**İSTANBUL TEKNİK ÜNİVERSİTESİ**
**ELEKTRİK-ELEKTRONİK FAKÜLTESİ**

# 3D OYUN TASARIMI

**Bitirme Ödevi**

**Ergün Mıhçıoğlu**

**Bölüm : Bilgisayar Mühendisliği**
**Anabilim Dalı: Bilgisayar Bilimleri**

**Danışman : Yrd. Doç. Dr. Feza Buzluca**

**Mayıs 2004**

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

# 3D OYUN TASARIMI

**Bitirme Ödevi**

**Ergün Mıhçıoğlu**

**Bölüm : Bilgisayar Mühendisliği**
**Anabilim Dalı : Bilgisayar Bilimleri**

**Danışman : Yrd. Doç. Dr. Feza Buzluca**

**Mayıs 2004**

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICS-ELECTRONICS FACULTY

# 3D GAME DEVELOPMENT

**Graduation Project**

**Ergün Mıhçıoğlu**

**Programme : Bilgisayar Mühendisliği**
**Department : Bilgisayar Bilimleri**

**Supervisor : Assoc. Prof. Dr. Feza Buzluca**

**May 2004**

## TABLE OF CONTENTS

# SUMMARY

In this project, a game application is developed using C++ language and OpenGL and DirectX libraries. This document first gives a brief introduction of concepts used in game programming. OpenGL and DirectX is introduced and the purpose of using these libraries are stated. Other concepts, like modelling, performance issues and networking, are also described. Then implementations are specified. Object diagrams of application parts are given. The solutions to performance problems are also given. After that network messaging system is told. At the end, the encountered problems during development are listed, then suggestions to later developments are given.

# ÖZET

Bu projede, nesne tabanlı programlama ile OpenGL ve DirextX kütüphaneleri kullanılarak 3 boyutlu bir oyun geliştirildi. Oyun programlamanın getirdiği zorluklar anlaşılmaya çalışıldı, ve bu problemlerin bir kısmı çözüldü.

Proje tamamen nesne tabanlı programlama mantığına uygun olarak geliştirildi. Uygulama çeşitli bölümlere ayrıldı ve bu bölümler kendi içinde tasarlandı. Nesnelerin çoğu ve bunlar arasındaki ilişkiler önceden belirlendi ve kodlama aşamasında bunlar geliştirildi. Polimorfizm ve encapsulation yoğun bir şekilde kullanıldı. Tasarlanan uygulama bölümlerinin bazıları başka oyun uygulamalarında da kullanılabilecek şekilde oluşturuldu.

Geliştirme aşamasında, oyun uygulamalarında genel olarak kullanılan bazı kütüphaneler de kullanıldı. Günümüzde oyun uygulamaları çok kapsamlı bir hale geldiğinden, geliştirilen bir oyunun baştan sona geliştirilerek ortaya üst düzey bir uygulama çıkarılması imkansız hale gelmiştir. Bu yüzden bir çok özellik hazır kütüphanelerden faydalanılarak geliştiriliyor. Bunun gibi yeni oyun projelerinde aynı özellikleri tekrar gerçeklemektense, gerçeklenmiş özelliklerin üzerine değişik özellikler eklenerek daha üst düzey uygulamalar geliştirilebiliyor.

Bu projede en sık kullanılan kütüphaneler OpenGL (Açık Grafik Kütüphanesi) ve DirectX kütüphaneleridir. OpenGL tüm grafik işlemlerinde kullanıldı. OpenGL sayesinde tüm çizimler ekran kartlarının tüm özellikleri kullanılarak çok hızlı bir şekilde işlenebiliyor. Geliştirilen görsel kaynaklar, OpenGL'in uygun yöntemlerini kullanarak bilgisayar ekranına çizildi. Böylece çok etkili grafik efektleri çok kısa zamanda geliştirebildi.

DirextX kütüphanesinin ise, DirectInput, DirectAudio ve DirectPlay kısımları kullanıldı. DirectInput sayesinde kullanıcı klavye ve fare aktivitelerini hızlı bir şekilde uygulamaya alabildik. DirectAudio ile ses efektleri gerçeklendi. DirectPlay ise ağ ve çok oyunculu oyun kısımlarında kullanıldı.

Bu kütüphaneler programcıya birçok özelliği kolay bir şekilde uygulamaya ekleme imkanı veriyorlar. Örneğin, DirectAudio ile 3 boyutlu ses efektleri eklemek, DirectPlay ile lobi sistemi yapmak ve oyun sunucunuzun ağ üzerinde tanıtımını yapmak son derece kolay. Uygulama geliştiriciler bu gibi özellikleri tekrar tekrar geliştirmek zorunda değiller. Zaten bunlar tekrar tekrar geliştirilseydi, bu gün piyasadaki oyunlar şimdikinden çok daha kısıtlı olurdu.

Projede ayrıca temel oyun programlama sorunları çözüldü. Oyunun işleyişinde gereken algoritmalar gerçeklendi ve oyun kuralları ve akışının kontrolü için gereken program kontrolleri oluşturuldu.

Oyun 3 boyutlu olduğundan, ekrana çizimin yeteri kadar hızlı olabilmesi için gerekn optimizasyonlar yapıldı. 3 boyutlu modellerin bellekte kapladığı alan çok olduğundan, programın bellek yönetimi minimum bellek kullanacak şekilde ayarlanmaya çalışıldı.

# 1. INTRODUCTION

Game industry is a part of computer industry for many long years. At the early times, there was special devices for playing games. This games were very simple, but they were still entertaining people, and became successful. With increased technology of computers and electronics, games also became more complicated, too. The demand for games was increasing. When PCs were created, game machines and computers became one, and people started to buy PCs for their works and also for entertainment.

This last decade, PCs and games advanced a lot. Now there is lots of very fancy and complicated games. Game applications are being more intelligent, graphics are improving and being more realistic. And game industry is one of the biggest part of computer industry now. Every day new games are releasing and maybe more than one thousand game is being developed now.

However, game development is very hard. A big team is required for development of a game; programmers, artists, musicians, writers, and so on. Also development of a game takes long times. So a game application is a very valuable value.

In addition to these, game industry accelerates computer industry's development. Because game industry always uses latest computer technologies and always demands more, other industries tries to develop technologies faster. Especially graphics cards are fastly advaced because of games. Most of the other areas doesn't require faster computers that much. However, computers are advancing and they are developing better applications too.

Currently biggest countries of game industry is USA and Japan. But other countries, especially UK, France and Germany also taking parts at game industry. The number of games released from this countries is increasing. Unfortunately, the size of game industry in Turkey is nearly zero. There is hardly one or two companies, but no commercial game released yet. Of course reason of this is copywright protection laws. Nobody developing commercial games, because they know it'll be sold pirate.

There is a variety of concepts in game development. There is application development, painting, modelling, musics and writing. For computer engineers, application development is most important part. This project is chosen as game development because of this. Game development is a very good implementation of application development.

In this project we tried to develop a between small and middle size game application. The target of development is design and using latest technologies, not very good graphics. Application is developed using C++, and nearly all concepts of object-oriented programming are applied. Most of art resources are taken from other resources.

The latest technologies are used at development of this project. The latest version of OpenGL is used for graphical operations, DirectX 8.0 is use for inputs, sound and network. Some features of current graphics cards used to speed up application.

In this report we tried to explain all concepts used in this project, and tell the development process of project. In section 2, the general concepts of game developmen is outlined. In section 3, development process is explained and implementations are given. In section 4, recommendations to other developments and last words are given.

# 2. GENERAL CONCEPTS

## 2.1 SOFTWARE KITS

### 2.1.1 3D Graphic Renderers

Computer Graphics is a very big study area. In 1980s mathematicians were researching mathematical equations to display graphics in a computer screen. Many algorithms were found, but these were only to display some simple objects on screen. After years computer graphics evolved a lot, there is lots of theorems and equations, and very complex things can be display in a computer screen.

However, learning all of these theorems and drawing methods is impossible. So if a person wants to make a graphics application, he will seldomly use these methods, but will use libraries using these methods.

3D Renderers are these kinds of libraries. They use many different algorithms to display the desired object on screen, but user will not see which algorithms are used. Users will only use functions of these libraries. This way developing better graphics applications are much easier.

#### 2.1.1.1 OpenGL API

OpenGL is one of the most widely used graphics application programming interface (API). It's been introduced at 1992. Now it's the premier development environment for developing portable, interactive 2D and 3D graphics applications. Because it's portable, the same software quality and performance can be achieved at a wide variety of current computer platforms.

OpenGL is first developed at 1992 by an independent consortium, the OpenGL Architecture Review Board. This consortium is composed of many of the industries leading graphics vendors. The ARB defines conformance tests and approves new OpenGL features and extensions. As of October 2003, voting members of the ARB include 3Dlabs, Apple, ATI, Dell Computer, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Matrox, NVIDIA, SGI, Sun.

One of the biggest strengths of OpenGL is it's well structured design and logical commands. For most developers, the structure of Direct3D is very complicated and hard to learn, but OpenGL, on the other hand, is very understandable. Learning OpenGL and developing applications are really easy. OpenGL drivers encapsulate information about underlying hardware, freeing the application developer from having to design for specific hardware features.

Another strength of OpenGL is it's documentation. Now, Internet is full of OpenGL tutorials and forums. There is lots of resources for a starter. Numerous books have been published about OpenGL.

OpenGL is highly portable. It has language bindings for C, C++, Fortran, Python, Perl, Ada and Java. Supported platforms are all Unix workstations, all Windows 95/98/NT/2000/XP, MacOS, OPENStep and BeOS. It also works with every major windowing system, including Win32, MaxOS, Presentation Manager and X-Window System.

OpenGL API Hierarchy:

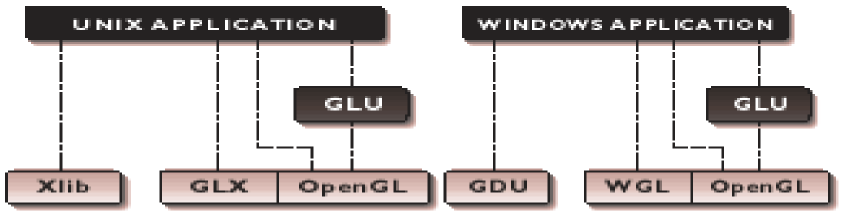


**Figure 2.1:** OpenGL API Hierarchy

- OpenGL applications use the window system's window, input, and event mechanism
- GLU supports quadrics, NURBS, complex polygons, matrix utilities, and more

The OpenGL Visualization Programming Pipeline:

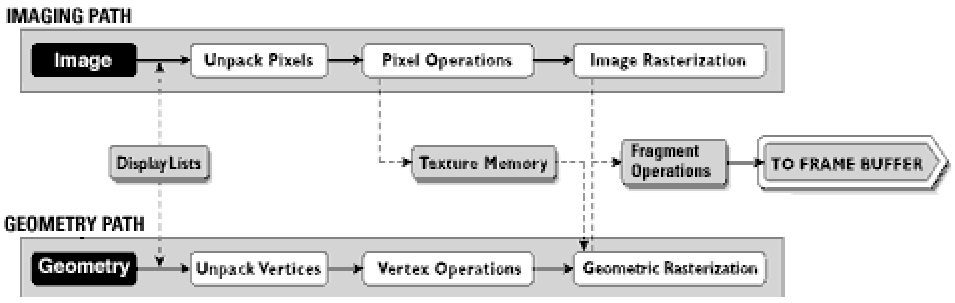


**Figure 2.2:** OpenGL Visualization Programming Pipeline

Note: Some of the informations in this part is taken from [1]

### 2.1.1.2 DirectX API

Microsoft DirectX is an advanced suite of multimedia APIs build into Microsoft Windows operating systems. DirectX provides a standard development platform for Windows-based PCs by enabling software developers to access specialized hardware features without having to write hardware-specific code. This technology is first introduced in 1995 and is a recognized standard for multimedia application development on Windows platform.

DirectX enables software and hardware talks to each other. It controls low-level functions and enables users to use a specific API for all hardware. However, DirectX API itself has a very steep learning curve. But after learning the structure of API, it's simple to develop applications using hardware like graphic cards, sound cards, keyboards, mice and joysticks.

Another problem of DirectX is its portability. Actually there is no portability of DirectX between different operating systems. There is no support to DirectX in Linux or MacOS.

Note: Some of the informations in this part is taken from [2]

## 2.1.2 User Events

### 2.1.2.1 Windows Events

In OSs like windows, that is OS with GUI and user interactive, it’s important for OS to distribute the events to applications. If application wants to respond to user events or other OS events, application must get this events from OS and do whatever it needs. Responses to events are responsibility of application, but they have to get events from OS first. So windows has a structure to distribute events to applications.

Every application running on Windows is basically a window. When an application is created it creates its own window structure, the structure that holds information about that application, and registers it to OS. Than OS gives application a handle to control its operations in system.

When registering its window structure, application puts a function pointer data to that structure. This function is responsible to get Windows messages. When a windows message occured, Windows sends this message to application through this function. Application catches messages it want to handle, and calls its own functions.

All keyboard and mouse events are also Windows messages. So if you want to handle keyboard events, you need to catch that message in your function then do your own things.

Not only keyboard and mouse events are messages. Activation, minimization, closing etc. of application is also messages that OS sends to application.

As we mentioned, to get Windows events you need to register a function. This function is WndProc function. WndProc function has a specified format, that is:

```
LRESULT CALLBACK WndProc( HWND hWnd,
                          UINT msg,
                          WPARAM wParam,
                          LPARAM lParam)
```

HWND is handle of application window. UINT is type of message. They are defined in *windows.h* file. And WPARAM and LPARAM are data of message. Like which key is pressed or position of mouse event.

With implementation of this function, you can get messages you want to get.

### 2.1.2.2 Direct Input

However, for some applications that needs faster access to user events, there is another way to get user events. Getting user events by Windows events are slow. Because messages are passing all levels of OS and then sent to application. An application may need a faster access, if user events are quickly happening. Games and simulations are this kind of applications. So there is another way to get events.

DirectInput is a response to this need. Application can get events from not OS but DirectX. DirectInput communicates directly with the hardware drivers rather than relying on Windows messages. So when your application creates a DirectInput event receiver, DirectInput instantly gets event from drivers when event occurred and sends it to application. So application gets event faster. Also, DirectInput enables an application to retrieve data from imput devices even when the application is in the background.

Note: Some of the informations in this part is taken from [2]

### 2.1.3 Sound Effects

#### 2.1.3.1 DirectX Audio

Game applications intensely use sound effects. In almost every user event, application gives a sound response to user to keep application alive. So application will need a high control over playing sound. Application may need to play sounds simultaneously, change the level of sound during runtime, etc. So applications need a library to achieve this goals. DirectX Audio come at this point.

DirectX Audio does much more than simply playing sounds. It provides a complete system for implementing a dynamic sountrack that takes advantages of hardware acceleration, Downloadable Sounds (DLS), DirectX Media Objects, and advanced 3-D positioning effects.

By using the DirectMusic and DirectSound imterfaces in your application, you can do the following:

- Load and play sounds from files or resources in MIDI, wave, or DirectMusic producer runtime format.
- Play from multiple sources simultaneously.
- Schedule the timing of musical events with high precision.
- Send tempo changes, patch changes, and other MIDI events programmatically.
- Use Downloadable Sounds. By using the DLS synthesizer, an application can be sure that message-based music sounds the same on all computers. An application can also play an unlimited variety of instruments and even produce unique sounds for individual notes and velocities.
- Locate sounds in 3-D environment.
- Easily apply pitch changes, reverb, and other effects to sounds.
- Use more than 16 MIDI channels. DirectX Audio breaks the 16-channel limitation and makes it possible for any number of voices to be played simultaneously, up to the limits of the synthesizer.
- Play segments on different audiopaths, so that effects or spatializtion can be applied individually to each sound.

- Capture MIDI data or stream ("thru") it from one port to another.
- Capture wave sounds from microphone or other input.

If you use source files from DirectMusic Producer or a similar application, you can do much more:

- Control many more aspects of playback at run time, for example by choosing a different set of musical variations or altering the chord progression.
- Play music that varies subtly each time it repeats.
- Play waves with variations.
- Map performance channels to audiopaths, so that different parts within the same segment can have different effects.
- Compose wholly new pieces of music at run time, not generated algorithmically but based on components supplied by a human composer.
- Dynamically compose transitions between existing pieces of music.
- Cue transitions, motifs, and sound effects to occur at specified rhythmic points in the performance.

These capabilities are the ones most often used by mainstream applications. DirectX Audio is designed to be used easily for the basic tasks, but it also allows low-level access to those who need it. It is also extensible. Specialized applications can implement new objects at virtually every stage on the audiopath, such as the following:

- Loaders to parse data in new or proprietary formats.
- Tracks containing any kind of sequenced data.
- Tools to process messages—for example, to intercept notes and apply transpositions, or to display lyrics embedded in a segment file.
- Custom sequencer.
- Custom synthesizer.
- Effects filters.

Also part of DirectX Audio is DirectMusic Producer, an application that enables composers to create DLS collections, chordmaps, styles, and segments—the pieces that let you take full advantage of the power of DirectMusic. DirectMusic Producer also makes it possible to create playable segments that contain multiple time-stamped waves. These waves can be in compressed or uncompressed format and can be either streamed at run time or wholly contained in memory.

As an application developer, you might never use DirectMusic Producer yourself, but it is a good idea to have a broad understanding of what it does so that you can work effectively with your sound design team. For an introduction from the application designer's perspective, see Compositional Music Elements. For more detail, see the DirectMusic Producer documentation.

DirectX Audio delivers full functionality on Microsoft® Windows® 95, Microsoft® Windows® 98, and Microsoft® Windows® 2000. However, support for hardware synthesizers is available only on Windows 2000 and Windows 98 Second Edition.

### 2.1.4 Network

#### 2.1.4.1 DirectPlay

The Microsoft DirectPlay API provides developers with the tools to develop multiplayer applications such as games or char clients.

DirectPlay provides a layer that largley isolates your application from the underlying network. For most purposes, your application can simply use the DirectPlay API, and enable DirectPlay to handle the details of network communication. DirectPlay provides many features that simplify the process of implementing many aspects of a multiplayer application, including:

- Creating an managing both peer-to-peer and client/server sessions
- Managing users and groups within a session
- Managing messaging between the members of a session over different network links and varying network conditions
- Enabling applications to interact with lobbies
- Enabling users to communicate with each other by voice

## 2.2 Game Engine

### 2.2.1 Object Models

#### 2.2.1.1 Modelling and 3D Models

In 3D applications, objects in 3D scene must be modelled before using in application. Applications need coordinates of objects in 3D space to draw them.

In earlier game applications, the world was not 3D. Objects in scene wasn’t modelled as 3D objects. They were just images. Mostly artists were making 2D images look like 3D. But applications draw that objects just as a 2D image. Creating a scene only using 2D images is really hard. Because every object in the scene must be drawed by artists. Also if objects are animated, all animation frames must be drawed and if object will look like 3D the display of object from other directions also must be drawed. So in a game application, there will be lots of images must be drawed.

Using 3D models eases this process. With an object in game modelled as 3D object, the object can be easliy rotated and animated by modelling programs. After artist models the object once, all animation and rotation can be done using same model. So the work of artist is reduced.

However, using 3D objects in application requires fast computers. Because when using 2D images, an object was just a rectangle. Computer draws just one polygon to draw an object. But using 3D models every object consist of lots of polygons, and when drawing one object computer draws lots of polygons. This is a lot of work for computer CPU and GPU.

But after the speed of both computer CPUs and GPUs increased, applications started using 3D models. While artists have less work, they created much better images and application graphics qualities are increased.

#### 2.2.1.2 3DS Max

3DS Max is one of the most popular 3D modelling programs. 3D modelling programs are used as an environment to model and texturize 3D objects efficiently. Using modelling programs, objects are modelled, and then they exported model data in files. Applications use this files to load model data like vertex coordinates and texture coordinates and draw model in scene.

There is lots of modelling programs currently available. Some of these are: 3dsMax, Maya, Bryce, Lightscape, and Rhino 3D. Most games and animations are created using these programs.

3dsMax is a very easy to use modeller. It’s not used just for modelling. There is lots of things you can do with Max. It can materialize objects and use that materials in applications.

### 2.2.1.3 Model File Formats

After object is modelled, it should be exported to application. Most modelling programs have their own file formats to save models. But this formats are mostly too complicated for being used in your own application. So most applications use a more general format and export model to this format from modelling application, or if this is also not satisfiying application's needs or too much for application, you can write your own format and exporter for your modeller.

There is lots of general formats available. Most commons are 3DS, MD2, MD3 and OBJ formats.

#### 2.2.1.3.1 3DS

3DS is main export format of 3dsMax. But other modelling programs are also capable to export this format.

This format contains most information about model. It's vertex, texture and normal coordinates, material info, and also light and camera info. So using 3ds file, a 3d world scene in a modeller application can be completely exported to a game application.

3DS format also contains animation data. But this data is very limited. It doesn't contain bone animation, it only contains rotating and transforming animations. So if one of the vertexes of object's position is changed it'll not be carried in 3DS format. So if model is changing shape, this will not effect 3DS animation.

So 3DS format is very suiatalbe for static objects, but for animated objects it is not convenient.

#### 2.2.1.3.2 MD2

MD2 is model format of Quake2. This format is very easy to use format and used in a lot of other games. This format contains vertex, and texture coordinates and materials. It also carries bone animation. So this format is good for animated objects. But this format has limits in vertex per object amounts.

#### 2.2.1.3.3 MD3

MD3 is model format of Quake3. It is an improved version of MD2 format. Stuff like vertex amount limit is improved. Vertex per object limit for MD3 is 4096 vertex.

This format also contains bone animation. So it's best format for animated models for now.

### 2.2.1.4 Model Loading

Modelled and exported objects must be loaded by application. Model files are binary files contains model information. So application loads all data about model from file. There is model loaders for both 3DS and MD3 formats. After model loaded this data used to draw model in OpenGL scene.

## 2.3 Performance

### 2.3.1 Rendering Tecniques

Before talking about rendering techniques, some information about how OpenGL works can be usefull.

OpenGL library operates as a pipeline. The data of objects which will be drawed to screen is entered to pipeline from one side, and it goes through all the pipeline. At the other side of pipeline, data exits as a data ready to draw to screen. Then this data is sent to frame buffer.

A macro view of this pipeline is like this:

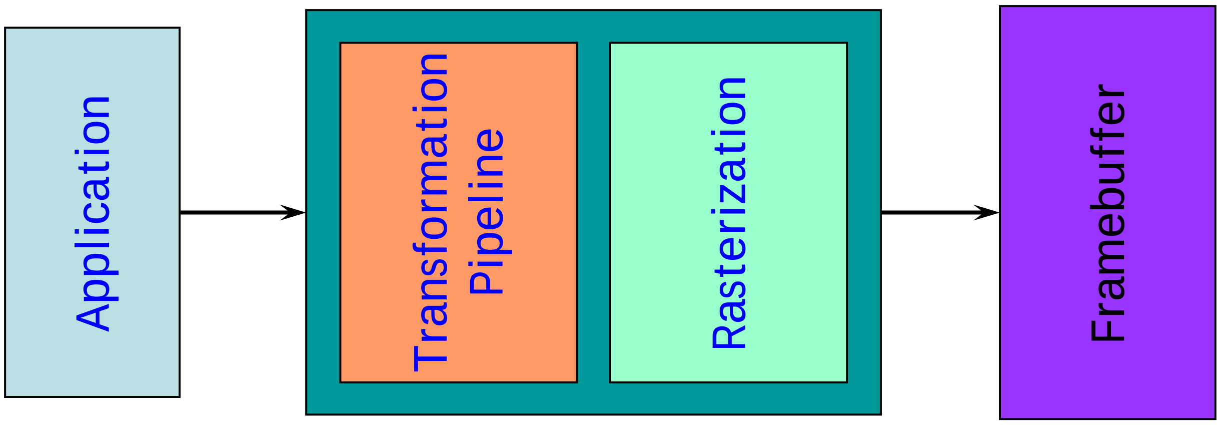


**Figure 2.3:** Macro view of OpenGL pipeline

Application transfers data to GL. First transformation operations are performed on data. Data rasterization operations are performed with results of transformations and then data transferred to framebuffer.

Every frame, this operations are performed again. So the the duration of this operations gives us the frame per second of our application.

This pipeline has levels to perform operations. For example lets look at transformation pipeline much closer:

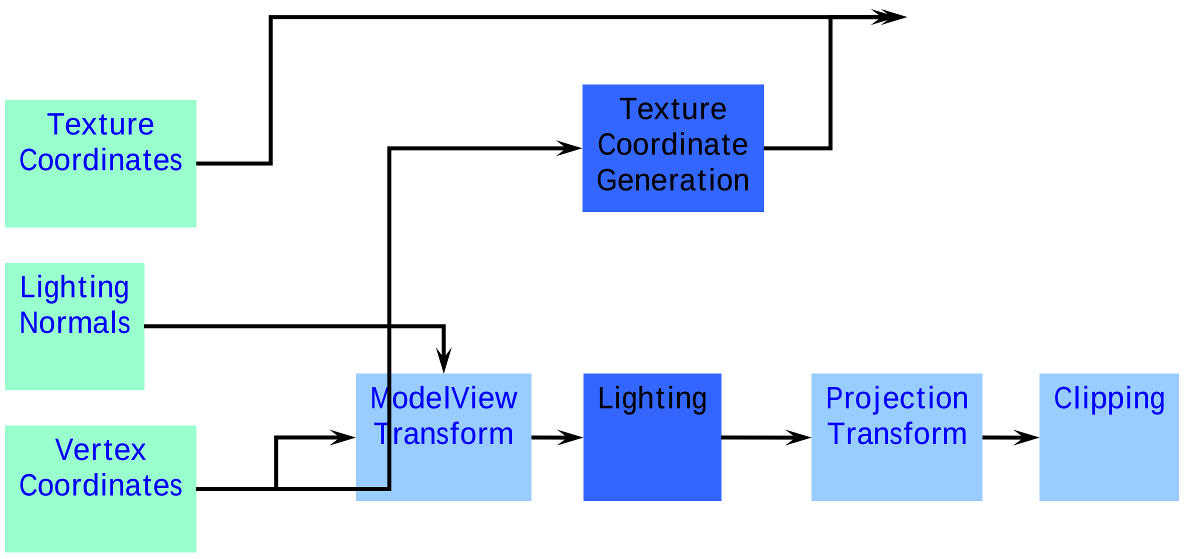


**Figure 2.4:** Transformation Piplene

After data is passed from one level, it is sent to next level. This operations are performed one by one, not simultaneously. So levels will wait the previous levels to finish their operations.

### 2.3.1.1 Immediate Mode

Immediate mode is using glBegin, glEnd, glVertex*, glTexCoord* funtions. In this mode, the data is sent to graphics pipeline immediately drawed to screen. After every glEnd command, data sent to graphics pipeline processed and drawed.

This way there will be a lot of subrotine calls to draw an object. For example if you'r drawing an object with 100 polygons with texturing and normals, there will be 3 glVertex call, 3 glTexCoord call and 1 glNormal call for every polygon. So it's 700 subroutine calls. When polygon amounts are increasing this subroutine calls will slow down application.

Also data will be transfered to GL in every function call. Because these functions are sending just small data, they will not use any advanced transferring method, so sending lots of data this way will be the slowest way.

An example of drawing one triangle in immediate mode:

```
glBegin(GL_TRIANGLES);
        glVertex3f(-1.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);
        glVertex3f(0.0f, 0.0f, 0.0);
glEnd();
```

This way all vertex data of triangle transfered to GL using *glVertex3f()* calls.

### 2.3.1.2 Display Lists

To make renderings of objects with lots of data faster, display list created. With display lists the data of an object is sent to graphics pipeline for once, only when display list is being created. But after this operations are performed, all operation results will be stored and not performed later. When a call to list occured, only the results are sent to framebuffer and objects drawed to screen. This way the rendering will really be much faster.

However, display lists also have some bad sides. With using display lists, you can't transform or rotate the object. Because all data is sent once and stored in graphics memory, later changes will not effect this data. So object will be stable.

Also display lists uses lots of memory, so user must be carefull when using more display list.

An example of drawing one triangle with display list:

First the display list must be prepared:

```
int id= glGenList(1);
glNewList( id, GL_COMPILE );
        glBegin(GL_TRIANGLES);
                glVertex3f(-1.0f, 1.0f, 0.0f);
                glVertex3f(1.0f, 1.0f, 0.0f);
                glVertex3f(0.0f, 0.0f, 0.0);
        glEnd();
glEndList();
```

Then display list will be drawed like this:

```
glCallList(id);
```

As you can see, first all data transferred to GL and compile. Now the object is ready and drawed with just a *glCallList()* call.

### 2.3.1.3 Vertex Arrays (VA)

Vertex arrays are used to transfer arrays of vertex data to GL with many fewer commands than previously necessary. There is six arrays, one each storing vertex positions, normal coordinates, color, color indices, texture coordinates, and edge flags.

The primary goal of vertex arrays is to decrease the number of subroutine calls required to transfer non-display listed geometry data to the GL. Using vertex arrays application first prepares all object data in arrays and specifies which arrays will be drawed. Then in one call all data will be sent to GL. A secondary goal is to improve the efficiency of the transfer, especially to allow direct memory access (DMA) hardware to be used to effect the transfer.

An example of drawing one triangle with vertex arrays:

First array is prepared:

```
GLfloat[3][3] array= { (-1.0f, 1.0f, 0.0f),
                       (1.0f, 1.0f, 0.0f),
                       (0.0f, 0.0f, 0.0f) };
```

Then this array is used to draw triangle:

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, array);

glDrawArrays( GL_TRIANGLES, 0, numOfVerts);
```

As you can see first GL_VERTEX_ARRAY client state is enabled, this tells OpenGL data there is data in vertex pointer. So when *glDrawArrays()* called, it draws data in vertex pointer which is *array*.

There is other client states like GL_TEXTURE_COORD_ARRAY and GL_NORMAL_ARRAY, to send all the data simultaneously.

#### 2.3.1.4 Vertex Buffer Objects(VBO)

VBO is very similar to vertex arrays. The main difference and it's goal is the data is stored in graphics memory, not in system memory. This gives to graphics hardware faster access to graphics data and decreases transfer times.

An example of drawing one triangle with VBO:

First array must be prepared and sent to graphics memory:

```
GLfloat[3][3] array= { (-1.0f, 1.0f, 0.0f),
                       (1.0f, 1.0f, 0.0f),
                       (0.0f, 0.0f, 0.0f) };
int id;
glGenBuffersARB(1, &id);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, id);
glBufferDataARB(GL_ARRAY_BUFFER_ARB,        numOfVerts,        array,
GL_STATIC_DRAW_ARB);
```

Then this array will be drawed:

```
glEnableClientState(GL_VERTEX_ARRAY);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, id);
glVertexPointer(3, GL_FLOAT, 0, (char*)NULL);

glDrawArrays(GL_TRIANGLES, 0, numOfVertes);
```

*glDeleteBufferARB(id);*

First part gets an id to reach the data in graphics memory, and sends data to graphics memory. Second part just looks like vertex arrays but before specifying vertex pointer we tell to OpenGL to use data in graphics memory by calling *glBindBufferARB()*. This way OpenGL gets data from graphics memory and draws them. The last call *glDeleteBufferARB()* frees memory in graphics hardware, if we'll not use this data again.

#### 2.3.1.5 Indices

Indices are used in vertex arrays and VBOs. As we mentioned vertex arrays transfers arrays of data at once and draws them. But an object may have same vertexes in this array. To decrease the number of vertexes, duplicate vertexes are not stored in array. But to produce the same object, an indice array is also sent which keeps indices of vertexes where they will be drawed. This reduces the amount of vertex transferred.

### 2.3.2 Cullings

Another way to increase the rendering speed is decrease the amount of polygons that will be rendered. To do this applications can make some simple calculations to see that if a polygon will be visible or not. If it will not be visible, application will not transfer it to GL, so the amount will be decreased.

#### 2.3.2.1 Backface Culling

One of this culling techniques is backface culling. The idea of backface culling is understanding if a face of 3D object is in front view or covered by another face. To simply understand this vertex of polygons are sent in a default order. Say clockwise or counter-clockwise. So you may say, ater rotations if the vertexes of an polygon is counter-clockwise, don't draw it. Because you know that you all sent front face polygons in clockwise order, so if it is counter-clockwise it turned backface and a frontface polygon will cover it.

This method only works with fully 3D objects. If an object is 2D, the polygon will not be drawed when it's rotated, but nothing will cover it.

Also with using this method, every polygon must be created in clockwise order or counter-clockwise order. If some are clockwise and some are counter-clockwise, this method will not work correctly.

To enable and disable backface culling:

*glEnable(GL_CULL_FACE) and glDisable(GL_CULL_FACE)*

functions will be used to specify clockwise or counter-clockwise you can use *glCullFace(GL_BACK) or glCullFace(GL_FRONT)*

will be used.

### 2.3.2.2 Frustum Culling

Frustum is the visible part of an OpenGL scene. OpenGL scene is a 3D space and objects may be in anywhere of this space, but only the objects is current frustum will be visible in screen. However, even if object is not in frustum, it'll pass all the pipeline operations and will take time to process. This will produce unnecessary operations.

To solve this problem, application can decide to transfer the object to GL or not looking if it's visible or not. Current frustum of OpenGL scene can be obtained from GL and application makes frustum test to see if object is in current frustum before transfering it. So there will be no unnecessary operations for an invisible object.

This method is very handy if world scene is very big. Because there will be a lot of object in a big world, if GL tries to render all of this objects it'll be too slow. So invisible objects will not be transfered and obly the visible objects will be rendered.

# 3. DEVELOPED PARTS

## 3.1. Windows Programming

### 3.1.1 Creating OpenGL Window

Applications, running on Windows OS and has a GUI, must have a window class (wc) structure. This structure contains information about application and registered at the beginning of application window creation. The information in this structure is used during the operation of application.

OpenGL applications are not different. Even if they have a different GUI than most of the windows applications, they still need to reside in a window. But this window must be specially created to draw OpenGL context. So at the initialization of program, an OpenGL window is created, and wc structure for this window is registered.

During creationg of our window all necessary information is set and registered. The most important informations for our application is listed here:

- Window class name.
- Resolution of display
- Frequency of monitor
- Color bits
- OpenGL supported pixel format
- Using DoubleBuffering while drawing

The diagram of main steps of creating a window:

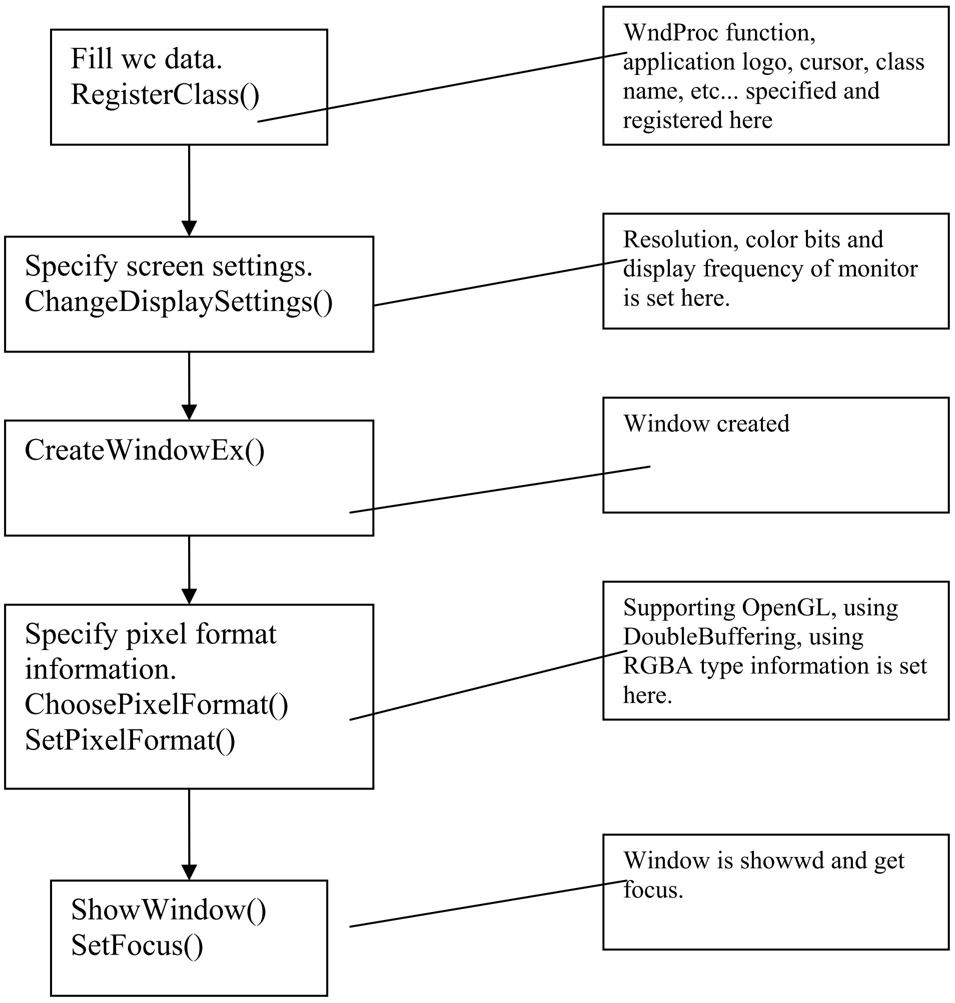

Fill wc data.
RegisterClass()
WndProc function, application logo, cursor, class name, etc... specified and registered here
Specify screen settings.
ChangeDisplaySettings()
Resolution, color bits and display frequency of monitor is set here.
CreateWindowEx()
Window created
Specify pixel format information.
ChoosePixelFormat()
SetPixelFormat()
Supporting OpenGL, using DoubleBuffering, using RGBA type information is set here.
ShowWindow()
SetFocus()
Window is showwd and get focus.

**Figure 3.1:** Window creationg diagram

### 3.1.1.1 GLWindow Class

To handle all windows creating operations, a class is created. This class creates the window and destroys it when application is finished. The properties of this class are:

- Creating window with specifications given at constructor
- Destroying the window at destructor
- Changing resolution
- Providing information about window

Declaration of constructor is here:

```
GLWindow(WNDPROC WndProc, char* name, char* title, int width= 640, int
height= 480, int bits= 16, bool fullscreenflag= true);
```

It gets all necessary information for window and creates it.

### 3.1.2 Windows Events

OS provides applications with every event occured on system. This way applications can handle events they need, and run smoothly on OS. As our game is also an windows application, it gets events from OS.

Events are passed to application by WndProc function. This game's WndProc function is registered to OS during windows creation. Application implements this function and gets some events.

The list of handled system events and reasons of handling:

- WM_ACTIVATE:

  This event is fired when application lost focus and get focus. When application lost focus DirectInput lost its connect with windows events. So when application regain focus, this connection must be rebuilt.

- WM_CLOSE:

  Application is closing, so delete pointers, or destroy objecs.

- WM_KEY events:

  All keyboard events. Game menu is not using DirectInput, so it gets events by windows events.

- WM_MOUSE events:

  All mouse buttons and move events. Because application is not using DirectInput for mouse it uses this events.

- WM_SIZE:

  If application is windowed, when size of window changes this event occurs. When size is changed OpenGL scene must be resized to.

There is two more event in our application. However, these two events are not system events, but events created by our own application.

When application is multithreaded, sometimes it is being necessary to notify main thread by side threads. At this need, side thread sends a message to main thread by PostMessage() system function. With this function, WndProc function of main thread is called with specified event.

This application has two special events:

- WM_INIT_COMBAT_AREA:

A network thread sends this event due to a network message.

- WM_NEW_NETWORK_MSG:

  A new network message received and put to buffer.

## 3.2 Class Designs & Game Classes

### 3.2.1 Message Handling

In different parts of the game, objects on screen will need to access an event. They would get the event and do something when that event occured. This need makes distribution of events essential. Objects must get an event if that event belongs to them.

Because of this need, an event distribution mechanism is developed. This system gets events and sends them where they should go. Lets say, there is four rectangle in our scene and one text field. When mouse clicked to one of the rectangles in the scene text field should display which one is pressed. So application must test the mouse location with rectangles and decide which one is clicked, and display the result in the text field. However, if amount of rectangles are very variable and this kinds of situations happening very often, some questions must be asked. 'What will test the mouse position and rectangles' positions?', and 'How it will access the data of rectangles and mouse event?'.

Our system coming handy in these situations. System specifies classes as container classes and event catcher classes. Event catcher classes must be added to container classes to catch events. When an event occured it's only sent to container class, and container class evaluates tests with event data and the objects on it, and sends event to objects which should get the event.

In every section of game, all game displays are composed of container objects and event catcher objects. In most situations, there is one main container object and other objects are added to this object. Container object is not only distributes events, but also manages drawings of objects on it.

Catcher classes has some empty event methods. When programmer extending a new class from this class, he should override methods which he want to catch. This methods are:

```
OnKeyDown(const KeyEvent& event)
OnKeyUp(const KeyEvent& event)
OnKeyPress(const KeyEvent& event)
OnMouseClick(const MouseEvent& event)
OnMouseDblClick(const MouseEvent& event)
OnMouseDown(const MouseEvent& event)
OnMouseUp(const MouseEvent& event)
OnMouseMove(const MouseEvent& event)
OnMouseOver(const MouseEvent& event)
OnMouseLeft(const MouseEvent& event)
```

These methods are called by container object when related event occured. For example, if mouse is moving over a rectangle, that rectangle objects mouse over method is called. Programmer implements OnMouseMove method and catchs mouse move event.

These are the main container and event catcher interfaces:

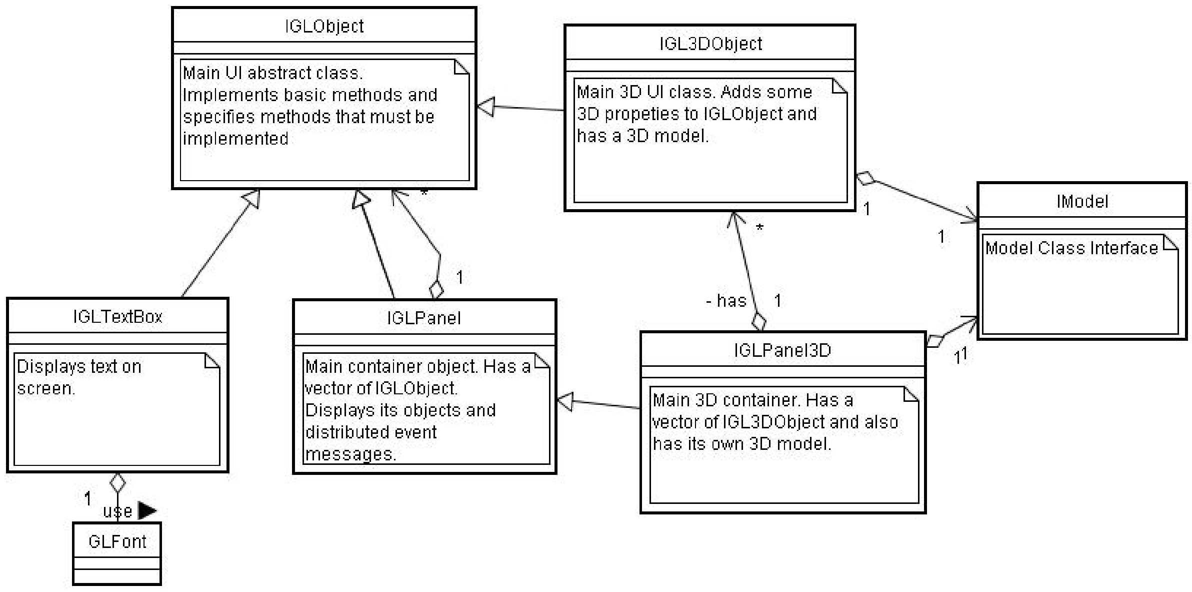

IGLObject
Main UI abstract class. Implements basic methods and specifies methods that must be implemented
IGL3DObject
Main 3D UI class. Adds some 3D propeties to IGLObject and has a 3D model.
IModel
Model Class Interface
IGLTextBox
Displays text on screen.
IGLPanel
Main container object. Has a vector of IGLObject. Displays its objects and distributed event messages.
IGLPanel3D
Main 3D container. Has a vector of IGL3DObject and also has its own 3D model.
GLFont
use
- has
1
*

**Figure 3.2:** Main container and catcher classes

*IGLPanel* is the main container object. As you can see from the diagram, it contains instances of *IGLObject* class. *IGLObject* class is the main event catcher class. Also, *IGLPanel* class derived from *IGLObject* class. So a panel is both a container and an event catcher. This means a panel can have other panels.

Container classes always have pointers of event catcher classes. So a container can have any class derived from its event catcher class. Using polymorphism the methods of inner classes are called when container called a method. Also destructors are defined as *virtual* and classes can destroy their own resources.

*IGLTextBox* is a general implementation of an event catcher class. This class displays a text and responds to events, like a mouse over and left event.

The list of event catcher and container interfaces:

- IGLObject
- IGL3DObject
- IGLPanel
- IGLPanel3D
- IGLTextBox
- IGLButton
- IGLEditBox

The transfer of events in application can be demonstrated with this diagram.

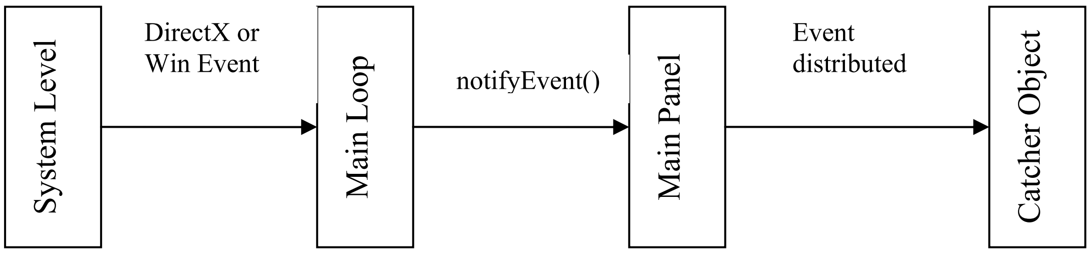


**Figure 3.3:** Transfer of events in application

### 3.2.2 Menu Classes

Menus are implementations of our event mechanism. There is a special panel class and other objects are added to this panel class. This way all keyboard and mouse events are distributed to our objects and we have an interactive game menu.

Using mouse over and mouse left events, we made our objetcs on panel to light up when mouse is our them. Also using mouse click event we made our text boxes to work as buttons performing operations.

This is the diagram of main menu classes:

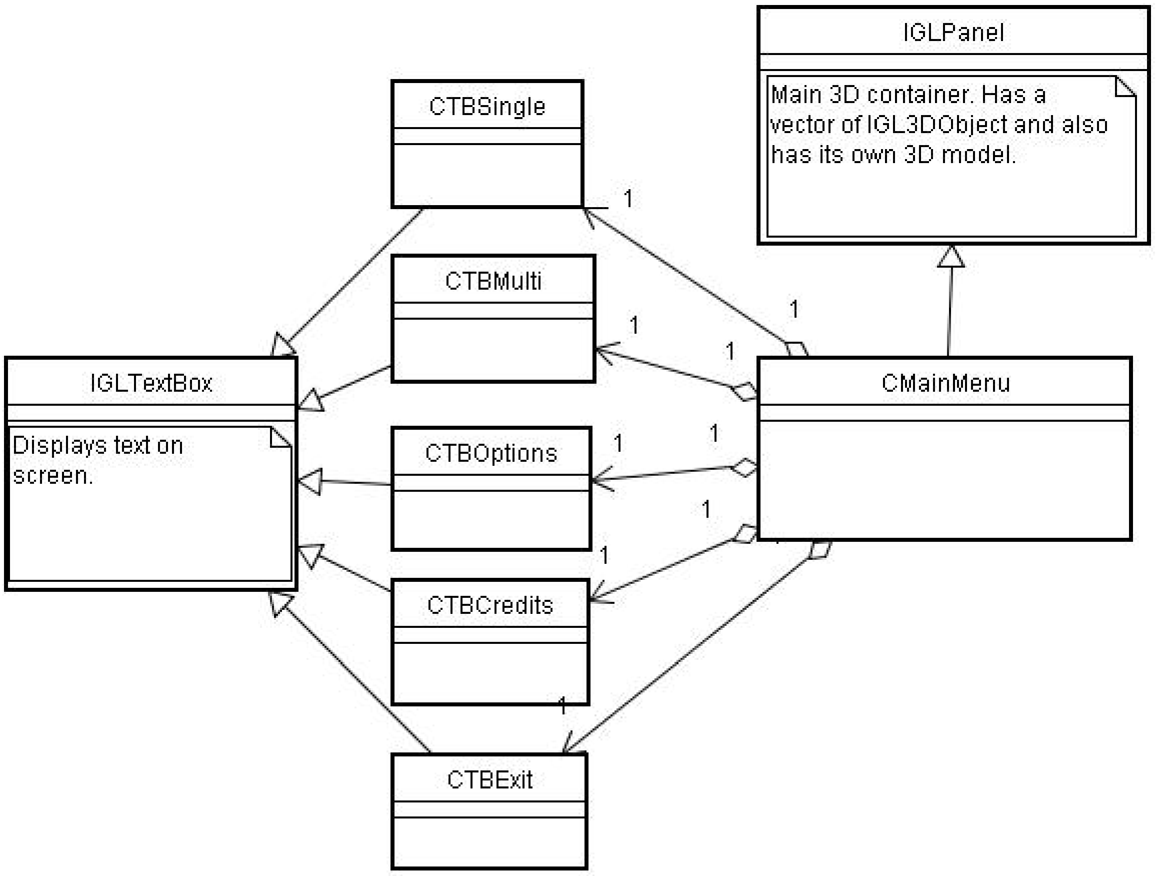


**Figure 3.4:** Menu classes

So you see that all our buttons are derived from IGLTextBox, and CMainMenu is derived from IGLPanel. CMainMenu has its own drawing function to display fog effected menu background.

All buttons has mouse over, mouse left and mouse click events implemented. They perform their specific operations at mouse click functions.

Here is a screenshot how our main menu looks like:

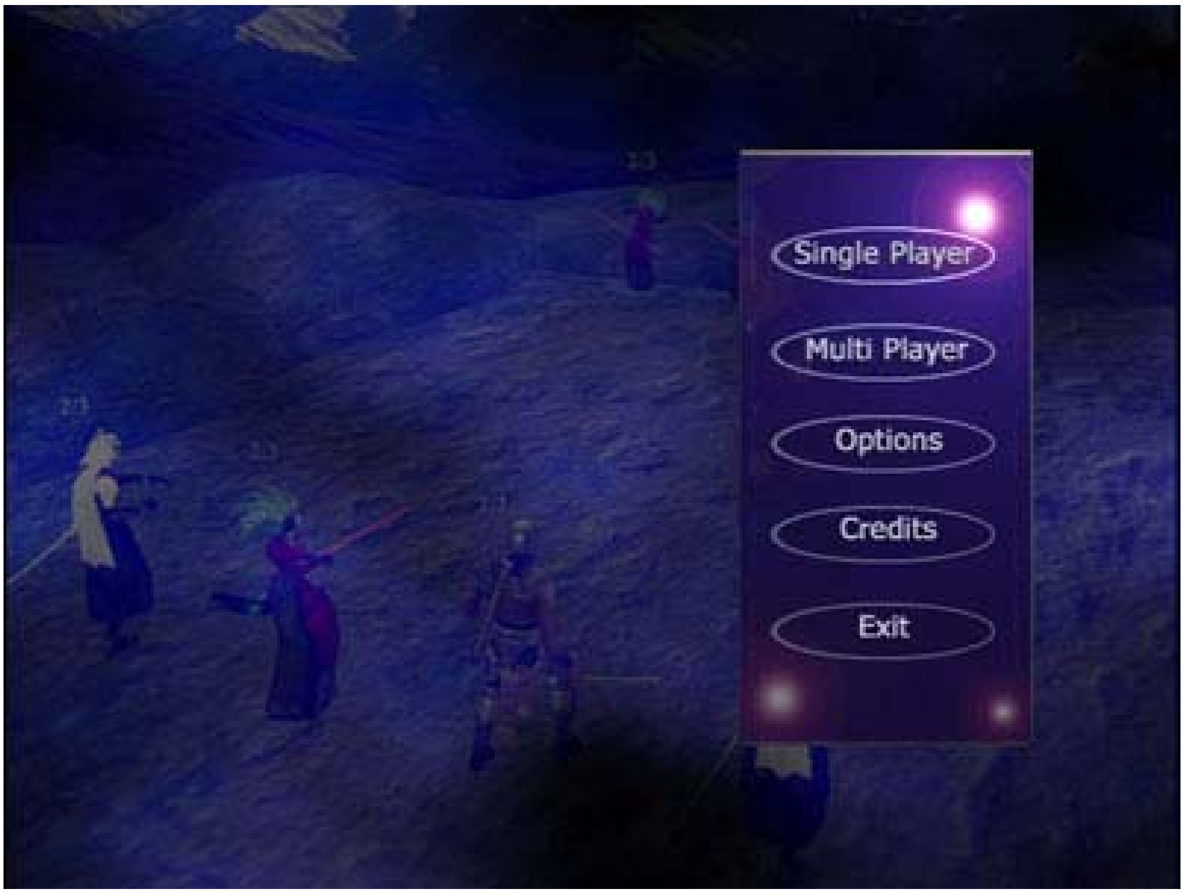


**Figure 3.5:** Main menu screenshot

### 3.2.3 Combat Area Classes:

Combat area classes manages the combat part of game. When game is started, combat area classes are being initialized, and displaying 3D game environment. All in game displays and game rules are in control of this part.

This part consist of:

- Combat area resource loader
- 3d combat environment
- Game user interface
- Combat area manager

Here is a diagram of this part:

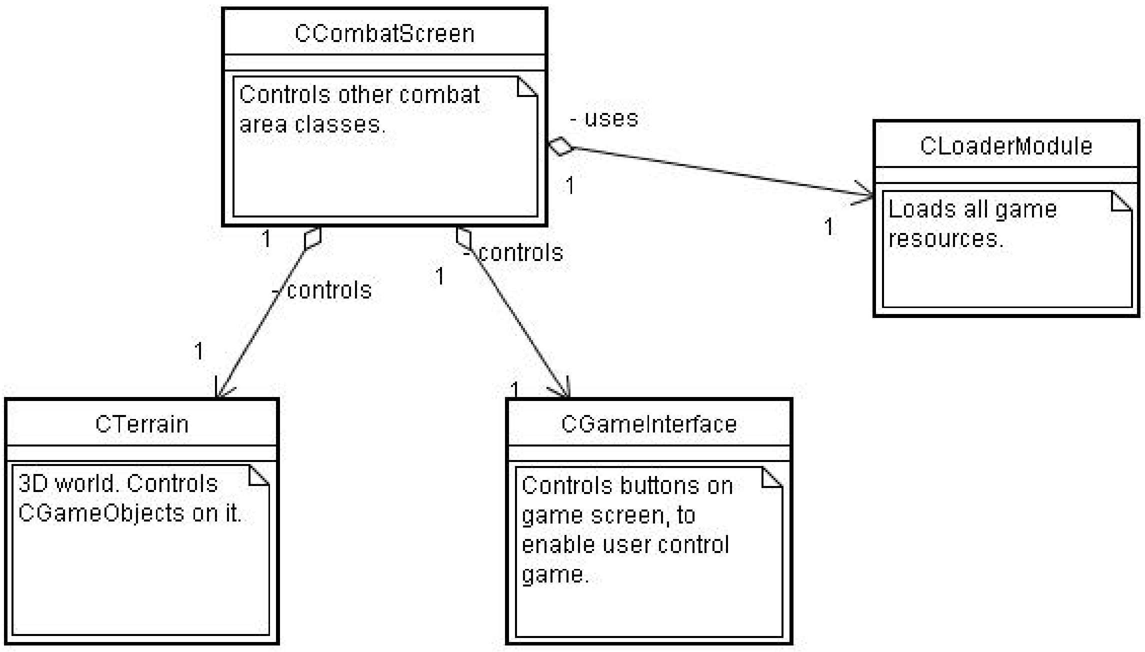

CCombatScreen
Controls other combat area classes.
- uses
1
1
CLoaderModule
Loads all game resources.
1
-controls
1
-controls
1
1
CTerrain
3D world. Controls CGameObjects on it.
1
CGameInterface
Controls buttons on game screen, to enable user control game.

**Figure 3.6:** Combat area classes

### 3.2.3.1 CCombatScreen class:

This is the combat area manager class. Other combat area classes are inside of this class. This class controls other classes and provides relations between classes.

Even if this is not a panel class, here this class first gets events from main thread. Because here events must be distributed between user interface and combat terrain. So this class gets events and first sends them to user interface. If user interface is not getting event, sends it to combat terrain.

### 3.2.3.2 CLoaderModule class:

All resource loading operations are performed by this class. When game is starting this module displays a loading screen and loads initial resources for combat area. During the game again this module is used to load additinal resources.

It loads game information data from text or binary files. For models or texture resources it uses 3D model loader classes.

### 3.2.3.3 CTerrain class:

This class is explained at next secion 3.2.4.1.

#### 3.2.3.4 CGameInterface class:

On combat screen, there is buttons to enable user to control game operations, like playing a different card. All this buttons makes the GUI(Game User Interface). GUI is controled this class.

For users, using a user interface is better than making every action by keys. Giving every action to keys makes it harder to use program at beginning. Providing a user interface is better at beginning.

Also in this game, user should see which card they are placing. So pictures of cards are displayed on screen, and users click this pictures to place cards.

Here is a screenshot of game combat area:

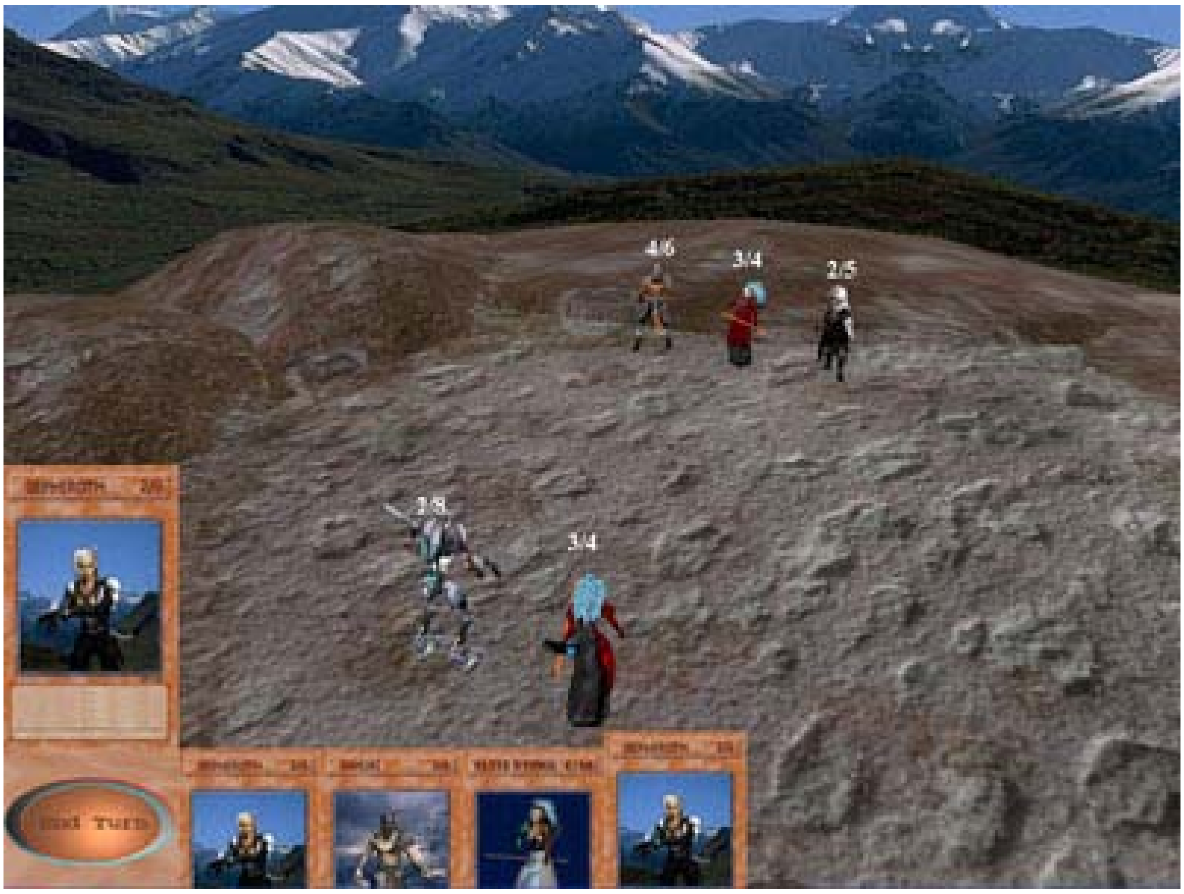


**Figure 3.7:** Combat area screenshot

### 3.2.4 In Game Classes:

Game classes are not different from main menu classes. They are also derived from event mechanism classes. The difference is these classes are derived from 3D classes. Game screen is some kind of main menu in design, but because objects are 3D it will provide a 3D world.

This diagram shows our game classes:

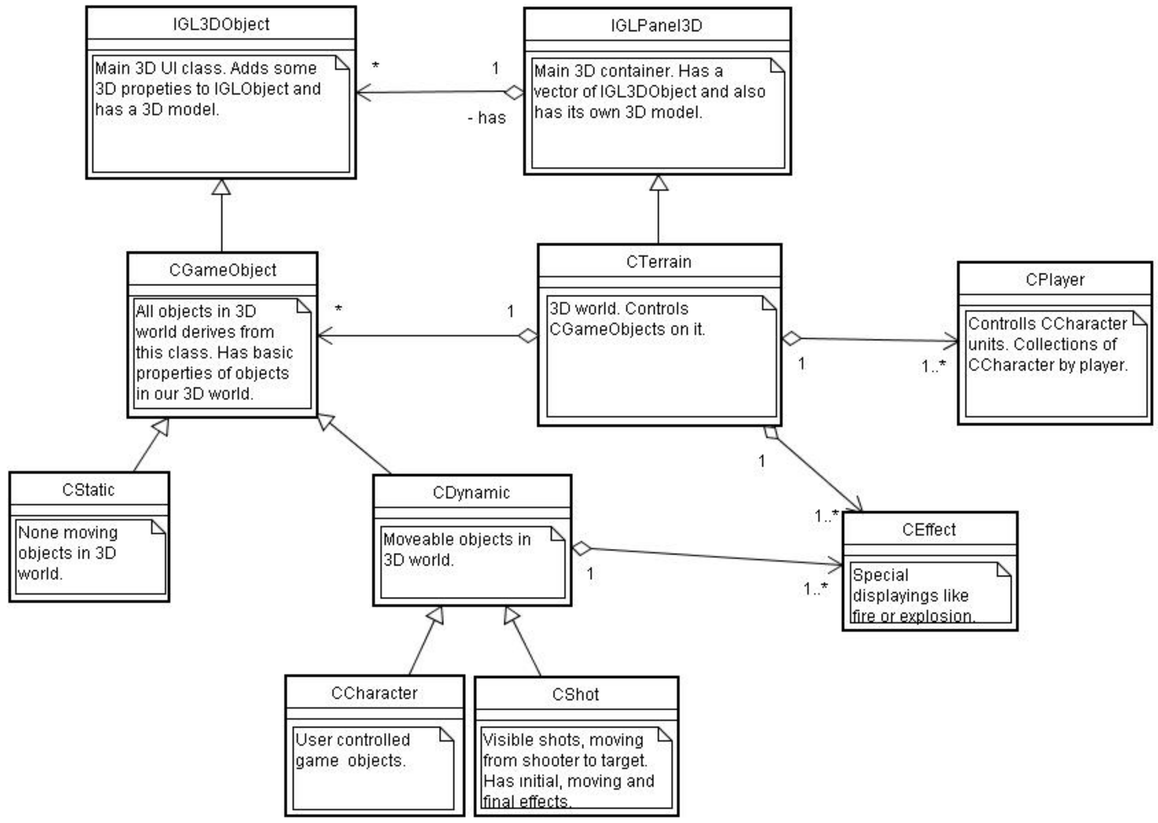

IGL3DObject
Main 3D UI class. Adds some 3D propeties to IGLObject and has a 3D model.
IGLPanel3D
Main 3D container. Has a vector of IGL3DObject and also has its own 3D model.
*
1
- has
CGameObject
All objects in 3D world derives from this class. Has basic properties of objects in our 3D world.
CTerrain
3D world. Controls CGameObjects on it.
CPlayer
Controlls CCharacter units. Collections of CCharacter by player.
1..*
CStatic
None moving objects in 3D world.
CDynamic
Moveable objects in 3D world.
CEffect
Special displayings like fire or explosion.
CCharacter
User controlled game objects.
CShot
Visible shots, moving from shooter to target. Has initial, moving and final effects.

**Figure 3.8:** In game classes

Now we can look closer to these classes.

### 3.2.4.1 CTerrain class:

This class is derived from IGLPanel3D class. It has its own 3DS model to draw our 3D world. At the beginning it will be an empty world. By changing 3DS model file, game can be played at different environments.

As other panel classes, this class also manages other objects. The class that CTerrain manages is CGameObject class. CTerrain will distribute events to CGameObject objects and manage their drawings.

Different than other panel classes, this class also contains CPlayer classes. With CPlayer, CTerrain manages players' informations. It also stores relations with game objects and players, like which object belongs to which player.

#### 3.2.4.2 CGameObject class:

Actually there is not much thing this class do. It only stands to specify a class layer between IGL3DObject class and CStatic and CDynamic classes.

#### 3.2.4.3 CStatic and CDynamic classes:

This classes will be the objects that users will see in our 3D world.

CStatic class is simple. It is objects in game that can not be moved. So they don’t have a move function.

CDynamic objects, on the other hand, are moveable objects. They can be moved by user or game. This class will implement the functions that all moveable objects have.

The lower classes are most specific classes. When an object will perform different action a class is derived from these classes and different functions are implemented and they are added to our world with the same manner as others.

#### 3.2.4.4 CPlayer class:

This class stores information about players. Informations are objects ids player has, the cards that player has, which cards player has at his hand, and others.

Storing all player data in a classes eases the control of player operations.

#### 3.2.4.5 CEffect class:

This class is used to display some special effects in our world. Because our effects will not get events, this class is not derived from IGLObject class.

CEffect class is an abstract class an it has Display() function that should be implemented by the classes derived from CEffect. Every different effect is a new classes derived from this class.

### 3.2.5 Model Classes

This game uses two different 3d models to display 3d objects. To manage operations and draw models, this models are kept in different classes. Other classes uses this model classes to display and load their models.

The diagram of model classes:

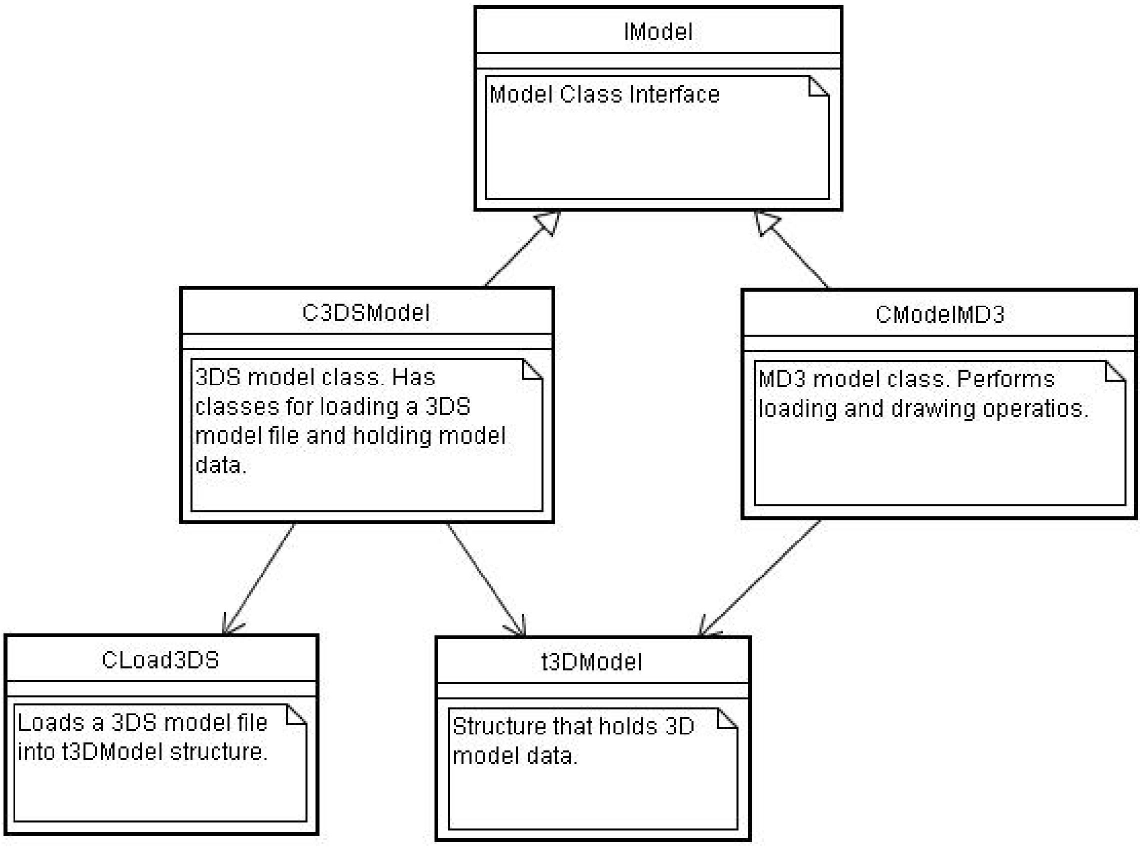

IModel
Model Class Interface
C3DSModel
3DS model class. Has classes for loading a 3DS model file and holding model data.
CModelMD3
MD3 model class. Performs loading and drawing operatios.
CLoad3DS
Loads a 3DS model file into t3DModel structure.
t3DModel
Structure that holds 3D model data.

**Figure 3.9:** Model classes

### 3.2.5.1 IModel Interface:

This interface specifies the common tasks of all model classes. This tasks are like loading model, drawing model, etc.

Classes using models has IModel interface. Using this interface eases the designing of other classes using models. Which model style the class uses is not decided at design, but decided at implementation.

### 3.2.5.2 C3DSModel Class:

This is one of the two model styles. This class performs specific operations to load and display 3DS models. It uses Cload3DS class to load from file.

After loading it stores model data at t3DModel strucure. This structure is common for both models. They uses same data to display the model. The main difference between the models are their file structures, after loading model data is similiar. However, because one model is static and other is animated, we made two different class.
Note: Some textures of .3DS files is taken from an Internet sit [4].

### 3.2.5.3 CModelMDS Class:

This is the other model style. Most operations are similiar, nut this class also has a capacity to display models animated.

Note: All .MD3 models are taken from an Internet site [3].

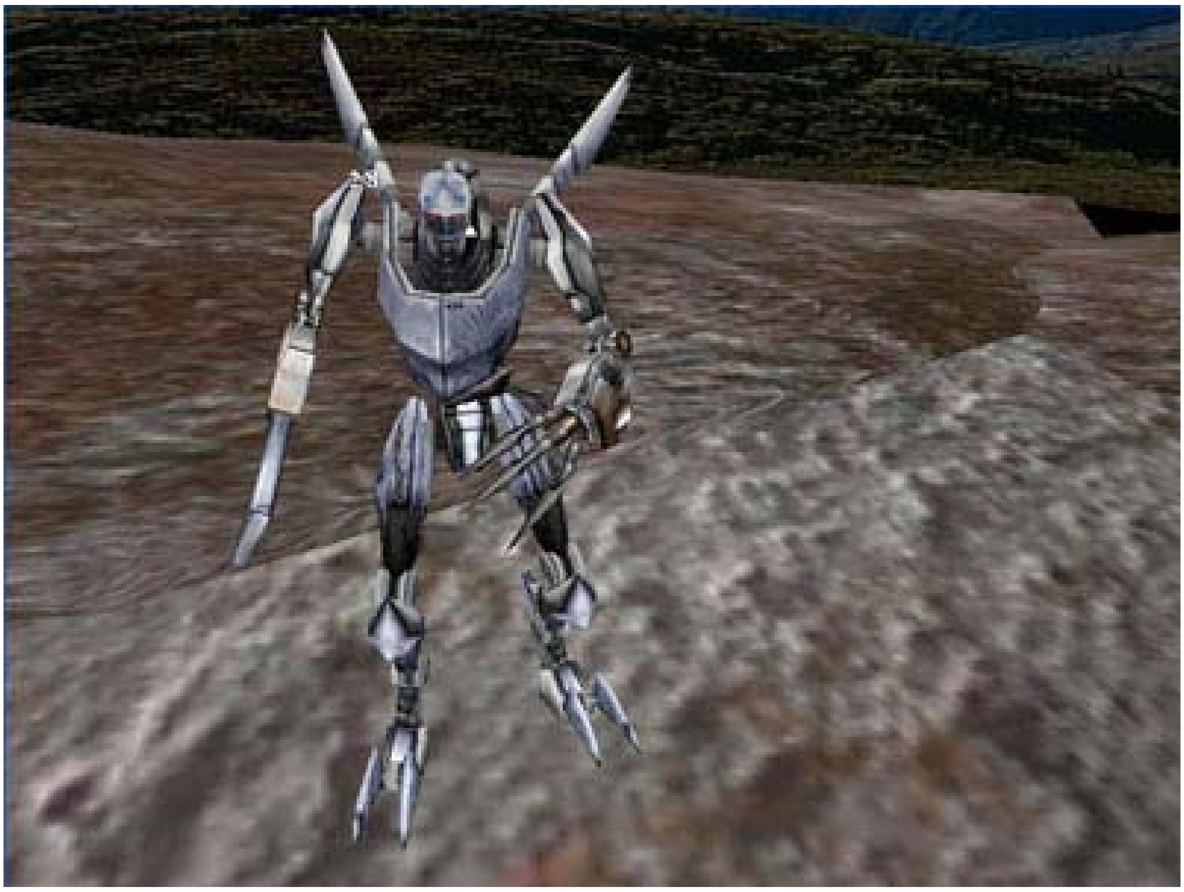


**Figure 3.10:** A .MD3 Model

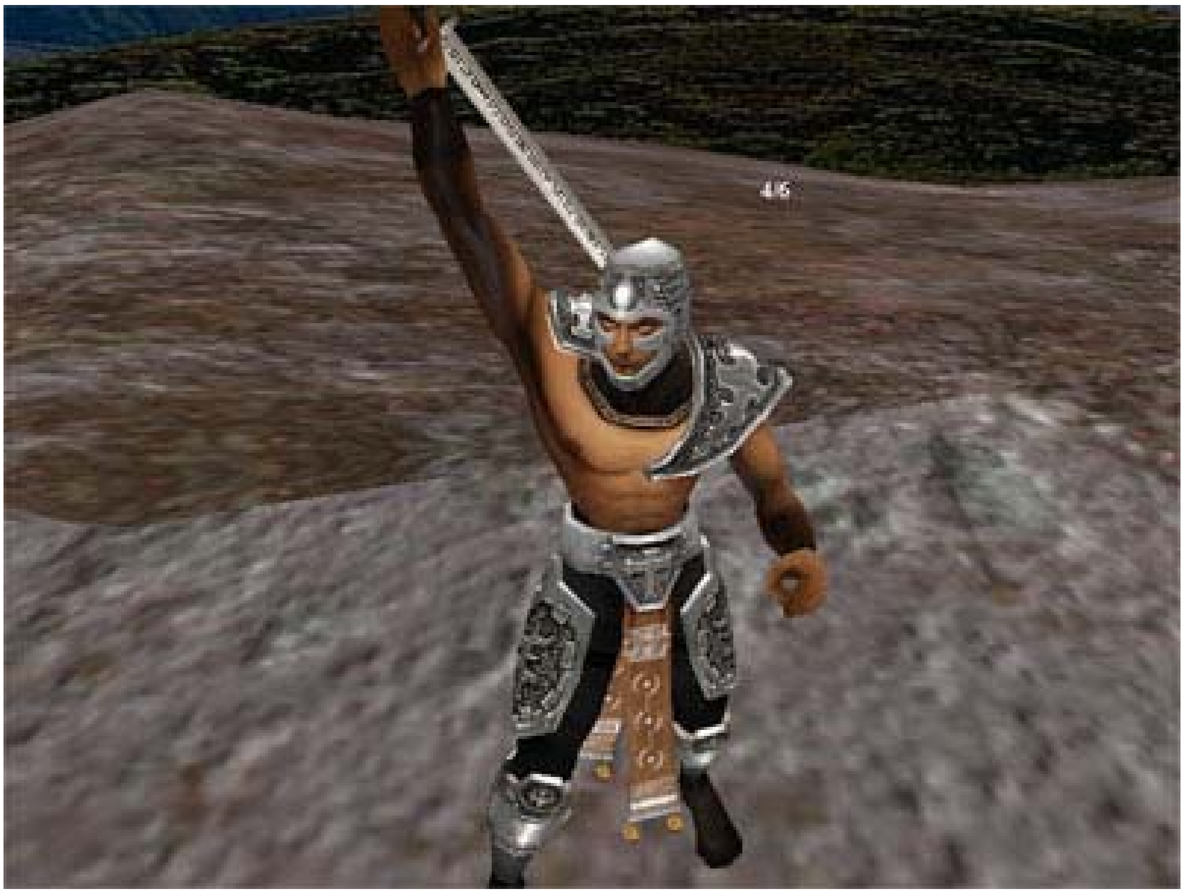


**Figure 3.11:** a .MD3 model

### 3.2.6 DirectInput & DirectAudio Classes

This game used DirectInput for getting keyboard events, and DirectAudio for playing sound. Using both of these APIs requires some initializations. Because of this classes created to perform initializations and provide some functions to other classes to use some features of DirectInput and DirectAudio.

#### 3.2.6.1 GLDInput8 Class

This class perform initialization and termination of DirectInput. When program is initializing, it must get a DirectInput device for keyboard from OS. End at termination this device must be released. This class performs these operations.

Other classes uses to update state of device and get which key is pressed. Also when window loses focus, the device must be reacquired. This class performs this too.

#### 3.2.6.2 GLDAudio8 Class

This class perform initialization and termination of DirectAudio. It creates necessary objects to play sound. It provides functions to load new sound effects and to play these effects.

The sounds effects this class uses can be .WAV or .MIDI files. It can be improved to play .MP3 files, too but currently it’s no playing .MP3 files.

## 3.3 PERFORMACE

### 3.3.1 Content of Game Scene

Game scene is mostly consist of models. The terrain and characters on terrain are both models. But, there is also other things than models. User interface of game is consist of textured polygons. Effects on scene are also some textured polygons.

However, from OpenGL's view of look, everything on game scene is textured polygons. The models are also a amount of textured polygons. When drawing a frame, OpenGL draws polygon by polygon. After finishing the frame, he program continues and at next loop it draws the frame again, maybe with new data.

So speed of drawing a frame specifies are applications speed. And amount of polygons is the biggest effect for speed of drawing a frame. So we'll give the amount of frames on our scene.

Here is the amount of polygons all other different models has:

| MODEL | # of Vertexes | # of Polygons |
|---|---|---|
| Terrain | 309 | 514 |
| BloodSeeker | 1818 | 1651 |
| Ryoko | 1156 | 1325 |
| Samjai | 1750 | 2106 |
| Sephiroth | 1333 | 1568 |

**Table 3.1:** Number of vertexes and polygons of models

Beside these, there will be effects and user interface, but total polygons of them will not exceed 100, so they are not important.

The maximum polygon amount is 2106, and the minumum is 514. But minimum character is 1325. In a scene there could be at most 10 characters and terrain. And minimum is only terrain. Here is the max and min polygons in a scene:

| MAX | MIN |
|---|---|
| 11044 | 514 |

**Table 3.2:** Max and min polygon amounts

With 100 more polygons for user interface and effects, it will be 11144 polygons at most. This is a reasonable value and it'll not cause a big speed problem in most machines. In next sections there will be a analyze of FPS(frame per second) on different machines.

### 3.3.2 Using VBO

At previous section, we said the biggest effect on applications speed is amount of polygons, and their drawing times. However, there is two different part when drawing a polygon, and they have different speeds.

These parts are transferring and rendering. When OpenGL draws a polygon, it first sends polygon data to graphics card, then renders that data and draws it. These two part consumes different amount of time and the speed of drawing a polygon is total of these two.

In current technology level, graphics cards(GPUs) has a very high speed at rendering polygons. They can render ten thousands of polygons in a second. However, this ten thousands of polygons' data first must be transferred to graphics cards, and transferring is way too slower then rendering. So speed of GPU will wasted, because it'll wait for next data to arrive.

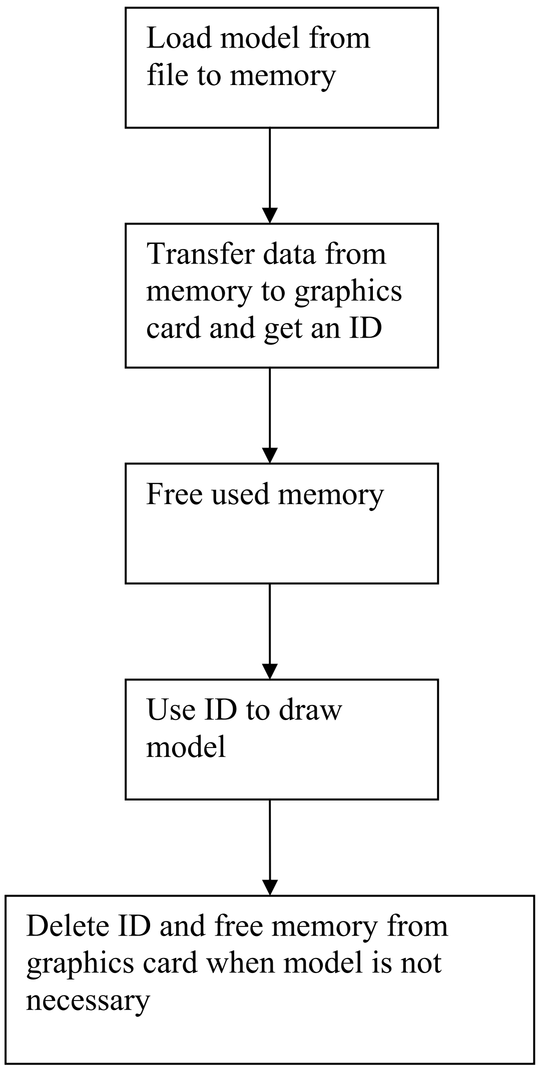
In this project, the maximum polygon amount is 11044 and transfer of this amount of data will take a very long time. So the application will be really slow.

This problem is solved by storing data on graphics memory. By storing data on graphics memory, transfer rates get very fast, and will be faster than rendering. So our limit will be rendering time.

VBO is used to use graphics memory. All polygon data is sent to graphics memory and drawed from there by VBO.

In our project, when models are loaded, they will first be loaded to main memory. Then the data in main memory will be transferred to graphics memory, and they'll be stored there. After storing this data, we'll get an ID for this data, and this ID will be used when drawing model.

The process of loading a model is like this.



Load model from file to memory
Transfer data from memory to graphics card and get an ID
Free used memory
Use ID to draw model
Delete ID and free memory from graphics card when model is not necessary

**Figure 3.12:** Diagram of loading a model

### 3.3.3 Memory Management

Game applications generally uses lots of memory space. Because resources of game must be loaded to memory, and sizes of these resources are mostly very big. Models, textures, sound effects, etc., are some of these resources.

This game is not an exception. It uses models, textures and sound files, and loading of these files to memory is necessary. Because of this, application uses a very big amount of memory. So application must use memory resource of computer carefully.

Normally in OpenGL applications, OpenGL provides most of the memory management. When textures loaded, OpenGL loads them somewhere, to main memory or graphics memory, and deletes them when application is destroyed.

However, in our application, the amount of memory these resources take is very big, and in process of game new resources will keep loading. So after game last for sometime, application will be using a very big amount of memory and most probably there will be a memory problem. So leaving memory management to OpenGL is not a good solution.

Because of these reasons, we made all our resource user classes to free all space they are using when they are destroyed. For example, a model class loads a model when it's created. It first loads them in main memory, and then transfers data to graphics memory. At the end the model data and textures will be kept at graphics memory. When the object is being destroyed, object will delete every space from graphics memory by OpenGL calls.

Because the space is taken by OpenGL, they must be delete by OpenGL too. The functions to delete resources are:

- glDeleteTextures()
- glDeleteBufferARB()

First one deletes loaded texture, and second one deletes model data in graphics memory.

The amount of memory used by our game models are in this table. It shows the amounts in main memory and in graphics memory both. It also shows how much space takes the textures of a model.

| **MODEL** | **Polygon Data On Main Memory (KB)** | **Polygon Data On Graphics Memory (KB)** | **Texture Data On Graphics Memory (KB)** |
|---|---|---|---|
| BloodSeeker | 22932 | 23028 | 4144 |
| Ryoko | 4068 | 4168 | 720 |
| Samjai | 16496 | 16812 | 1520 |
| Sephiroth | 2968 | 2992 | 1740 |

**Table 3.3:** Memory consumes of models

## 3.4 NETWORK

All network operations of this project is handled by DirectPlay. DirectPlay is networking API of DirectX 8.0. If we have used socket programming for this project, we'll have had to develop all the networking parts, message sending and receiving, message format and multithreading. However, DirectPlay supplies all of these, after a short initializtion process. All we have to do is the format of messages to make clients and servers talking each other, which messages will be sent, and how server or client reacts to network messages. Network operations under these are managed by DirectPlay.

Both server and client use DirectPlay objects, and initialize DirectPlay when multiplayer game is starting. When initializing DirectPlay application gives a function poniter to DirectPlay. DirectPlay notifies our application about network events through this function. For example, when a message is receive, message received event will be sent to this function. Declaration of this function:

```
HRESULT WINAPI DPServerMessageHandler( PVOID pvUserContext,
                                        DWORD dwMessageId,
                                        PVOID pMsgBuffer);
```

DirectPlay sends message id and contents of message through this function. Server and client gets different events from DirectPlay and reacts them. List of events server and client get are:

Server events:

- DPN_MSGID_CREATE_PLAYER:
  Create a new player. Player name and id passed through message content.

- DPN_MSGID_DESTROY_PLAYER:
  Destroy player. Player id passed through message content.

- DPN_MSGID_TERMINATE_SESSION:
  Termination of current DirectPlay session.

- DPN_MSGID_RECEIVE:
  New message received from a client. Message and client id is passed through message content.

Client events:

- DPN_MSGID_ENUM_HOSTS_RESPONSE:
  After searching for servers, the result of search returns with this event. Message content contains information about found servers.

- DPN_MSGID_TERMINATE_SESSION:
  Terminte current session.

- DPN_MSGID_RECEIVE:
  New message received from server. Message and client id is passed through message content.

After server or client getting a event, they processes content of the message and performs operations for that message. However, this DirectPlay functions runs on a different thread than our main server, and this thread must perform its operation and end quickly. Because new threads will be created after this thread if new messages will arrive. So the program must not stay at this function long. But network events, especially initialization of a multiplayer game requires a long initialization process.

To solve this problem we made this DirectPlay thread sending events to main thread. When an network event occured and it requires a long process, it sends an event to main thread to perform this process. This diagram show how it works:

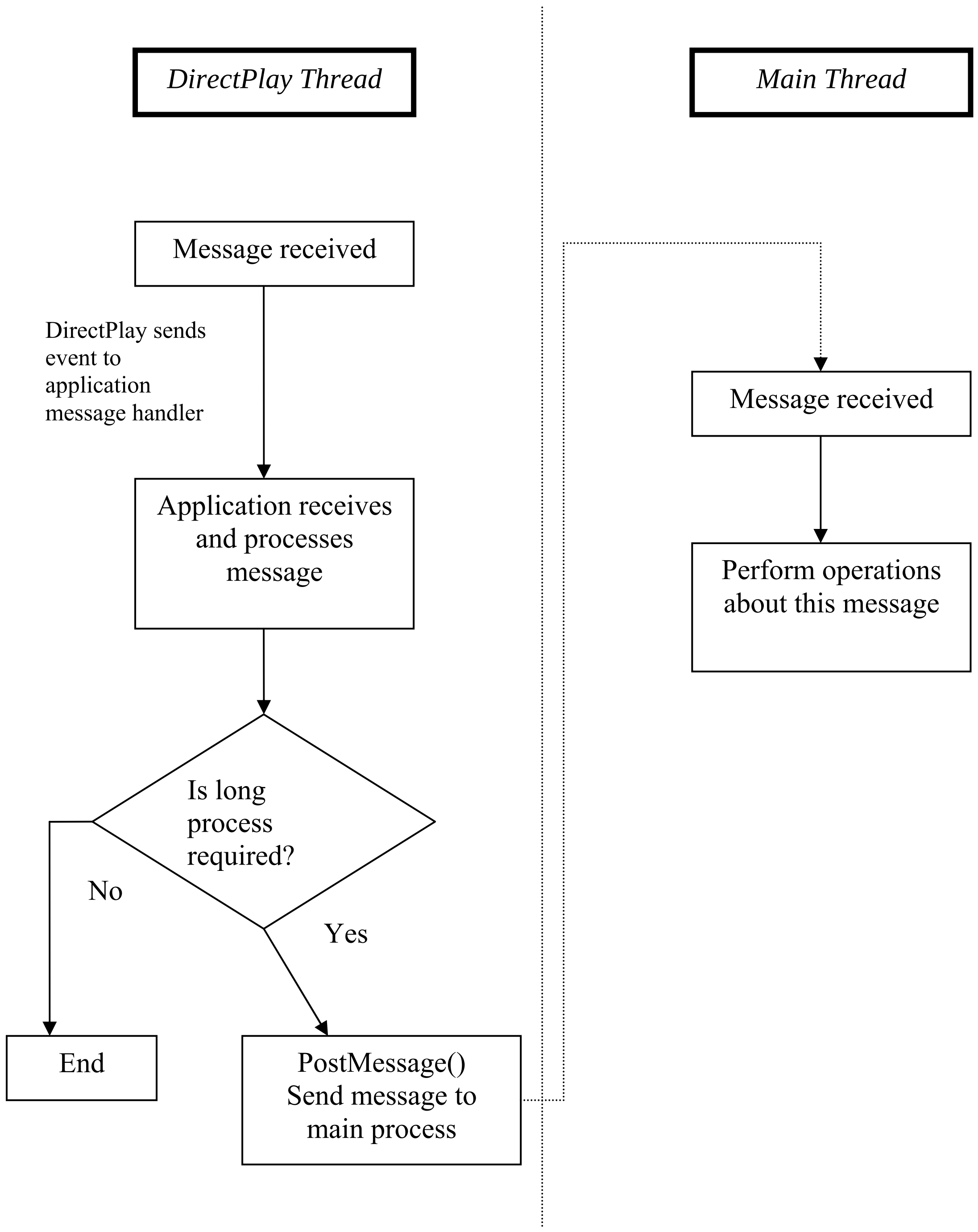

DirectPlay Thread
Main Thread
Message received
DirectPlay sends event to application message handler
Application receives and processes message
Is long process required?
No
Yes
End
PostMessage() Send message to main process
Message received
Perform operations about this message

**Figure 3.13:** Network message diagram

# 4. IMPLEMENTATION

Here we’ll give information about game play and game rules. First the list of game play:

- Game is played by two persons
- Players have their own decks of cards
- The decks has 15 cards
- Every card represents a character or a spell
- Players draws random cards from deck
- Played characters fights with each other and reduces others defenses
- Players places cards, and fights with characters cards are representing
- Players can use spell cards to give damage to opponent characters or heal their own characters

Now the rules of game:

- A player starts to place card first, but other draws first
- Players can have max 4 cards at their hands
- Players can have max 5 characters on area
- In a turn player can place only one card
- Every two turn player draws a new card
- Characters on area can move only once in a turn
- When there is a character at attack position, opponents characters can’t attack characters at defend position
- Only archers or magicians can attack from defend position
- A player loses when he both have no character on area and no card at hand

# 5. CONCLUSION & RECOMMENDATIONS

In this project we tried to develop a between small and medium size game application. In the end of the project, we have a playable game, and basic systems for later improvements. The project last nearly 4 months. However, we can't say this game is finished. It requires some parts to be developed to be finisihed. Here is a list of parts requires to be deleveloped:

- Options menu: graphics options and performance options should be added
- Deck construction menu: a menu to enable users to see their decks before game and edit the deck
- More cards: different cards and characters must be added to make game more strategical
- Different environments: Single environment of combat area bores people, at least 3 different environments must be added.
- Network menu: Network menu must be improved, it must display all available servers and pings.

With 2 months more work, these parts can be added to game, and then this game will be a medium size game application. However, to make this game a more complicated commercial game more tasks required:

- An adventure part, where user can explore the game world and find opponents
- A story-line to make game world living
- Better models and textures
- Full sound effects, every action must have a sound effect, different kinds of musics
- More multiplayer options

Completing these tasks, this game can be at a quality of a commercial game. However, completion of these tasks with only one person is nearly impossible. Even if it's done, it'll not be good enough. Because one person can't develop both models and sound effects. At least 3 or 4 people must work on this project to finish it.

For later developers, we recommend working in at least two persons team. One programmer, and one programmer-modeller. Maybe one more person for sound effects, but it's not necessary.

Using OpenGL for graphics operations, and DirectX for other purposes, like audio, network and inputs is a good combination. Direct3D and DirectDraw are hard to learn, and we don't recommend them. Also there is OpenAL for audio operations, but we didn't use it, so no idea.

# 6. REFERENCES

[1] Microsoft DirectX 8.0 C++ Documentation
[2] Website www.opengl.org
[3] Website www.polycount.com
[4] Website www.3dcafe.com