
		string format = "A8R8G8B8";
	>;

	sampler ShadowMap = sampler_state
```

```
{
	Texture = (shadow_map);
	ADDRESSU = CLAMP;
	ADDRESSV = CLAMP;
	MAGFILTER =gaussianquad;
	MINFILTER =gaussianquad;
	MIPFILTER =none;
};



float shadowBias
<
	string UIName = "Bias";
	string UIWidget = "slider";
	float UIMin = 0;
	float UIMax = 1000;
	float UIStep = 0.0001;
>  = 0.0f;//0.000005f;

float backProjectionCut
<
	string UIName = "Back Projection Cut";
	string UIWidget = "slider";
	float UIMin = 0;
	float UIMax = 1000;
	float UIStep = 0.0001;
>  =80.0f;



// Projected Spotlight Texture
texture2D spotLight
<
	string ResourceName = "spotlight.jpg";
	string UIName = "SpotLightTex";
	string ResourceType = "2D";
>;

// Spotlight Sampler
sampler SpotLight = sampler_state
{
	Texture = (spotLight);
	ADDRESSU = CLAMP;
	ADDRESSV = CLAMP;
	MAGFILTER =linear;
	MINFILTER =linear;
	MIPFILTER =none;
};

struct VS_ShadowMap_OUT
{
	float4 Pos              :     POSITION;
	float3 viewVec          :     TEXCOORD0;
	float4 shadowCrd  :     TEXCOORD1;
};

//////////////////////////////////////////////////
//    Scene Vertex Shader
```

```
////////////////////////////////////////////////////
VS_ShadowMap_OUT vs_shadowMap_main(float4 inPos : POSITION)
{
	VS_ShadowMap_OUT Out;

	// Transform vertex into final world space
	inPos  = mul(inPos,matWorld);

	//Find distance from camera
	Out.viewVec = distanceScale * (view_position - inPos.xyz);

	//Project the object into the lights space
	float4 sPos = mul(inPos,lightViewProjection);

	// Use projective texturing to map the position of each fragment
	// to its corresponding texel in the shadow map.
	Out.shadowCrd.x = 0.5 * (sPos.z + sPos.x);
	Out.shadowCrd.y = 0.5 * (sPos.z - sPos.y);
	Out.shadowCrd.z = 0.0f;
	Out.shadowCrd.w = sPos.z;

	Out.Pos  = mul(inPos,matViewProjection);

	return Out;

}

////////////////////////////////////////////////////
//    Scene Pixel Shader
////////////////////////////////////////////////////
float4 ps_shadowMap_main(     float3 viewVec: TEXCOORD0,
                              float4 shadowCrd: TEXCOORD1) : COLOR
{
	// Radial distance
	float dep = length(viewVec);

	// The depth of the pixel closest to the light
	float shadowMap = tex2Dproj(DepthRenderTarget, shadowCrd).r;
	// A spot image of the spotlight
	float spotLight = tex2Dproj(SpotLight, shadowCrd).r;
	// If the depth is larger than the stored depth, this pixel
	// is not the closest to the light, that is we are in shadow.
	// Otherwise, we're lit. Add a bias to avoid precision issues.
	float shadow = (dep < (shadowMap + shadowBias));
	// Cut back-projection, that is, make sure we don't lit
	// anything behind the light. Theoretically, you should just
	// cut at w = 0, but in practice you'll have to cut at a
	// fairly high positive number to avoid precision issue when
	// coordinates approaches zero.
	shadow *= (shadowCrd.w > backProjectionCut);
	// Modulate with spotlight image
	shadow *= spotLight;


	// Shadow any light contribution except ambient
	float4 colour =  shadow;
	colour *= lightColour;
	colour += shadowColour;
	return colour;
```

```
        }

//--------------------------------------------------------------//
// Technique Section for Shadow Map
//--------------------------------------------------------------//
technique ShadowMap
{
   pass pShadowMap
   {
          VertexShader = compile vs_2_0 vs_shadowMap_main();
          PixelShader = compile ps_2_0 ps_shadowMap_main();
   }
}
```

The ShadowMap technique is responsible for creating the shadow map which is blended with the normal view of the scene to create the shadowed scene. The vertex shader is used to calculate the distance from the camera to the vertex in world space. Another transformation is used to calculate the projected texture coordinate of each pixel in the depth map to the positions of the shadow map (as illustrated by the original diagram). The vertices are then transformed into their final wold-view-projection position and passed along to the pixel shader.

In the shadow pixel shader, the shadow map is projected into the scene at the scene coordinates calculated in the vertex shader. Another texture containing a spotlight image is also projected into the scene using the same coordinates to create a more realistic representation of a spotlight and a softer edge to the light. Next the shadow value is calculated by performing the depth test. If the length of the vector from the current pixel to the camera is smaller than the value stored in the depth map then the pixel is in shadow. A back projection cut is used to stop the light projecting backwards. Finally, the shadow value is modulated with the spotlight texture image and a user defined light and shadow colour.

5.1.2.5 Blending with the Scene

The final step in shadow mapping is to blend the shadow map with the scene:

```
	//Begin Drawing to Back Buffer
	if(SUCCEEDED(base->D3dDevice->BeginScene()))
	{

		base->D3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER ,
			D3DCOLOR_COLORVALUE(0.0f,0.0f,1.0f,1.0f), 1.0f, 0L);



		base->camera = defaultCam;				//Set Camera back to default



		//Draw all subsets as normal and clear subsetmanager(no more geometry)
		base->subsetManager->Render();
		base->subsetManager->clear();

		//Set Render states to multiply next colour by the one on the back buffer
		base->D3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
		base->D3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR);
		base->D3dDevice->SetRenderState(D3DRS_SRCBLENDALPHA, D3DBLEND_ZERO);

		//Draw the shadowMaps sprite (a sprite with the shadow map as its texture)
		for(int i = 0; i < numLights; i++)
		{
			lightCams[i]->shadowMap->sprite->Render();
		}

		//Reset render states
		base->D3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
		base->D3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
		base->D3dDevice->SetRenderState(D3DRS_SRCBLENDALPHA, D3DBLEND_SRCALPHA);
		base->D3dDevice->EndScene();

		base->D3dDevice->Present(0, 0, 0, 0);



	}else
	{
		base->messageBox(NULL,"Could Not begin base->scene");
	}
```

The scene is rendered to the backbuffer without a global effect, i.e. using each material from the scene as it would normally be rendered. Once the scene is on the backbuffer, blending is used to modulate the backbuffer scene with the shadow map for each light which is drawn onto a sprite on the backbuffer.

5.1.3 Evaluation

As the image above shows, shadow mapping is capable of rendering shadows cast on multiple curved and flat surfaces in a visually pleasing manner. The main advantage of shadow mapping over other techniques for shadows such as shadow volumes, is that there is no extra geometry added to the scene. The scene is rendered multiple times but is not changed between renders. Shadow mapping is a screen space effect which is advantageous in that the rendering times for a simple and complex scene are almost exactly the same and it does not suffer so much from the fill-rate problems involved in rendering the large triangles often generated by shadow volumes. Shadow mapping does, however suffer some drawbacks due to inaccuracy and its use of floating point textures which are not supported on all hardware.

The implementation of shadow mapping for this assignment is far from perfect, there are some issues with the shadows when the camera gets too close to the lit area of the scene, this is simply a balancing issue with setting the “shadowBias” parameter in the shadow mapping effect. Apparently getting this value right for each scene is one of the trickier aspects of using shadow mapping. There are also problems with the sprite sizes on some machines which stem

from an issue with the DirectX ID3DXSprite interface. While there are a few issues with the current implementation it is successful in displaying a technique for rendering shadows within a complex scene at a relatively low cost.