GAME DEVELOPMENT with DirectX 9

Game Development with DirectX 9

Marco Bouterse
mcbouter@cs.vu.nl
1142828

Vrije Universiteit, Amsterdam
Faculty of Exact Sciences

October 2005

Bachelor Project Computer Science

Supervised by Dhr. Anton Eliëns

Contents

Introduction

Since the invention of the videogame in the 50s and 60s, which is most clearly marked by the creation of the well-known game “Pong”, it took a few more decades for this new form of multimedia to really break through. With the introduction of Personal Computers in the early 80s, the need for entertainment on this new medium also started to grow. Over the past years the game industry has rapidly grown into a billion-dollar industry, making computer games a driving force in the multimedia sector. While in the early days it was possible for a single person to create a commercial game, nowadays large development teams work multiple years with budgets of millions of dollars. Although the market for console games has outgrown the PC game market, the latter is still very important. In this market, DirectX is the language to speak. Almost every PC game has been developed using DirectX. The only concurrent, OpenGL, is sometimes also supported, but very few games are OpenGL only.

The goal of this project is to explore the possibilities of DirectX by building a library of wrapper classes on top of it. This library should hide the complex details of DirectX and provide simple easy-to-use functions. To demonstrate the use and functionality of the resulting library a small demonstration game will also be created. The library should be a reusable, extendable basis that can be used as the starting point of building games or other DirectX based multimedia application.

The organization of this document is as follows. Chapter 1 provides a brief history of DirectX and shows its basic structure. In chapter 2 the design for the library is presented. Chapter 3 provides an overview of the most important functions of the library. The design of the game is explained in chapter 4. The working of the game classes is the subject of chapter 5. Finally an evaluation of the project is given in chapter 6. Additional information and a complete overview of the library can be found in appendices A, B, C and D.

Chapter 1: DirectX

This chapter gives a short history of the DirectX library, describes the biggest changes in version 9 and gives an overview of the structure of DirectX.

1.1 History

The first version of DirectX was released in 1995 by Microsoft to give game developers the performance in Windows they could only get through DOS before, without having to support all kinds of different hardware. In version 2.0 Direct3D was added to support the hardware capabilities of upcoming video cards and the separate graphic accelerator cards that appeared in that period (3DFX). With DirectX 3.0 Microsoft finally won over the game development community by creating an API (Application Programming Interface) that was efficient, abstract and not hard to use [2]. At the time DirectX 5.0 was released (for some reason there has never been a version 4.0) most of the hardware manufacturers supported and developed drivers for it. In DirectX 6.0 the Retained Mode was dropped and Immediate Mode became the standard graphics mode to use. The changes in version 7.0 were mostly dealing with the new hardware Transformation & Lighting features of the new generation video cards. DirectX 8.0 was the first version since 5.0 that introduced major changes. The former separate graphics components Direct3D and DirectDraw (2D) were merged into the new component DirectX Graphics and a lot of new features were introduced to enable support for the latest video cards. One of the biggest changes was the introduction of vertex and pixel shaders (up to version ps_1_4 in DirectX 8.1).

1.2 DirectX 9

The API did not change much in version 9.0, but the biggest improvement is the introduction of better vertex and pixel shaders. Standards up to vs_3_0 and ps_3_0 are supported and a High-Level Shader Language (HLSL) has been introduced. The HLSL is a special shader programming language that replaces the old assembly like syntax [1]. The latest version is DirectX 9.0c and is freely available from the Microsoft website. The SDK that is needed to create programs that use DirectX is also freely available. It is continuously being improved to support the latest developments in graphics card technologies.

1.3 Organization of DirectX

DirectX 9 consists of six different components that can be used more or less independently. For example: it is possible to use the input component of DirectX with the OpenGL library for the graphics. The components of DirectX are:

- **DirectX Graphics**. Introduced in DirectX 8.0 merging DirectDraw and Direct3D to simplify the library and use the memory more efficient. This component handles everything that the graphics card can produce.

- **DirectX Audio**. Also introduced in version 8.0 merging DirectSound and DirectMusic. This component is an interface to the sound card.

- **DirectInput**. This component takes care of all input devices that a computer can handle. From mouse and keyboard to steering wheel and joysticks. It also handles force feedback effects.

- **DirectPlay**. This is the networking component and provides an interface to program multiplayer games.

- **DirectShow**. This component supports capturing and playback of multimedia streams. Formerly known as DirectX Media.

- **DirectSetup**. A small component that can check for the installed version of DirectX and can install another version. Useful for writing install programs.

To implement DirectX, Microsoft used a Hardware Abstraction Layer (HAL) and the Component Object Model (COM). These techniques enable maximum speed, a common interface and backward compatibility. The HAL is an abstraction layer right on top of the graphics card driver. The HAL represents the capabilities of a graphics card. Games can check the HAL for those capabilities and switch certain features on or off. A fallback method for switched off features may be implemented. Up to DirectX 7.0 a Hardware Emulation Layer (HEL) that could emulate certain features in software was provided. Since DirectX 8.0 an interface to use an HEL or pluggable software device is provided, but it is (almost) never used. Software developers have to write it themselves. This makes games using DirectX 8 or higher very dependent on what the graphics card can provide.

To handle version changes and backward compatibility the COM interface design is used. A COM component is normally a .DLL (Dynamic Link Library) file that can be accessed in a consistent and defined way. It has the following key properties:

- COM interfaces can never change.
- COM is language independent
- Only the methods of a COM component are accessible, never the data.

Every time that a component is changed, a new interface has to be provided. The old interfaces still have to be supported, which result in the ability to play DirectX 3 games if you have DirectX 9 installed on your computer. DirectX 9 introduced IDirect3D9 as the interface for Direct3D, but still supports IDirect3D2, which was used in DirectX 3.
Using this technology Microsoft has created a very powerful library that gives game developers an interface to get the ultimate performance out of the latest hardware.

Chapter 2: Design of the Library

This chapter presents the design of the library of classes built on top of DirectX 9. First an overview of the library is given, showing the different components and its classes. Next the different components are briefly discussed. The design presented here is heavily based on the library design from the book "Game Programming All in One" [2]. I have used the library presented in this book as the basis for this project and adapted it to support DirectX 9 (the book uses DirectX 8). Where the book only goes into 2D graphics, I have extended the Graphics component considerably to support 3D too, using several tutorials [7-10] and the book "Beginning Direct3D Game Programming" [1]. The purpose of this library is to create a reusable code-base that can relieve the game programmer from having to deal with the inner workings of DirectX and have more understandable game code.

2.1 Overview

Figure 2.1 shows the overall design of the library, which consists of five more or less separate components.

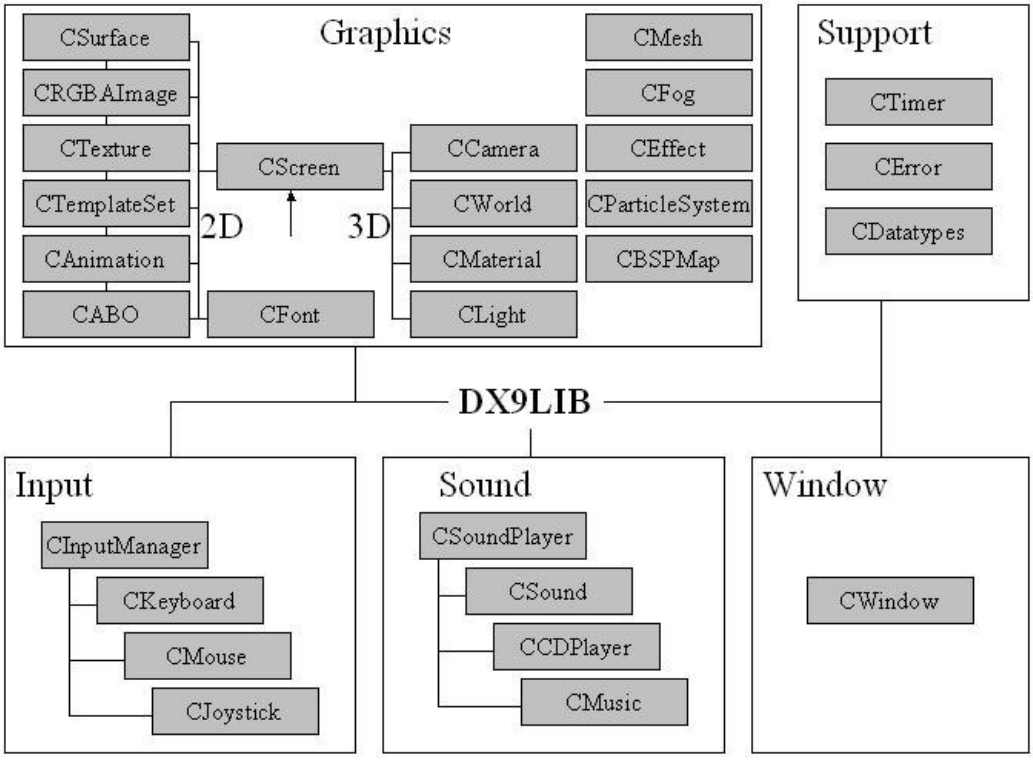

Figure 2.1 - Overview of the library

The **Support** component handles supportive tasks like error checking and timing. The **Window** component takes care of the creation of the base window (needed for every Microsoft Windows compatible application) and hides the Windows specific code. The **Sound** component provides classes to play well-known audio formats like WAV, MP3 and MIDI and also supports CD playback. The **Input** component provides an interface to the keyboard, the mouse and game-specific input devices (like gamepads or joysticks). The biggest and most complicated is the **Graphics** component, which contains classes for rendering 2D and 3D graphics and special effects.

2.2 Support Component

The Support Component contains classes that are not directly using DirectX, but are needed for the rest of the library. The `CTimer` class provides time and date functionality. `CDatatypes` defines a couple of useful macro's and datatypes, providing names that contain the size and type of the datatype to improve portability. Finally `CError` contains the definitions of all possible errors that can be generated by the library.

2.3 Window Component

The Window Component contains a single class, `CWindow`, that takes care of all Windows related code that is needed to set up a basic window, needed for any application to run on the Windows platform. It provides a few simple calls to quickly setup an empty window.

2.4 Sound Component

The Sound Component has a main class `CSoundPlayer`, that initializes the DirectSound component and contains three other classes: `CSound`, `CMusic` and `CCDPlayer`. `CSound` represents a sound buffer for sound effects in the WAV format and provides functionality to control pan, volume and pitch. `CMusic` adds support for other music formats like MP3 and Midi, that are mainly used for background music. Finally `CCDPlayer` provides an interface to playing audio CD's using the computer's CD-ROM player.

2.5 Input Component

The Input Component's main class is `CInputManager` that initializes the DirectInput component. `CKeyboard` acts as an interface to the keyboard, `CMouse` provides the same function for the mouse and `CJoystick` provides an interface to game control devices like gamepads and joysticks.

2.6 Graphics Component

The most complex component is the Graphics Component, that is initialized by the `CScreen` class. `CRGBAImage`, `CSurface`, `CTexture`, `CTemplateSet`, `CAnimation`, `CABO` and `CFont` provide 2D functionality. They represent image buffers, background images, textures, animations and text. `CABO` uses most of the other classes to represent a 2D animated object, that can contain multiple animations, collision detection and more. With these classes, almost all 2D games can be built. The other classes add 3D functionality. `CCamera` represents the viewer in a scene. `CWorld` is an interface to the world matrix. `CMaterial` represents a material, used for lighting calculations. `CLight` represents a light (can be a spotlight, directional light or point light). `CMesh` represents a 3D model that can be loaded from a .X file that is created with some sort of 3D modeling program. `CFog` represents fog, that can be added to a scene to limit visibility or set a certain mood. `CEffect` represents a shader effect that can be loaded and used to render objects with (to replace the classic material/light method). `CParticleSystem` is a multipurpose special effects class that uses point sprites to achieve effects like snow, fountains, fire or smoke. Finally `CBSPMap` can be used to load 3D scenes from .BSP maps and render them.

Chapter 3: Implementation of the Library

This chapter presents an overview of the most important functionality of the classes of the library and the difficulties encountered when programming them. After each class or set of classes a test program was created to test the working. These test programs can be found on the CD-ROM. An overview of all test programs is given in appendix A. The order followed here is the order in which the classes were created during the project. For an extensive overview of the classes of the library and all their functions see appendix D in which a complete documentation is provided.

3.1 Programming for Windows

To get any application running on a computer that is running the Microsoft Windows operating system, a window has to be created first. To show even a simple empty window, quite some code is needed. One of the intentions of this project is to create a library that simplifies the low-level DirectX interface and supplies a reusable base to develop games and other multimedia applications. Therefore the first thing to do is develop a reusable window class that provides a single call to create an empty window.

3.1.1 Support Component

Before starting to program the window class, there are a few supporting files to create first. `CDatatypes.h` contains type definitions of often used types and assigns names to them that indicate the type, length and whether they are signed or unsigned. For example an unsigned long (32 bits) integer becomes "`u_int32`".
`CError.h` will contain all error definitions that can be generated by the library. Having a unique return value for every possible error simplifies debugging and enables testing for specific errors. As the library is being developed, all possible error values will be added to this file.
The final supporting file is a little more complicated. `CTimer` is a general purpose timer class that uses the built-in hardware timer to supply accurate timing information. The most important function is `GetDelta()` that returns the time that passed between the last two calls to `Update()` in seconds. This timer is very accurate and useful for real-time applications. Finally it also supplies functions to get the current system date and time information.

3.1.2 Window Component

After the supporting files the `CWindow` class can be developed. As said before this class shields all overhead of creating an empty window and takes care of the message processing. To create a simple window a user defined class must inherit from `CWindow` and implement the virtual `Frame()` function. This function is called whenever there are no messages to process, thus will normally be the place where all the application processing and rendering is done. An instance of this class must call `Create()` with the appropriate parameters to set up the window and `Run()` to enter the message loop. To leave the message loop (thus quitting the application) the `Frame()` function must return '`b_false`'. Additional functions to change size and position are also provided. A detailed description of all the functions and parameters can be found in appendix D. The first test program shows an empty window with title, using these first classes.

3.2 The Graphics Component (2D)

3.2.1 Direct3D Basics

Now that the window basics are covered, the DirectX programming can really start. The `CScreen` class deals with setting up Direct3D. It hides the details of setting up a Direct3D object and device (video adapter) by providing a simple interface. The `Init()` function creates a DirectX 9 object. With the `SetMode()` function the default display adapter is used to set up a window with the as parameters provided properties. It also provides functions for clearing the screen to a certain color and drawing some basic 2D shapes (line, square, circle). To make sure only one instance of this class can exist it is implemented as a singleton. To demonstrate the drawing functions a test program is created that shows a Direct3D window with a line, a square and a circle. Every second one more vertex is added to the circle, which makes the circle smoother.

3.2.2 Simple Surfaces

The next classes deal with 2D graphics. The first one, `CRGBAImage`, represents a raw image: a buffer that contains all color information for each pixel in the image. The important functions are `LoadFromBitmap()` and `LoadFromTarga()`, that load image information from existing BMP and TGA files into a buffer. This class is not entirely useful by itself, but is used by the following classes to hold image data. The first class that uses it is `CSurface`, which represents a static image that can be displayed on the screen, but has limited capabilities. It is only usable for static images that stay inside the screen boundaries (no clipping), like backgrounds. The most important functions are `Create()` and `Render()`, used to create and display an image. The third test program shows all classes developed up until now in action: it creates a window with a Direct3D screen and displays a background picture.

3.2.3 Animated Objects

`CTexture` also represents an image using a raw image as its source, but this image is used more flexibly by the classes that follow to obtain images that are scalable, can be rotated and support transparency. `Create()` is the most important function of this class and off course is used to create a new texture from a raw image object. The following class uses `CTexture` and is used to represent a template set, an organized set of images, laid out on a grid. `CTemplateSet` uses a texture containing a grid of images and provides access to the individual cells of the grid. It is mainly used for animation as implemented in the `CAnimation` class that will be discussed next. The important functions are `Create()` and `GetUV()`, which return the rectangle of the texture that contains the image in a specific cell of the grid. By using a template set it is possible to have lots of smaller images in one texture, needing only one source file, which is more efficient. `CAnimation` is the class that represents a single animation and uses a `CTemplateSet` object to hold the frames for the animation. `Create()`, `Update()` and `Render()` are the most useful functions of this class. The functionality of `Create()` and `Render()` is obvious, `Update()` advances the animation one frame.
The final 2D graphics class puts everything together, using almost all classes that have been created up until now. `CABO` represents a 2D object with support for multiple animations, transformations (rotate, scale) and collision detection. It also provides a single easy-to-use function to load an Animated Blittable Object (ABO)

from file, hiding all details of the previous classes. `LoadFromFile()` accepts a file name as a parameter, referring to a file with a specific format, that contains the filename of a source image (containing a grid of frames), the type of image, the transparent color (color key), the number of animations and for each animation the number of frames and the cell coordinates of each frame. See the documentation for `CABO` in Appendix D for the specifics of the format. With this class the 2D graphics part of the library is finished and can be used to create the graphics for standard 2D games like Tetris and BreakOut. The test program shows some of the features of the ABO class, by displaying a very basic animation (only 2 frames).

3.3 The Input Component

Before getting into the difficulties of 3D programming, I decided to first develop the library for the other (simpler) components of DirectX. The first one is DirectInput, the component that deals with all input devices. This component is very important for a game library, because it provides the property that is fundamental for games: interactivity.

The base class in this component is `CInputManager`. This class just initializes the DirectInput object of DirectX and its most import method is `Init()`, that takes care of the initialization. The class is implemented as a singleton to ensure there is only one instance at a time.

The next class is `CKeyboard`, that takes care of all input from the keyboard. The `Init()` function initializes the device object and must always be called first. `Update()` refreshes the key information and will record the current state. It must be called before `IsButtonUp()` and `IsButtonDown()`, that are used to query the state of a specific key. A small test program shows the use of this class. The animation can be moved around using the arrow keys, ESC exits the test program.

The `CMouse` class is very similar to the keyboard class, only it has three more functions: `GetXAxis()` and `GetYAxis()` for retrieving the relative movement of the mouse and `Clear()` to reset the mouse buffer. The test program now allows mouse control and the left and right mouse buttons rotate the animation.

The final input class is `CJoystick`, which represents a joystick or gamepad device. The provided functions are comparable with the mouse except for the `Init()` function that needs some extra configuration data. The test program shows the class working and is similar to the mouse test program. During testing it appeared that this class causes a crash if `Init()` is called, when there is no gamepad attached to the computer. Unfortunately I haven't been able to solve this problem.

3.4 The Sound Component

The next component handles sound, another very important aspect of video games. The organization of this component is very similar to the input component.
`CSoundPlayer` is the entry point of the component that provides the `Init()` function for initializing the DirectSound object.
`CSound` represents a single sound object. A sound can be loaded from a .wav file using `LoadFromFile()` and played using `Play()` and `Stop()`. Additionally there are some functions to alter the properties of the sound (frequency, volume, pan).

The final class of this component does not actually use DirectX, but uses the Media Control Interface to control CD playback functionality. The class `CCDPlayer` provides the functions `Eject()`, `Play()` and `Stop()` to control the system's CD player in an recognizable way. The test program for this component shows a couple of the functionalities. An explosion sound is repeatedly played and its properties can be controlled using the keyboard. The CD playback is also shown, although the next number function doesn't seem to work really smooth: it skips more than one number.

3.5 The Graphics Component (3D)

3.5.1 Vertex Buffers

After exploring the other components of DirectX a little bit it is time to return to the core of DirectX, Direct3D and try to do real 3D graphics programming. The `CScreen` class is adapted slightly (the capability of the graphics card to do hardware vertex processing is investigated and the result is used when creating the device object). The first test program displays a colored triangle using a vertex buffer. This is done by creating a buffer with vertex data (consisting of x, y , z and color components), creating a vertex buffer and copying the vertex data into the vertex buffer (after locking it). Using `SetStreamSource()` a specific pipeline on the graphics card is assigned to the vertex buffer and with `SetFVF()` the format of the vertices is set. Finally the triangle is drawn using `DrawPrimitive()`.
The next step is to set up the perspective and the camera view. To control the perspective a function `SetPerspective()` is created in the `CScreen` class that accepts three parameters: field of view, near clipping plane and far clipping plane. A special class is created to represent the camera. `CCamera` provides functions to set the camera properties (eye position, the point it is looking at, up vector) and the call `SetCamera()` that needs to be called every frame to position the camera according to its current properties. To test the new class there is a little program that displays the triangle in a perspective volume, viewed by a camera. By pressing the up and down keys it is possible to zoom in or out.

3.5.2 Displaying Text

Before moving on with 3D, the `CFont` class is created, that allows for a very simple interface to set up a font and display text with it. The important functions are `Init()` to set up the type of font, `SetPosition()` or `SetTextArea()` to control the position of the text and `DrawText()` to display a string in a specific color. Again there is a test program, showing different types of fonts being used.

3.5.3 Spinning Cubes

To manipulate objects in world space, the world transformation is used to achieve rotation, translation and scaling of objects. To provide an interface to the DirectX 9 world matrix, the `CWorld` class is used. This class can alter the world matrix and provides the functions `RotateX()`, `RotateY()`, `RotateZ()`, `Translate()` and `Scale()`. A test program shows the use of it by showing a colored cube that can be manipulated by pressing certain keys. Another test program shows a small cube orbiting a large cube plus a frames per second counter and uses time based instead of frame based animation.

3.5.4 Texture Mapping

The next step is to add textures to the cubes. We simply use the `CTexture` class developed earlier to load and set the active texture. One new function is added to this class: `SetModulate()` can be used to switch color modulation between texture and object color on and off. This is demonstrated by another test program.

3.5.5 Turn on the Lights!

To create more realistic looking images, lights must be added to the scene. A class `CLight` is created that provides the functionality to do this. The static function `EnableLighting()` can be used to enable or disable lighting in DirectX. Another static function `SetAmbientLight()` can be used to add ambient light (constant light factor that is everywhere in the scene). With the `Create()` function, the parameter specifies the type of light that should be created. There are three different types of lights: a point light has a position and emits light in all directions, a directional light has no position, but emits parallel rays of light in a specific direction (like the sun), finally a spot light has a position, a direction and a cone that limits the range of the light (like a point light, only focussed on a specific point). The `Switch()` function is the virtual light switch: if the light is on, it will turn the light off and vice versa. The rest of the functions can be used to manipulate the light properties. To use light on objects in the scene, these objects must have material properties set. Therefore the class `CMaterial` is created, that is used to represent a material. After creating a material it can be used to call the `SetActive()` function and draw the object that uses the material. The test program that shows this has also a couple of modifications in the way that the cubes are drawn. The vertex buffer now stores normals instead of colors and because the normals differ per face of the cube, fewer vertices can be shared, so each face is drawn separately now (before all sides were drawn as one triangle strip).

3.5.6 Complex Models

Until now I used a vertex buffer in the test programs to render cubes. The `CMesh` class provides support for more complex models in an easy way. The `LoadFromX()` function can read a 3D model from a file in the Microsoft .X format. This type of file can be generated as or converted from output of 3D Modeling programs like 3DStudio MAX or Maya. After loading the model is rendered by simply calling the `Render()` function. In the test program I used the tiger model that is used in the DirectX SDK sample programs.

3.5.7 Advanced Camera

To support the camera control needs for gaming, the `CCamera` class developed earlier is extended with a couple of functions. `Move()` and `Strafe()` allow camera movement in the [x, z]-plane. `Move()` moves the camera forward or backward and `Strafe()` takes the camera to the left or right. `Rotate()` comes in two flavors: the first one takes an angle and a rotation vector as parameters and rotates the camera around it's own position, the second takes as extra parameter a point and rotates the camera around this point. The second is useful for creating 3rd person games. The rotating functions use quaternions to calculate the rotations. Quaternions are four dimensional vectors, the fourth component being a rotation. The advantage of using quaternions is that they are faster and require less code. Again there is a test program to show the new camera features being used to create controls for a 1st person view.

3.5.8 Fog

Before getting to advanced effects, the `CFog` class is created to allow simple fog effects. The further away an object in the scene is, the more it will be blended with the fog color, effectively reducing the visibility range. It can be used to boost performance (all objects further than the fog range don't need rendering or just to add realistic looking fog effects. In the test program the fog color is the same as the background color, making the model seem to disappear slowly when the camera goes further away.

3.5.9 Shader Effects

Up until now I created classes to support the old fashioned way of creating lighting effects. With the advent of DirectX 8.1 vertex and pixel shaders were introduced. Vertex shaders should replace the task of the former Transformation & Lighting phase in the graphics pipeline and pixel shaders are there to create effects formerly achieved by multi-texturing. To write these shaders DirectX 9 has introduced a High Level Shader Language (HLSL), a C-like programming language that can be used to write all kinds of effects. To load these effect files into a scene, the `CEffect` class is created. It has the Create() function to create a new effect, the `SetMatrix()` and `SetVector()` functions to provide variables to the shader programs, the `SetTechnique()` function to select a specific rendering technique (defined in the shader program) and the `RenderMesh()` function that is capable of rendering a mesh using the selected shader technique. The big advantage is that a different effect can be achieved by loading a different file and without having to re-build the program. The first of the shader test programs shows a very basic shader that transforms the vertices using the current world, view and projection matrix and provides ambient lighting only, so every pixel is lit the same amount. The second test program adds diffuse lighting: there is a single directional light that shines on the model. The third basic test adds specular light to the lighting model, resulting in the appearance of reflection of light off the model. Finally the last basic shader test adds a texture to the model.

I also extended the `CTexture` class with a `CreateFromFile()` function to load textures directly from file, instead of first having to create a raw image. It also supports more file types.

The final shader program uses a point light instead of a directional light, which adds an attenuation factor to the light calculation. The further away the light source is, the less influence it has on the model.

3.5.10 Particle System

To create special effects like fire, sparks, snowflakes, smoke, etc. often a particle system is used. A particle system is a collection of individual elements (particles) that have individual attributes like position, speed, direction and color. The system itself also has properties like origin and external forces. By manipulating these properties all kinds of effects can be simulated. The `CParticleSystem` class is an implementation of a multi-purpose particle system. It can be configured with a `ParticleInfo` structure, containing all kinds of parameters that are used for creating new particles. These parameters can also be altered while the particle system is active to provide even more flexibility. A test program shows the particle system in action. By pressing the buttons 1-4, a different effect can be chosen.

3.5.11 Viewports

Without using multiple computers and networking, it is still possible to create multiplayer games. This is done by splitting the screen in multiple viewports, that each show a different part of the game world, related to each player. The `CViewPort` class allows the creation of multiple viewports. With `Create()` a new viewport with specified properties is created. Using viewports changes the normal way of rendering between `StartFrame()` and `EndFrame()`. Instead for each viewport the `Begin()` function must be called, that takes the clear color for that viewport as a parameter. After rendering to the viewport is done, the `End()` function must be called, unless it is the last viewport within the frame. After the rendering for the last viewport has been done `EndLastViewPort()` must be called. This function takes care of presenting everything to the screen.

3.6 The Sound Component: CMusic

To support a wider array of audio formats, like MP3 and MIDI, the Sound Component is extended with an extra class, `CMusic`, that provides that kind of functionality. It provides two functions to load these formats: `LoadMP3()` and `LoadMidi()`. The latter also supports the WAV format as well. With the `Play()` function, playback is started and `Stop()` stops it. Finally the `IsPlaying()` function can be used to check if the music is currently playing.

3.7 Useful Extensions

3.7.1 Collision Detection

A very important aspect of game programming is collision detection: the technique that is used to find out when two objects collide with each other. Although this is not a real graphics issue the DirectX library has some functions to support it and therefore I also added basic support for it to the library. The method used is called Axis Aligned Bounding Box (AABB), which creates a box that is aligned with all three axis around game objects and uses two points of this box (minimum and maximum bounds) to do simple collision detection. The collision detection functionality is added to the `CMesh` class. `InitBoundingBox()` calculates the AABB in model space. This function should be called after creating the mesh. `UpdateBoundingBox()` takes the current world matrix as a parameter and transforms the AABB to a Bounding Box in world space (doesn't need to be axis aligned anymore) and transform this box back to an AABB. This function should be called before rendering the mesh. To check for collisions the function `CheckCollisionWith()` can be used, that takes another mesh as parameter. This method of collision detection is not very accurate, but it is very fast and realizes acceptable results.

3.7.2 Binary-Space Partitioning Maps

To speed up the rendering of game worlds, often Binary-Space Partitioning (BSP) Trees are used to sort and store the geometry. By storing polygons using their relations (in front of, in back of another) the engine can quickly decide which ones to draw and which ones not. It also offers the possibility to do collision detection with world geometry. The `CBSPMap` class only supports the loading and rendering of Quake 3 BSP maps using textures and light maps. It doesn't offer support for collision detection and model loading. Only TGA textures are supported.

3.7.3 Lost Device Handling

Whenever the bit depth of the screen is changed while in windowed mode, or the application loses focus while in full screen mode, the Direct3D device maintained by `CScreen` changes from operational state to lost state. This can be detected by checking the return value of `EndFrame()` or (`EndLastViewPort()` when viewports are used). The application must take steps to recover from this situation. Whenever it occurs, the `CheckDeviceState()` function should be called before drawing. If it returns `errorDeviceLost`, the device cannot be recovered yet and the frame should be skipped (just return `b_true`). If the return value is `errorNotReset`, the device is ready to be reset. First all device objects must be invalidated, the easiest way is to call a function `InvalidateDeviceObjects()` that calls the `Invalidate()` function on all library objects that are being used and have this function. If the application manages any Direct3D resources itself that are created with D3DPOOL_DEFAULT, these must also be released. After invalidating all device objects, the device can be reset, by calling the `Reset()` function of `CScreen`. Next all objects must be restored / recreated again. Provide a `RestoreDeviceObjects()` function that first resets the perspective, restores any changes to the default render states, sets up lighting and recreates all device objects that were released (or calls their `Restore()` function if they have one) in the `InvalidateDeviceObjects()` function. The `RestoreDeviceObjects()` function can also be used for initializing the device objects. After restoring, the device should be operational again. However there still seems to be a bug in the process, requiring to invalidate/restore two times, before everything works again. For example, when in full screen, ALT-TAB out of the application and back will not correctly display the objects, ALT-TAB out and back again does restore the objects correctly.

Chapter 4: Design of the Game

In this chapter the design is presented of the demo game created to show the usage and possibilities of the wrapper library.

4.1 General Overview

The game is designed to be very simple, but still showing as much functionality of the library as possible. The main concept is that it is an arcade style space racegame. The player controls a spaceship that flies around a circular track and has to pick up bonus objects to score points and dodge mines that can destroy the ship. There are also repair power-ups to repair the damage to the ship. After each lap the speed is increased and it gets harder to hit the bonus objects and dodge the mines, but the points awarded for hitting a bonus object also increase, so the further the player gets, the higher the award for hitting a bonus object. The goal of the game is to get as far as possible and register the highest score ever.

4.2 Target System and Requirements

The game uses DirectX and therefore will only be playable on the Microsoft Windows platform, that has the latest DirectX 9 runtime installed. It will need an Intel Pentium 4 or AMD Athlon XP processor and about 512 MB of RAM memory. A graphics card that is fully DirectX 9 compatible will also be needed. To install the game on harddisk about 7 MB of free space is needed.

4.3 Theme: Graphics and Sound

The setting of the game is on a rough planet, somewhere in the universe. The graphics will be somewhat dark and the planet consists mainly of rock. The music theme and sounds are inspired by old arcade style games.

4.5 Main Menu

- **Start**. The player will start a new game with an undamaged ship and no score.
- **Resume**. Only available when the player was playing a game and pressed ESC to go back to the menu. The game will continue were it was before switching to the menu.
- **Settings**. Here the player can switch the background music on and off.
- **Highscores**. Here an overview is given of the highest scores achieved with the game.
- **Quit**. The game exits and the player will be back in Windows.

4.6 Playing a game

After a short introduction sequence the main menu will be shown. To start playing the player selects “Start” and presses return, space or the left mouse button. A short fly-through of the scene is shown, which can be skipped by pressing space or enter. Now the camera shows the ship from behind. After a countdown from 3 the ship starts moving and the player gets control. With the arrow keys, the player can move the ship to the left or the right. Doing this he can hit green bonus objects, that generate points, which will be added to his score. There are also mines on the track that will move towards the player if he gets near. These mines should be evaded, otherwise they will explode and result in damage to the ship, shown with a damage bar on the screen and

by smoke, coming from the ship. If the ship is damaged the player can repair the it by hitting special repair power-ups. When the player finishes a lap, the ship's speed is increased, the mines move towards the player more aggressively, but the bonus objects are also worth more points. The game continues until the player fills up the damage bar (after hitting mines) and explodes the ship. The main objective is to try to achieve the highest score ever.

4.7 Code Design

Figure 4.1 gives an overview of the code design for the game. The entry point is `GameMain`, which contains the `WinMain()` function, the Windows equivalent of `main()`. It creates a new custom `GameWindow`, that implements the `Frame()` function of the `CWindow` class. Inside the `Frame()` function, that is called each frame, the `Process()` and `Render()` functions of `CGame` are called.

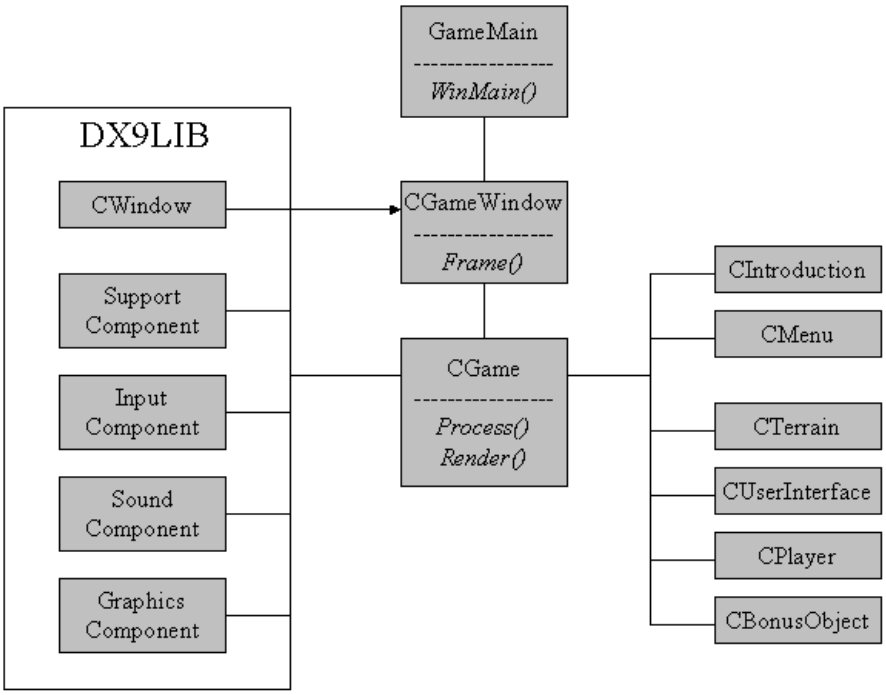

Figure 4.1 Global design of the game

`CGame` is a state machine that determines which classes need processing and rendering and dependent on the current game state, the `Process()` and `Render()` function of one or more of the classes to the right are called. `CGame` also takes care of input processing. `CIntro` handles the introduction sequence and `CMenu` controls the main menu. `CTerrain` is responsible for the scene in which the game takes place. `CUserInterface` handles all status information that is displayed on the screen during gameplay (score, highscore, current lap, damage). `CPlayer` contains all player information and is responsible for the ship model, that the player controls. Finally `CBonusObject` represents an object on the track that the player can interact with, which can be a point scoring object, a repair power-up, but also a mine. For this project I haven't developed a very sophisticated game engine / architecture, because that was not the focus of this project. For serious games, a more efficient and complicated game engine should be constructed.
Chapter 5 provides more detailed information about the inner workings of the classes.

Chapter 5: Programming the game

This chapter describes the inner workings of all the game classes that were presented in the previous chapter. A description of the most important functions and their tasks is given. The source code of the game classes can be found on the CD-ROM (see also Appendix A).

5.1 GameMain

This is the entry point of the application and contains only one function: `WinMain()`, which is needed for any application using windows, like a normal C/C++ program must start with `main()`. With a few simple calls a `CGameWindow` object is created and initialized. If initialization succeeds, the `Run()` method is called to enter the real-time message loop, which handles Windows-related messages and calls the `Frame()` function of `CGameWindow`, whenever there are no messages to process. This code can be used for any windows application, changing only the window title and possibly the window size.

5.2 CGameWindow

This class initializes the graphics component by calling the `Init()` and `SetMode()` functions of `CScreen`. Also the `Init()` function of `CGame` is called to initialize the game classes. In the Frame() function the Process() function of `CGame` is called to do the needed processing. If this function returns a negative value, the game has exited and `b_false` is returned to stop the message loop and exit to Windows. After processing the frame buffer is cleared and between `StartFrame()` and `EndFrame()` the `Render()` function of `CGame` is called to render the current frame to the screen. The rest of the code is occupied with lost device recovery, which was explained in more detail in chapter 3.

5.3 CGame

This is the main class of the game. It controls the flow of the game and calls the remaining classes when needed. It is organized as a state machine, that decides which functions to call depending on the current game state. The game states are:

- `GameStateIntroduction`: Short intro displaying text
- `GameStateMenu`: Main menu
- `GameStateFlyThrough`: Cinematic flythrough of the scene
- `GameStateCountDown`: Countdown before player gets control
- `GameStatePlaying`: Actual game is running
- `GameStateGameOver`: The player's ship has been destroyed

The `CGame` class also manages all input and the star field particle system that is shown as background during the introduction and the menu.
The `Init()` function initializes the Input component, the Sound component and all other game classes. The `Process()` function first calls `ProcessInput()` to handle the input and does the processing depending on the current state, what usually comes down to calling the `Process()` function of one or more of the other classes. The `Render()` function works similar, calling the `Render()` functions of other classes to render all screen objects. `InvalidateDeviceObjects()` and

`RestoreDeviceObjects()` are concerned with lost device recovery and call their equivalents of the other game classes. The private functions `ProcessFlyThrough()` and `ProcessCountDown()` are used to move the camera through the scene and to show a 3, 2, 1, GO! countdown respectively.

5.4 CIntro

When the game is started a star field is shown (controlled by `CGame`) and a number of sentences fade in and out in the middle of the screen. `CIntro` takes care of this. `Process()` determines the level of fading depending on a timer and switches to the next line, when needed. If all lines have been shown, it returns –1 to indicate the `CGame` class that is has finished and the game state should be advanced. The `Render()` function sets the text area to be the entire screen and renders the text using the current alpha level in the middle of the screen.

5.5 CMenu

After the introduction (which can be skipped by pressing escape, the spacebar or enter) the main menu is shown. This is where `CMenu` comes in. It is also organized as a state machine, where the different menu states are:

- `MenuStateBuildup`: The menu is appearing from the middle of the screen.
- `MenuStateSelect`: The player must highlight and select his choice
- `MenuStateBeginGame`: The player has selected “Start”
- `MenuStateResumeGame`: The player has selected “Resume”
- `MenuStateQuitGame`: The player has selected “Quit”
- `MenuStateSettings`: The player has selected “Settings”
- `MenuStateHighscores`: The player has selected “Highscores”

Depending on the current state `CMenu` does its processing by calling `Process()` and the menu is rendered with `Render()`.

5.6 CTerrain

The terrain class contains a terrain generator and additionally manages the background and the scene objects. The terrain generator is based on the one presented in chapter 8 of the book “OpenGL Game Programming” [3]. The OpenGL code was translated to DirectX code. It loads the terrain using a heightmap, a bitmap with grayscale values (white is the highest point and black the lowest). In the `Init()` function this map is loaded into a vertex buffer. A buffer for the track is also created. The terrain does not need much processing. The `Process()` function only triggers vulcano eruptions at random times during the flythrough state. The Render() function calls different functions to display the different parts of the scene. `DrawSkybox()` shows a space environment by drawing a cube around everything with space panorama textures on the inside. `DrawTerrain()` shows the planet surface with the information from the heightmap and a rock texture. It also draws the track using a different texture. Next the tunnel is drawn, which is a `CMesh` object that needs to be rotated and translated into the correct position. Finally the particle system is rendered if needed to show the eruptions.

5.7 CPlayer

In the `CPlayer` class most of the game logic is situated. The `Process()` function takes care of the ship movement. To achieve smooth controls the movement to the left and the right is computed using the basic physics of applying forces to an object. If no force is applied, the ship is slowed down, to simulate friction. The `Render()` function is used to position and display the ship at the right position. If necessary the smoke and explosion particle system are shown. `MovePlayer()` can be used by other classes to apply a force to the ship. `AdjustDamage()` and `AdjustScore()` enable access to these properties of the player. The `SetCamera()` function calculates the position and orientation of the camera based on the player's position. Finally there are a few functions to enable other classes to retrieve important values.

5.8 CBonusObject

First designed to represent a single bonus object containing a `CMesh` object holding the model and some additional information, but completely rewritten, because this proved to be highly inefficient and resource consuming. This version contains only three `CMesh` objects, one for each separate object type. The class contains an array of structures that holds the important information of all the bonus objects in the game. In the `Init()` function the materials are created for the different objects and also the sounds are loaded. The `Reset()` function (re)fills a single entry in the array of structures. First the type of the object is determined using a random value. The chances are: 75% to get a point scoring object, 24% to get a mine and 1% to get a repair power-up. Next a random position is assigned, that is not right in front of the player's current position. Finally the rendering flag is set to true to make sure the object will be shown. `ResetAll()` just goes through the entire array and calls `Reset()` for each entry. `Process()` is the most important function of this class. It determines whether the object is visible to the player or not and sets the rendering flag accordingly. It also does the collision detection between the objects and the player and takes the appropriate action if a collision occurs. Finally the `Render()` function displays the visible objects at the proper positions.

5.9 CUserInterface

This class is responsible for displaying status information using 2D graphics. It uses `CABO` for the graphical part and `CFont` for the numbers. The `Process()` function retrieves the current lap, damage and score values from the `CPlayer` class, formats the numbers into a string and selects the right animation frame for the damage bar. The number of frames per second is also retrieved and will be rendered to the top left of the screen if the player presses 'F' during the game. The `Render()` function puts everything on the screen.

Chapter 6: Conclusions

With this project I have studied the workings of a part of DirectX, Microsoft's interface between multimedia code and hardware. Although there is a lot left unexplored, I did use a considerable number of functions from different components of the DirectX library (DirectX Graphics, DirectX Audio and DirectInput). The goal set beforehand was to develop a library that is a reusable base for multimedia projects and provides a simpler interface than the original DirectX library. I think that the resulting library of this project cannot be considered complete enough to be used in practice, but it provides a very good starting point to build a custom interface library to be used in projects. To be useful in practice the library must be extended more to support at least also DirectPlay functionality for multiplayer games. It should also be better debugged on different machines to reveal bugs I haven't discovered. The second part of the goal is clearly achieved. In most cases a considerable amount of code is hidden behind a few simple function calls, for example to set up an empty window or to initialize Direct3D.
Creating a simple game was a very good way to evaluate the use of the library. While programming a few bugs and weak points were discovered, bot in general it was good to work with. The result is a very simple, but complete game that shows most of the library classes in action.

Besides everything I have learnt about DirectX, this project was also very valuable for my C++ programming skills. During the project I have encountered numerous bugs and code that wouldn't work, but most of the time I managed to solve the problems, using the DirectX documentation, internet tutorials and programming books. Beside this I also got a modest impression of the time and effort that goes into developing a real game. Even my simple game took much longer to develop and debug than I had originally planned. All the little things like finding sounds and making simple models took up more time than expected. No wonder big productions are almost always delayed a few months, sometimes even years.

Beside the points of frustration, when something just would not work, I have enjoyed the project and it raised the interest I had in programming with DirectX even more. If there would have been more time there are things that I would have added to the library, like more advanced shader effects (environment mapping, shadows) or a whole new Network component that covers DirectPlay. I would also like to go a level higher and develop a game architecture that enables the development of more complex games.

Appendix A: Contents of the CD-ROM

The CD-ROM contains all the files that were developed during this project. The file organization is as follows:

Demo Game
This directory contains two subdirectories:

1. *Project Files*: Contains all source code and resource files that were used to develop the demo game.
2. *Release Build*: Contains the executable and resources needed for the demo game to run.

Documentation
Contains this document.

Test Programs
This directory contains all test programs created during the project. All test programs come with complete source code of the project at the time the test program was created. The following test programs were created:

1. *Basic Window* Just creates an empty window with title bar
2. *Direct3D Basic* Shows a white D3D screen with a line, a square and a circle, that has a vertex added every second.
3. *Surface* Shows a window with a background image.
4. *Animation* Shows a very simple animation, consisting of two frames.
5. *Keyboard Input* Shows keyboard input, use the arrow keys to move the animation around. From now on all test programs can be exited by pressing Escape.
6. *Mouse Input* Same as the previous, only now the mouse controls the movement. Press a mouse button to rotate the animation.
7. *Joystick Input* Same as previous, only now a gamepad / joystick controls movement. Only works if a game device is connected.
8. *Sound* Repeatedly plays an explosion sound. Up and Down arrows adjust the frequency, + and – the volume and Left and Right arrows control pan (2d sound position). Audio CD playback is also possible.
9. *Vertex Buffer* Uses a hardware accelerated vertex buffer to display a colored triangle.
10. *Camera* Shows (very) basic camera control. Press Up or Down to zoom in or out on the triangle.
11. *Fonts* Shows the capabilities of the font class.
12. *World Transformations*
    a. World_test Shows a cube that can be rotated, translated and scaled.
    b. World_test2 Shows a rotating cube that is orbited by a smaller cube.

13. *Texture Mapping* Same as World_test2, only now with textured cubes. Modulation between cube and texture colors can be switched on and off by pressing 'M'.
14. *Lights & Materials*
    a. Light_test Same as the previous program, but with lighting. A point light is added to the scene.
    b. Spotlight_test Instead of a point light, a spotlight is used here. Also a specular component is added to the small cube.
15. *X Models* Loads and renders a tiger model in the DirectX .X format.
16. *Advanced Camera* Shows more complex functions of the camera class, by realizing First Person Shooter controls to navigate.
17. *Fog* Shows simple fog effect. Move away from the model to make it slowly disappear.
18. *Shader Basics*
    a. Shader_test1 A very simple shader that only shows ambient light.
    b. Shader_test2 Here diffuse light from a directional source is added.
    c. Shader_test3 This shader adds specular lighting to the model.
    d. Shader_test4 Finally a texture is added to the model.
19. *Point Light* Same as the final basic shader, but using a point light, that can be moved around.
20. *Particle System* This test shows the particle system class that uses point sprites to achieve various special effects.
21. *Viewports* Divides the screen into 4 separate render surfaces. Shows the same model from a different angle in each viewport.
22. *Collision detection* Shows a falling box, that stops falling when a collision with another box is detected.
23. *Mp3 & Midi* Plays a mp3 and a midi file.
24. *BSP Maps* Loads and renders a very simple BSP map (textured box).
25. *Device Loss* Shows how resources can get lost by pressing ALT-TAB and how to restore them.

Appendix B: Technical Information

Development Platform

This project was developed in the C++ programming language using Microsoft Visual Studio 6 as programming environment. The computer running it had the following properties:

Operating System: Microsoft Windows XP Professional
Processor: AMD Athlon XP 3000+
Internal Memory: 1024 MB RAM
Graphics Card: ATI RADEON 9800 Pro (fully DirectX 9 compatible)
Sound Card: nVidia nForce APU / SoundStorm

SDK versions

For this project the original DirectX 9 SDK has been used. In the mean time several updates have been released, but the main reason not to use a more recent version is that the recent SDK versions do not support Visual Studio 6 anymore.

Visual Studio 6 Settings

To successfully compile and link the files created during this project, Visual Studio should be configured correctly. Under **Tools → Options → Directories** the *Include* and *Lib* directories of the SDK must be added and moved to the top of the list. Under **Project → Settings → Link** the following libraries must be present:

- dsound.lib (needed for Sound Component)
- strmiids.lib (needed for Sound Component)
- winmm.lib (needed for Sound Component)
- dinput8.lib (needed for Input Component)
- dxguid.lib (needed for Input Component)
- d3d9.lib (needed for Graphics Component)
- d3d9x.lib (needed for Graphics Component)

Appendix C: Resources

This section provides an overview of the resources used for this project. A distinction is made between books, websites and software. The number before the resource is used throughout the document as a reference.

Books

[1] Wolfgang F. Engel, *Beginning Direct3D Game Programming*, Premier Press, Game Development Series, 2003.
[2] Bruno Miguel Teixeira de Sousa, *Game Programming: All in One,* Premier Press, Game Development Series, 2002.
[3] Kevin Hawkins and Dave Astle, *OpenGL Game Programming*, Premier Press, Game Development Series, 2001.
[4] Leen Ammeraal, *Basiscursus C++*, Academic Service, 1999.
[5] Brian Kernighan and Dennis M. Ritchie, *C Handboek*, Prentice Hall, 1990.

Websites

[6] http://msdn.microsoft.com/directx (DirectX SDK documentation)
[7] http://www.andypike.com/tutorials/directx8/
[8] http://www.ultimategameprogramming.com
[9] http://www.codesampler.com/dx9src.htm
[10] http://www.toymaker.info/Games/html/collisions.html

Software

[11] Microsoft Visual C++ 6.0
[12] Microsoft DirectX 9 Software Development Kit
[13] 3D Studio Max 7
[14] Macromedia Fireworks

Appendix D: Library Documentation

This appendix contains a total overview of all the classes developed for the library. For each class a short description is given and the basic usage of the class is explained. Also a complete list of all functions is provided, with information about the parameters, return value and a short description. The order in which the classes are presented is the same as in chapter 3.

CTimer

Class Description: This class provides timer functionality to support an accurate timing mechanism that is needed by games. This class depends on the availability of a hardware timer. If none is present, no fallback mechanism is provided.

Basic Usage: Call Update() every frame. With GetDelta() the time between two calls to Update() can be retrieved.

void Update (void)	
Parameters	- -
Return value	- -
Description	This function queries the hardware timer and calculates the difference with the previous value. The result can be requested with GetDelta(). It also updates the current time structure. This function should be called prior to all other timing functions.

real32 GetDelta (void)	
Parameters	-
Return value	The amount of seconds passed between the last two calls to Update()
Description	Returns the time that has passed between the last calls to Update()

u_int32 GetSeconds (void)	
Parameters	-
Return value	Seconds of current system time recorded by Update()
Description	Returns the seconds part of the system time

u_int32 GetMinutes (void)	
Parameters	-
Return value	Minutes of current system time recorded by Update()
Description	Returns the minutes part of the system time

u_int32 GetHours (void)	
Parameters	-
Return value	Hours of current system time recorded by Update()
Description	Returns the hours part of the system time

GetDay / GetMonth / GetYear	
Parameters	-
Return value	System date recorded by Update()
Description	Returns the 3 parts of the system date

CWindow

Class Description: This class provides a basic reusable window shell, that can be used as the starting point for any Windows application.

Basic Usage: To use the CWindow class, a custom class derived from it must implement the Frame() function. Next the Create() function must be called, using the appropriate parameters and finally the Run() function must be called to start the window. Every frame, the Frame() function will be executed then, so here all the application processing and rendering must be done.

rr32 Create (HINSTANCE hInstance, LPSTR szTitle, int iWidth, int iHeight, u_int32 iStyle)		
Parameters	hInstance	Instance that was passed to WinMain()
	szTitle	String containing the window title
	iWidth	Initial width of the window
	iHeight	Initial height of the window
	iStyle	Window style (see Win32 API)
Return value	noError	Window successfully created
	errorRegisterClass	The class could not be registered
Description	This function creates a new window based on the passed parameters. Only the first two parameters are obligatory, the size and style are optional. If they are not present, default values will be used.	

void Run (void)	
Parameters	-
Return value	-
Description	This function enters the Windows real-time message loop. It checks for messages and processes them. If there are no messages the Frame() function is called. The function stops when a WM_QUIT message is received (user has closed window).

virtual bool32 MessageHandler (UINT iMessage, WPARAM wParam, LPARAM lParam)		
Parameters	iMessage	Message type
	wParam	Data specific to message
	lParam	Data specific to message
Return value	b_true	Message handled
	b_false	Message not handled
Description	This default message handler only checks for WM_CLOSE messages, in which case the application will exit.	

virtual bool32 Frame (void)		
Parameters	-	
Return value	b_true	Frame processed, continue.
	b_false	Application done, quit.
Description	This virtual function needs to be implemented in a class deriving from CWindow. In this function all the processing of the application must be done. It is called every time there are no Windows messages to process.	

void SetPosition (int iX, int iY)		
Parameters	iX	Horizontal position of the window
	iY	Vertical position of the window
Return value	-	
Description	Displays the window's upper left corner at position (iX, iY) on the screen.	

void GetPosition (int iX, int iY)		
Parameters	pkPosition	Pointer to a POINT structure
Return value	-	
Description	After calling this function, the current position of the upper left corner of the window will be in the POINT structure.	

void SetSize (int iWidth, int iHeight)		
Parameters	iWidth	Width of the window
	iHeight	Height of the window
Return value	-	
Description	Scales the window to the supplied values.	

void GetSize (POINT * pkSize)		
Parameters	pkSize	Pointer to a POINT structure
Return value	-	
Description	After calling this function, the current size of the window will be in the POINT structure.	

HWND GetWindowHandle (void)	
Parameters	-
Return value	Handle to the window
Description	Used to obtain a handle to the window object, which is necessary for a range of Windows related operations.

void Show (int iShow)		
Parameters	iShow	Show state (e.g.: SW_SHOW, SW_HIDE, etc.)
Return value	-	
Description	This function changes the show state of the window. See Windows API documentation for details.	

CScreen

Class Description: This class initializes Direct3D, sets up the default device (graphics card) and provides some basic drawing primitives.

Basic Usage: Call Init() to initialize Direct3D, then call SetMode() to set the display mode. If using 3D, call SetPerspective() to define the clipping region. To render, first call Clear() to clear the frame buffer, then call StartFrame(), then call your rendering functions and finally call EndFrame().

err32 Init (HWND hWindow)		
Parameters	hWindow	Handle to parent window
Return value	noError	Initialization succeeded
	error…	See CError.h for possible error values
Description	This function initializes Direct3D.	

err32 SetMode (u_int32 iFullScreen, u_int16 iWidth, u_int16 iHeight, u_int16 iDepth, bool bHardware)		
Parameters	iFullScreen	0 = windowed, 1 = full screen
	iWidth	Screen width in pixels
	iHeight	Screen height in pixels
	iDepth	Screen color depth
	bHardware	Hardware acceleration on / off
Return value	noError	Mode correctly set
	error…	See CError.h for possible error values
Description	This function configures the screen used, some settings may only be available in full screen mode.	

void SetPerspective (real32 fieldOfView, real32 nearPlane, real32 farPlane)		
Parameters	fieldOfView	Field of view factor (0 - PI)
	nearPlane	Near clipping plane
	farPlane	Far clipping plane
Return value	-	-
Description	This function creates a viewing frustum, determining what part of the world is visible on the screen..	

D3DXMATRIXA16* CScreen::GetProjectionMatrix (void)		
Parameters	-	-
Return value	Pointer to projection matrix	-
Description	This function returns a pointer to the current projection matrix.	

err32 Clear (u_int8 iRed, u_int8 iGreen, u_int8 iBlue, u_int8 iAlpha)		
Parameters	iRed	Red clear color component
	iGreen	Green clear color component
	iBlue	Blue clear color component
	iAlpha	Alpha channel of the clear color
Return value	noError	Screen cleared to specified color
	error…	See CError.h for possible error values
Description	Clears the screen to the specified color.	

err32 StartFrame (void) / err32 EndFrame (void)		
Parameters	-	-
Return value	noError	Everything OK
	error…	See CError.h for possible error values
Description	Prepare / End one frame for the screen, put all rendering code between these calls (not when using viewports).	

err32 CheckDeviceState (void)		
Parameters	-	-
Return value	noError	Device operational
	error…	See CError.h for possible error values
Description	Check for lost device, if so it must be reset.	

err32 Reset (void)		
Parameters	-	-
Return value	noError	Device reset
	error…	See CError.h for possible error values
Description	Resets device to recover from lost state.	

err32 DrawLine (real32 fX1, real32 fY1, real32 fX2, real32 fY2, u_int8 iRed, u_int8 iGreen, u_int8 iBlue, u_int8 iAlpha)		
Parameters	fX1, fY1	Start position of the line
	fX2, fY2	End position of the line
	iRed, iGreen, …	Color components of the line
Return value	noError	Line drawn
	error…	See CError.h for possible error values
Description	Draws a colored line from specified start to end point.	

err32 DrawRectangle (real32 fX1, real32 fY1, real32 fX2, real32 fY2, u_int8 iRed, u_int8 iGreen, u_int8 iBlue, u_int8 iAlpha)		
Parameters	fX1, fY1	Upper left of the rectangle
	fX2, fY2	Lower right of the rectangle
	iRed, iGreen, …	Color components of the rectangle
Return value	noError	Rectangle drawn
	error…	See CError.h for possible error values
Description	Draws a colored rectangle using specified corners.	

err32 DrawCircle (real32 fCenterX, real32 fCenterY, real32 iRadius, u_int8 iRed, u_int8 iGreen, u_int8 iBlue, u_int8 iAlpha, u_int32 iVertices)		
Parameters	fCenterX/Y	Coordinates of center of the circle
	iRadius	Radius of the circle
	iRed, iGreen, …	Color components of circle
	iVertices	Number of vertices used for circle
Return value	noError	Circle drawn
	error…	See CError.h for possible error values
Description	Draws a colored circle, based on its parameters. The more vertices, the smoother the displayed circle will be.	

bool32 IsModeSupported (u_int16 iWidth, u_int16 iHeight, u_int16 iDepth)		
Parameters	iWidth	Width of queried mode in pixels
	iHeight	Height of queried mode in pixels
	iDepth	Color depth of queried mode
Return value	b_false	Mode is not supported
	b_true	Mode is supported
Description	Check if a certain screen mode is supported by the system.	

bool32 CheckMode (u_int16 iWidth, u_int16 iHeight, D3DFORMAT kFormat)		
Parameters	iWidth	Width of queried mode in pixels
	iHeight	Height of queried mode in pixels
	kFormat	Format of queried mode
Return value	b_false	Mode is not supported
	b_true	Mode is supported
Description	Check if a certain screen mode is supported by the system. View DirectX documentation for possible formats.	

void ShowCursor (u_int32 iShowCursor)		
Parameters	iShowCursor	0 = hide cursor, 1 = show cursor
Return value	-	-
Description	Show or hide the cursor.	

void SetDefaultStates (void)		
Parameters	-	-
Return value	-	-
Description	Set default values for rendering / texture states	

err32 GetCapabilities (D3DCAPS9* pCaps)		
Parameters	pCaps	Pointer to D3DCAPS9 structure
Return value	Error value	See CError.h
Description	Retrieves the capabilities of the current device.	

LPDIRECT3DDEVICE9 GetDevice (void)		
Parameters	-	-
Return value	Pointer to the Direct3D device	
Description	Function to retrieve access to the Direct3D device (representation of the graphics card).	

u_int32 GetFPS (void)		
Parameters	-	-
Return value	Number of frames per second	
Description	Retrieve the current FPS of the application	

u_int32 GetFormat (void)		
Parameters	-	-
Return value	Current format used	
Description	Retrieve the D3DFormat currently used	

u_int32 GetBitdepth (void)		
Parameters	-	-
Return value	Current bitdepth	
Description	Retrieve the bitdepth currently used	

CRGBAImage

Class Description: This class represents a raw image buffer. It is basically just a big multidimensional array containing color information.

Basic Usage: Mainly used by other classes of the library. To load an image use LoadFromBitmap() or LoadFromTarga(), depending on the image format.

err32 LoadFromBitmap (LPSTR lpszFilename)		
Parameters	lpszFilename	File name of the bitmap to load
Return value	Error code	See CError.h
Description	Load a bitmap from a .bmp file into an image buffer	

err32 LoadFromTarga (LPSTR lpszFilename)		
Parameters	lpszFilename	File name of the targa file to load
Return value	Error code	See CError.h
Description	Load a bitmap from a .tga file into an image buffer	

void SetColorKey (u_int8 iRed, u_int8 iGreen, u_int8 iBlue)		
Parameters	iRed	Red component of color key
	iGreen	Green component of color key
	iBlue	Blue component of color key
Return value	-	-
Description	Set the color components of the transparent color	

void SetWidth (u_int32 iWidth) / SetHeight (u_int32 iHeight)		
Parameters	iWidth / iHeight	Dimensions to be set
Return value	-	-
Description	Set the width / height of the raw image.	

void SetColor (u_int32 iX, u_int32 iY, u_int8 iRed, u_int8 iGreen, u_int8 iBlue, u_int8 iAlpha)		
Parameters	iX, iY	Position of target pixel
	iRed, iGreen, …	Color components
Return value	-	-
Description	Set the pixel at (iX, iY) to the specified color.	

void SetImageBuffer (u_int32 * pImage)		
Parameters	pImage	Pointer to an image buffer
Return value	-	-
Description	Set the image buffer of this object.	

u_int32 GetWidth / GetHeigth (void)		
Parameters	-	-
Return value	Width / height of the image buffer	-
Description	Get width / height of the raw image	

u_int32 GetColor (u_int32 iX, u_int32 iY)		
Parameters	iX, iY	Position in image buffer
Return value	Color at position (iX, iY)	-
Description	Get the color of the pixel (iX, iY) in the image buffer	

u_int32 * GetImageBuffer (void)		
Parameters	-	-
Return value	Pointer to the image buffer	-
Description	Get a pointer to the image buffer of this object.	

CSurface

Class Description: This class represents a static image surface that can be rendered to the screen. This is mainly useful for background images, because rotating, scaling and clipping are not supported.

Basic Usage: Load an image into a CRGBAImage object. Call Create() providing a pointer to the raw image object as a parameter. Call Render() to display it.

err32 Create (CRGBAImage * pkRawImage)		
Parameters	pkRawImage	Pointer to raw (source) image
Return value	Error code	See CError.h for error descriptions
Description	Create a new image surface, using a raw image object.	

err32 Update (void)		
Parameters	-	-
Return value	Error code	See CError.h for error descriptions
Description	Update the current surface (when raw image is changed).	

err32 Render (POINT * pkDestRect, RECT * pkSourceRect = NULL)		
Parameters	pkDestRect	Target screen area to display surface (NULL means entire screen)
	pkSourceRect	Source area from raw image to be used (NULL means entire image)
Return value	Error code	See CError.h for error descriptions
Description	Display (a part of) the surface on (specified part of) the screen	

void SetRawImage (CRGBAImage * pkRawImage)		
Parameters	pkRawImage	Pointer to raw image to be used
Return value	-	-
Description	Set the raw image that needs to be used for this surface.	

CRGBAImage * GetRawImage (void)		
Parameters	-	-
Return value	Pointer to raw image that is being used	-
Description	Get a pointer to the current raw image used for this surface.	

CTexture

Class Description: This class represents a texture, another image representation that can be used more dynamically than surfaces.

Basic Usage: Creating a texture can be done with Create() using a raw image object or directly using CreateFromFile(). Call SetActiveTexture() to activate it: all objects rendered will be using the texture until another texture is activated.

err32 Create (CRGBAImage * pkRawImage)		
Parameters	PkRawImage	Pointer to raw (source) image
Return value	Error code	See CError.h for error descriptions
Description	Create a new texture, using a raw image object.	

err32 CreateFromFile (LPCSTR lpszFileName)		
Parameters	LpszFileName	Name of the texture file
Return value	Error code	See CError.h for error descriptions
Description	Create a new texture directly from a source file.	

err32 Update (void)		
Parameters	-	-
Return value	Error code	See CError.h for error descriptions
Description	Update the texture data.	

void SetRawImage (CRGBAImage * pkRawImage)		
Parameters	pkRawImage	Pointer to raw image to be used
Return value	-	-
Description	Set the raw image that needs to be used for this texture.	

CRGBAImage * GetRawImage (void)		
Parameters	-	-
Return value	Pointer to raw image that is being used	-
Description	Get a pointer to the current raw image used for this texture.	

void Invalidate (void)		
Parameters	-	-
Return value	-	-
Description	Releases all resources used that are related to the device. This function must be called before resetting the device.	

err32 Restore (void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	Recreates the texture. Only use if created with a raw image, otherwise just recreate the texture from file to restore.	

void SetActiveTexture (void)		
Parameters	-	-
Return value	-	-
Description	Set this texture to be the current active texture so it will be used when rendering.	

static void SetModulate (bool32 bModulate)		
Parameters	bModulate	b_true = color modulation on b_false = color modulation off
Return value	-	-
Description	Switch color modulation (combining of color values) on or off. If this is turned on, the texture colors will be combined with the original colors of the surface to which it is mapped.	

u_int32 GetID (void)		
Parameters	-	-
Return value	Texture ID of this texture	-
Description	Get the texture ID of this texture object.	

static u_int32 GetActiveTexture (void)		
Parameters	-	-
Return value	Texture ID of current active texture	-
Description	Get the texture ID of the current active texture object.	

CTemplateSet

Class Description: This class represents a template set: a texture divided into cells. It also provides access to the individual cells. With template sets a single texture file can contain multiple images, which can be used for animations.

Basic Usage: This class is not intended to be used manually, but acts as a supporting class for CAnimation.

void Create (CTexture * pkTexture, u_int32 iTextureWidth, u_int32 iTextureHeight, u_int32 iCellWidth, u_int32 iCellHeight)		
Parameters	PkTexture	Pointer to source texture.
	iTextureWidth / Height	Texture dimensions
	iCellWidth / Height	Cell dimensions
Return value	-	-
Description	Create a new template set using the specified texture.	

void GetUV (CCellID kPosition, CRectText * pkUVRect)		
Parameters	KPosition	Position of the source cell
	pkUVRect	Pointer to destination texture rectangle
Return value	-	-
Description	Get the texture rectangle of a specific cell.	

void SetActiveTexture (void)		
Parameters	-	-
Return value	-	-
Description	Set texture used by this template set as the active texture.	

u_int32 GetTextureWidth (void) / u_int32 GetTextureHeight (void)		
Parameters	-	-
Return value	Width / Height of the texture	-
Description	Get the dimensions of the source texture	

u_int32 GetCellWidth (void) / u_int32 GetCellHeight (void)		
Parameters	-	-
Return value	Width / Height of a single cell	-
Description	Get the dimensions of a single cell in the template set.	

CAnimation

Class Description: This class represents a single two-dimensional animation consisting of multiple frames.

Basic Usage: An animation can be created from a template set using Create() and rendered with the Render() function. Call Update() to advance the animation by 1 frame. It is easier to use the CABO class, which is able to load multiple animations from file.

Void Create (CTemplateSet * pkTemplateSet, u_int32 iFrames, CCellID * pkPosition)		
Parameters	pkTemplateSet	Pointer to a template set object
	iFrames	Number of frames of this animation
	pkPosition	Starting position of animation in grid
Return value	-	-
Description	This function creates a new animation object, consisting of a number of frames (cells in a template set).	

void Update (void)		
Parameters	-	-
Return value	-	-
Description	This function moves the animation one frame forward. If it is in the last frame, it will go back to the first one.	

err32 Render (RECT kDestRect, u_int32 iColor, real32 fAngle)		
Parameters	kDestRect	Position and size of the animation
	iColor	Color filter
	fAngle	Angle of rotation for the animation
Return value	-	-
Description	This function displays the current frame of the animation. It uses top and left from the destination rectangle as the position and bottom and right values as size parameters. The frame is rotated by the specified angle (in degrees).	

void SetCurrentFrame (u_int32 iFrame)		
Parameters	iFrame	Frame ID that needs to be displayed
Return value	-	-
Description	This function sets the specified frame number as the current frame.	

u_int32 GetCurrentFrame (void)		
Parameters	-	-
Return value	Current frame ID	-
Description	This function returns the current frame ID	

CABO

Class Description: This class represents an Animated Blittable Object (ABO), a graphical 2D object that supports multiple animations, standard transformations (scale, rotate and translate) and collision detection.

Basic Usage: Load a configuration file (format shown below) with LoadFromFile(). Make sure all colors are shown by using SetColor(255,255,255,255) (or other values to achieve color effects). To display it call the Render() function. For transformations the Rotate(), SetPosition() and SetSize() can be used.

ABO Format	**Example**
[Source texture] [Image Type (1=bitmap, 2=targa)]	Graphics/image.bmp 1
[Alpha red] [Alpha green] [Alpha blue]	255 128 255
[Cell width] [Cell height]	128 128
[Number of Animations]	2
[Number of frames 1st animation]	2
[Cell pos. first frame]	0 0
[Cell pos. second frame]	0 1
[Number of frames 2nd animation]	1
[Cell pos. first frame]	1 0

void Create (u_int32 iAnimations, CAnimation * pkAnimations)		
Parameters	iAnimations	Number of animations
	pkAnimations	Pointer to array of animations
Return value	-	-
Description	This function creates a new ABO with the specified animations.	

void Update (void)		
Parameters	-	-
Return value	-	-
Description	Updates the current animation.	

void SetAnimation (u_int32 iAnimation, CAnimation * pkAnimation)		
Parameters	iAnimation	Animation ID
	pkAnimation	New animation object
Return value	-	-
Description	Set the animation on position iAnimation in the array to the specified animation.	

err32 LoadFromFile (LPSTR lpszFilename)		
Parameters	lpszFilename	Name of ABO configuration file
Return value	See CError.h for possible error values	
Description	Creates a new ABO using configuration data from the file.	

err32 Render (void)		
Parameters	-	-
Return value	-	-
Description	Displays current animation frame of ABO at current position.	

void Rotate (real32 fAngle, u_int32 iAccumulate)		
Parameters	fAngle	Angle of rotation (in degrees)
	iAccumulate	0 = fAngle is absolute 1 = fAngle is relative
Return value	-	-
Description	This function rotates the ABO around its center.	

bool32 Collide (CABO & rkABO, u_int32 iUseSphere)		
Parameters	RkABO	Reference to other ABO
	iUseSphere	1 = Bounding Spere, 0 = Bounding rectangle
Return value	b_false	No collision
	b_true	Collision
Description	This function does collision detection between this ABO and another that is provided as parameter. It uses bounding sphere or rectangle methods, depending on second parameter.	

bool32 ContainsPoint (u_int32 iX, u_int32 iY)		
Parameters	iX, iY	Pixel coordinates on the screen
Return value	b_false	Point is not inside bounding rectangle
	b_true	Point is contained in bounding rectangle
Description	This function checks wheter a given point is contained in the ABO's bounding rectangle.	

void Invalidate (void)		
Parameters	-	-
Return value	-	-
Description	Releases all resources used that are related to the device. This function must be called before resetting the device.	

void Restore (void)		
Parameters	-	-
Return value	-	-
Description	Restores the object, must be called after resetting the device.	

void SetCurrentAnimation (u_int32 iAnimation)		
Parameters	iAnimation	The animation ID that needs to be set
Return value	-	-
Description	This function sets the specified animation to be the current.	

u_int32 GetCurrentAnimation (void)		
Parameters	-	-
Return value	Current animation ID	-
Description	This function returns the current animation ID	

void SetPosition (u_int32 iX, u_int32 iY)		
Parameters	iX, iY	Coordinates of new position
Return value	-	-
Description	This function displays the ABO at the specified position.	

void SetSize (u_int32 iWidth, u_int32 iHeight)		
Parameters	iWidth	Width of the ABO
	iHeight	Height of the ABO
Return value	-	-
Description	This function can be used to change the size of the ABO.	

void SetColor (u_int8 iAlpha, u_int8 iRed, u_int8 iGreen, u_int8 iBlue)		
Parameters	iAlpha	Alpha component of color
	iRed, iGreen, iBlue	Color components
Return value	-	-
Description	This function can be used to change the ABO's color	

void SetRadius (real32 fRadius)		
Parameters	fRadius	Bounding sphere radius
Return value	-	-
Description	This function sets the radius of the bounding sphere that is used for collision detection.	

u_int32 GetXPosition (void) / u_int32 GetYPosition (void)		
Parameters	-	-
Return value	Current X / Y coordinate of ABO	-
Description	Get the current position of the ABO.	

u_int32 GetHeight (void) / u_int32 GetWidth (void)		
Parameters	-	-
Return value	Current size of ABO	-
Description	Get the current size of the ABO.	

u_int32 GetColor (void)		
Parameters	-	-
Return value	Current color components of ABO	-
Description	Get the current color of the ABO.	

real32 GetDirection (void)		
Parameters	-	-
Return value	Current angle of the ABO	-
Description	Get the current angle of the ABO.	

real32 GetRadius (void)		
Parameters	-	-
Return value	Current collision radius of ABO	-
Description	Get the current radius of the ABO's bounding sphere.	

CInputManager

Description: This class initializes the DirectInput object, which must be done before the other classes of the Input component can be used.

Basic Usage: Just call the Init() function before initializing the other input classes.

err32 Init (HINSTANCE hInstance)		
Parameters	hInstance	Handle to window instance
Return value	Error value	See CError.h for error descriptions
Description	This function initializes the DirectInput object.	

LPDIRECTINPUT8 GetInput (void)		
Parameters	-	-
Return value	Pointer to DirectInput object	
Description	This function returns a pointer to the DirectInput object.	

static CInputManager * GetSingleton (void)		
Parameters	-	-
Return value	Access point to InputManager	
Description	This function returns a pointer to the InputManager object.	

CKeyboard

Description: This class represents the keyboard and handles all input from it.

Basic Usage: Call Init() to initialize the keyboard. Call Update() every frame to refresh the key states. Use IsButtonUp() and IsButtonDown() to check the status of a specific key, using its code. The key codes can be found in the DirectX documentation.

err32 Init (HWND hWindow)		
Parameters	hWindow	Handle to parent window
Return value	Error value	See CError.h for error descriptions
Description	This function initializes the keyboard device.	

err32 Update (void)		
Parameters	-	-
Return value	Error value	See CError.h for error descriptions
Description	This function updates the key buffer with the current state.	

bool32 IsButtonDown (u_int32 iButton)		
Parameters	iButton	DirectX key code (see DirectX documentation)
Return value	b_false	The button is not down
	b_true	The button is being pressed
Description	This function checks the status of a button on the keyboard.	

bool32 IsButtonUp (u_int32 iButton)		
Parameters	iButton	DirectX key code (see DirectX documentation)
Return value	b_false	The button is being pressed
	b_true	The button is not down
Description	This function checks the status of a button on the keyboard.	

CMouse

Description: This class represents the mouse and handles all input coming from it.

Basic Usage: Call Init() to initialize the mouse. Use Update() to refresh the mouse information every frame. With IsButtonDown() and IsButtonUp() the mouse button states can be evaluated. GetWheelMovement() can be used to check the wheel of the mouse. The movement of the mouse can be retrieved using GetXAxis() and GetYAxis().

err32 Init (HWND hWindow)		
Parameters	hWindow	Handle to parent window
Return value	Error value	See CError.h for error descriptions
Description	This function initializes the mouse device.	

err32 Update (void)		
Parameters	-	-
Return value	Error value	See CError.h for error descriptions
Description	This function updates the mouse buffer with the current state.	

bool32 IsButtonDown (u_int32 iButton)		
Parameters	iButton	Mouse button (0-3)
Return value	b_false	The button is not down
	b_true	The button is being pressed
Description	This function checks the status of a button on the mouse.	

bool32 IsButtonUp (u_int32 iButton)		
Parameters	iButton	Mouse button (0-3)
Return value	b_false	The button is being pressed
	b_true	The button is not down
Description	This function checks the status of a button on the mouse.	

int32 GetXAxis (void) / int32 GetYAxis (void)		
Parameters	-	-
Return value	Relative movement along X / Y axis	-
Description	This function returns the relative movement of the mouse.	

int32 GetWheelMovement (void)		
Parameters	-	-
Return value	Movement of mouse wheel	-
Description	This function returns the relative movement of the mouse wheel. A positive value indicates upward movement and a negative value indicates downward movement.	

err32 Clear (void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	This function removes all data from the mouse buffer.	

CJoystick

Description: This class represents a joystick / game device and handles all input coming from it.

Basic Usage: Call Init() to initialize the joystick, providing values for minimal and maximum value and a deadzone value (dependant on device). Use Update() to refresh the device information every frame. With IsButtonDown() and IsButtonUp() the joystick button states can be evaluated. The movement of the game device can be retrieved using GetXAxis() and GetYAxis().

err32 Init (HWND hWindow, int32 iMin, int32 iMax, int32 iDeadZone)		
Parameters	hWindow	Handle to parent window
	iMin	Minimum of joystick range
	iMax	Maximum of joystick range
	iDeadZone	Deadzone (percentage of range in which no movement is reported)
Return value	Error value	See CError.h for error descriptions
Description	This function initializes the joystick device.	

err32 Update (void)		
Parameters	-	-
Return value	Error value	See CError.h for error descriptions
Description	This function updates the device data with the current state.	

bool32 IsButtonDown (u_int32 iButton)		
Parameters	iButton	Joystick button
Return value	b_false	The button is not down
	b_true	The button is being pressed
Description	This function checks the status of a joystick button.	

bool32 IsButtonUp (u_int32 iButton)		
Parameters	iButton	Joystick button
Return value	b_false	The button is being pressed
	b_true	The button is not down
Description	This function checks the status of a joystick button.	

u_int32 GetXAxis (void) / u_int32 GetYAxis (void)		
Parameters	-	-
Return value	Relative movement along X / Y axis	-
Description	This function returns the relative movement of the joystick.	

CSoundPlayer

Description: This class initializes the DirectSound object.

Basic Usage: Just call Init() to initialize Direct Audio before using the other classes of the Sound component.

err32 Init (HINSTANCE hInstance)		
Parameters	hInstance	Handle to window instance
Return value	Error value	See CError.h for error descriptions
Description	This function initializes the DirectSound object.	

LPDIRECTSOUND8 GetSound (void)		
Parameters	-	-
Return value	Pointer to DirectSound object	
Description	This function returns a pointer to the DirectSound object.	

IDirectMusicPerformance8* GetPerformance (void)		
Parameters	-	-
Return value	Pointer to Performance object	
Description	This function returns a pointer to the DirectMusic performance object.	

IDirectMusicLoader8* GetLoader (void)		
Parameters	-	-
Return value	Pointer to Loader object	
Description	This function returns a pointer to the DirectMusic loader object.	

static CSoundPlayer * GetSingleton (void)		
Parameters	-	-
Return value	Access point to CSoundPlayer	
Description	This function returns a pointer to the CSoundPlayer object.	

CSound

Description: Represents a sound buffer that can be manipulated and used for playback.

Basic Usage: Load a sound file in the WAV format using the LoadFromFile() function. Call Play() to start the playback and Stop() to stop it.

Note: Not all WAV files are supported, always check the return value of LoadFromFile().

err32 LoadFromFile (LPSTR lpszFileName)		
Parameters	lpszFilename	Name of sound file (.wav)
Return value	Error value	See CError.h for error descriptions
Description	This function loads a sound from file into a buffer	

err32 SetVolume (u_int32 iVolume)		
Parameters	iVolume	Volume of sound (0-100)
Return value	Error value	See CError.h for error descriptions
Description	This sets the volume of the sound.	

err32 SetFrequency (u_int32 iFrequency)		
Parameters	iFrequency	Frequency of sound (5000-100000)
Return value	Error value	See CError.h for error descriptions
Description	This sets the frequency of the sound.	

err32 SetPan (int32 iPan)		
Parameters	iPan	Pan of sound: -100 (left speaker) – 100 (right speaker)
Return value	Error value	See CError.h for error descriptions
Description	This sets the pan of the sound.	

err32 Play (u_int32 iLoop)		
Parameters	iLoop	0 = Play only once 1 = Loop playback
Return value	Error value	See CError.h for error descriptions
Description	This function starts the playback of the sound buffer	

err32 Stop (void)		
Parameters	-	-
Return value	Error value	See CError.h for error descriptions
Description	This function stops the sound playback.	

CCDPlayer

Description: Provides functionality to play audio CD's.

Basic Usage: Call Update() to refresh CD information, check the status of the CD player with IsReady() and use Play() and Stop() to control playback.

void Eject (void)		
Parameters	-	-
Return value	-	-
Description	Ejects the CD.	

void Play (u_int32 iTrack)		
Parameters	iTrack	Track of audio CD
Return value	-	-
Description	Playback the specified audio track.	

void Stop (void)		
Parameters	-	-
Return value	-	-
Description	Stop CD playback.	

void Update (void)		
Parameters	-	-
Return value	-	-
Description	Updates the CD information (length + nr of tracks)	

u_int32 GetNumberOfTracks (void)		
Parameters	-	-
Return value	Number of tracks on CD	-
Description	Get the number of tracks that are on the CD.	

int8 * GetLength (void)		
Parameters	-	-
Return value	String containing length of current CD	-
Description	Get the total length of the current CD..	

u_int32 GetCurrentTrack (void)		
Parameters	-	-
Return value	Track number	-
Description	Get the number of the track that is currently being played.	

bool32 IsReady (void)		
Parameters	-	-
Return value	b_false	CD player not ready
	b_true	CD player ready to be used
Description	Returns the current status of the CD player.	

CCamera

Description: Represents the camera in a 3D scene.

Basic Usage: Position and point the camera in the preferred direction using SetEye(), SetLookAt() and SetUp(). Use the other functions to move the camera through the scene.

void SetCamera (void)		
Parameters	-	-
Return value	-	-
Description	Set camera in scene according to current values.	

void SetEye (real32 eyeX, real32 eyeY, real32 eyeZ)		
Parameters	eyeX, eyeY, eyeZ	Position of the camera
Return value	-	-
Description	Set the position of the camera in a 3D scene.	

void SetLookAt (real32 lookX, real32 lookY, real32 lookZ)		
Parameters	lookX, lookY, lookZ	Point the camera is focused on
Return value	-	-
Description	Set the point the camera in a 3D scene is looking at.	

void SetUp (real32 upX, real32 upY, real32 upZ)		
Parameters	upX, upY, upZ	Up vector of the camera
Return value	-	-
Description	Set the axis that is up relative to the camera in a 3D scene	

D3DXMATRIXA16* GetMatrix (void)		
Parameters	-	-
Return value	Pointer to the view matrix	-
Description	Get a pointer to the current view matrix	

void Zoom (real32 fFactor)		
Parameters	fFactor	Zoom factor (0.0-1.0: zoom out, > 1.0: zoom in)
Return value	-	-
Description	Effectively moves the camera to or from the focus point.	

void Move (real32 fSpeed)		
Parameters	fSpeed	Movement speed
Return value	-	-
Description	Moves the camera forward / backward in the x,z plane.	

void Strafe (real32 fSpeed)		
Parameters	fSpeed	Movement speed
Return value	-	-
Description	Moves the camera left / right in the x,z plane.	

void Rotate (real32 fAngle, real32 fX, real32 fY, real32 fZ)		
Parameters	fAngle	Angle of rotation
	fX, fY, fZ	Rotation axis
Return value	-	-
Description	Rotates the camera at its position around the specified axis.	

void Rotate (D3DVECTOR3 vPoint, real32 fAngle, real32 fX, real32 fY, real32 fZ)		
Parameters	vPoint	Center of rotation
	fAngle	Angle of rotation
	fX, fY, fZ	Rotation axis
Return value	-	-
Description	Rotates the camera around the specified point / axis combination.	

CFont

Description: Font object that can be used to display text in a specific font or color.

Basic Usage: Call Init() with the preferred font properties as parameters. Set the position with SetTextArea() or SetPosition(). Use DrawText() to display a string.

err32 Init (int32 iHeight, int32 iWidth, LPCTSTR lpszFace)		
Parameters	iHeight	Font size
	iWidth	Average font width (0 = automatic)
	lpszFace	Name of face (e.g.: "Arial")
Return value	Error value	See Cerror.h
Description	Initializes the font.	

void DrawText (LPCTSTR lpszText, DWORD dwFormat, u_int32 iColor)		
Parameters	lpszText	Text string
	dwFormat	Format options (see DirectX documentation)
	iColor	Color of the text
Return value	-	-
Description	Draws a string to the screen in the specified color. Format options like DT_CENTER, DT_RIGHT etc. can be used to align the text. See for more information DirectX documentation about ID3DFont::DrawText().	

void SetTextArea (u_int32 iMinX, u_int32 iMinY, u_int32 iMaxX, u_int32 iMaxY)		
Parameters	iMinX, iMinY	Top left position of text area
	iMaxX, iMaxY	Bottom right position of text area
Return value	-	-
Description	Set up an area in which the text is to be displayed (text outside area won't be visible.	

void SetPosition (u_int32 iX, u_int32 iY)		
Parameters	iX, iY	Position to start text (use DT_LEFT as format parameter to make sure text starts at this location)
Return value	-	-
Description	Set the position where the text should start.	

void Invalidate (void) / void Restore (void)		
Parameters	-	-
Return value	-	-
Description	Invalidate() releases all resources used that are related to the device. This function must be called before resetting the device. Restore() restores it. Must be called after resetting the device.	

CWorld

Description: Represents the world matrix and allows transformations it to rotate, scale and translate objects in 3D space.

Basic Usage: Call Reset() to move back to the center of coordinate space (0,0,0). Make calls to Rotate(), Translate() to determine position and orientation. Call Scale() to adjust the size. Now the Render() function of the object that needs to be drawn can be called.

void SetWorld (void)		
Parameters	-	-
Return value	-	-
Description	Set world matrix to current matrix.	

void Reset (void)		
Parameters	-	-
Return value	-	-
Description	Reset the world matrix by loading the identity matrix.	

D3DXMATRIXA16* GetMatrix (void)		
Parameters	-	-
Return value	Pointer to world matrix	-
Description	Returns a pointer to the current world matrix	

void LoadMatrix (D3DXMATRIX* matWorld)		
Parameters	matWorld	Pointer to a transformation matrix
Return value	-	-
Description	Loads the matrix pointed to by matWorld and sets it as the current world matrix.	

void RotateX (real32 fAngle)		
Parameters	fAngle	Angle of rotation (in degrees)
Return value	-	-
Description	Applies a rotation around the x-axis.	

void RotateY (real32 fAngle)		
Parameters	fAngle	Angle of rotation (in degrees)
Return value	-	-
Description	Applies a rotation around the y-axis.	

void RotateZ (real32 fAngle)		
Parameters	fAngle	Angle of rotation (in degrees)
Return value	-	-
Description	Applies a rotation around the z-axis.	

void Rotate (real32 fAngle, real32 fX, real32 fY, real32 fZ)		
Parameters	fAngle	Angle of rotation (in degrees)
	fX	X component of arbitrary axis
	fY	Y component of arbitrary axis
	fZ	Z component of arbitrary axis
Return value	-	-
Description	Applies a rotation around an arbitrary axis.	

void Translate (real32 fX, real32 fY, real32 fZ)		
Parameters	fX	Movement along x-axis
	fY	Movement along y-axis
	fZ	Movement along z-axis
Return value	-	-
Description	Applies a translation in the specified direction.	

void Scale (real32 fX, real32 fY, real32 fZ)		
Parameters	fX	Scale factor along x-axis
	fY	Scale factor along y-axis
	fZ	Scale factor along z-axis
Return value	-	-
Description	Applies a scaling using the specified factors.	

void Push (void)		
Parameters	-	-
Return value	-	-
Description	Pushes the current world matrix onto the matrix stack.	

void Pop (void)		
Parameters	-	-
Return value	-	-
Description	Pops the top matrix from the matrix stack and uses it as the current world matrix.	

void MultLocal (D3DXMATRIX* matLocal)		
Parameters	matLocal	Pointer to a transformation matrix
Return value	-	-
Description	Applies a local matrix multiplication with the top element of the matrix attack.	

CMaterial

Description: Represents a material that describes the properties of a specific object. Used for lighting calculations.

Basic Usage: Use Create() to quickly create a material of the specified color. Use SetSpecular() and SetEmission() to set additional properties. Call SetActive() to activate this material.

void Create (u_int32 iRed, u_int32 iGreen, u_int32 iBlue)		
Parameters	iRed, iGreen, iBlue	Color of the material
Return value	-	-
Description	Creates a new material, using the specified parameters for the diffuse and ambient components of the material.	

void SetActive (void)		
Parameters	-	-
Return value	-	-
Description	This material will be set as the active material, meaning that everything drawn from this call until another material is set active will be using this material's properties.	

void SetDiffuse (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
Return value	-	-
Description	The way diffuse lighting is affecting this material is changed. The color components indicate the amount of light of each component is reflected.	

void SetAmbient (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
Return value	-	-
Description	The way ambient lighting is affecting this material is changed.	

void SetSpecular (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha, real32 fPower)		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
	fPower	Sharpness of highlight
Return value	-	-
Description	The way specular lighting is affecting this material is changed. Specular light will cause a highlight on the object the sharpness of this highlight is indicated by fPower.	

void SetEmission (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha);		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
Return value	-	-
Description	This will cause the material to appear to emit light itself.	

CLight

Description: Light object that is used to control the lighting in a scene. It represents a single light source.

Basic Usage: Enable lighting by calling the static function EnableLighting(b_true). Add ambient light to a scene using the static function SetAmbientLight(). To create a light source use Create(), call the appropriate functions to set its parameters and activate it using Switch().

Note: This class assumes the support for a maximum of 8 lights, which can be a problem. Change the value of MAX_LIGHTS and recompile if needed.

void Create (D3DLIGHTTYPE iType)		
Parameters	iType	Type of light to be created. Use L_POINT, L_DIRECTIONAL or L_SPOT
Return value	-	-
Description	Creates a new light using standard properties (bright white)	

err32 Switch (void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	Light switch, if light is on it will be turned off, otherwise it will be turned on, unless the maximum active lights is reached.	

void SetPosition (real32 fX, real32 fY, real32 fZ)		
Parameters	fX, fY, fZ	Position of the light
Return value	-	-
Description	Sets the position of this light. Only useful for point / spotlights.	

void SetDirection (real32 fX, real32 fY, real32 fZ)		
Parameters	fX, fY, fZ	Direction of the light
Return value	-	-
Description	Sets the direction of this light. Only useful for directional / spotlights.	

void SetDiffuse (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
Return value	-	-
Description	Sets the diffuse component of this light.	

void SetAmbient (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
Return value	-	-
Description	Sets the ambient component of this light.	

void SetSpecular (u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	iRed, iGreen, iBlue, iAlpha	Color components
Return value	-	-
Description	Sets the specular component of this light.	

void SetRange (real32 fRange)		
Parameters	fRange	Range of the light
Return value	-	-
Description	Sets the range of the light (distance traveled before it is faded away). Not for directional lights.	

void SetAttenuation (real32 fA0, real32 fA1, real32 fA2)		
Parameters	fA0, fA1, fA2	Attenuation constants
Return value	-	-
Description	Sets the way how light fades over distance. See DirectX documentation for details.	

void SetTheta (u_int32 iTheta)		
Parameters	iTheta	Angle of inner cone (in degrees)
Return value	-	-
Description	Sets the angle of the inner cone of the spotlight. This cone will be fully lighted. See also DirectX documentation.	

void SetPhi (u_int32 iPhi)		
Parameters	iPhi	Angle of outer cone (in degrees)
Return value	-	-
Description	Sets the angle of the outer cone of the spotlight. The light intensity decreases between the outside of the inner cone and the outside of the outer cone. See also DirectX documentation.	

void SetFalloff (real32 fFalloff)		
Parameters	fFalloff	Falloff constant
Return value	-	-
Description	Determines the way light fades between inner and outer cone. Default is 1.0f. See also DirectX documentation.	

static void SetAmbientLight (u_int32 iRed, u_int32 iGreen, u_int32 iBlue)		
Parameters	iRed, iGreen, iBlue	Color components
Return value	-	-
Description	Sets the amount of ambient light of the scene.	

static void EnableLighting (bool32 bEnable)		
Parameters	bEnable	b_true = enable lighting b_false = disable lighting
Return value	-	-
Description	Switches the use of lighting in a scene on or off.	

CMesh

Description: This class holds a 3D model that can be loaded from a .x file and rendered to the screen. It also provides functions to create basic shapes and supports collision detection between meshes.

Basic Usage: Either use LoadFromX() or Create…() to load / create a model. Use Render() to display it.
For collision detection call InitBoundingBox() after creation and call UpdateBoundingBox() with the transformation matrix used to position the model (the world matrix that is valid just before Render() is called). Finally use CheckCollisionWith() to do the collision detection.

err32 LoadFromX (LPSTR lpszFileName)		
Parameters	lpszFileName	Filename of a .x model
Return value	Error value	See CError.h
Description	Loads a 3D model from a .x file	

err32 CreateBox (real32 fWidth, real32 fHeight, real32 fDepth, u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	fWidth, fHeight, fDepth	Dimensions of the box
	iRed…iAlpha	Components of box color
Return value	Error value	See CError.h
Description	Creates a box shape with the specified dimensions and color.	

err32 CreateSphere (real32 fRadius, u_int32 iSlices, u_int32 iStacks, u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	fRadius	Radius of the sphere
	iSlices, iStacks	Level of detail
	iRed…iAlpha	Components of sphere color
Return value	Error value	See CError.h
Description	Creates a sphere shape with the specified dimensions and color. Increase number of slices and stacks to increase the number of polygons used to build the sphere.	

err32 CreateCylinder (real32 fRadiusTop, real32 fRadiusBottom, real32 fLength, u_int32 iSlices, u_int32 iStacks, u_int32 iRed, u_int32 iGreen, u_int32 iBlue, u_int32 iAlpha)		
Parameters	fRadiusTop	Radius of top (0 to create cone)
	fRadiusBottom	Radius of bottom
	fLength	Length of cylinder
	iSlices, iStacks	Level of detail
	iRed…iAlpha	Components of cylinder color
Return value	Error value	See CError.h
Description	Creates a cylinder shape with the specified dimensions and color. Increase number of slices and stacks to increase the number of polygons used to build the shape.	

void Render (void)		
Parameters	-	-
Return value	-	-
Description	Renders a 3D model to the screen.	

void Render (bool32 bMaterial, bool32 bTextures)		
Parameters	bMaterial	Use materials of mesh
	bTextures	Use textures of mesh
Return value	-	-
Description	Renders a 3D model to the screen, specifies wheter the current active material/texture should be used or the ones provided by the mesh.	

void Invalidate (void)		
Parameters	-	-
Return value	-	-
Description	Releases all resources used that are related to the device. This function must be called before resetting the device.	

err32 InitBoundingBox (void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	Calculates the bounding box for the mesh in model space.	

void UpdateBoundingBox (D3DXMATRIX* matWorld)		
Parameters	matWorld	Pointer to the current world matrix
Return value	-	-
Description	Calculates the current bounding box of the mesh in world space.	

bool32 CheckCollisionWith (CMesh* pMesh)		
Parameters	pMesh	Pointer to mesh that needs to be checked against this mesh for collisions.
Return value	b_false	No collision between meshes
	b_true	Collision occurred between meshes
Description	Checks whether a collision has occurred between this mesh and the one passed as parameter to the function.	

D3DXVECTOR3 GetMinBounds (void)	
Parameters	-
Return value	Vector representing the minimum bounds of the bounding box
Description	Returns the lower / left / front corner of the current bounding box.

D3DXVECTOR3 GetMaxBounds (void)	
Parameters	-
Return value	Vector representing the maximum bounds of the bounding box
Description	Returns the upper / right / back corner of the current bounding box.

LPD3DXMESH GetMesh (void)		
Parameters	-	-
Return value	Pointer to the mesh object.	-
Description	Gives a pointer to the actual ID3DMESH object.	

CFog

Description: This class can be used to add fog to a 3D scene (objects that are further away are blended with a specific color).

Basic Usage: Call Create() to initialize the CFog object. Call SetVisible() to show or hide the fog.

err32 Create (u_int32 iColor, DWORD dwMode)		
Parameters	iColor	Fog color
	dwMode	Fog mode (D3DFOG_LINEAR, D3DFOG_EXP, D3DFOG_EXP2)
Return value	Error value	See CError.h
Description	Creates a new fog object using default parameters and the specified mode and color. See DirectX documentation for detailed information about the different modes.	

err32 SetVisible (bool32 bVisible)		
Parameters	bVisible	b_true = Enable fog b_false = Disable fog
Return value	Error value	See CError.h
Description	Enables or disables the fog.	

err32 SetLinearFog (real32 fStart, real32 fEnd)		
Parameters	fStart	Starting position of fog
	fEnd	End position of fog (fully fogged)
Return value	Error value	See CError.h
Description	Sets the parameters for linear fog. The amount of fog is linearly distributed between the start and end position.	

err32 SetExponentialFog (real32 fDensity)		
Parameters	fDensity	Fog density
Return value	Error value	See CError.h
Description	Sets the density factor for exponential fog. See DirectX documentation for details about this factor.	

CEffect

Description: This class is used to load and use shader effects, that are written in the High Level Shader Language (HLSL).

Basic Usage: Call Create() to load an effect from a .fx file. To enable a technique defined in the shader file, use SetTechnique(). To draw a mesh using the enabled technique, call RenderMesh().

err32 Create (LPSTR lpszFileName)		
Parameters	lpszFileName	File name of effect to load
Return value	Error value	See CError.h
Description	Creates a new effect from a .fx source file	

err32 SetVector (D3DXHANDLE hParameter, CONST D3DXVECTOR4* pVector)		
Parameters	hParameter	Name of a vector variable
	pVector	Pointer to the vector to use
Return value	Error value	See CError.h
Description	Supplies a vector variable value to a shader effect	

err32 SetMatrix (D3DXHANDLE hParameter, CONST D3DXMATRIX* pMatrix)		
Parameters	hParameter	Name of a matrix variable
	pMatrix	Pointer to the matrix to use
Return value	Error value	See CError.h
Description	Supplies a matrix variable value to a shader effect	

err32 SetTechnique (LPCSTR pName)		
Parameters	pName	Name of the technique to use
Return value	Error value	See CError.h
Description	Enables a specific technique defined in the shader	

err32 RenderMesh (CMesh* pMesh)		
Parameters	pMesh	Pointer to a mesh
Return value	Error value	See CError.h
Description	Renders a mesh object using the technique currently set	

void Invalidate (void)		
Parameters	-	-
Return value	-	-
Description	Releases all resources used that are related to the device. This function must be called before resetting the device.	

LPD3DXEFFECT GetEffect (void)		
Parameters	-	-
Return value	Pointer to the effect object	-
Description	Provides access to the effect object for more advanced control.	

CParticleSystem

Description: This class is used to create and control a particle system that can be used to achieve a wide variety of special effects, such as explosions, fire, smoke, snow, sparks etc.

Basic usage: Call Create() to create a new particle system. Create a ParticleInfo structure, fill it with zeros and fill in the fields that are necessary for this particle system. Call Init() to initialize the particle system. Call Update() every frame and Render() to show the particle system.

Note: A drawback of this class is that it enables alpha blending and disables the z-buffer to achieve the effects and therefore the particles are not tested against the z-buffer. This means that the particles are visible through other objects, even when they are actually behind those objects. Also sizes of point sprites seem to show differently on different computers.

err32 Create (u_int32 iMaxParticles, D3DXVECTOR3 vOrigin, D3DXVECTOR3 vForce, LPCSTR lpszTextureName)		
Parameters	iMaxParticles	Max. amount of particles to use
	vOrigin	Point of emission
	vForce	External force (gravity, wind, etc.)
	lpszTextureName	Filename of texture to use
Return value	Error value	See CError.h
Description	Creates a new particle system that must be initialized first by calling Init(..).	

void Init (ParticleInfo * pParticleInfo)		
Parameters	pParticleInfo	Pointer to a ParticleInfo structure describing the properties of the particle system.
Return value	-	-
Description	Initializes the particle system. The ParticleInfo structure contains all needed info to set the properties of the particle system. It must be maintained by the application and stay available as long as the particle system is needed. The properties can be changed dynamically while the system is running.	

void Invalidate (void)		
Parameters	-	-
Return value	-	-
Description	Releases all resources used that are related to the device. This function must be called before resetting the device.	

void Update (real32 fTimePassed)		
Parameters	fTimePassed	Fraction of time that has passed since last call (in seconds)
Return value	-	-
Description	Updates calculation for all particles, creating and destroying them when needed.	

err32 Render (void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	Renders all particles of the system as point sprites (always facing towards viewer).	

int Emit (u_int32 iNrOfParticles)		
Parameters	iNrOfParticles	Number of particles to create
Return value	Number of particles that could not be created	
Description	Creates a number of new particles.	

void Reset (void)		
Parameters	-	-
Return value	-	-
Description	Sets the number of particles to 0, destroying all current particles. To prevent new particles from being created the iParticlesPerSecond field of the ParticleInfo structure should be set to zero.	

void Move (D3DXVECTOR3 vDirection)		
Parameters	vDirection	Vector describing the movement to make in x, y and z direction
Return value	-	-
Description	Moves the entire particle system (emission point) towards the specified direction.	

void SetOrigin (D3DXVECTOR3 vOrigin)		
Parameters	vOrigin	New origin position
Return value	-	-
Description	Sets a new origin (emission point) for the particle system.	

void SetForce (D3DXVECTOR3 vForce)		
Parameters	vForce	Vector describing the external force affecting the particles.
Return value	-	-
Description	Sets a new external force to influence the particles.	

D3DXVECTOR3 GetOrigin (void)		
Parameters	-	-
Return value	Current origin	-
Description	Returns the current origin (emission point) of the system.	

D3DXVECTOR3 GetForce (void)		
Parameters	-	-
Return value	Current force	-
Description	Returns the current external force affecting the system.	

u_int32 GetNrOfParticles (void)		
Parameters	-	-
Return value	Current number of particles	-
Description	Returns the amount of particles the system currently uses.	

CViewPort

Description: This class represents a viewport, a part of the screen that can be rendered to. By using multiple viewports, split-screen multiplayer games can be constructed.

Basic Usage: When using viewports, the StartFrame() and EndFrame() functions of CScreen should **not** be called. Instead call Begin() and End() for each viewport, but the last. Use Begin() and EndLastViewPort() for the last.

void Create (u_int32 iX, u_int32 iY, u_int32 iWidth, u_int32 iHeight)		
Parameters	iX, iY	Starting position of viewport
	iWidth, iHeight	Dimensions of the viewport
Return value	-	-
Description	Creates a new viewport starting at (iX,iY) with the specified width and height.	

err32 Begin (u_int8 iRed, u_int8 iGreen, u_int8 iBlue, u_int8 iAlpha)		
Parameters	iRed…iAlpha	Clear color components
Return value	Error value	See CError.h
Description	Activates this viewport and clears it with the specified color.	

void End (void)		
Parameters	-	-
Return value	-	-
Description	Must be called after rendering to this viewport, unless this is the last viewport that has been rendered to (frame ready).	

err32 EndLastViewPort (void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	Must be called after rendering to the last viewport in a frame.	

void SetViewArea (RECT kViewArea)		
Parameters	kViewArea	RECT describing the wanted viewport dimensions
Return value	-	-
Description	Sets the viewport area to start at the (top, left) values from the RECT and uses the (bottom, right) values as height and with.	

void SetDepth (real32 fMinZ, real32 fMaxZ)		
Parameters	fMinZ	Minimum depth
	fMaxZ	Maximum depth
Return value	-	-
Description	Sets the depth values for this viewport. Values must be between 0.0 and 1.0. See DirectX documentation for more information.	

void GetViewArea (RECT* kViewArea)		
Parameters	kViewArea	Pointer to a RECT structure.
Return value	-	-
Description	Fills the RECT with the current view area parameters.	

real32 GetMinZ (void)		
Parameters	-	-
Return value	Current minimum depth value	-
Description	Returns the current minimum depth value.	

real32 GetMaxZ (void)		
Parameters	-	-
Return value	Current maximum depth value	-
Description	Returns the current maximum depth value.	

CMusic

Description: This class represents a background music track, loaded from a mp3 or midi file.

Basic Usage: Use LoadMP3() or LoadMidi() to load the media. Use Play() and Stop() to control playback.

err32 LoadMP3 (LPCSTR lpszFileName)		
Parameters	lpszFileName	Name of mp3 file
Return value	Error value	See CError.h
Description	Loads the specified mp3 file.	

err32 LoadMidi (LPCSTR lpszFileName)		
Parameters	lpszFileName	Name of midi file
Return value	Error value	See CError.h
Description	Loads the specified midi file.	

err32 Play(u_int32 iNrOfRepeats)		
Parameters	iNrOfRepeats	Number of times to repeat the track
Return value	Error value	See CError.h
Description	Plays the loaded mp3/midi track, repeating it the specified amount of times (only for midi).	

err32 Stop(void)		
Parameters	-	-
Return value	Error value	See CError.h
Description	Stops the playback of the loaded mp3/midi track.	

bool32 IsPlaying(void)		
Parameters	-	-
Return value	b_false	Track is currently not playing
	b_true	Track is playing
Description	Returns the current playing status of the mp3/midi track.	

CBSPMap

Description: This class provides functionality for loading and rendering maps in the Quake 3 BSP format.

Basic Usage: Load the map with LoadFromFile() and display it using Render().

Note: This class only handles geometry, texture maps and light maps, it doesn't support collision detection, model loading and JPG textures. To be really useful in practice it should be extended to support these features.

.

err32 LoadFromFile (LPCSTR lpszFileName)		
Parameters	lpszFileName	Filename of .bsp map
Return value	Error value	See CError.h
Description	Loads a Quake 3 BSP map / level from file.	

void Render (void)		
Parameters	-	-
Return value	-	-
Description	Renders the map to the screen.	

void Invalidate (void)		
Parameters	-	-
Return value	-	-
Description	Releases all resources used that are related to the device. This function must be called before resetting the device.	