

**DOKUZ EYLÜL UNIVERSITY  
GRADUATE SCHOOL OF  
NATURAL AND APPLIED SCIENCES**

**A TOOL TO CREATE 3D ANIMATION FILMS**

**by  
Yunus Emre ALPÖZEN**

**December, 2006  
İZMİR**

# **A TOOL TO CREATE 3D ANIMATION FILMS**

**A Thesis Submitted to the  
Graduate School of Natural and Applied Sciences of Dokuz Eylul University  
In Partial Fulfillment of the Requirements for the Degree of Master of  
Science in Computer Engineering**

**by  
Yunus Emre ALPÖZEN**

**December, 2006**

**İZMİR**

## M.SC THESIS EXAMINATION RESULT FORM

We have read the thesis entitled “**A TOOL TO CREATE 3D ANIMATION FILMS**” completed by **Yunus Emre ALPÖZEN** under supervision of **Prof. Dr. R. Alp KUT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Alp Kut

---

Supervisor

---

(Jury Member)

---

(Jury Member)

---

Prof.Dr. Cahit HELVACI

Director

Graduate School of Natural and Applied Sciences

## ACKNOWLEDGEMENTS

This thesis has been prepared at the time of working with a great effort in a different city. Entire thesis implementation consists of 120.000+ lines of code and most of these codes are written during weekends and late at nights. Nevertheless, after this exhausting study, I am very proud of having such a thesis like this. I believe that know how I gathered during this study is invaluable.

I would like to express my deep and sincere gratitude to my supervisor, Professor Alp KUT, Chair of the Department of Computer Engineering. His understanding, encouraging and personal guidance have provided a good basis for the present thesis.

I wish to express my warm and sincere thanks to Associate Professor Yalçın Çebi for his essential assistance and moral support.

If this thesis help someone to build up something that works interest of humanity and for a better world, I would have more vital reasons to be proud of this work.

## **A TOOL TO CREATE 3D ANIMATION FILMS**

### **ABSTRACT**

The aim of this study is to have know how on creating 3D animation films, discipline 3D animation film generation process and develop a sample tool to create 3D animation films.

Current 3D animation film industry use computers that have built in graphics engines and dedicated for this purpose. Uncommonly, underlying hardware is produced specifically for the software to produce films. Most of these computers are operated on Sun Systems and have costs in million dollars rank. It is not possible to reduce these requirements for a personal computer unless this generation process taken under control.

3D animation film generation process can be managed by applying standard SDLC (Software Development Life Cycle). However, execution step requires a robust IDE (Integrated Development Environment) and built in graphics engine that provides abstraction on rendering details and let user to take care about big picture. Development environment code named as “Weendigo”. Weendigo is a Red Indian belief on migration of the human beings' capabilities, courage, power, and skills when he was eaten by another human being. Pneuma is the vital spirit which makes men the most valuable creature in the world. By the way, Weendigo refers to development environment and Pneuma is the name of the underlying graphics engine. Weendigo has a component based architecture that allows add or remove components at runtime by implementing some interfaces. Therefore Weendigo gathers all skills and capabilities from the components added and pneuma let them born. Also this functionality provides different points of view for the people have different roles in this process which will lead to discipline from beginning to end.

**Keywords:** 3D, 3D animation film, movie generation, IDE, Weendigo, Graphics Engine, Pneuma, Microsoft .NET Framework, Microsoft DirectX

# ÜÇ BOYUTLU ANİMASYON FİLM GELİŞTİRME ARACI

## ÖZ

Bu çalışmanın amacı üç boyutlu animasyon film geliştirme konusunda tecrübe edinilmesi, üç boyutlu film geliştirme sürecinin disipline edilmesi ve örnek bir üç boyutlu animasyon film geliştirme aracının geliştirilmesidir.

Şu an ki üç boyutlu animasyon film sanayisinde kullanılan bilgisayarlar donanımsal grafik motorlarına sahip bu amaç için tasarlanmışlardır. Sıradışı olarak, kullanılan donanımsal altyapı kullanılan yazılımlara özel olarak üretilmiştir. Bu bilgisayarların bir çoğu Sun Systems üzerinde çalışan milyon dolarlık edlere sahiptir. Oluşturma süreci kontrol altına alınmadıkça bu gereksinimlerin kişisel bilgisayarlar seviyesine indirilebilmesi olası değildir.

Üç boyutlu animasyon film geliştirme süreci SDLC izlenerek yönetilebilir. Buna karşın geliştirme adımı kullanıcının detaylara takılmadan büyük resim ile ilgilenebilmesine izin veren, görsellik ihtiyaçlarına soyutlama yapabilecek, gömülü bir grafik motoruna sahip güçlü bir entegre geliştirme ortamına (IDE) gereksinim duymaktadır. Bu projedeki geliştirme ortamı “Weendigo” kod adıyla anılmaktadır. Weendigo, kıvırlı derili inanışına göre bir insanın başka bir insanı yediğinde onun yeteneklerine, cesaretine, gücüne ve kapasitesine sahip olması durumudur. Pneuma ise insanları diğer canlılardan daha değerli yapan ruh anlamındadır. Weendigo bu projedeki entegre geliştirme ortamının, Pneuma ise grafik motorunun ismidir. Weendigo bileşen temelli mimarisiyle belirli arayüzleri destekleyen bileşenlerin çalışma zamanında eklenip çıkartılmasına izin vermektedir. Weendigo bu bileşenlerin yeteneklerine sahip olurken, Pneuma bu bileşenlere can vererek yeteneklerin açığa çıkartır. Ek olarak bu özellik, farklı rollerdeki kişilere farklı bakış açıları sunarak sürecin başından sonuna kontrol altında olmasını sağlamaktadır.

**Anahtar Sözcükler:** Üç boyut, Üç boyutlu animasyon film, Film oluşturma, IDE, Weendigo, Grafik Motoru, Pneuma, Microsoft .NET Framework, Microsoft DirectX

## CONTENTS

M.Sc THESIS EXAMINATION RESULT FORM.....	ii
ACKNOWLEDGEMENTS.....	iii
ABSTRACT.....	iv
ÖZ.....	v
<b>CHAPTER ONE INTRODUCTION.....</b>	<b>1</b>
<b>CHAPTER TWO PREVIOUS WORK .....</b>	<b>5</b>
<b>CHAPTER THREE MATERIALS AND METHODS .....</b>	<b>8</b>
3. 1 Game Development Process at a Glance .....	8
3.2 Microsoft DirectX (Graphics API).....	11
<b>CHAPTER FOUR WEENDIGO .....</b>	<b>17</b>
4.1 Weendigo Startup Page.....	17
4.2 Weendigo Solution Explorer.....	19
4.3 Weendigo Toolbox .....	26
4.4 Property Window.....	35
4.5 Timeline Track Bar.....	41
4.6 Resource Management.....	43
4.7 Weendigo Docking Library.....	48
4.8 Dialog Management.....	58

<b>CHAPTER FIVE SCENARIO MANAGEMENT .....</b>	<b>60</b>
5.1 Scenario (Game) Engine .....	60
5.2 Scene Design User Interface .....	64
5.3 Scene Compilation.....	71
5.4 Scenario Manager .....	76
 <b>CHAPTER SIX PNEUMA.....</b>	 <b>86</b>
6.1 Pneuma Design.....	86
6.2 Hardware Enumeration .....	95
6.3 Common Controls in 3D Environment.....	102
6.4 Darken Scene Algorithm.....	115
6.5 Rendering Static Meshes.....	118
6.6 Rendering Animated Meshes .....	125
6.7 Rendering Text .....	135
6.8 Scene Background .....	140
6.9 Camera Usage .....	143



<b>CHAPTER SEVEN POST IMPLEMENTATION REVIEW .....</b>	<b>150</b>
7.1 Cross Threading Issue.....	150
7.1.1 Cross-threading Vulnerability .....	150
7.2 Dynamic Code Injection .....	160
7.2.1 Why Weendigo Needs On-the-fly Code Generation? .....	162
7.2.2 What is Code Dom? .....	165
7.2.3 Weendigo On-the-fly Code Generation .....	166
7.3 Exception Handling .....	169
7.3.1 ADPlus .....	172
7.3.2 Application Verifier .....	174
7.3.3 Dr. Watson.....	174
7.3.4 WinDbg .....	175
7.4 Performance Issues .....	176
 <b>CHAPTER EIGHT</b>	
 <b>CONCLUSION.....</b>	<b>180</b>
 <b>REFERENCES.....</b>	<b>182</b>
 <b>APPENDICES .....</b>	<b>183</b>

## **CHAPTER ONE**

### **INTRODUCTION**

Weendigo is a Red Indian belief on migration of the human beings' capabilities, courage, power, and skills when he was eaten by another human being. Pneuma is the vital spirit which makes men the most valuable creature in the world. By the way, Weendigo refers to development environment and Pneuma is the name of the underlying graphics engine. Weendigo has a component based architecture that allows add or remove components at runtime by implementing some interfaces. Therefore Weendigo gathers all skills and capabilities from the components added and Pneuma let them born. From now on, Weendigo will refer for development environment and Pneuma will refer for graphics engine.

Weendigo is a complete set of design tools for building animation films. Visual Basic, Visual C++, Visual C#, Visual J# and including all other languages supported by .NET Framework all use the same integrated design environment, which allows them to share facilitates in the creation of a film. Weendigo consist more than 120.000 lines of code. Supporting Microsoft .NET technologies is an important feature also this is why it has a similar interface with Microsoft Visual Studio .NET 2005 development environment. It is not possible to write codes on Weendigo. But weendigo also has a built in compiler. It only compiles project and does not produce any output until user decides to preview video or produce video. This compilation just warns user for specific design time errors that might cause run time exceptions.

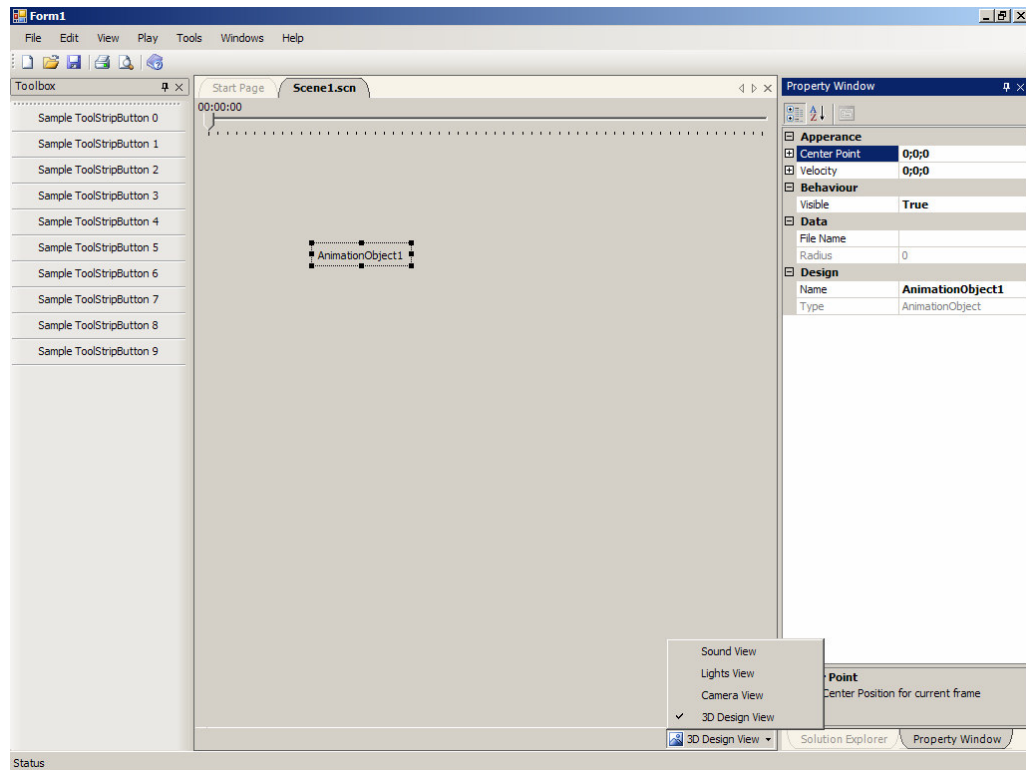


Figure 1.1 Weendigo has a user interface similar with Visual Studio .NET 2005

As you have seen the figure above it is very similar to Visual Studio .NET 2005 development environment. To efficiently manage the items that are required by your development effort, such as components and scenes and media files, Weendigo has a built in container named as Weendigo Solution. Solution Explorer is provided as part of the Weendigo Integrated Development Environment (IDE).

There is a toolbox like Visual Studio .NET style, which allows you add/remove components supported by Weendigo. It is possible to add items from this toolbox to a scene by using drag and drop. By right clicking this toolbox, you will be able to enumerate GAC (Global Assembly Cache), COM+ (Component Object Model) components, and specify a custom .NET component to use your weendigo project.

There is a property grid like Visual Studio .NET style, which allows you to set properties of objects at design time. When you click an object all of the properties of this object are enumerated on this grid. Some of these properties might be marked as

read-only, weendigo automatically senses these properties and shows these properties in a state that cannot be altered.

In all scenes, there is a duration bar which is an essential need for animation films. Scenes duration must be defined during design time. All changes made on scenes at specific times are recorded by weendigo in XML form and being ready to use for scenario manager at the time of playing movie.

All of the containers included in Weendigo are dockable containers. Docking, resizing and repositioning of these containers are designated for ease to use.

By using drag and drop, you can prepare a scene for playing. After all it becomes a big problem to find out any design time errors. Current development environments, including Microsoft Visual Studio .NET 2005, the source files are compiled but the design time errors are not handled by compilers. Design time errors are causes exceptions during run time. It is not an acceptable situation, if you are producing a 3D animation films. It might take hours to find out what is wrong. To overcome this issue, weendigo introduces a new style of compilation. Weendigo compilation performs compilation on each design time object separately. Thus provides an efficient approach on producing films. A sample snapshot of compilation process is given below.

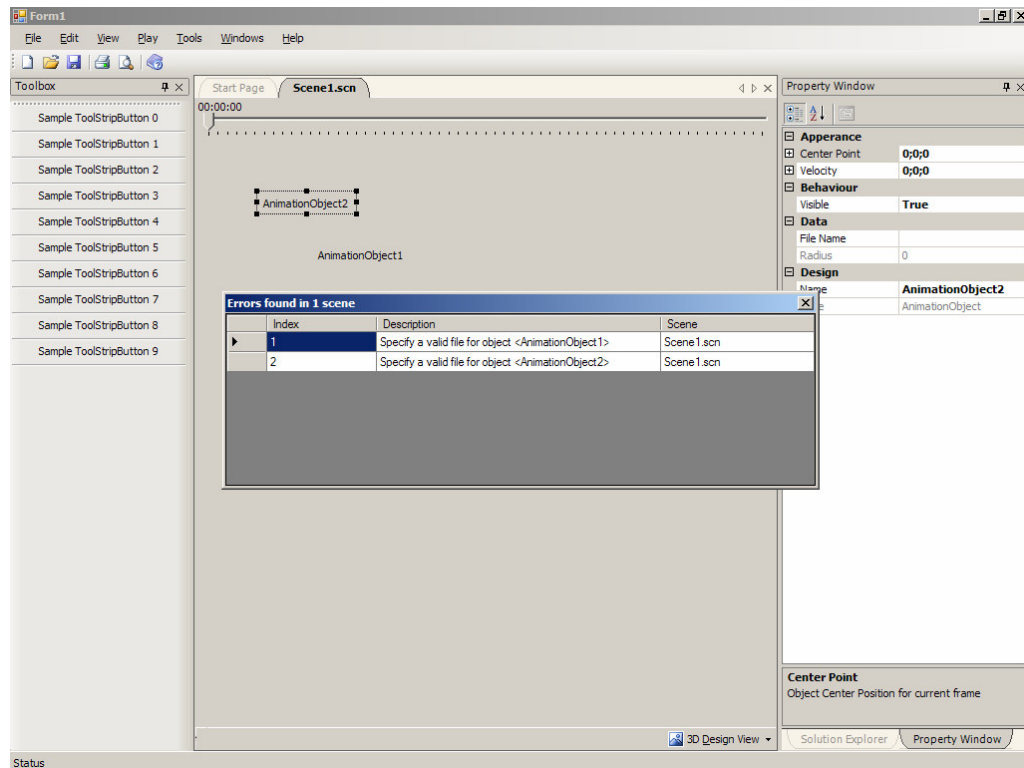


Figure 1.2 Weendigo performs a compilation on prepared scene to check design time error which might cause a run time exception

In this scene, weendigo design time compiler complains that no valid file is specified for Animation Object in “scene1.scn”. There are two different objects in our scene but definition of a valid animation object file for these instances was forgotten.

## **CHAPTER TWO**

### **PREVIOUS WORK**

In film, the term 3-D (or 3D) is used to describe any visual presentation system that attempts to maintain or recreate moving images of the third dimension, the illusion of depth as seen by the viewer.

The principle involves taking two images simultaneously, with two cameras positioned side by side, generally facing each other and filming at a 90 degree angle via mirrors, in perfect synchronization and with identical technical characteristics. When viewed in such a way that each eye sees its photographed counterpart, the viewer's visual cortex will interpret the pair of images as a single three-dimensional image.

The stereoscopic era of motion pictures begins in the late 1890s when British film pioneer William Friese-Greene files a patent for a 3-D movie process. In his patent, two films are projected side by side on screen. The viewer looked through a stereoscope to converge the two images. Because of the obtrusive mechanics behind this method, theatrical use was not practical. Frederic Eugene Ives patented his stereo camera rig in 1900. The camera had two lenses coupled together 1 3/4 inches apart. In 1903, 3-D films were shown at the Paris Exposition under the auspices of the Lumiere Brothers. While it is unconfirmed, the footage may have been a remake of their film *L'Arrivée du Train*. Regardless, they later filmed this footage stereoscopically in the late 1930s.

After computers becoming an important part of our lives, cartoon and film industry began to replace their old fashioned technologies with new ones which are computer aided software. There are still some films support 3D views. However, this idea brought out 3D animation films. The very first totally computer generated animation movie was *Toy Story*.

A computer-animated film commonly refers to feature films that have been computer-animated to appear three dimensional on a movie screen. While traditional 2D animated films are now done primarily on computers, the technique to render realistic 3D computer graphics (CG), or 3D Computer-generated imagery (CGI), is unique to using computers to create movies. Here is a list of well known animation films chronological by release date:

- 1995
  - Toy Story
- 1998
  - Antz
  - A Bug's Life
- 1999
  - Toy Story 2
- 2000
  - Dinosaur
- 2001
  - Shrek
  - Final Fantasy: The Spirits Within
  - Monsters, Inc.
  - Jimmy Neutron: Boy Genius
- 2002
  - Ice Age
  - Jonah: A VeggieTales Movie
- 2003
  - Finding Nemo
- 2004
  - Ark
  - Homeland (film)
  - Shrek 2
  - Shark Tale
  - Terkel in Trouble (Denmark, "Terkel i knibe")
  - The Incredibles

As seen in the list, number of produced film by release date is increasing year by year. A full list of animation films is given in appendices, see also upcoming films by year. Most of the films given above are generated on Sun Systems Graphics Servers using Pixart's software. Uncommonly, underlying hardware is produced specifically for this software. It is not possible to get a demo version or allow these films to be generated using a personal computer.

There are several tools to prepare video using a 3D scene. 3D Studio Max is capable to produce a video by playing an animation according to key frames. But this feature in 3D Studio Max is embedded and not available for a team work. Also, scene merging and some other important features are not enabled. Weendigo serves a complete solution from beginning to end in a 3d animation film. 3D Studio Max solution is limited with rendering arbitrary scenes not intended to prepare a 3D film.



## **CHAPTER THREE**

### **MATERIALS AND METHODS**

Game development and animation film development processes are familiar with each but differs in some key points. Creation of a 3D animation film process can be considered as a software development project. By the way, there are different roles as this process contains art in addition to computer science. This chapter includes game development process, and technologies used in this project.

#### **3. 1 Game Development Process at a Glance**

A computer game is a computer-controlled game. A video game is a computer game where a video display such as a monitor or television is the primary feedback device. The term "computer game" also includes games which display only text (and which can therefore theoretically be played on a teletypewriter) or which use other methods, such as sound or vibration, as their primary feedback device, but there are very few new games in these categories. There always must also be some sort of input device, usually in the form of button/joystick combinations (on arcade games), a keyboard & mouse/trackball combination (computer games), or a controller (console games), or a combination of any of the above. Also, more esoteric devices have been used for input. Usually there are rules and goals, but in more open-ended games the player may be free to do whatever they like within the confines of the virtual universe.

In common usage, a "computer game" or a "PC game" refers to a game that is played on a personal computer. "Console game" refers to one that is played on a device specifically designed for the use of such, while interfacing with a standard television set. "Video game" (or "videogame") has evolved into a catchall phrase that encompasses the aforementioned along with any game made for any other device, including, but not limited to, mobile phones, PDAs, advanced calculators, etc.

Development of computer and video games is undertaken by a developer, which may be a single person or a business. Typically, large-scale commercial games are developed by development teams within a company specializing in computer or video games. A typical modern video or computer game costs from \$1 million up to \$15 million to develop. Development is normally funded by a publisher. A contemporary game can take from one to three years to develop, though there are exceptions.

While in the early era of home computers and video game consoles in the early 1980s, a single programmer could handle almost all the tasks of developing a game, the development of modern commercial video games involves a wide variety of skill-sets and support staff. As a result, entire teams are often required to work on a single project.

Game development process can be easily considered a software project management process. As SDLC (Software Development Life Cycle) process suggests following steps exist in a software project:

- **Project Definition (Initial Request):** project is defined by non IT people generally by the sponsor of the project.
- **Requirement Specification:** Project Manager forms a cross functional project team to define the detail project scope including the technical solution and alternatives
- **Functional Analysis:** Project Manager conducts meetings with appropriate project stakeholders to review, modify and/or approve the requirements document
- **Project Planning:** Project Manager leads the project team to develop a detail and comprehensive project plan based on the preliminary project plan and requirements document
- **Application Specification:** application specification including user interface design and business logic controls is defined by business analysts.
- **Technical Analysis:** technical details and technical constraints are defined by a lead developer.
- **Development:** project team executes the project plan, implements demanded application in order to predefined application specification and obeys the preceding technical analysis design.

- **System Test:** functionality, unit and integration tests including end to end tests are performed. Usually this is duty of business analyst who has prepared application specifications.
- **User Acceptance Test:** end user tests are performed to finalize project. Usually this is duty of project owner. This includes providing project plan status to project stakeholders and obtaining customer approval.
- **Deployment:** the project team implements the project into production environment.
- **Post Implementation Support:** This includes managing changes, taking corrective action and user education.
- **Post Implementation Review:** this is a measurement of project success.

A typical present-day game development team usually includes:

- One or more producers to oversee production
- At least one game designer
- Artists
- Programmers
- Level designers
- Sound engineers (composers, and for sound effects)
- Testers

Some members of the team may handle more than one role. For example, the producer may also be the designer, or the lead programmer may also be the producer. However, while common in the early video game era, this is increasingly more uncommon now for professional games. Also these roles can easily be mapped to roles in SDLC process.

3D computer graphics are works of graphic art that were created with the aid of digital computers and specialized 3D software. In general, the term may also refer to the process of creating such graphics, or the field of study of 3D computer graphic techniques and its related technology.

3D computer graphics are different from 2D computer graphics in that a three-dimensional representation of geometric data is stored in the computer for the purposes of performing calculations and rendering 2D images. Sometimes these images are later displayed in a pre-rendered form, and sometimes they are rendered in real-time.

### 3.2 Microsoft DirectX (Graphics API)

Microsoft DirectX is a set of low-level application programming interfaces (APIs) for creating games and other high-performance multimedia applications. It includes support for high-performance 2-D and 3-D graphics, sound, and input. In a perspective of creating a 3D animation film requires 3D graphics and 3D sound ability. Weendigo includes a 3D framework based on Microsoft DirectX technologies. Weendigo film generation only includes 3D Graphics facilities.

Microsoft Direct3D is a low-level graphics application programming interface (API) that enables you to manipulate visual models of 3-dimensional objects and take advantage of hardware acceleration, such as video graphics cards.

The graphics pipeline provides the horsepower to efficiently process and render Direct3D scenes to a display, taking advantage of available hardware. This figure conceptually illustrates the building blocks of the pipeline:

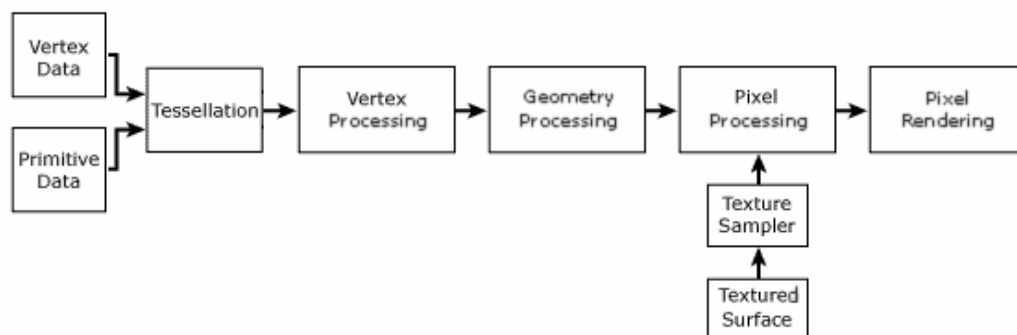


Figure 3.1 Microsoft Direct 3D graphics pipeline

Here is a brief description of each block:

- **Vertex Data**, Untransformed model vertices are stored in vertex memory buffers.
- **Primitive Data**, Geometric primitives, including points, lines, triangles, and polygons, are referenced in the vertex data with index buffers.
- **Tessellation**, The tessellator unit converts higher-order primitives, displacement maps, and mesh patches to vertex locations and stores those locations in vertex buffers.
- **Vertex Processing**, Direct3D transformations are applied to vertices stored in the vertex buffer.
- **Geometry Processing**, Clipping, back face culling, attribute evaluation, and rasterization are applied to the transformed vertices.
- **Textured Surface**, Texture coordinates for Direct3D surfaces are supplied to Direct3D through the IDirect3DTexture9 interface.
- **Texture Sampler**, Texture level-of-detail filtering is applied to input texture values.
- **Pixel Processing**, Pixel shader operations use geometry data to modify input vertex and texture data, yielding output pixel color values.
- **Pixel Rendering**, Final rendering processes modify pixel color values with alpha, depth, or stencil testing, or by applying alpha blending or fog. All resulting pixel values are presented to the output display.

This implementation details are hidden from a 3D animated film designer in Weendigo. Vertex and/or primitive data are provided in Microsoft X Files. Transformation texture mapping are performed by Weendigo automatically. Other hardware details are performed by Microsoft Direct3D libraries.

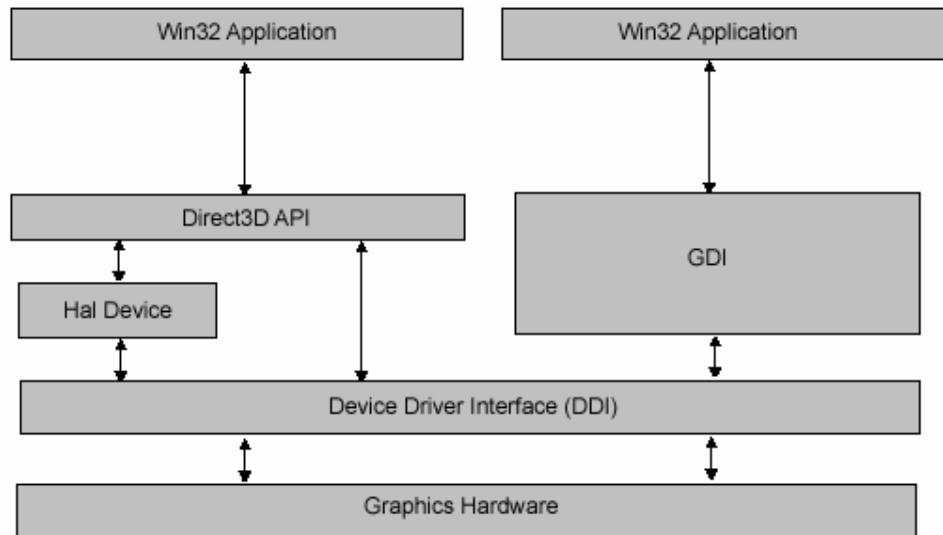


Figure 3.2 Shows the relationships between a Window application, Direct3D,GDI, and the hardware

This figure is taken from Microsoft DirectX 9.0 SDK documentation library. This figure shows only what Microsoft DirectX does. However, Weendigo graphics engine called “Pneuma” performs too many things based upon these layers.

Direct3D exposes a device-independent interface to an application. Direct3D applications can exist alongside GDI applications, and both have access to the computer's graphics hardware through the device driver for the graphics card. Unlike GDI, Direct3D can take advantage of hardware features by creating a HAL (Hardware Acceleration Layer) device.

A HAL device provides hardware acceleration to graphics pipeline functions, based upon the feature set supported by the graphics card. Direct3D methods are provided to retrieve device display capabilities at run time.

The primary device type is the HAL device, which supports hardware accelerated rasterization and both hardware and software vertex processing. If the computer on which your application is running is equipped with a display adapter that supports Direct3D, your application should use it for Direct3D operations. Direct3D HAL

devices implement all or part of the transformation, lighting, and rasterizing modules in hardware.

Applications do not access graphics adapters directly. They call Direct3D functions and methods. Direct3D accesses the hardware through the HAL. If the computer that your application is running on supports the HAL, it will gain the best performance by using a HAL device.

Direct3D supports an additional device type called a reference device or reference rasterizer. Unlike a software device, the reference rasterizer supports every Direct3D feature. Because these features are implemented for accuracy rather than speed and are implemented in software, the results are not very fast. The reference rasterizer does make use of special CPU instructions whenever it can, but it is not intended for retail applications. Use the reference rasterizer only for feature testing or demonstration purposes.

Weendigo supports both HAL and REF devices and allow users to switch between them by clicking a button or pressing F3 button.

Typically 3D graphics applications use two types of Cartesian coordinate systems: left-handed and right-handed. In both coordinate systems, the positive x-axis points to the right and the positive y-axis points up. You can remember which direction the positive z-axis points by pointing the fingers of either your left or right hand in the positive x-direction and curling them into the positive y-direction. The direction your thumb points, either toward or away from you, is the direction that the positive z-axis points for that coordinate system. The following illustration shows these two coordinate systems.

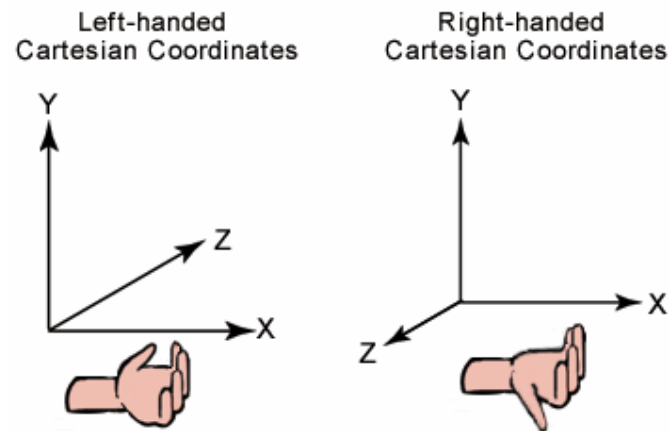


Figure 3.3 There are two distinct cartesian coordinates: Left-handed Cartesian Coordinates and Right-handed Cartesian Coordinates. By the way, Weendigo only support Left-handed Cartesian Coordinates which is commonly preferred.

Animated Objects have different approaches on calculating transforms. Transformation of World, View and Projection matrices and using camera options will be documented in another article. Direct3D uses the world and view matrices that you set to configure several internal data structures. But important point is setting these matrices are time consuming. Weendigo has an implementation of setting these matrix transformations in an optimized way. Each time you set a new world or view matrix, the system recalculates the associated internal structures. Setting these matrices frequently-for example, thousands of times per frame-is computationally time-consuming.

Lights are used to illuminate objects in a scene. When lighting is enabled, Direct3D calculates the color of each object vertex based on a combination of:

- The current material color and the texels in an associated texture map.
- The diffuse and specular colors at the vertex, if specified.
- The color and intensity of light produced by light sources in the scene or the scene's ambient light level.

When you use Direct3D lighting and materials, you allow Direct3D to handle the details of illumination for you.



How you work with lighting and materials makes a big difference in the appearance of the rendered scene. Materials define how light reflects off a surface. Direct light and ambient light levels define the light that is reflected. You must use materials to render a scene if lighting is enabled. Lights are not required to render a scene, but details in a scene rendered without light are not visible. At best, rendering an unlit scene results in a silhouette of the objects in the scene. This is not enough detail for most purposes. Weendigo currently has a view to designate the lights in scene. But this facility is not implemented, yet. Currently, Weendigo uses only one light at a scene. Further versions should allow user administer lights on scene at each frame.

With DirectX 9.0, developers can take advantage of DirectX multimedia functionality and hardware acceleration while using managed code. DirectX 9.0 for Managed Code enables access to most of the original unmanaged DirectX functionality. Using managed distribution of Microsoft DirectX, provides an abstraction over programming language and memory management facilities underlying operating system.

As a conclusion, choosing Microsoft DirectX 9 and managed extension in Weendigo implementation involved the following advantages:

- Platform Abstraction
- Hardware Abstraction
- Operating System Abstraction
- Device Driver Abstraction
- Graphics Pipeline Abstraction
- Programming Language Abstraction

These advantages add worth Weendigo to have as a built in facility. Therefore, Weendigo has the facilities given above which competitor does not. Most of the competitors use technologies and hardware specific for their implementation.

## **CHAPTER FOUR**

### **WEENDIGO**

Weendigo is a robust development environment to create 3D animation films. In a programming perspective Weendigo is a Win32 application. By the definition, In developement process, there were same problems with any other Win32 application. Also, there are some advantages of implementing Weendigo as an Win32 applicaiton. For instance, entire user interface is being prepared by using visual inheritance.

#### **4.1 Weendigo Startup Page**

Hyper Text Markup Language (HTML) is a markup language designed for the creation of web pages with hypertext and other information to be displayed in a web browser. Preparing rich text content in a windows application requires too much effort. For a rapid application approach, embedding HTML content in a windows application is a commonly used trick. Weendigo start page is implemented using this approach.

In Microsoft .NET Framework 2.0, “System.Windows.Forms.WebBrowser” control exist in class library. This control lets you host Web pages and other browser-enabled documents in your Windows Forms applications. You can use the WebBrowser control, for example, to provide integrated HTML-based user assistance or Web browsing capabilities in your application. Additionally, you can use the WebBrowser control to add your existing Web-based controls to your Windows Forms client applications.

In Weendigo implementation, WebBrowserWindow inherits from WebBrowser control class. This class automatically load an HTML file. StartPageWindow implements BasePage which is a requirement for hosting in Weendigo IDE. There is composition relation between StartPageWindow and WebBrowserWindow. In

StartPageWindow constructor prepares and initializes WebBrowserWindow control. Constructor implementation is given below:

```
public StartPageWindow(string pFullPath,DockingContainerWindow pParent)
    : base(pParent)
{
    fullPath = pFullPath;
    if (fullPath.Length>0)
    {
        wWindow= new WebBrowserWindow(pFullPath);
    }
    else
    {
        wWindow = new WebBrowserWindow();
    }
    wWindow.IsWebBrowserContextMenuEnabled = false;
    wWindow.Dock = System.Windows.Forms.DockStyle.Fill;
    wWindow.ScrollBarsEnabled = false;
    this.Controls.Add(wWindow);
}
```

Both of these classes' implementation details are given below for clarity:

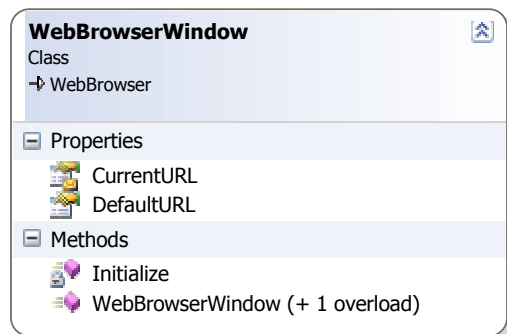


Figure 4.1 Web Browser Window class interface

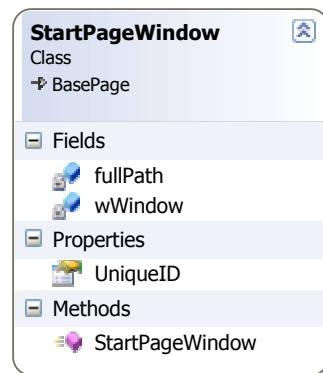


Figure 4.2 Start Page Window class interface

It is possible to embed some menu items functionality in this page like enumerating recent projects, opening and loading solutions. Only handling link click events would be enough. A sample snapshot of start page is given below.

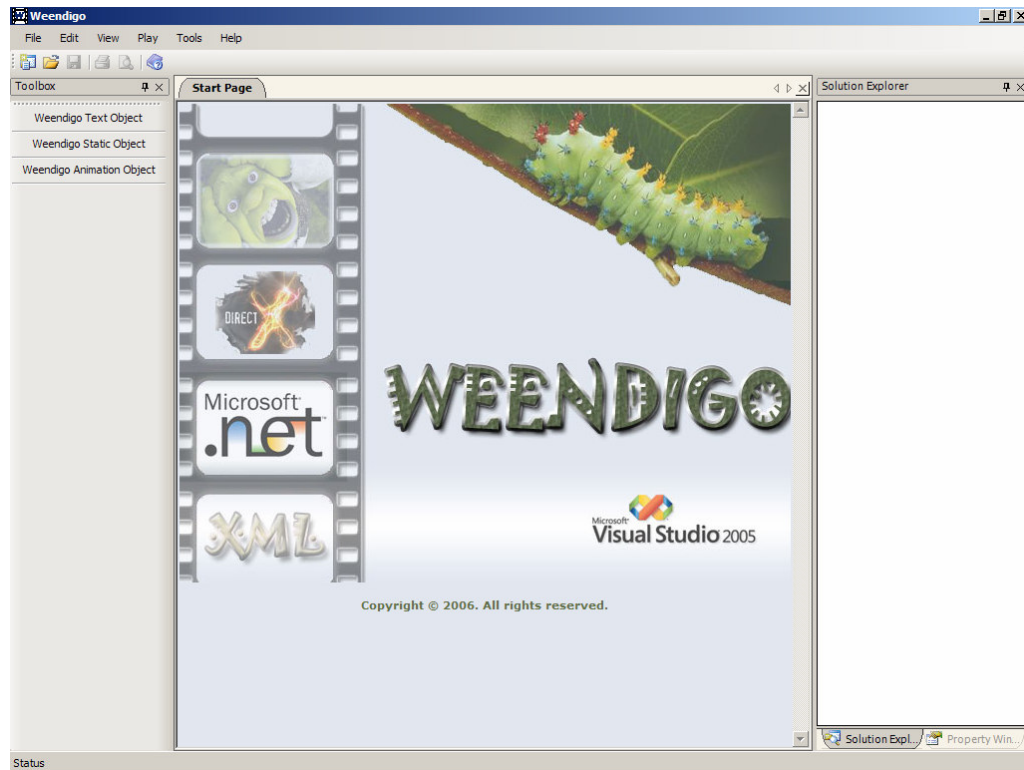


Figure 4.3 A sample snapshot of startup page. Also it is possible to customize this user interface by only modifying HTML source code

## 4.2 Weendigo Solution Explorer

One of the common needs of an Integrated Development Environment is organizing your files and objects. Weendigo Solution Explorer provides you with an organized view of your project files (scenes) as well as ready access to the commands that pertain to them. Weendigo Solution Explorer has a standard view represents the active solution as a logical container for one 3D movie project and scenes associated with them. You can open project items for modification and perform other management tasks directly from this view.

Weendigo solutions include scenes, scenes include scene objects. Weendigo Solution Explorer represents only scenes. Scene objects are represented when you open related scene. Weendigo Solution Explorer is a tree view designated to show currently loaded scene. A set of functions (including opening, closing solution files, and etc.) are exposed to integrate Weendigo Solution Explorer with Weendigo IDE.

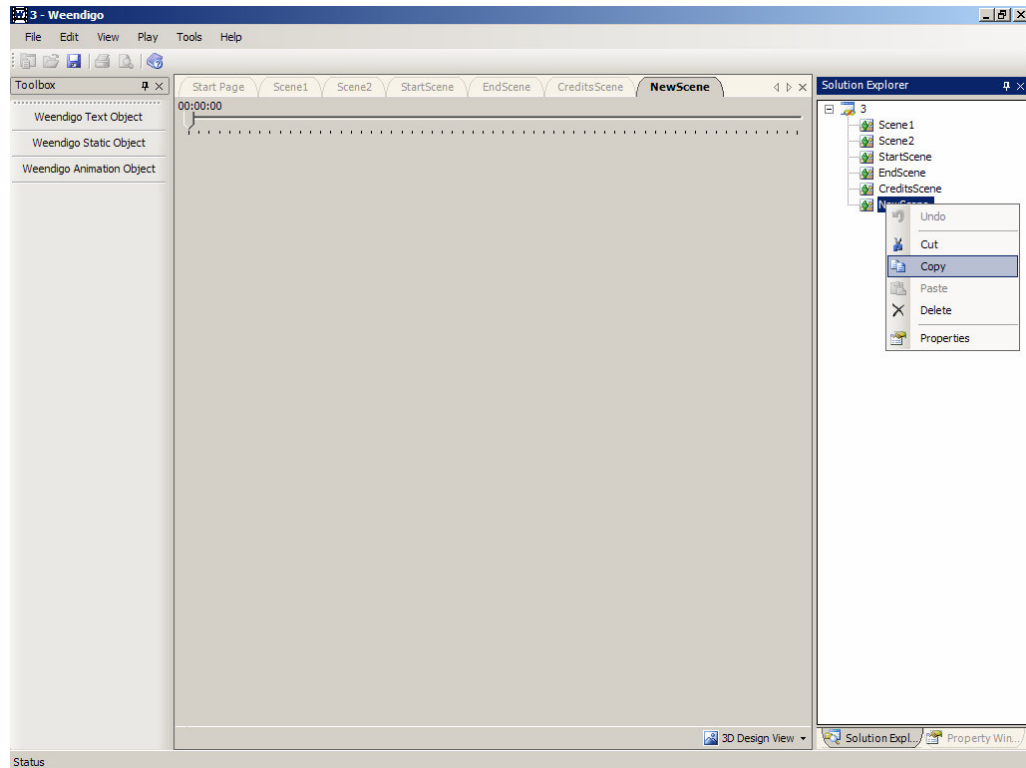


Figure 4.4 Weendigo Solution Explorer has a similar properties with Visual Studio .NET 2005

Weendigo Solution Explorer is built up using three different classes:

- SolutionExplorer
- SolutionTreeNode
- SolutionConfiguration

Solution Explorer class inherits “TreeView” class and intended to be a communication point between Solution and Weendigo IDE. Weendigo IDE can only have a single instance of Weendigo Solution Explorer. Nevertheless, Weendigo Solution Explorer is not marked as a sealed class by design.

SolutionTreeNode class inherits “TreeNode” class and used by Solution Explorer. All of the items (including solution node itself) in tree view are constructed using this class. This class is not publicly exposed by Weendigo IDE.

SolutionConfiguration class inherits “ConfigurationBase” class which is provided by Weendigo IDE for use of any purpose configurations. A solution configuration is matched with a single weendigo solution file.

Weendigo Solution Explorer serves the following facilities:

- Creating a new solution is accomplished by calling “InitiateOnFile” method.
- Loading a solution file is accomplished by calling this class’s “ConstructFromFile” method.
- Closing a loaded solution is accomplished by calling “UnloadSolution” method.
- Adding a new item is accomplished by calling “AddNewSolutionItem” method.
- Removing an existing item is accomplished by calling “RemoveSolutionItem” method.
- Saving a solution is accomplished by calling “StoreFile” method.

Other facilities exposed by Weendigo Solution Explorer are followings:

- When user double clicks on a solution item, SolutionItemSelected event is fired thus causes Weendigo IDE to load selected solution item. If selected item is already loaded, Weendigo IDE automatically brings front selected solution item.
- When user left clicks on a solution item, SolutionItemLeftClick event is fired. This is an informative event for Weendigo IDE. Left clicking on a solution item once enables user to edit selected scene name.

- When user right clicks on a solution item, `SolutionItemRightClick` event is fired thus forces Weendigo IDE to show a popup menu for selected solution item.
- When a solution item is renamed by left clicking one time on a solution item or using F2 shortcut key, Weendigo Solution Explorer fires a `SolutionItemRenamed` event.
- When a solution item is deleted by choosing delete from popup or using Del shortcut key, Weendigo Solution Explorer fires a `SolutionItemRemoved` event after removing selected item.
- When user right clicks on solution (root node), `SolutionRightClick` event is fired thus forces Weendigo Solution Explorer to show a solution specific popup menu.

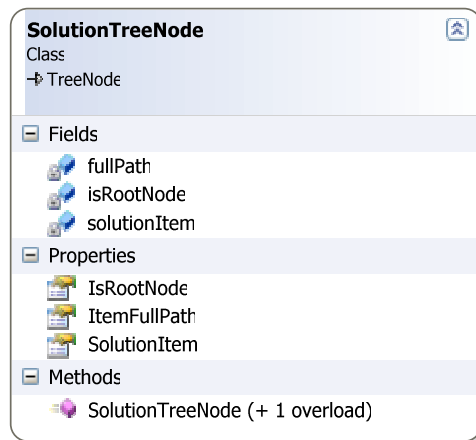


Figure 4.5 Solution Tree Node class interface

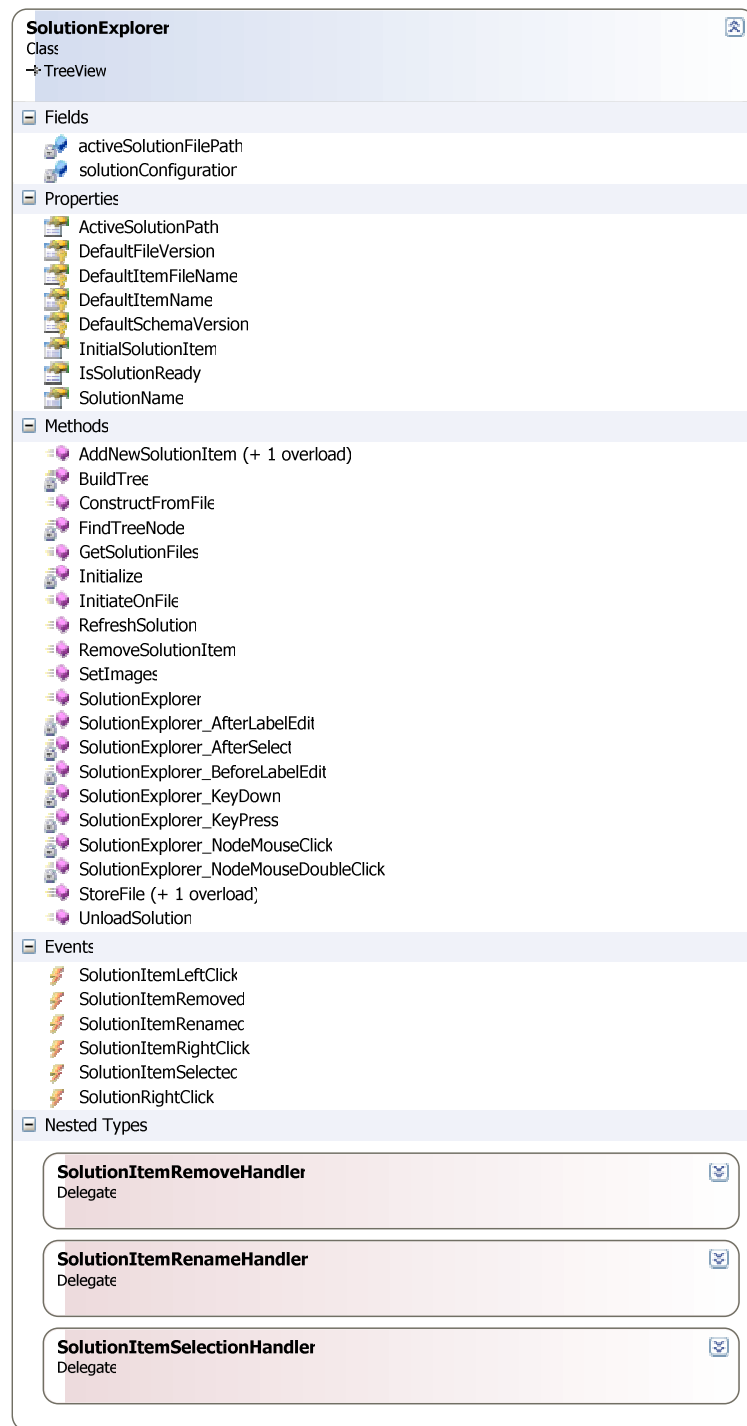


Figure 4.6 Solution Explorer class interface



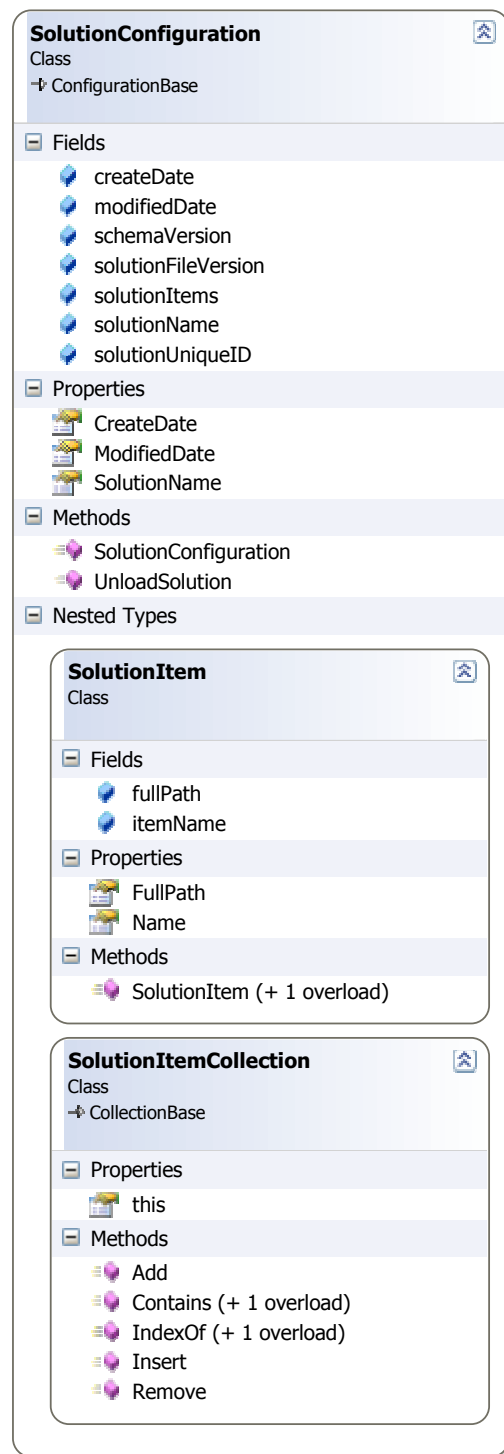


Figure 4.7 Solution Configuration class interface

As shown in diagram, SolutionConfiguration includes two internal class declarations for solution items (SolutionItem) and solution item list

(SolutionItemCollection). This class is a serializable class to map between physical solution file and Weendigo Solution Explorer.

Weendigo solution files contain SolutionConfiguration instance in XML serialized form like following:

```
<?xml version="1.0"?>
<SolutionConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <FileVersion>1.0</FileVersion>
  <SchemaVersion>1.0</SchemaVersion>
  <SolutionGUID>1e6c1c64-debf-47c4-aa82-59eb3f359e61</SolutionGUID>
  <SolutionName>3</SolutionName>
  <CreateDate>2006-04-19T17:31:01.65625+03:00</CreateDate>
  <ModifiedDate>2006-05-14T15:18:36.359375+03:00</ModifiedDate>
  <Contents>
    <SolutionItem Name="Scene1" Path="D:\Workspace\documents\test1\3\Scene1.scn" />
    <SolutionItem Name="Scene2" Path="D:\Workspace\documents\test1\3\Scene2.scn" />
    <SolutionItem Name="StartScene" Path="D:\Workspace\documents\test1\3\StartScene.scn" />
    <SolutionItem Name="EndScene" Path="D:\Workspace\documents\test1\3\EndScene.scn" />
    <SolutionItem Name="CreditsScene"
Path="D:\Workspace\documents\test1\3\CreditsScene.scn" />
    <SolutionItem Name="NewScene" Path="D:\Workspace\documents\test1\3\NewScene.scn" />
  </Contents>
</SolutionConfiguration>
```

XML Schema definition is given below:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified"
elementFormDefault="qualified">
  <xs:element name="SolutionConfiguration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FileVersion" type="xs:decimal" />
```

```

<xs:element name="SchemaVersion" type="xs:decimal" />
<xs:element name="SolutionGUID" type="xs:ID" />
<xs:element name="SolutionName" type="xs:string" />
<xs:element name="CreateDate" type="xs:dateTime" />
<xs:element name="ModifiedDate" type="xs:dateTime" />
<xs:element name="Contents">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="SolutionItem">
        <xs:complexType>
          <xs:attribute name="Name" type="xs:string" use="required" />
          <xs:attribute name="Path" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

Weendigo Solution Explorer has an extensible interface which will allow further changes. Implementation details are hidden from Weendigo IDE. This approach has important advantages. For instance, adding support for multiple solution files (similar to Microsoft Visual Studio .NET 2005 Project files) is mitigated. For further improvements, Microsoft Visual Source Safe integration might be added to have source control management features.

### 4.3 Weendigo Toolbox

The Toolbox displays pre-registered items for use in Weendigo projects. There are several built in components including Weendigo Static Object, Weendigo Animation Object, and Weendigo Text Object available for use. Also, it is possible to add or remove these items at runtime. The items available can include .NET components, and COM components that support a common interface and marked with attribute

named as “ToolboxDescriptorAttribute”. All components listed in toolbox can easily be added your scene by a simple drag and drop operation. Weendigo IDE hosts Toolbox itself. But implementation details are hidden from Weendigo IDE.

Weendigo “ToolboxControl” is a class derived from ToolStrip control. Each item listed in toolbox is represented with “ToolboxItem” class which is derived from ToolStripButton control.

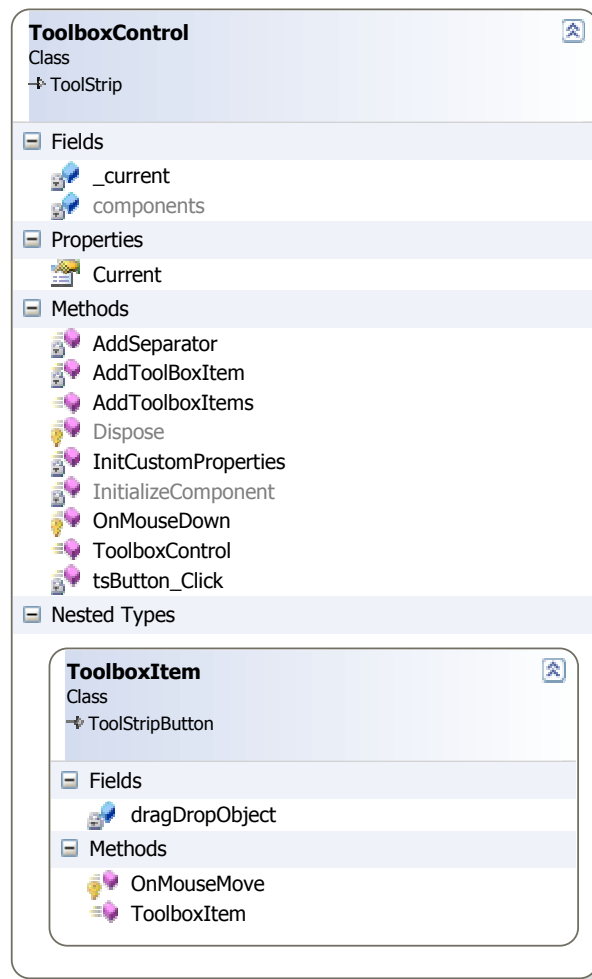


Figure 2.8 Toolbox Control class interface

Each **ToolboxItem** has a `dragDropObject` which encapsulates drag and drop object inside. When user clicks a toolbox and moves it to scene in design view, selected

toolbox item adds this encapsulated drag and drop object to clipboard. It is Weendigo IDE responsibility to handle drag and drop events.

ToolboxControl is implemented with singleton design pattern and can be accessible via Current static property. Having multiple instances of toolbox control is not a desired expectation. This is why this class is implemented with singleton design pattern. Toolbox control is a dockable control embedded in Weendigo IDE.

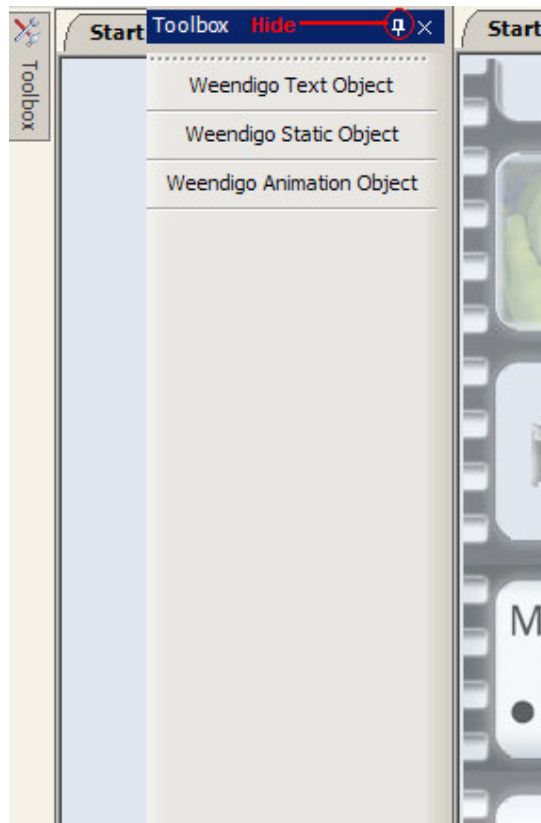


Figure 4.9 Weendigo Toolbox control enumerates registered controls available to use. Also hiding this toolbox control is possible by clicking pin icon

Drag and drop object simply has a reference for BaseDisplayObject which is the base interface for Pnema architecture. Also, “DragDropEncapsulationObject” is used by drag and drop events inside 3D scene design view in Weendigo.

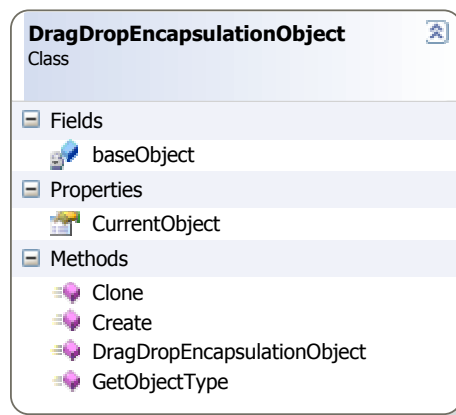


Figure 4.10 Drag Drop Encapsulation Object class interface. Drag and Drop operations encapsulates any instance of object by using this class

When user right clicks on toolbox, a context menu is shown to allow user to choose toolbox items:

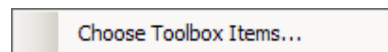


Figure 2.11 Choose Toolbox Items context menu

After clicking “Choose Toolbox Items...” a dialog box is shown which includes a list of assemblies in Global Assembly Cache (GAC). Each computer where the common language runtime is installed has a machine-wide code cache called the global assembly cache. The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer. Assemblies deployed in the global assembly cache must have a strong name. When an assembly is added to the global assembly cache, integrity checks are performed on all files that make up the assembly. The cache performs these integrity checks to ensure that an assembly has not been tampered with, for example, when a file has changed but the manifest does not reflect the change.

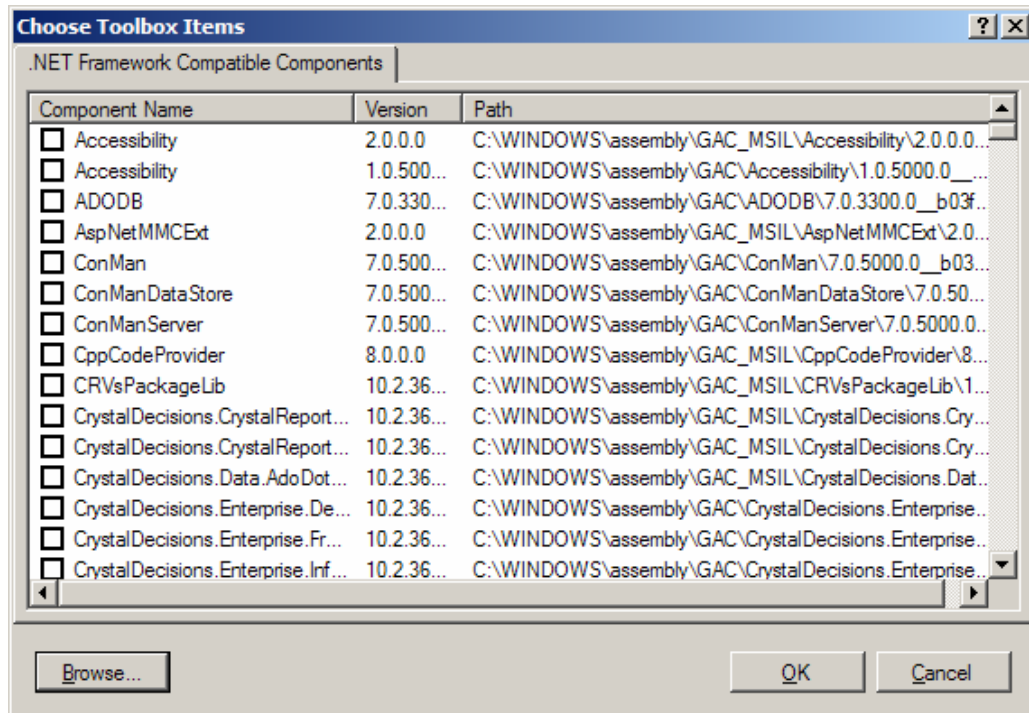


Figure 2.12 GAC Enumeration is available in Weendigo

Accessing GAC is not possible within Microsoft .NET Framework built in classes. GAC enumeration in Weendigo is accomplished by making native calls to “fusion.dll” with a full native and managed interface and enumeration mapping. Here is a list of used enumerations:

- ASM\_NAME\_PROPERTY
- NameDisplayFlags
- CacheFlags

The list of interfaces implemented is given below:

- IAssemblyCache
- IAssemblyEnum
- IAssemblyName

“AssemblyInfo” struct and “GAC” sealed class is also implemented.

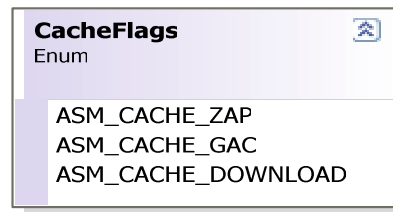


Figure 4.13 Cache Flags enumeration

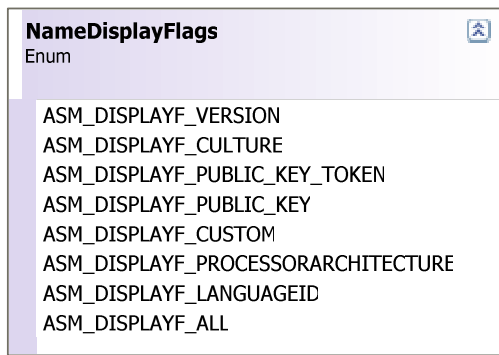


Figure 4.14 Name Display Flags enumeration

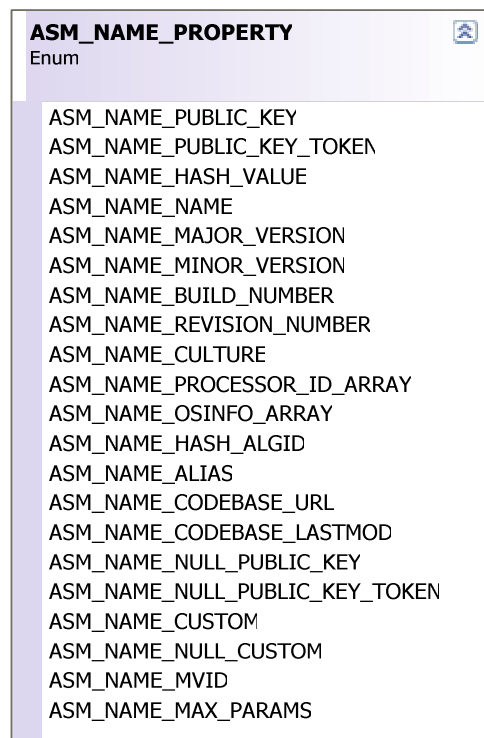


Figure 4.15 Assembly name property enumeration



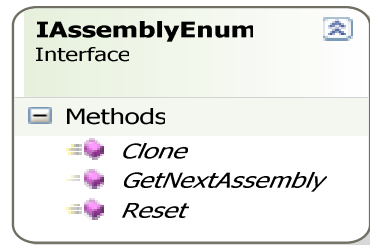


Figure 4.16 IAssemblyEnum interface

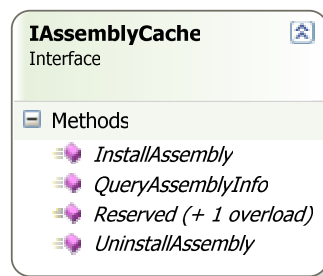


Figure 4.17 IAssemblyCache interface

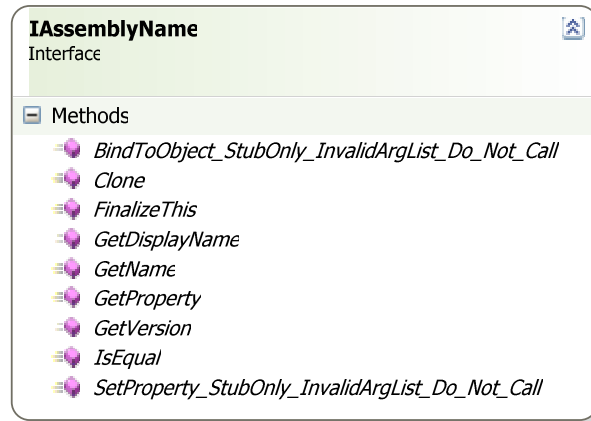


Figure 4.18 IAssemblyName interface

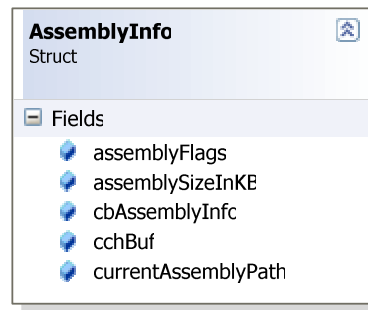


Figure 4.19 AssemblyInfo struct

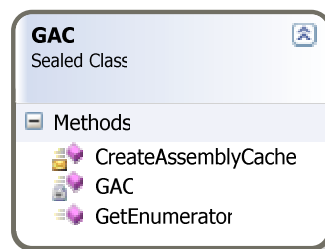


Figure 4.20 GAC class interface

After enumerating global assembly cache, gathered data from different interface are collected in a single class named as “GACComponent”. This class intended to use for any purposes. In Weendigo any instance of this class maps to single row in assembly list. The whole list is represented in a single instance single class named as “GACComponentCollection”.

Also, there is support for adding assemblies to toolbox by clicking browse button displayed in Toolbox Items dialog. Weendigo Toolbox is a container for components implemented BaseDisplayObject and Weendigo IDE is a container for Weendigo Toolbox. Weendigo Toolbox allows developers to built their own components and distribute assembly files. When a film designer wants to use this component, adding assembly to toolbox will be enough. Thus proves that Weendigo has an extensible framework.

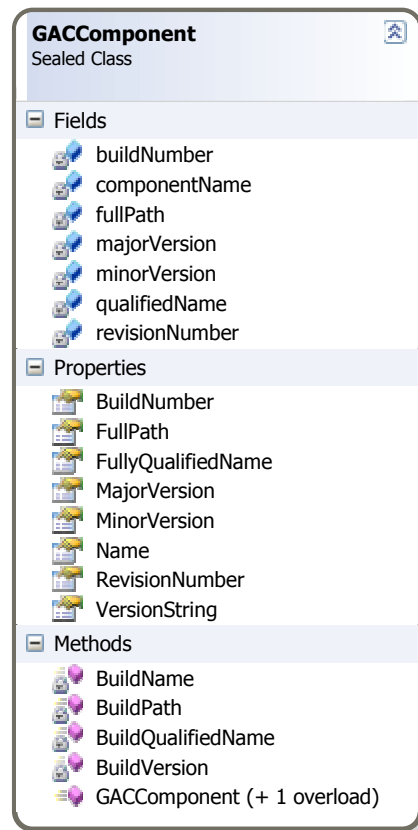


Figure 4.21 GACComponent class interface

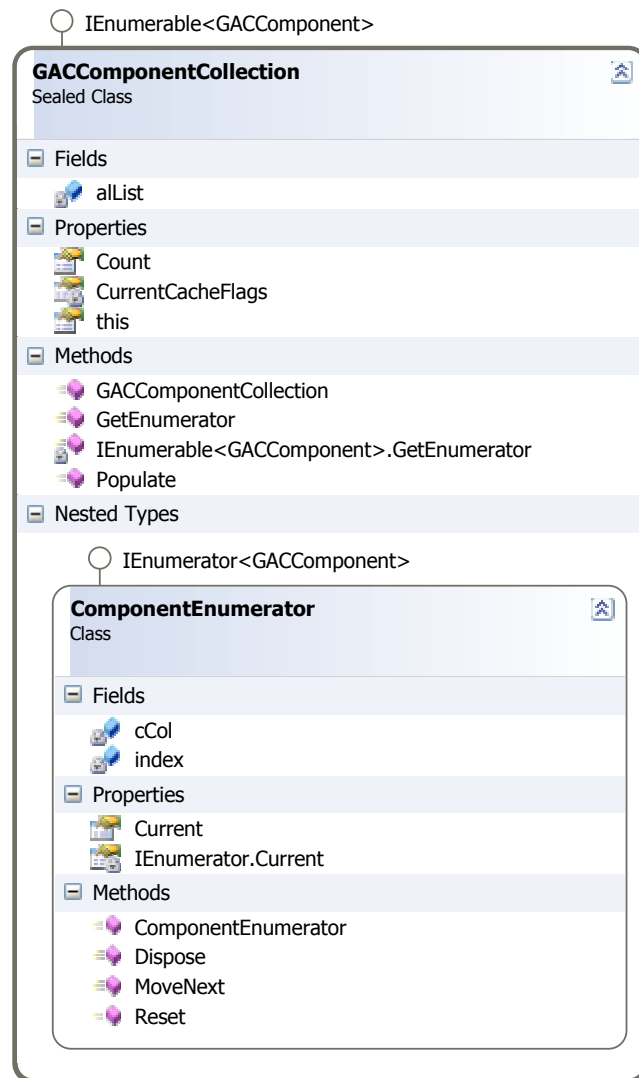


Figure 4.22 GACComponentCollection class interface

## 4.4 Property Window

Weendigo serves a properties window that enable users to view and change the design-time properties of selected objects that are located in editors and designers. This properties window can also be used to edit and view file, project, and solution properties. This Properties Window is available from the View menu.

The Properties window displays different types of editing fields, depending on the needs of a particular property. These edit fields include edit boxes, drop-down lists, and links to custom editor dialog boxes. Properties shown in gray are read-only.

There are two views of properties window available to use in Weendigo IDE.

- **Categorized:** Lists all properties and property values for the selected object, by category. You can collapse a category to reduce the number of visible properties. When you expand or collapse a category, you see a plus (+) or minus (-) to the left of the category name. Categories are listed alphabetically.
- **Alphabetic:** Alphabetically sorts all design-time properties and events for selected objects. To edit an undimmed property, click in the cell to its right and enter changes.

Also there is a description pane which is intended to show the property type and a short description of the property. A sample property window items are displayed in the figure 2.23

Property window enumerates an object's public properties and information shown in property window must be explicitly set by using custom attributes. For instance, Center Point public property of BaseDisplayObject is given below:

```
[Browsable(true),
Bindable(false),
ReadOnly(false),
Category("Apperance"),
DesignOnly(false),
Description("Object Center Position for current frame"),
DisplayName("Center Point")]
public virtual Point3D CenterPoint
{
    get { return objectCenter; }
    set { objectCenter = value; }
}
```

Code snippet given above shows that this object has a property named as `CenterPoint` and encapsulates `objectCenter` private member. `Browsable` attribute accepts a boolean value. If true, this property will be shown in property window, otherwise not. `Bindable` attribute accepts a boolean value. It is reserved for future use, which is intended to bind 3D objects using information gathered from a data source like a database. `ReadOnly` attribute accepts a boolean value. If true, this property will be shown in gray and will be read only by the definition. `Category` attribute accepts a string. This string specifies the name of the category in which to group the property when displayed in Property Window. `Description` attribute accepts a string. This string specifies a description when displayed in Property Window. This string is shown in “Description Pane”. `DisplayName` attribute accepts a string. This string specifies the property name when displayed in Property Window.

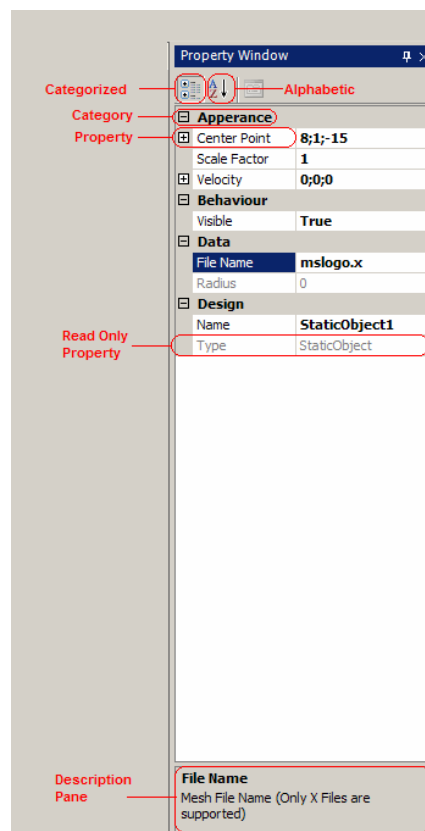


Figure 4.23 Property Window Properties

Point3D is defined as a serializable struct. In addition, this struct has a built in TypeConverter which is used by Property Window to show any instance of Point3D. Point3D struct definition is given below:

```
[Serializable, TypeConverter(typeof(Point3DConverter))]  
public struct Point3D  
{  
}
```

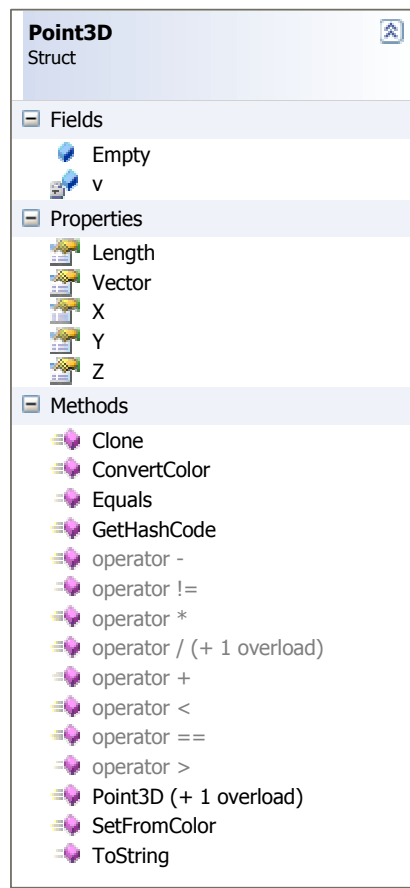


Figure 4.24 Point3D class interface

Point3D encapsulates a vector inside, and has operator overloading implementations. Considering Property Window, this class implementation is out of scope. Point3DConverter is a class inherits from TypeConverter class. When property window wants to display a property in type of Point3D, related a new

instance `TypeConverter` class is initiated and displayed in Property Window. `Point3DConverter` class details are given below:

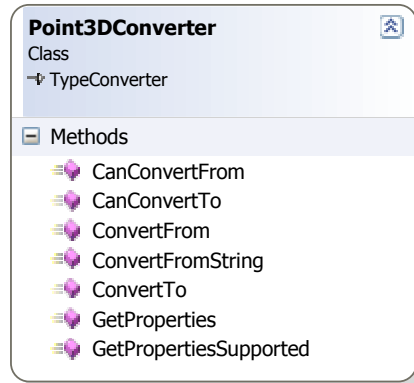


Figure 4.25 `Point3D Converter` class interface

This class implementation is given below:

```
public class Point3DConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context, Type sourceType)
    {
        if (sourceType.Equals(typeof(String)))
            return true;
        return base.CanConvertFrom(context, sourceType);
    }

    public override object ConvertTo(ITypeDescriptorContext context,
        System.Globalization.CultureInfo culture, object value, Type destinationType)
    {
        if (destinationType.Equals(typeof(String)))
            return value.ToString();
        return base.ConvertTo(context, culture, value, destinationType);
    }

    public override bool CanConvertTo(ITypeDescriptorContext context, Type destinationType)
    {
        if (destinationType.Equals(typeof(String)))
```



```

        return true;
        return base.CanConvertTo(context, destinationType);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
System.Globalization.CultureInfo culture, object value)
    {
        if (value is String)
        {
            return ConvertFromString(value.ToString());
        }
        return base.ConvertFrom(context, culture, value);
    }

    public override bool GetPropertiesSupported(ITypeDescriptorContext context)
    {
        return true;
    }

    public override PropertyDescriptorCollection GetProperties(ITypeDescriptorContext context,
object value, Attribute[] attributes)
    {
        return TypeDescriptor.GetProperties(value, new Attribute[] {
BrowsableAttribute.Default});
    }

    public static new Point3D ConvertFromString(string value)
    {
        string[] strArr = value.Split(';');
        try
        {
            return new Point3D(float.Parse(strArr[0]), float.Parse(strArr[1]),
float.Parse(strArr[2]));
        }
        catch (Exception)
        {
            throw new InvalidCastException(value);
        }
    }
}

```

There are built in type converters for types like Color. When a color is shown in property grid a type converter and a type editor is initiated automatically to show and allow user to edit.

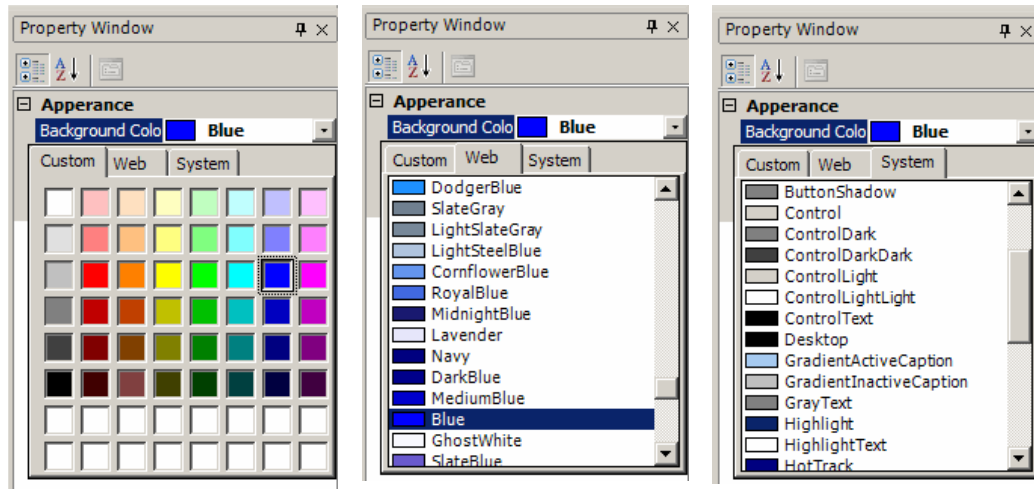


Figure 4.26 Color Type Converter user interface

## 4.5 Timeline Track Bar

A timeline is a description of a series of events in chronological order, chronological arrangement of occurrences. While we are talking about designing a 3D animation film, it is required to have track bar to control movie over a timeline. Weendigo has a built in timeline track bar allow user to review or modify the timing in the active scene.

Weendigo timeline track bar enables user to perform following actions inside a scene:

- Set the scene length of the active scene including extending and shortening scene. A scene length is set to 60 by default.
- Seek to a specific time to edit the object properties at this specific time. Currently, sensitivity of timeline is limited with the units of seconds.

Weendigo timeline track bar is implemented as a control and embedded into Weendigo IDE. Class details are given below for clarity.

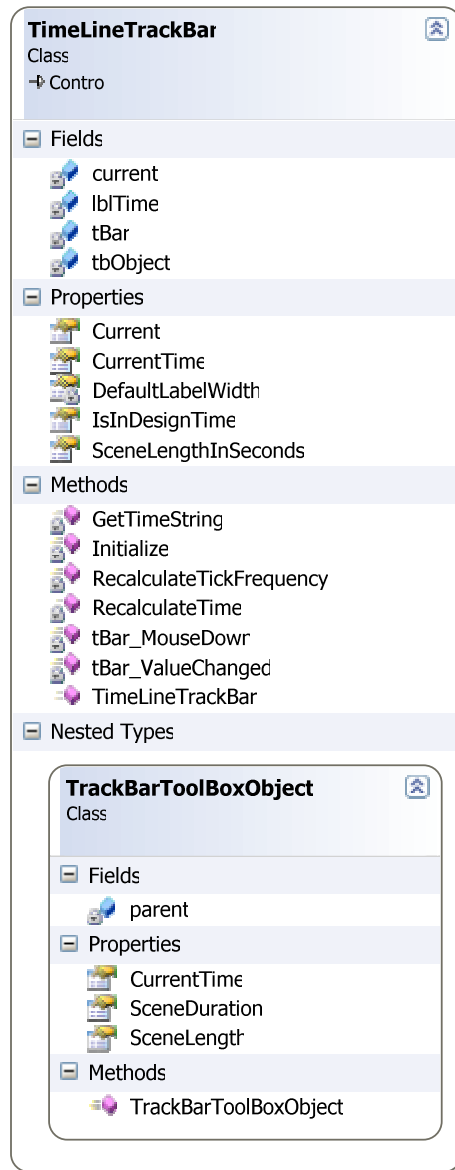


Figure 4.27 Timeline Trackbar Class interface

This implementation is very close general track bar implementation. Additionally it has a duration string relying above of the timeline track bar.



Figure 4.28 Weendigo Timeline track bar has a duration string relying at the top.

As a requirement for a fully integration with Weendigo IDE, Weendigo timeline track bar exposes a toolbox object compatible with Weendigo Toolbox. Additional to these implementations, “TimeSpanUIEditor” which inherits `UITypeEditor` is implemented to able user to view and modify duration in the property window. Class details of “TimeSpanUIEditor” is given below;

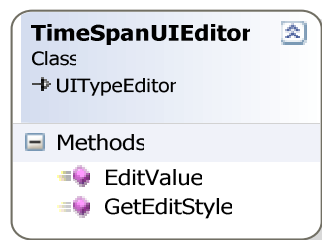


Figure 4.29 TimespanUIEditor  
class interface

Weendigo timeline track bar has the following properties that are enumerated by Weendigo IDE:

- Data
  - **Current Time:** active time of the current scene.
  - **Scene Duration:** total duration of the scene.
  - **Scene Length:** Equivalent to “Scene Duration” property it is the total length of the scene length. But the time is in the units of seconds.

## 4.6 Resource Management

Textures, icons, mesh files are commonly used resources in Weendigo implementation. Thus requires a management ability to coordinate and provide a caching mechanism for a resource access. In weendigo access to any resource

managed using a single class. Also icons used in Weendigo design editor are managed. But this management is done automatically by Microsoft .NET Framework.

Textures and mesh files are loaded to memory after the first load request. This loading is done only once for a file using whole execution life cycle. Thus means unloading these files are not done unless the application is closing. There are different types of resources managed by Weendigo. A list of these resources is given below:

- Textures
- Effects
- Fonts
- Meshes

Resources are cached in a class implemented with Singleton design pattern. ResourceCache class holds references of instances load demanded objects. Each of these resources can be easily identified with a string. For instance, a file based object can be identified by its full path. By the way, a font can be identified by its name. Each of these objects is stored in hash tables with their unique identifiers. When a texture file is demanded to load, file is loaded to memory at first demand and added to texture cache with its full path. Afterwards when a request is demanded to access this file, file is not read and loaded to memory again. Cached copy is returned to callee. By the way, this means Weendigo does not have a support to handle changes on a resource at run time due to performance reasons. “ResourceCache” class details are given below for clarity:

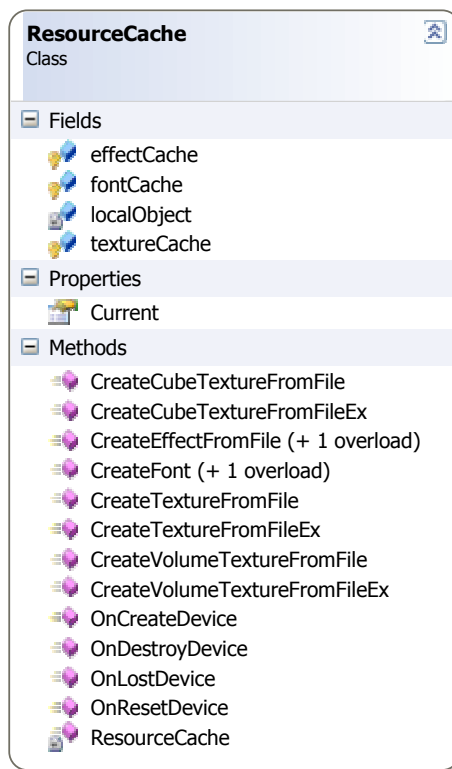


Figure 4.30 Resource Cache class interface

Only device reset and lost events forces to empty and reload cached items. Additionally, icons used in weendigo design editors are cached and managed by Microsoft .NET Framework. Microsoft .NET Framework has a built in support for resource management. All icons are used in Weendigo editor is sized at 16x16. Only one bitmap image is used to access these icons. Thus, forces us to extract relevant images at run time.

All Microsoft .NET Framework managed resources are referenced using their indexes. Whole picture is given below with a zoom (200%):



Figure 4.31 Icons in Weendigo Editor

This image consists of fourteen 16x16 images. Image descriptions are given below consecutively:

- Properties Window Image
- Solution Explorer Window Image
- Delete Image
- Close Project Menu Image
- View Scenario Image
- Save Menu & Toolbox Image
- Compile Project Menu Image
- Toolbox Window Image
- Play Movie Image
- Play Scene Image
- New Weendigo Solution Image
- Weendigo Solution Image (in Solution Explorer Window)
- Weendigo Scene Image (in Solution Explorer Window)
- Save as Image

Each of these images are extracted at run time and managed by Microsoft .NET Framework. As seen in the figure above, whole image has a pink layer. This is required to use color keying. Pink is used as a chroma key in this implementation. A chroma key is the removal of a color (or small color range) from one image to reveal another image "behind" it. The removed color becomes transparent. This technique is also referred to as "color keying", "colour-separation overlay" ("CSO").

The principal subject is photographed / filmed against a background having a single color or a relatively narrow range of colors, usually in the blue, green or pink (in this case pink). When the phase of the chroma signal corresponds to the preprogrammed state or states associated with the background color(s) behind the principal subject, the signal from the alternate background is inserted in the composite signal and presented at the output. When the phase of the chroma signal deviates from that associated with the background color(s) behind the principal

subject, video associated with the principal subject is presented at the output. This process is commonly known as "keying", "keying out" or simply a "key."

The best known example is television weather broadcasts, where the meteorologist is filmed in front of a flat, evenly colored green or blue screen. The background color is removed electronically, and replaced with a weather map which the meteorologist points to (by glancing at monitors' off-camera). The meteorologist must not wear clothing with any color close to the background color, or else part of the clothing will be replaced with the background video.

Blue is used for weather maps and movie special effect because it is complementary to human skin tone and therefore is easier to key out. However, in many instances green has become the favored color because digital cameras retain more detail in the green channel and it requires less light than blue. Although green and blue are used most often, any color can be used. Occasionally a magenta background is used.

With better imaging and hardware many companies are avoiding the confusion often experienced by weather presenters by lightly projecting a copy of the CSO image onto the CSO blue/green background. This allows the presenter to accurately point and look at the map without referring to the monitors.

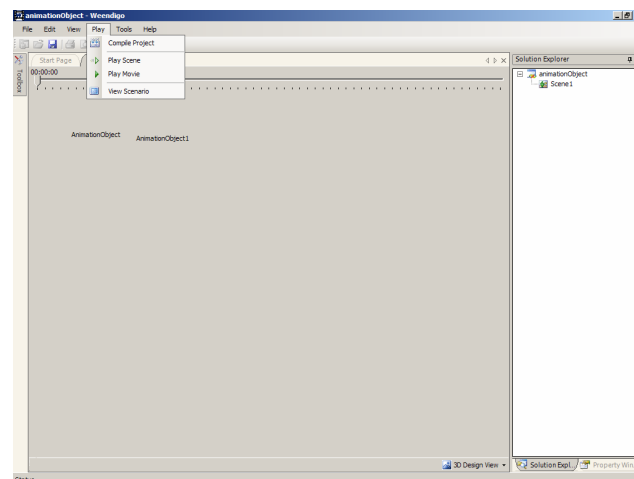


Figure 4.32 A sample snapshot of commonly used icons



## 4.7 Weendigo Docking Library

The Dock is a graphical user interface feature first introduced in the NeXTSTEP and OPENSTEP operating systems, and radically changed and refined in Mac OS X, where it behaves more like the Apple Newton's Newton OS Dock. Windows does not have a native dock equivalent, but many applications like Microsoft Visual Studio .NET have built in docking feature inside with a limited usage. Weendigo has an important built in library inside to serve complex graphical user interface in Weendigo Design Editor.

There are too many features implemented in Weendigo Docking Library but several features are included and used in Weendigo editor. The rest of these features are reserved for further releases. Following features are included and used in current version:

- Auto Hide Panels
- Tab page support for Panels
- Floating forms and panels
- MDI support for tab pages

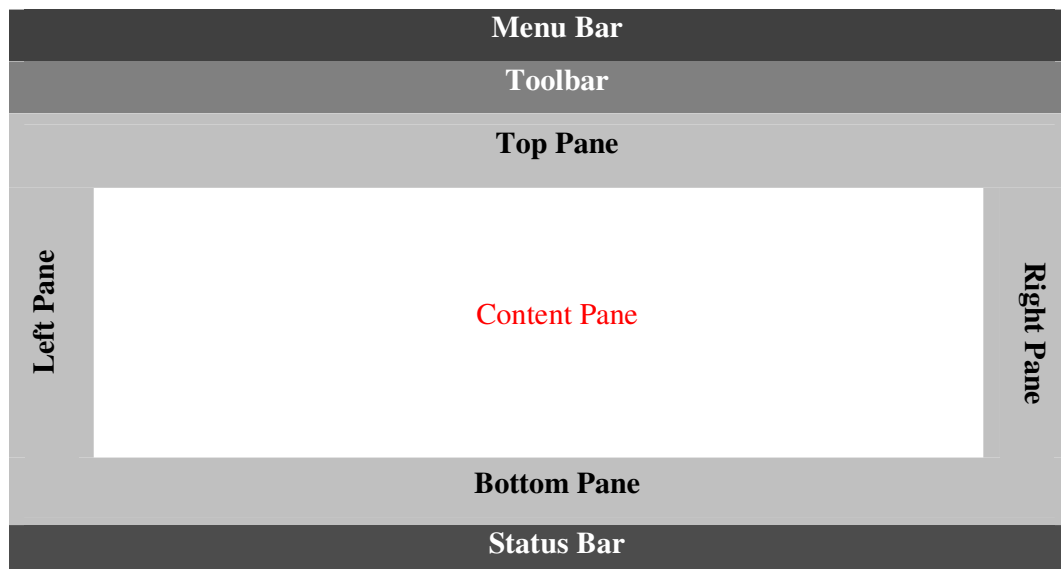


Table 4.1 Weendigo User Interface Structure

Menu bar, status bar, and the toolbar is visible unless marked as invisible by user. Other panes including top, right, bottom, left and content pane are logical panes therefore they are invisible. There three floating form inside weendigo design editor. They are given below:

- Solution Explorer
- Toolbox
- Property Window

All of these forms are embedded inside floating forms. Thus allows user to re-dock, move all of these controls to any pane. Also content pane contains a tabbed page control which allow user to switch between them easily. It has similar features with MDI forms.

Graphical computer applications with a Multiple Document Interface (MDI) are those whose all windows reside under a single parent window (usually with the exception of modal windows), as opposed to all windows being separate from each other (single document interface). In the usability community, there has been much debate over which interface type is preferable. Generally SDI is seen as more useful in cases where users work with more than one application.

The disadvantage of MDI usually cited is the lack of information about the currently opened windows: In order to view a list of windows open in MDI applications, the user typically has to select a specific menu ("window list" or something similar), if this option is available at all. With an SDI application, the window manager's task bar or task manager displays the currently opened windows.

In Weendigo implementation "tabs" to show the currently opened windows in an MDI application is preferred. By the way, sometimes this approach is called as "tabbed document interface" (TDI). When tabs are used to manage windows, individual ones can usually not be resized.

In the Weendigo implementation, the panes are referred as “hot zones” and implemented in “HotZone” class under “Weendigo.Controls.Docking” namespace. This class has virtual method declaration; nevertheless it is not defined as an abstract class as these methods are not defined as pure virtual. This class details are given below for clarity:

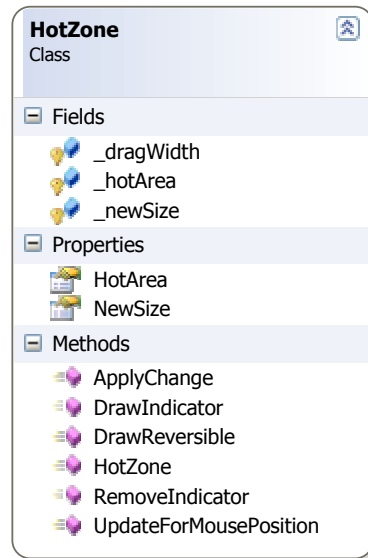


Figure 4.33 Hotzone class interface

Floating hot zones are implemented in one of the inheritor of “HotZone” class called as “HotZoneFloating”. This class allows floating forms to be added. These zones might include zero or more docked controls inside them. While this zones are dragging and being dropped this class instance behaves as a container. This class has implementation of predefined virtual methods and behaves different which is an expected result of polymorphism. This class details are given below:

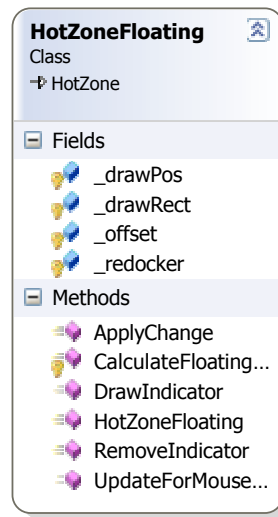


Figure 4.34 HotZoneFloating class

Another inheritor of HotZone class is “HotZoneReposition” this implementation allows reposition child controls in a given order. Order is set by using “Position” enumeration. This class details are given below:

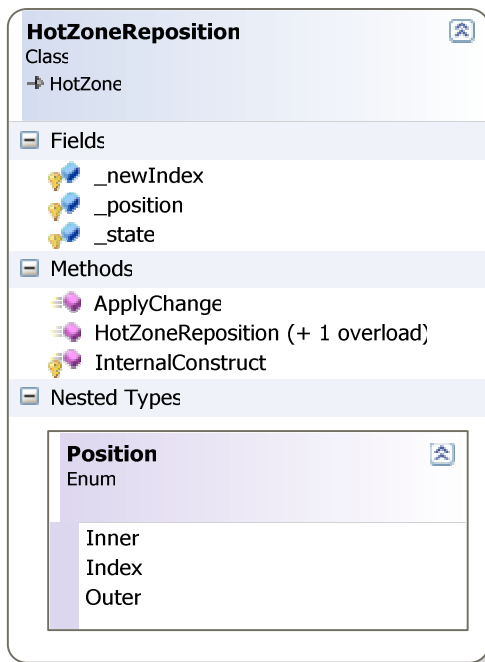


Figure 4.35 HotZoneReposition class interface

Also for tabbed grouped controls “HotZoneTabbed” class is implemented as a inheritor of “HotZone” in order to suspend layout in Weendigo Editor.

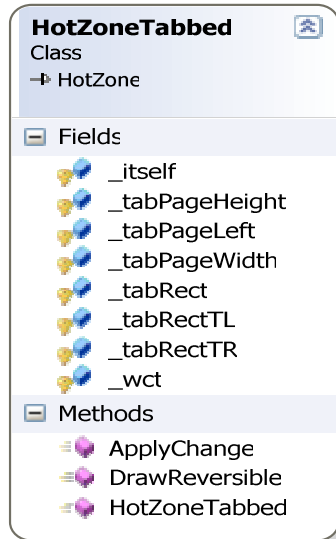


Figure 4.36 HotZoneTabbed class

Each control in these zones must be a window to provide abstraction and provide a fully integration. Therefore “Window” class is implemented which is an inheritor of “ContainerControl”. Container control allows adding controls and behaves as a host for all child controls. These classes’ details are given below for clarity:

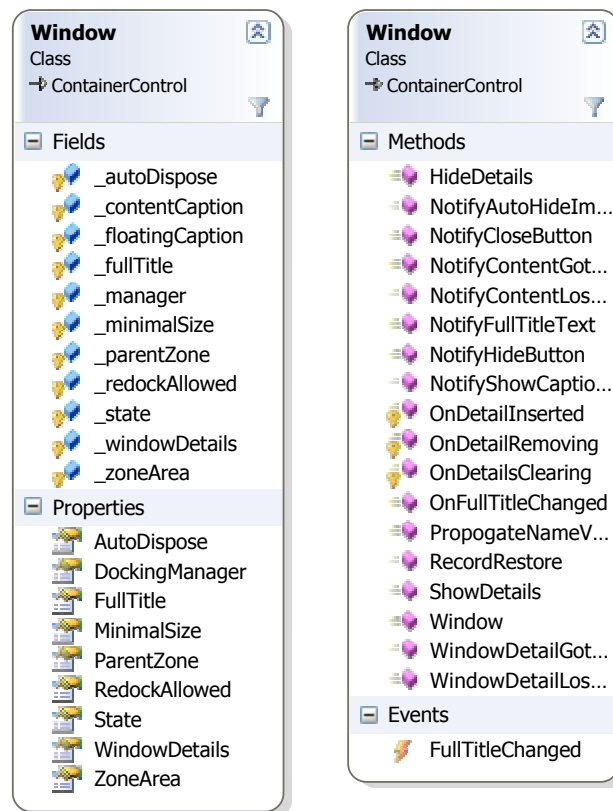


Figure 4.37 Window class interface

Each instance of this class must have a valid reference of Docking Manager. This control is similar to windows forms by the way these control must be added to an existing zone.

Window Content is an inheritor of Window class. This class intended to ease use of window class. There are several useful implementations like disposing itself when there is no child controls. As these class have events and consuming system resources these a nice feature for an application. Also this class has a functionality to bring it self to top among other instances in the same zone. This class details are given below:

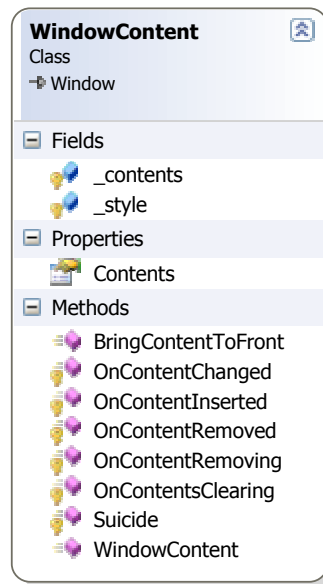


Figure 4.38 Window Content class interface

Auto hiding and pinning window contents is an important feature to allow user to customize designer items layout according to their habits. This feature is implemented as a component but this component is not intended to be use as a toolbox item. Therefore, toolbox support is disabled by overriding toolbox attributes inside class declaration. This class is an inheritor of panel control in windows forms. Any instance of this class automatically docks itself as a requirement. This auto hide panel is always hidden until some contents are added. When the main Weendigo editor window is resized there are some special actions to be taken. This action forces to invalidate and reconstruct this control. It is not a functionality requirement, it just prevents drawing artifacts. Also each instance of these auto panels adds itself to the main application filtering list. Only mouse events are filtered when this form is activated and contains focus on controls inside this panel and the panel is visible.

Additionally, this class has a nested class inside. This class is a host panel and performs its own paintings to serve the functionality of hiding and showing panel at run time. It might have more than one child control inside. But there must be at least one child control. Otherwise object might be constructed and will not be visible until

a child control is added. Auto hide panel and this nested class of auto host panel class details are given below:

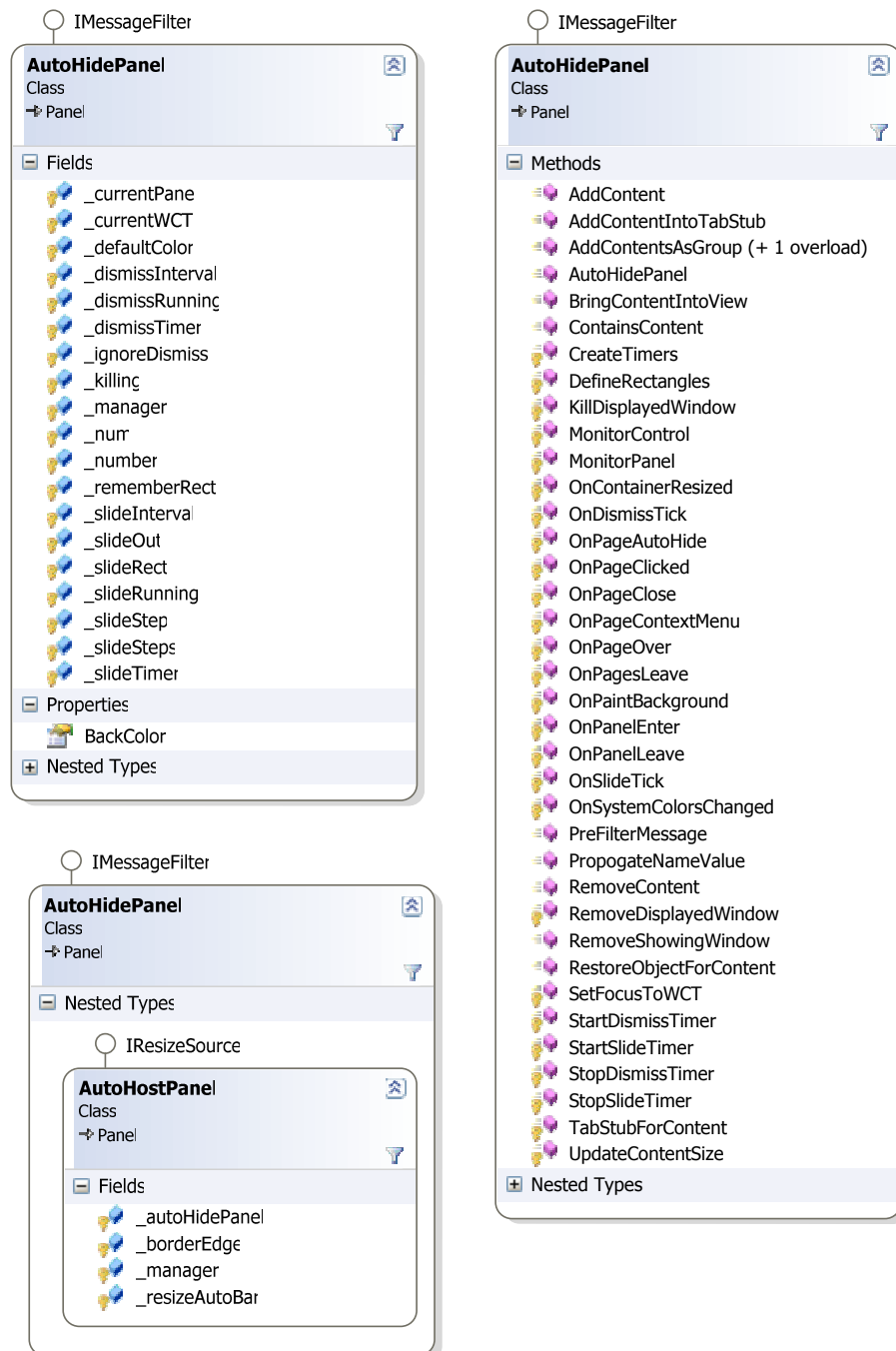


Figure 4.39 Auto Hide Panel class interface



Whole of docking operations are managed by a single class. Only one instance of this class is used during runtime, but it has a built in support for multiple instances. This is why this class is not implemented using singleton design pattern. A Form can cause the child controls to be reordered after the initialization but before the “Form Load” event. Therefore this class hooks into the event and force the auto hide panels to be ordered back into their proper place to handle this activity. This class has the ability to load window layout from a configuration file and store it in a configuration file. This class details are given below:

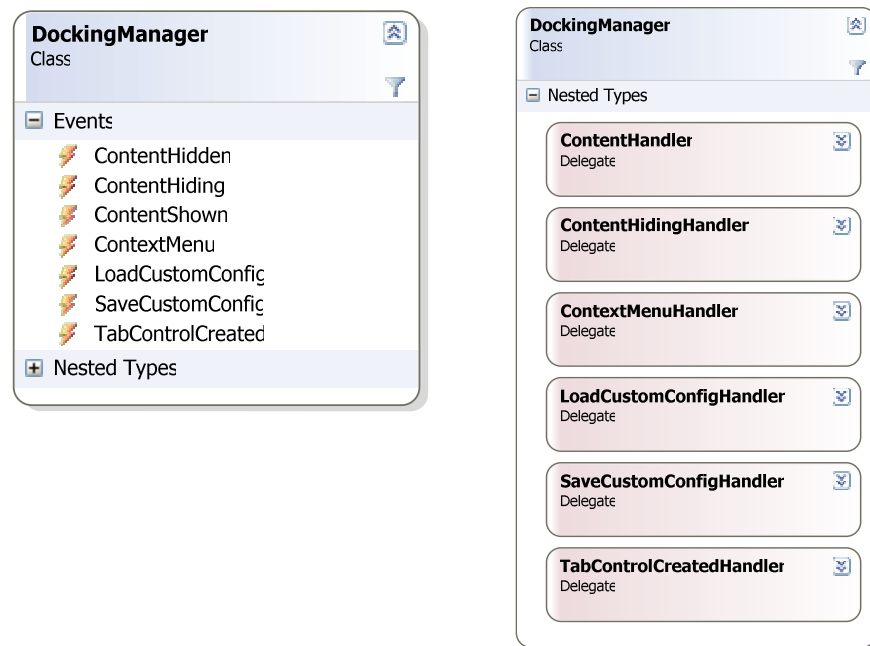


Figure 4.40 Docking Manager class interface

There are several Win32 native API calls made in docking manager. All of these calls are collected inside classes according to related library. GDI32 class contains calls made to gdi32.dll and User32 class contains calls made to user32.dll.

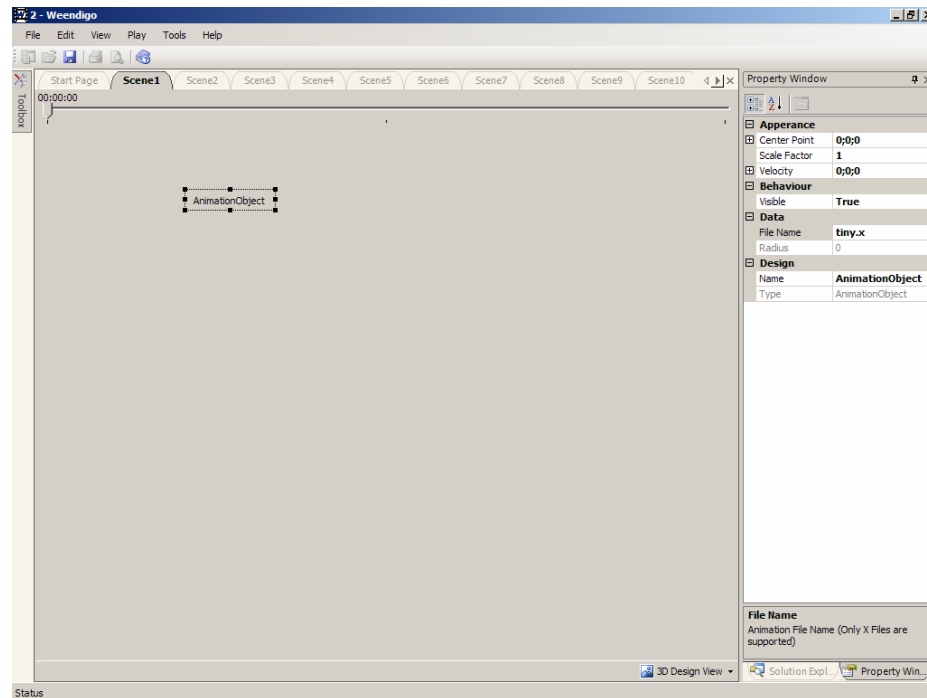


Figure 4.41 It is possible to add a numerous of content panel inside content zone. Titles are scrolled when needed.

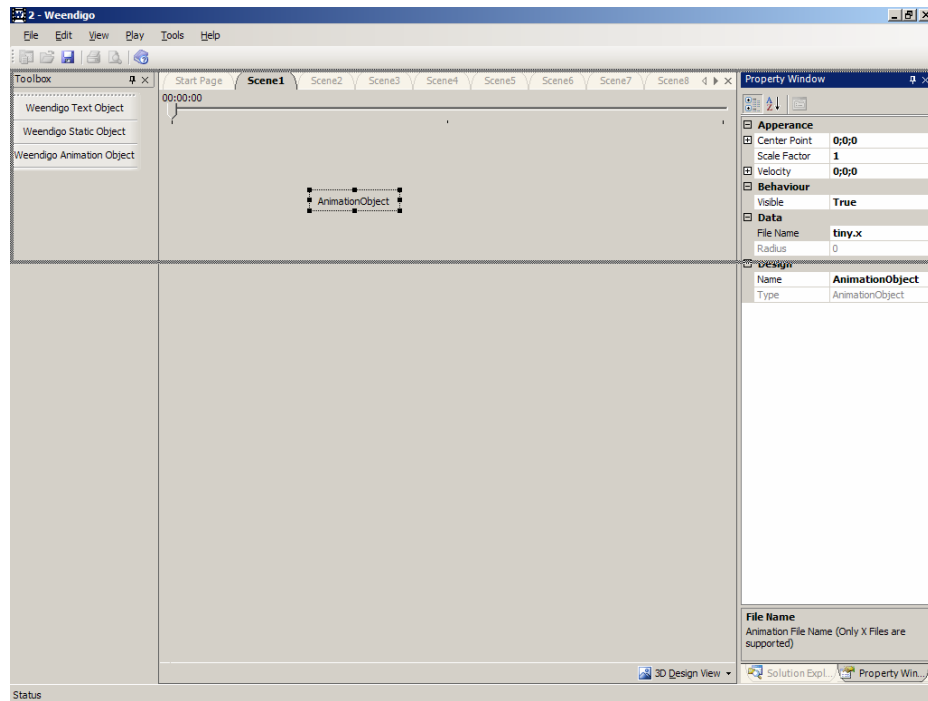


Figure 4.42 Any panel can be dragged and docked into another zone, virtual result is shown at then time of dragging

## 4.8 Dialog Management

Dialogs used in Weendigo are managed by a single controller. This provides a valuable flexibility that allows you customize user dialogs whenever you want. Dialogs used in Weendigo are managed via single class implemented using singleton design pattern. As weendigo design, it is not a requirement to be thread safe for this class implementation. Weendigo interacts with a single user at a time therefore only one user dialog can be shown. It is not feasible to show multiple dialog boxes simultaneously. All of the dialogs including splash form are handled inside this class implementation. Class details are given below for clarity:

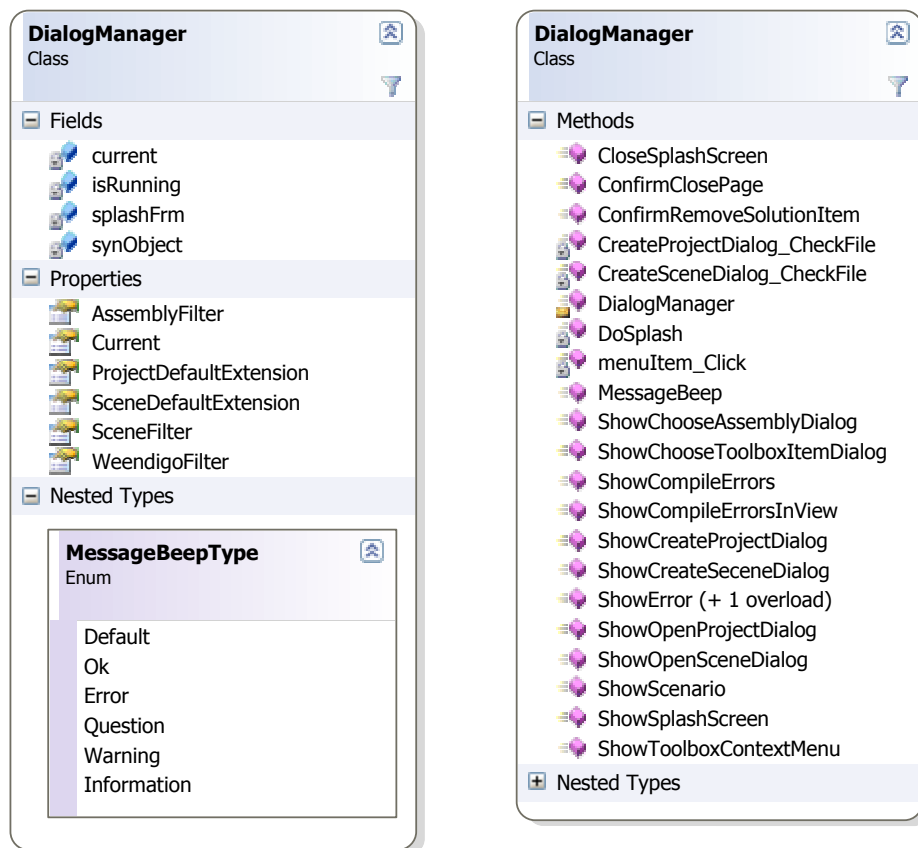


Figure 4.43 Dialog Manager class interface

Dialog box usage is summarized in below.

- **Confirm Dialogs:** prompts a question to user with choices accept, decline and cancel. There are several cases in usage, for instance as user closing a design page, a confirmation is required.
- **Information Dialogs:** display an information about user request like compile result of a scene.
- **Splash Screen:** shown at startup, creates a new thread and allows required operations to be completed in a background thread, yields CPU to perform this operations.
- **Open Dialogs:** displays a file dialog box to open an existing file. There are several cases in usage, for instance as user demands to load an existing project.
- **Save Dialogs:** displays a file dialog box in save mode to save some data on a non existing file. There are several cases in usage, for instance as user demands to create a new project.
- **Context Menu:** display a context menu according to the selected object. Context menus are especially useful on user right clicks.
- **Message Beep:** plays a waveform sound. This functionality is a wrapper for Win32 message beep. After adding the play sound request to queue, Win32 message beep function returns control to the calling function and plays the sound asynchronously. If it cannot play the specified alert sound, then attempts to play the system default sound. If it cannot play the system default sound, the function produces a standard beep sound through the computer speaker. Also users can disable the warning beep by using the Sound Control Panel application. This functionality is useful especially in usage of dialog boxes to warn user about an operation is being performed.

## **CHAPTER FIVE**

### **SCENARIO MANAGEMENT**

A scenario (from the Italian, that which is pinned to the scenery) is a brief description of an event or a series of events, an outline of entrances, exits, and action describing the plot of a play that was literally pinned to the back of the scenery. A scenario is also an account or synopsis of a projected course of action, events or situations. Scenario development is used in policy planning, organizational development and, generally, when organizations wish to test strategies against uncertain future developments. Scenario management has an important factor on the film appreciation.

#### **5.1 Scenario (Game) Engine**

Although there is no need for a game engine in order to run a 3D animation video properly. There is a need for an engine to manage scenario. Infact, the game engine embedded in Weendigo behaves like a rule based engine and scenario steps are rules and entire scenario inside a scene represents a flow. This is why this engine is called as game engine.

A game engine is the core software component of a computer or video game or other interactive application with real-time graphics. It provides the underlying technologies, simplifies development, and often enables the game to run on multiple platforms such as game consoles and Microsoft Windows. By the way, Weendigo game engine performs object attribute mapping objects in scene based on predefined scenario.

Weendigo game engine is not a common game engine it is specific for a 3D animation film. A 3D animation film requires a robust graphics engine. Game engine does not perform too much operation. Most of the CPU time is consumed by graphics engine. During Weendigo graphics engine and game engine integration tests and individual graphics engine CPU usage rate for the graphics is approximately 98

% of over all CPU usage. Hence this is a visual application this result is expected and acceptable.

In the main message loop of the Weendigo application, arbitrary calls are made to game engine and graphics engine. Game engine controls scene flow by the time. There are two type of times in Weendigo:

- Application time
- Elapsed time

Application time is time elapsed from the beginning, elapsed time is time elapsed since last frame. Also this value is used to calculate frame per second. But these times are important for the game engine. Application time equals to video time, elapsed time is used to be notified scene changes. If the absolute value of application time and the absolute value of the application time minus elapsed time differs, scene has to be changed.

Interpolation of object properties such as position and velocity (this can be also referred as acceleration) is done by Scenario Manager in the units of seconds. But the interpolation of object properties is done by game engine. As this directly depends on underlying hardware specifications and working process simultaneously, it is not guessable during at design time.

Also there is a darken scene algorithm for each scene passes. Thus forces game engine to wait scene pass and be aware of scene pass algorithm. Darken scene algorithm is implemented in graphics engine, game engine has no idea on what is being done between scene passes, but the game engine has to know to wait it. Game engine class details are given in the figure below.

Game engine has another important duty; managing camera. Because graphics engine does not know which objects are being displayed, game engine calculates the bounding box and decides camera's best position. Mostly, camera management is

implemented as a graphics engine feature. By the way using this approach will let user to manage camera position and angles at design time in further versions of Weendigo.

Graphics engine served in Pneuma is an abstract class and has exposed interfaces. These interfaces are both implemented in a single class called “SceneView”. This class does not perform too much things inside. But this class is the only way to access graphics engine. This class may be considered as a façade of the graphics engine.

There is another important class like game engine called Pneuma application. This class is implemented using singleton design pattern like game engine class. This class instantiates game engine and graphics engine and initializes video. Details of these classes are given in the figure below:

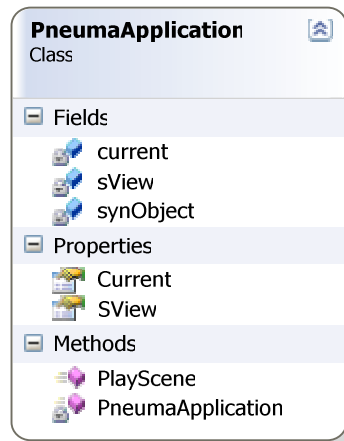


Figure 5.1 Pneuma Application class interface

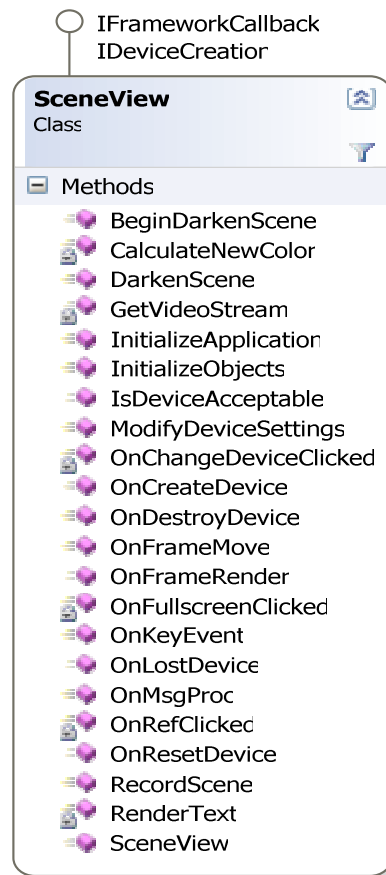


Figure 5.2 SceneView class interface



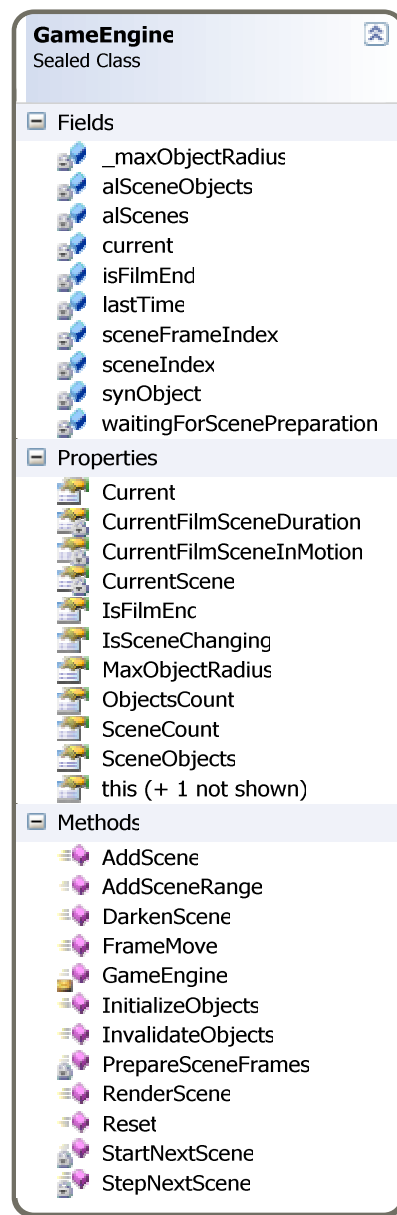


Figure 5.3 GameEngine class interface

## 5.2 Scene Design User Interface

Weendigo has an interface that enables users to design a movie scene. Specify object properties, runtime and design time positions with a simple drag and drop operation. Users are able to cut / copy / paste / delete objects like modifying a text in a text document.

Weendigo compatible objects (any object inherits from `BaseDisplayObject`) are enumerated in Weendigo Toolbox. User drags and drops these components on a scene. A scene is an object inherits from `BaseDesignView`. Using same programming semantic, also cut / copy / paste / delete operations on these objects are possible.

Whenever user right clicks on an object that inherits from `BaseDisplayObject`, Weendigo IDE will automatically show a popup context menu like following figure. User is free to choose any of these operations.

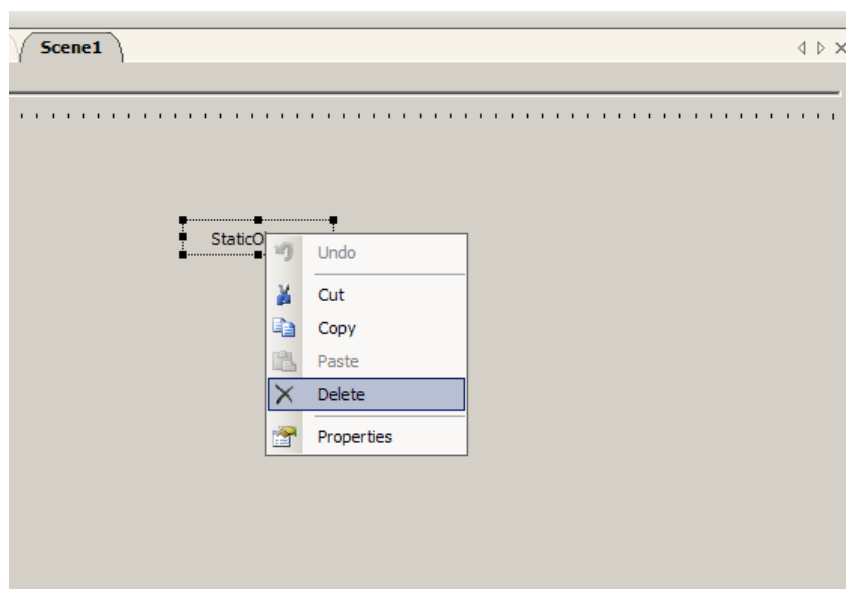


Figure 5.4 Standard text based operations are also available for Weendigo Objects

Embedding an object that inherits from `BaseDisplayObject` is not supported (also not required) this is why “Paste” operation is disabled, as seen in the figure above. By the way, whenever user right clicks on scene background, Weendigo will show a popup context menu like following:

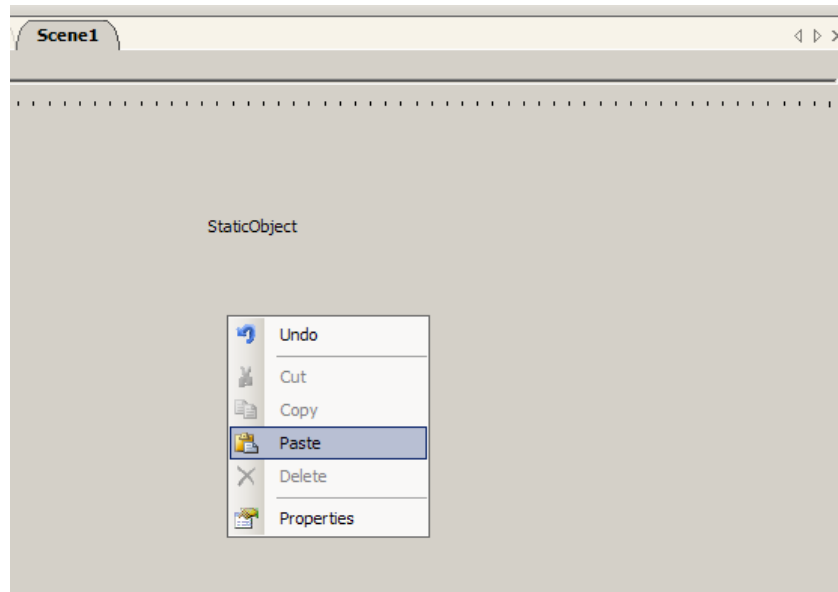


Figure 5.5 Also weendigo objects can be pasted inside same scene. Weendigo will initiate a new instance which is cloned version of the original one.

If Weendigo Clipboard (which differs from Microsoft Windows Clipboard object) contains an object inherits from `BaseDisplayObject` “Paste” operation in this popup context menu is enabled, otherwise it will be disabled by default. When user chooses “Paste” operation in this case a new object will be added to scene that inherits from “StaticObject”. All of the properties including private members of object will be copied to a new instance of object. However, because of the requirement of each object must have a unique name inside a scene; newly initiated object have a different name. Object will be renamed by adding a counter value as a suffix for the copied object name. Also, all of these operations can be done dragging object with Ctrl key is pressed. Additionally, these operations can be done by using “Ctrl + X”, “Ctrl + C”, “Ctrl + V”, “Ctrl + Del”.

As a programming perspective, all of these operations are done using XML serialization. For instance, loading a scene from a file is done by XML de-serialization. A scene is defined with “FilmScene” Class details are given in below:

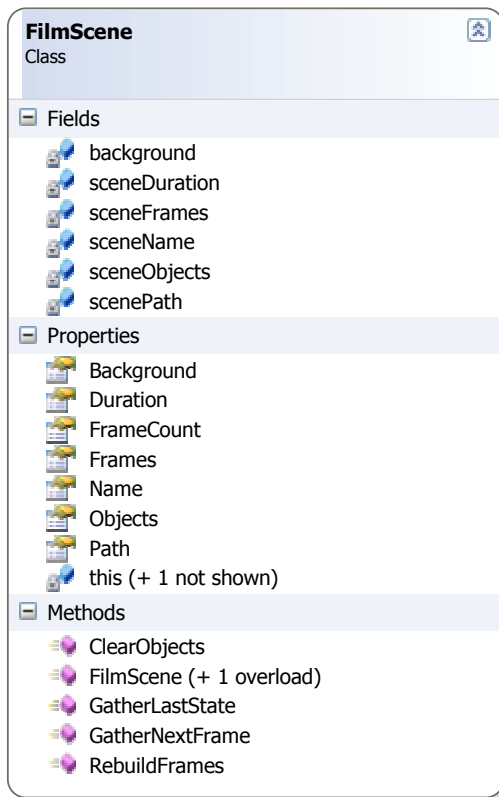


Figure 5.6 Film Scene class interface

A `FilmScene` is built up from a “SceneFrame” collection and “SceneObject” collection. Scene Object directly maps to an instance of `BaseDisplayObject` at design time. SceneObject contains required information to build objects at runtime. This class details are given below:

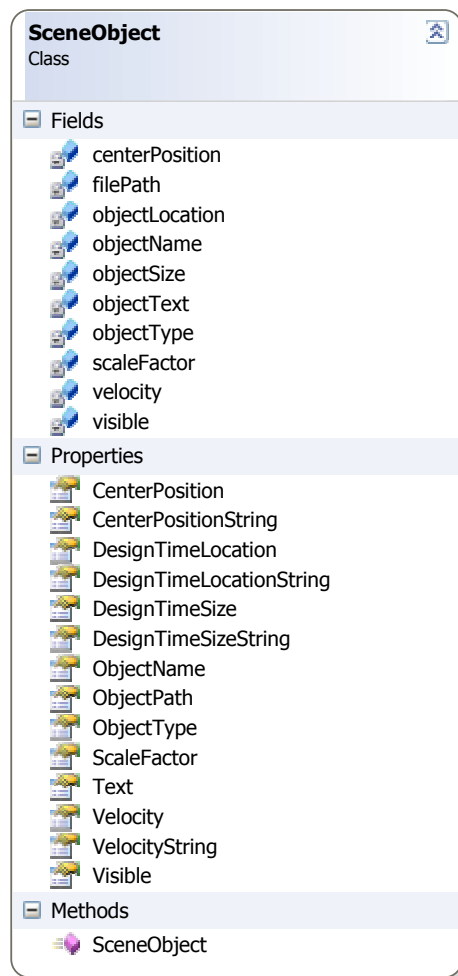


Figure 5.7 Scene Object class interface

As shown in the figure above, this is an entity class that means it only holds data does not operate on it.

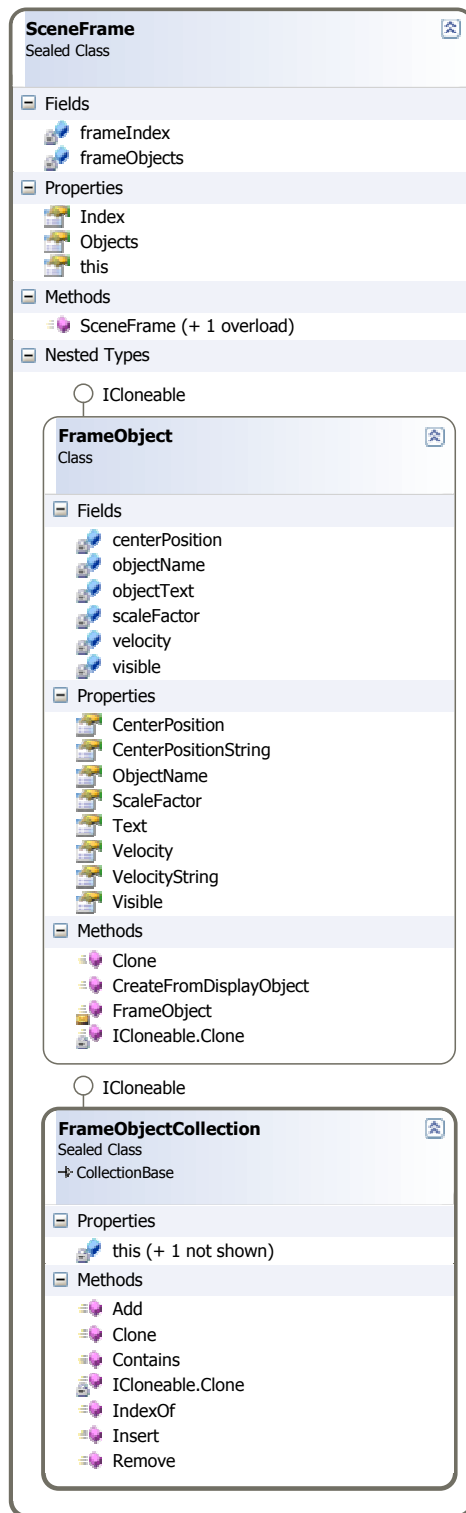


Figure 5.8 Scene Frame class interface

These classes are used to transfer data between design time and runtime. Xml Serialized version of an instance is given below for clarity:

```
<?xml version="1.0"?>
<Scene xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" Name="Scene1" Duration="5"
Path="D:\Workspace\documents\test1\mslogo\Scene1.scn">
  <Background Color="255;0;0" Stencil="0" Z-Depth="1" ClearFlags="Target, ZBuffer" />
  <Objects>
    <Object CenterPosition="0;0;0" Velocity="0;0;0" ScaleFactor="2" Visible="true"
Path="crypt.x" Name="StaticObject" DesignTimeLocation="123;98" DesignTimeSize="105;28"
Assembly="Weendigo.PNeuma.StaticObject" />
    <Object CenterPosition="8;1;-15" Velocity="0;0;0" ScaleFactor="1" Visible="true"
Path="mslogo.x" Name="StaticObject1" DesignTimeLocation="218;209" DesignTimeSize="105;28"
Assembly="Weendigo.PNeuma.StaticObject" />
  </Objects>
  <Frames>
    <Frame Index="1">
      <Objects />
    </Frame>
    <Frame Index="2">
      <Objects />
    </Frame>
    <Frame Index="3">
      <Objects />
    </Frame>
    <Frame Index="4">
      <Objects />
    </Frame>
    <Frame Index="5">
      <Objects />
    </Frame>
  </Frames>
</Scene>
```

Saving a scene state in Weendigo UI, performs XML Serialization on active scene.

### 5.3 Scene Compilation

Compilation is the process of translating text written in a computer language into another computer language that is meaningful for a computer. Generally output of compilation process is either assembly language or machine language. Lexical analyzing, preprocessing, parsing, and semantic analysis, code optimizations, code generation are the steps of a compilation process. Compilation may include many or all of these operations. Nevertheless these operations are not enough for a tool to create 3D animation films. The missing operation in this process flow is logical analyzing.

Lexical analysis is a scanning process that breaks statements into formerly predefined tokens. For instance, following statement will be broken into identifiers and operators tokens.

`result= a + b * c / d`

These statement will be tokenized like following:

result	=	a	+	b	*	c	/	d
--------	---	---	---	---	---	---	---	---

Static analysis organizes produced tokens into syntax tree that describes structure. Only syntax errors are handled in this process. Syntax analysis uses grammars. There is just one grammar for a language. Here is the syntax tree of the previous example;



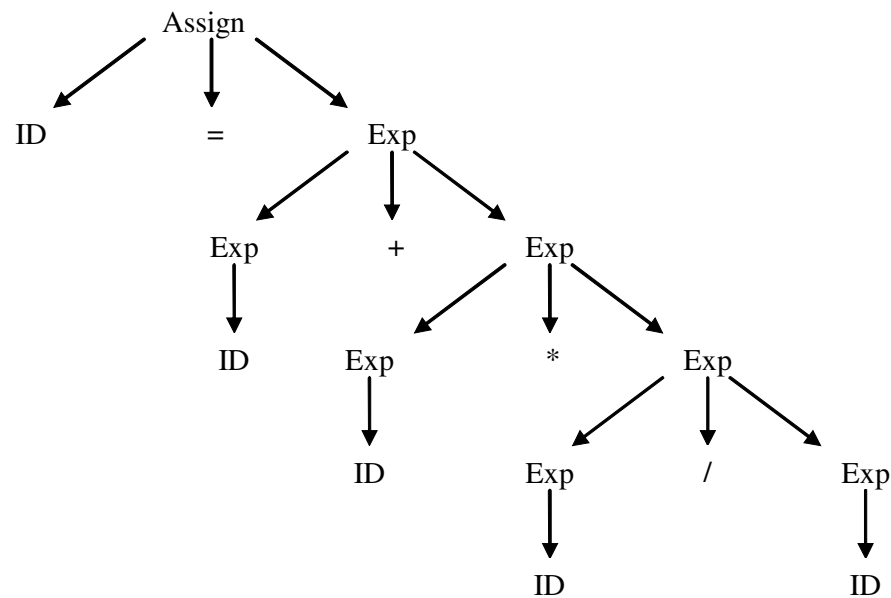


Figure 5.9 Syntax tree of given example

Semantic analysis is the process of intermediate code generation and validates meaning. Type and error checking is done at this step. After this step, a code optimization step might occur. Machine independent code optimizations are performed to improve efficiency. By the way, finding optimal code is NP (Nondeterministic polynomial time) complete problem.

Code generation process translates intermediate code to real machine code. Memory management, register allocation, instruction selection, and instruction scheduling are the sub processes.

Generating a 3D animation movie needs additional compilation process which was not done in the whole of these steps. Compilers could not catch runtime errors. In fact, this is possible to detect at design time. For instance, an animated object should have a valid mesh file name. If not, it is certain that we will have a run time error. Due to cost of testing a 3D animation film, a 3D animation film tool should handle these design time errors. Weendigo has a capability to compile each object independently and inform user on these design time errors. In most cases, this could

be done by compiler. But there is not any compiler have built in support for this facility. Consider following code snippet:

```
using(StreamReader sr = File.OpenText("C:\\input.txt"))
{
    // Do something
}
```

Compiler can easily perform a check on whether a valid text file named as “input.txt” exists under drive lettered as “C”.

Every scene in a Weendigo project is designated with a SceneEditorWindow which’s details are given in figure 3.10.

Each file under the active project can have only one SceneEditorWindow instance. However, there can be different views for a scene. For example, a visual art designer might want to have a Camera view to have different perspectives. Every time, when user builds solution “CompileScene” method of each “SceneEditorWindow” instance is called by Weendigo IDE.

Inside “SceneEditorWindow” ’s “CompileScene” method. All visible views are compiled independently. This is done by calling “CompileView” method of each instance which is inherited from “BaseDesignView”. Class details of “BaseDesignView” are given in figure 3.11.

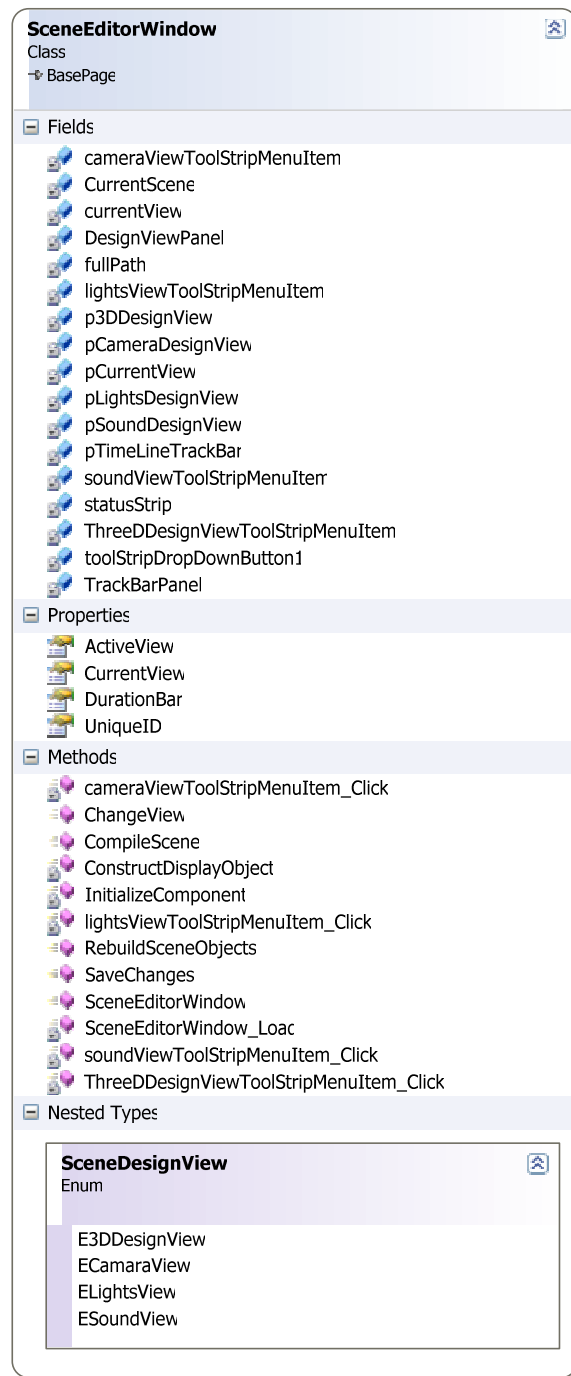


Figure 3.10 Scene Editor Window class interface

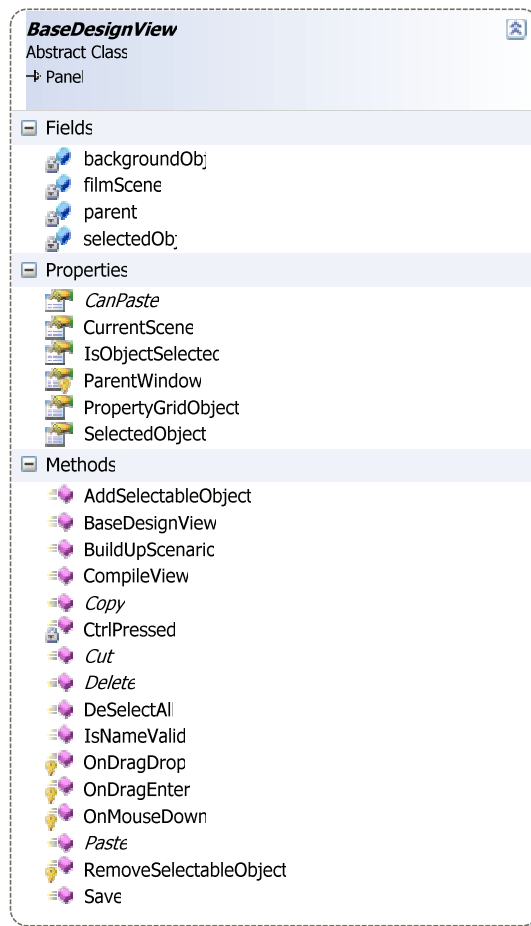


Figure 5.11 Base Design View class interface

Inside “CompileView” of an instance “BaseDesignView”, each object is compiled by calling “BaseDisplayObject”’s “IsObjectValid” method.

For clarity, Weendigo Static Object’s compilation conditions are given below:

- A file name should be specified for mesh object.
- Specified filename must exist under Weendigo Media Path.
- Specified filename must be a valid X File which is the only mesh file type supported by Weendigo IDE.

If any of these conditions are not met, a compiler error generated but compilation process is not broken. After whole projects compilation, all errors are reported user like following screen.

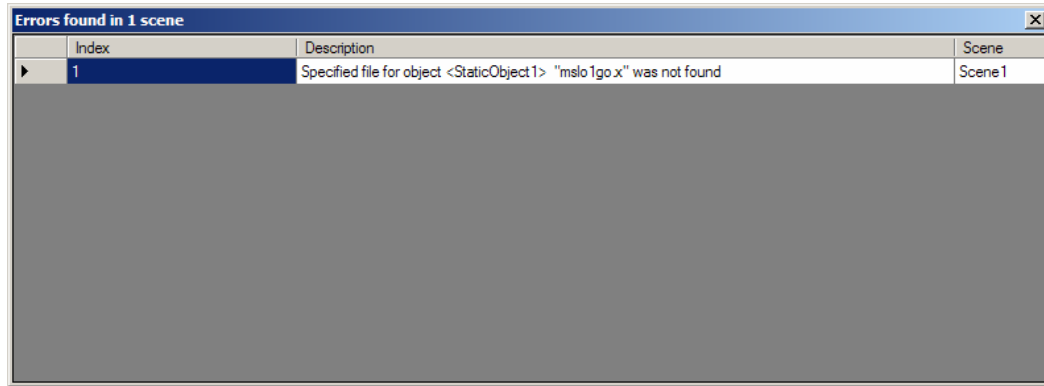


Figure 5.12 Compiler Errors are displayed in a dialog box and contains detailed information on error

Weendigo compiler complains that the object named as “StaticObject1” in Scene1 has a file name which is not exists under the Weendigo Media Path.

Weendigo support any components developed with a Microsoft .NET compatible language. Additionally Weendigo compiles project against design time problems. This is not a required feature for a tool to create 3D animation films, but this is a nice to have feature.

## 5.4 Scenario Manager

A scenario (from the Italian, that which is pinned to the scenery) is a brief description of an event or a series of events, an outline of entrances, exits, and action describing the plot of a play that was literally pinned to the back of the scenery. A scenario is also an account or synopsis of a projected course of action, events or situations. Scenario development is used in policy planning, organizational development and, generally, when organizations wish to test strategies against uncertain future developments. Scenario management has an important factor on the film appreciation.

Scenario management is similar to project management. Scenario management has familiar steps with Software Development Life Cycle (SDLC). Before starting a project, project manager has to define the subject and give a brief description on what are the expectations and focus points. It is hard to guess what are the requirements and costs of this project during this step. The project manager has to define project scope and prepare a draft project plan to estimate project cost. After retrieving the approval of sponsor, project scope is circumscribed and role players are chosen. Project manager finalizes project plan defining each tasks in details and maps these tasks to resources. These resources are staff assigned to this project in software projects. At this time, all milestones are defined and it is clear to evaluate project status at a specific time. In software projects, sometimes everything does not go well and there can be latency in times. Fortunately, in scenario management there is no doubt if an entry will occur at a specific time. Scenario management differs from project management at this point.

Current version of Weendigo has a sensitivity limited at a second while managing the scenario. By the way, a human eye is capable to interpret twenty four frames at a second. Thus means if you prepare a scene with twenty four frame and play this sequentially in a second. You will gather a movement. Weendigo covers these sequentially order and lets scenario manager to define the object positions in a time interval. Interpolation between two arbitrary seconds is done by Weendigo automatically. This will save time on designing a movie and lets scenario manager to concentrate on whole scenario instead of time frames. For instance, following illustration shows a movement of an object in the twenty four frames.

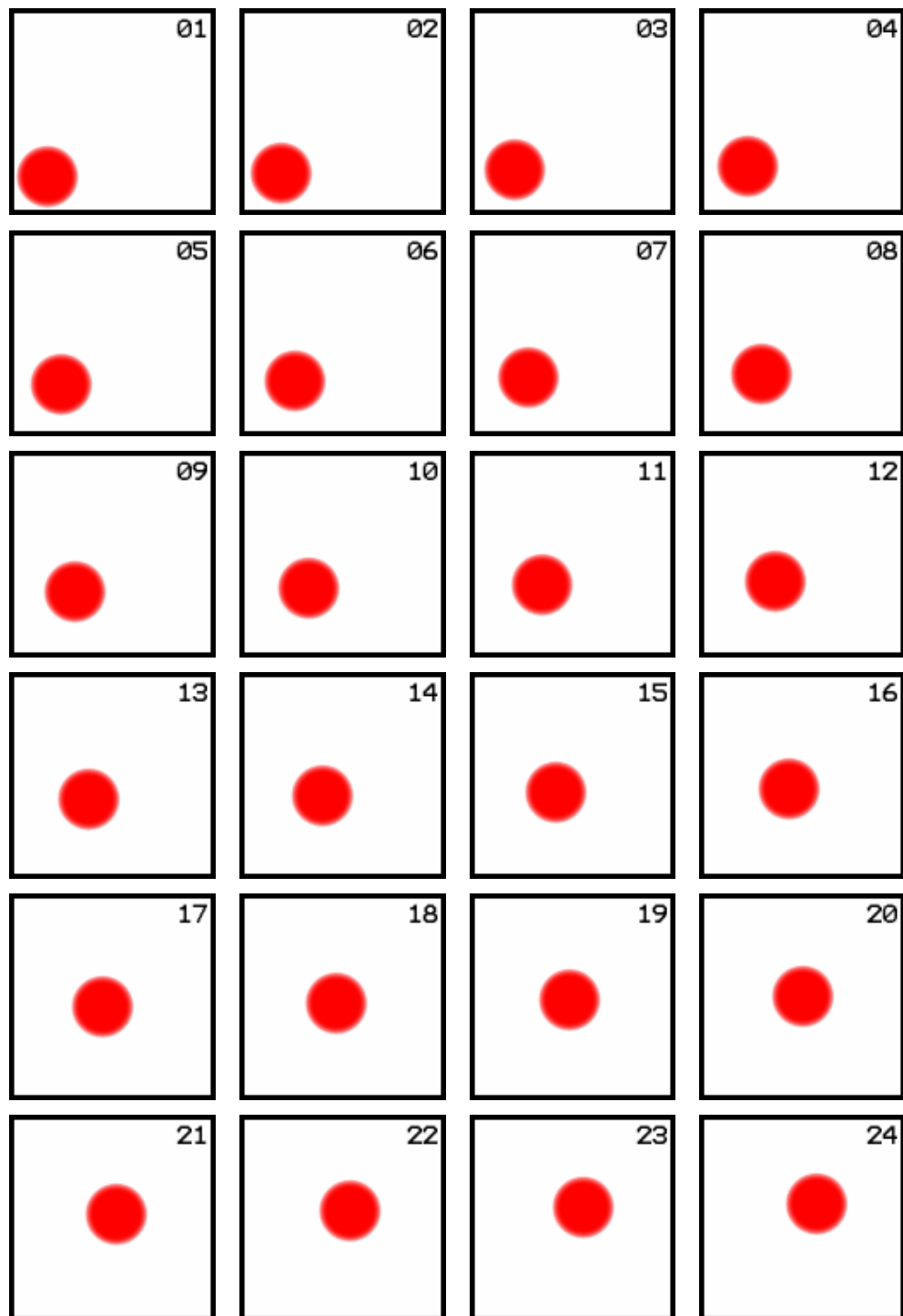


Figure 5.13 Movement of an object in twenty four frames

First image has a red ball positioned nearly left bottom corner and moved to right and up by two pixels at each time frame. Assume that you have an application that representing each image arbitrarily from starting first indices to the twenty fourth one. The time to show each image is must set to 1/24 seconds approximately 42 milliseconds. You will see that the red ball is moving to the right corner smoothly. Here is a sample code snippet:

```
for (int index = 0; index < this.ImageList1.Images.Count; index++)
{
    this.pictureBox1.Image = this.ImageList1.Images[index];
    System.Threading.Thread.Sleep(41);
}
```

Note that 41 millisecond is used as a delay interval, because of using an algorithm like this. We should loose 1 millisecond between thread switches according to underlying operating system and hardware performance.

Familiar with this, scenario manager in weendigo must decide ach object position according to timeline. Nowadays, most of computer video has prepared with thirty frames at a second for best view. A more sensitive frame count at a second is useless and not need since human eye capability.

Scenario management is done in a separate project named as “Scenario Manager”. All classes used in scenario management are derived by BaseCrew abstract class. The class diagram is given below:



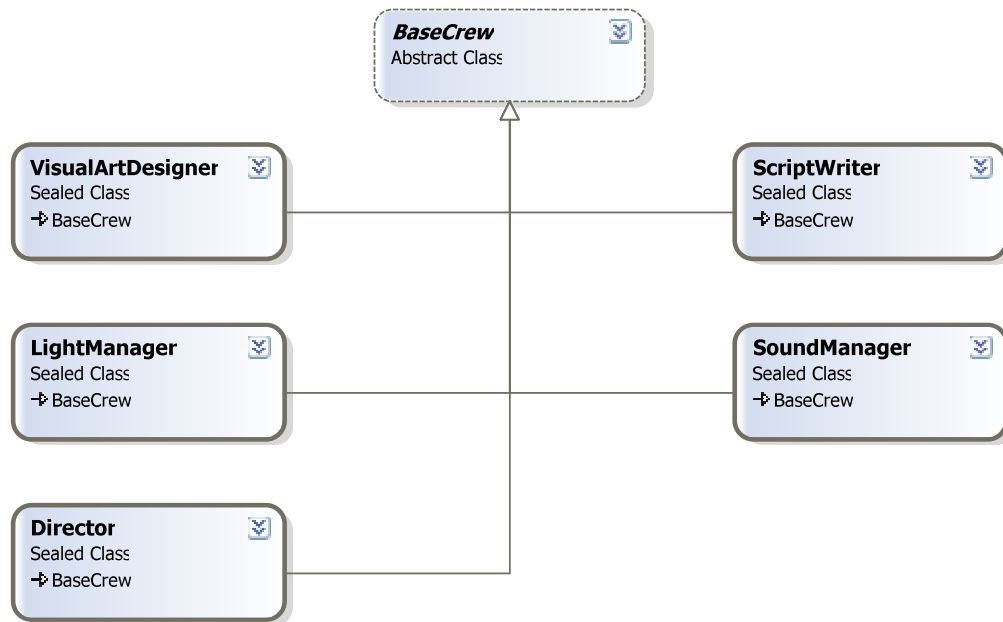


Figure 5.14 Class diagram of the classes used in scenario management

- **Virtual Art Designer** is not implemented yet. It is reserved for future versions and intended to manage special visual effects.
- **Light Manager** is not implemented yet. It is reserved for future versions and intended to manage lights on each scene.
- **Director** is implemented to manage all other crews and also itself.
- **Script Writer** is implemented to manage scenario according to user decisions at design time.
- **Sound Manager** is not implemented yet. It is reserved for future versions and intended to manage sounds used in video and synchronize sounds with video.

BaseCrew is an abstract class and enforces inheritors to implement a method to do their tasks. This class details are given below for clarity:

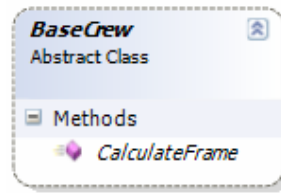


Figure 5.15 Base Crew class interface

ScriptWriter is the class to extract frames from an existing scenario. This extraction process involves linearly interpolation of each object between time intervals. This interpolation is not specific for positioning objects also includes velocity interpolation which is also known as acceleration. ScriptWriter class details are given below. Note that this class implemented using singleton design pattern.

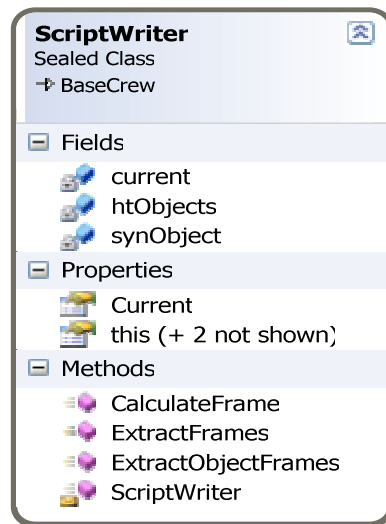


Figure 5.16 Script Writer class interface

Director class loads scenario, builds frame scenes and manages all other crews. By the way, there is only one crew ScriptWriter which is available to use.

A film is built up film scenes and a film scene is built up role players and scene environment which is known as scenery. In weendigo implementation, a film is described with a collection of film scenes. A film scene contains film scene frames. Each film scene frame has the scene object references.

A scene object is implemented inside SceneObject class. Any design time object (Weendigo Static Object, Weendigo Animation Object, Weendigo Text Object) is mapped to a single instance of this class. Any instance of this object contains over all information in design time, object size, position, type and any other feature. This class details are given in the figure below.

A series of scene objects are defined in SceneObjectCollection class. This class inherits CollectionBase class and allow add or remove scene objects at run time. This class details are given in the figure below.

A scene frame contains references for the objects in scene and what changes will be done on them. A scene frame contains scene frame object list and scene frame objects. This object is an entity object and does not perform any modification.

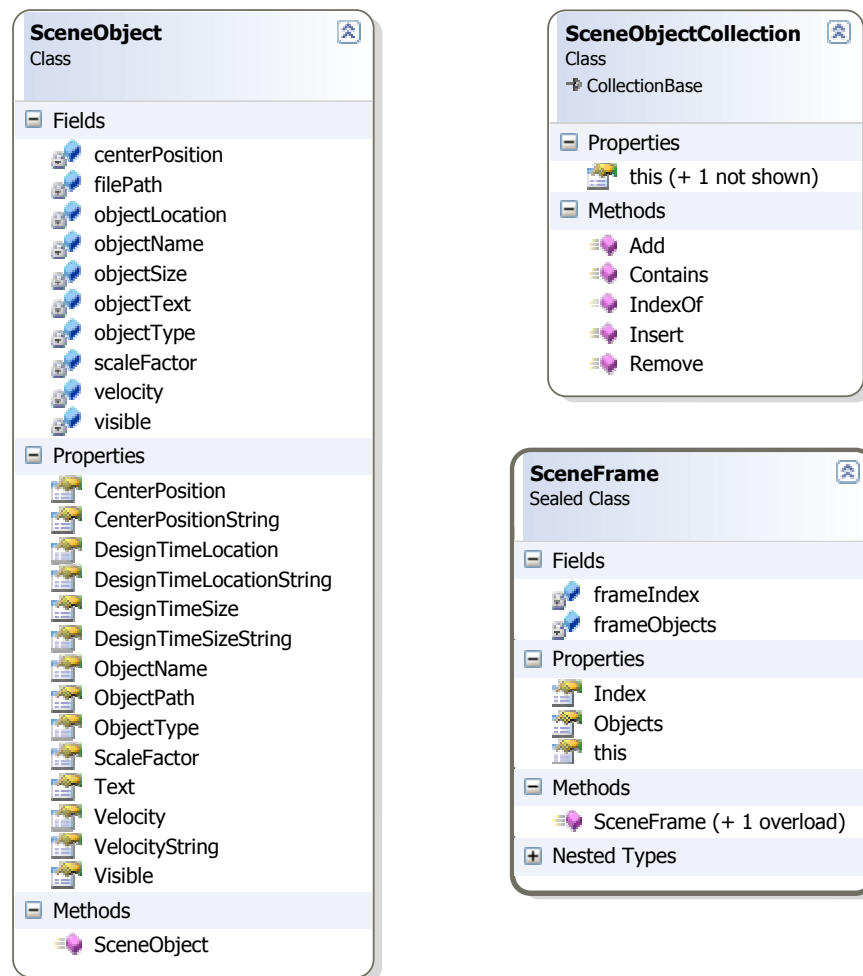


Figure 5.17 Scene Frame, Scene Object, and Scene Object Collection class interfaces

Frame Object contains references for the objects inside a scene frame and new calculated values for their common properties which are set at design time. Frame object collection is a class to hold a series of frame object and allow add or remove at run time. Frame object and frame object collection classes implement the ICloneable interface to support cloning. Clone operation is an important need for extracting whole frames and frame object from an optimized scenario.

Film Scene, Scene Frame Collection, Frame Object and Frame Object Collection class details are given below for clarity.

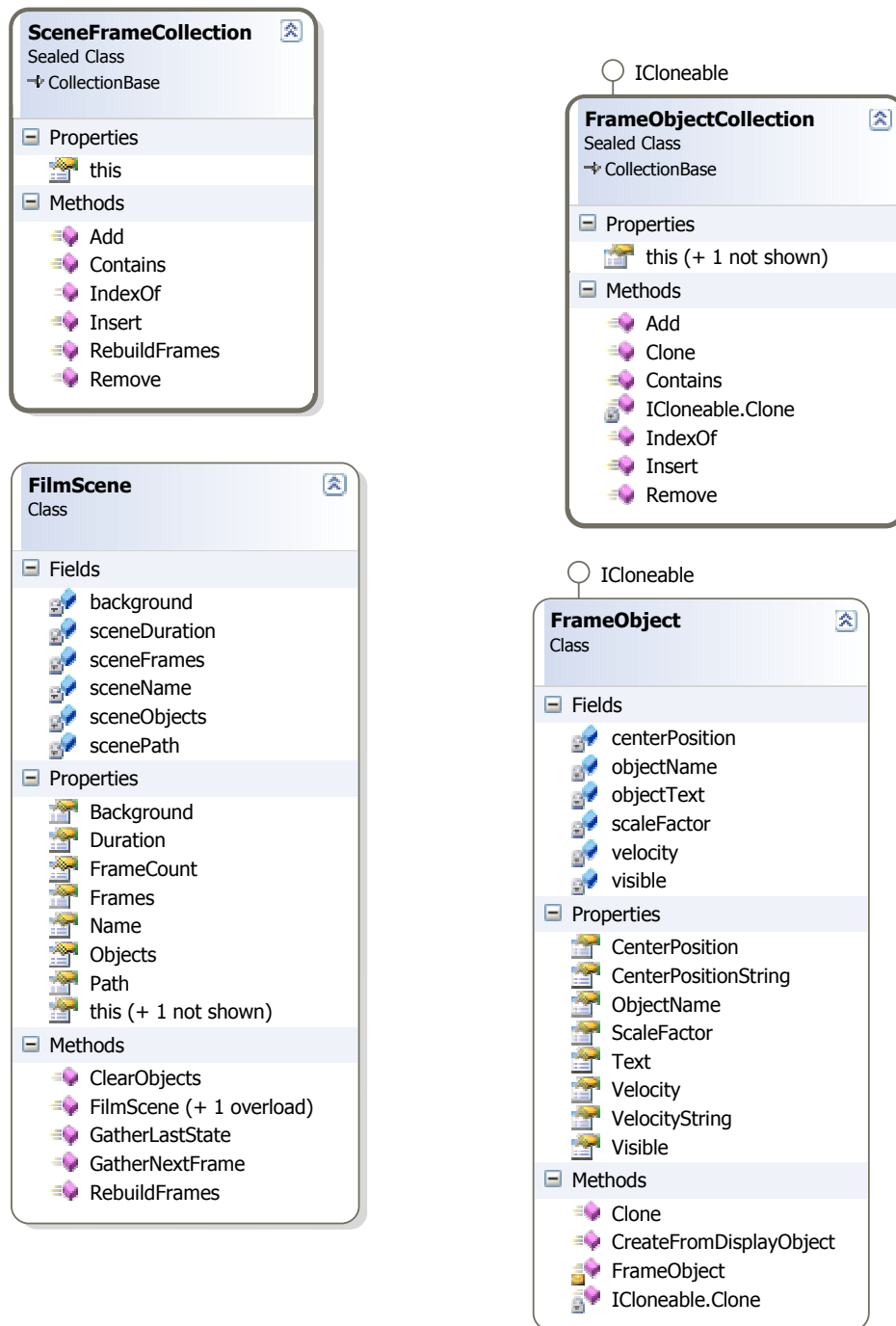
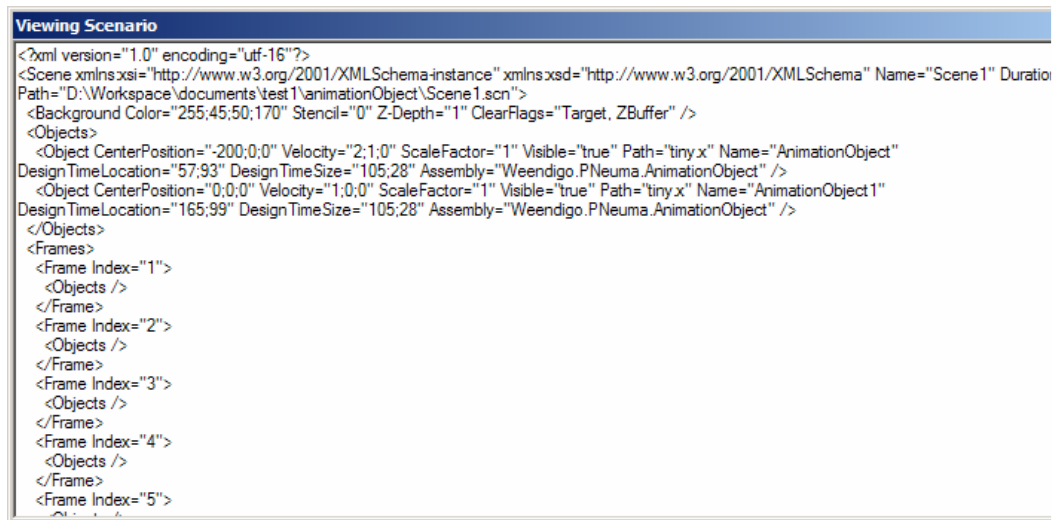


Figure 5.18 Scene Frame Object, Film Scene, Frame Object, Frame Object Collection class interfaces

All of these objects are serializable from an xml document and each scene file is serialized to these objects to construct a film scene. Additionally it is possible to view scene at design time but modifying XML serialized is not permitted. Designing scene using Weendigo editor is more simple and recommended for referential integrity between scene objects and frame objects. User must choose Play → View Scenario to view scenario in XML form at design time.



```

Viewing Scenario
<?xml version="1.0" encoding="utf-16"?>
<Scene xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" Name="Scene 1" Duration
Path="D:\Workspace\documents\test 1\animationObject\Scene 1.scn">
  <Background Color="255;45;50;170" Stencil="0" Z-Depth="1" ClearFlags="Target, ZBuffer" />
  <Objects>
    <Object CenterPosition="-200;0;0" Velocity="2;1;0" ScaleFactor="1" Visible="true" Path="tiny.x" Name="AnimationObject"
Design TimeLocation="57;93" DesignTimeSize="105;28" Assembly="Weendigo.PNeuma.AnimationObject" />
    <Object CenterPosition="0;0;0" Velocity="1;0;0" ScaleFactor="1" Visible="true" Path="tiny.x" Name="AnimationObject 1"
Design TimeLocation="165;99" DesignTimeSize="105;28" Assembly="Weendigo.PNeuma.AnimationObject" />
  </Objects>
  <Frames>
    <Frame Index="1">
      <Objects />
    </Frame>
    <Frame Index="2">
      <Objects />
    </Frame>
    <Frame Index="3">
      <Objects />
    </Frame>
    <Frame Index="4">
      <Objects />
    </Frame>
    <Frame Index="5">

```

Figure 5.19 Active scenario can be viewed via choosing View Scenario from Play menu.

## **CHAPTER SIX**

### **PNEUMA**

Pneuma is the name of the graphics engine embedded in Weendigo implementation to provide abstraction on graphics implementation details. Therefore, Pneuma performance and success affects directly Weendigo performance and success.

#### **6.1 Pneuma Design**

A graphics engine is the core component of a computer or video game or other interactive application with real-time graphics. It provides the underlying technologies, simplifies development, and often enables the application to run on multiple platforms. A graphics engine only deals with rendering process. Rendering is the final process of creating the actual 2D image or animation from the prepared scene. Hence Weendigo is a tool to create 3D animation films; graphics engine is the one of the core components. Pneuma has the meaning of vital spirit; this is why Weendigo Graphics engine is called as Pneuma.

Most often, graphics engines are built upon a graphics API such as Direct3D or OpenGL which provides a software abstraction of the GPU or video card. Pneuma is built on Microsoft DirectX API. One of the important requirements of a graphics engine is performance. Graphics engines are generally compared with their performances on a specific hardware. Each lines of code become important, when we are talking about performance like graphics engines. Graphics hardware manufactures are designing their own graphics engine which is specific for the hardware they are producing. Currently, these kinds of products not ready for sale. Also, this would be an unwanted approach if each defines their standards. There must be consortium to define these graphics engine's standards like W3C (World Wide Web Consortium).

Weendigo implementation contains a software graphics engine design as used to be. By the way, algorithms used in graphics engines are generally same. By the way, graphics engines differ with their design. Class design and programming language decision becomes more important. Weendigo is fully implemented with Microsoft C# 2.0. There is no exception for graphics engine. Implementing graphics engine in native C++ (might using naked functions for performance considerations) and exposing a managed interface for interaction with graphics engine is an approach. But this brings interoperation and marshalling cost.

The relation between objects can be defined using the following approaches in UML standards:

- Association
- Aggregation
- Composition
- Generalization

Generalization can be done by using inheritance but dynamic dispatch and dynamic types cost too much effort at run time. A method becomes “virtual” at the highest level in the hierarchy in which it is declared as virtual. If static dispatch is used, the compiler can determine at compile time which function definition will be used where. The compiler can hard code this definition (or link to it) into object code. If dynamic dispatch is used, the runtime has to do a query when the method is call since it can not tell what method is being called. Dynamic dispatch involves overhead which may be non-trivial, especially for classes that are used a lot.

Graphics engine classes are called frequently. And the call count of render method in a time interval is tightly related with the performance. Graphics engine main class is implemented with using the whole approaches given above by considering performance issues. Main class is implemented with singleton design pattern which guarantees there is no other instance in the process scope.



In Pneuma implementation an event based (delegated) approach is used. A graphics engine should be able to detect and report device lost events. A device lost event is fired when rendering target is lost generally render target window lost focus. An event is fired when device lost, this implemented in Direct3D. By the way, while considering performance, raising an event could be costly. An event is already fired, raising another event would increase this cost. To overcome this cost, delegates are used. An interface called “IDeviceCreation” is defined. Also calling render and frame move methods at runtime could be costly. Using same approach an interface called “IFrameworkCallback” is defined.



Figure 6.1 IDeviceCreation and IFrameworkCallback interfaces

A built in device handler and device event argument class are already defined for reporting device events to multiple recipients.

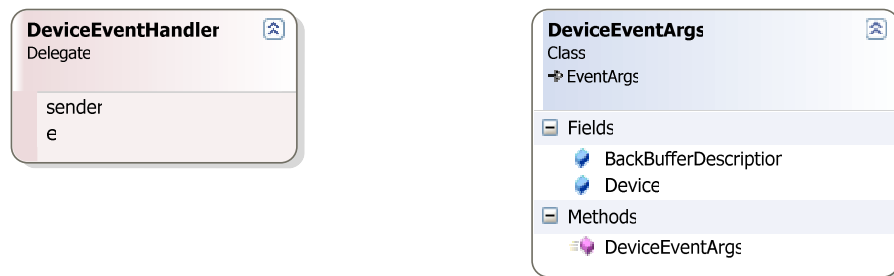


Figure 6.2 DeviceEventHandler delegate and DeviceEventArgs class interfaces

Also for the render target window a standard Win32 window callback function is defined as follows:

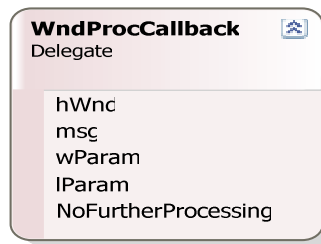


Figure 6.3 WndProcCallback  
delegate interface

Also a graphics engine should have a timer. This timer usually used for measuring statistical data. But also used for reporting the time consumed between calls. This consumed time is important for a game application and in this case, this is important for Weendigo to prepare scene for the moment according to predefined scenario. Data needed by timer is defined as follows:

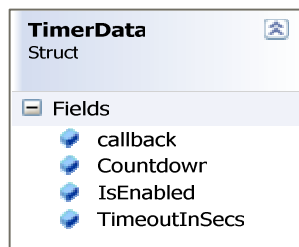


Figure 6.4 TimerData interface

“FrameworkTimer” class is defined to measure consumed time using “TimerData” struct. This timer has a generic interface like Windows based timers. Same operations are supported like start, stop, advance, and reset. This class details are given below for clarity:

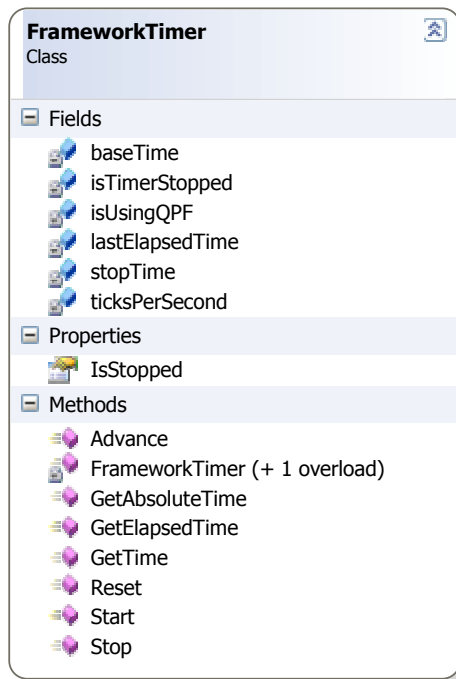


Figure 6.5 Framework Timer class interface

Although related Pneuma classes are implemented with using singleton design pattern. Pneuma needs to hold too much information thus make Pneuma classes stateful contrast design pattern which requires classes to be stateless. Device information and callback method references are hold in a single class called “D3DFrameworkData”. This class contents are built up at startup time and minor changes are done at runtime except device events (device lost events, switch between full screen and windowed mode, etc). This class details are given below for clarity (note that there is not any method, this class does nothing only holds graphics engine’s state):

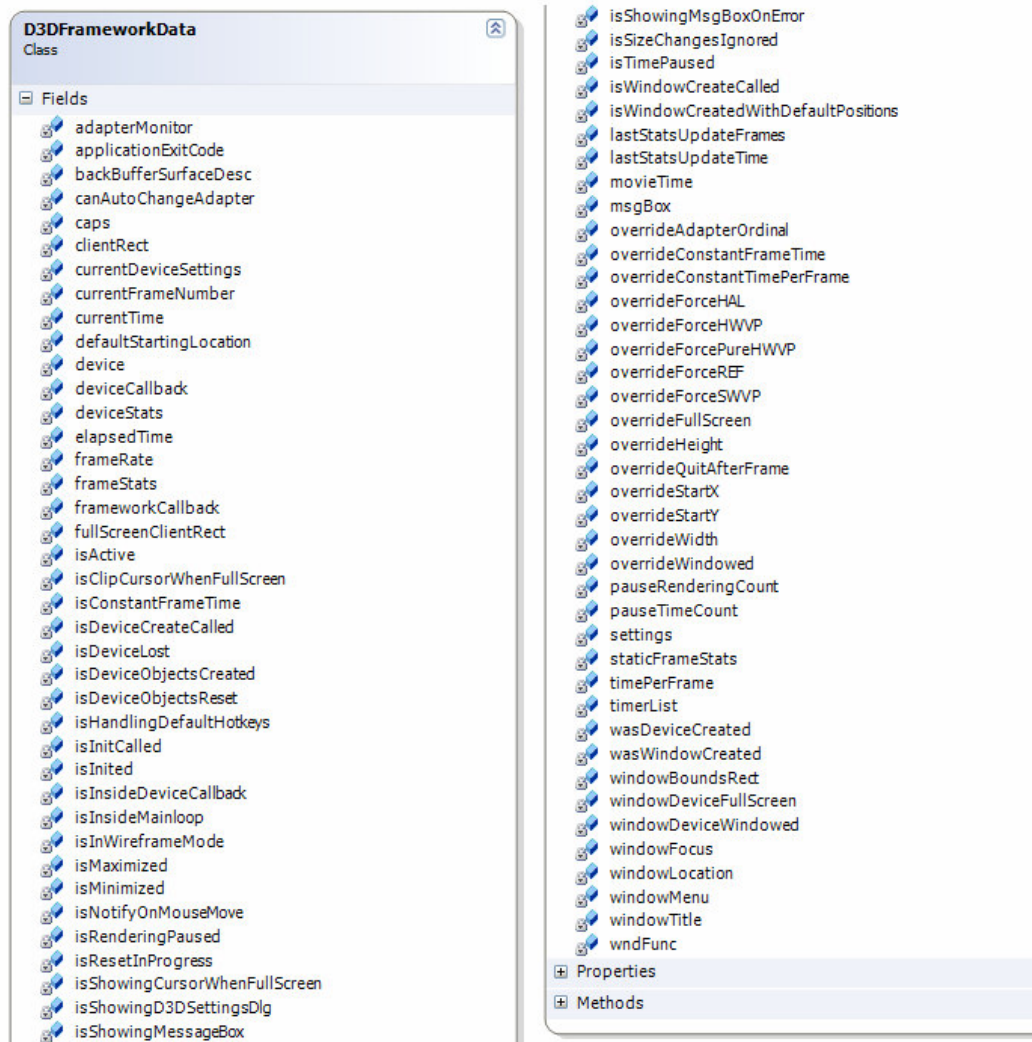


Figure 6.6 D3Dframework Data class interface

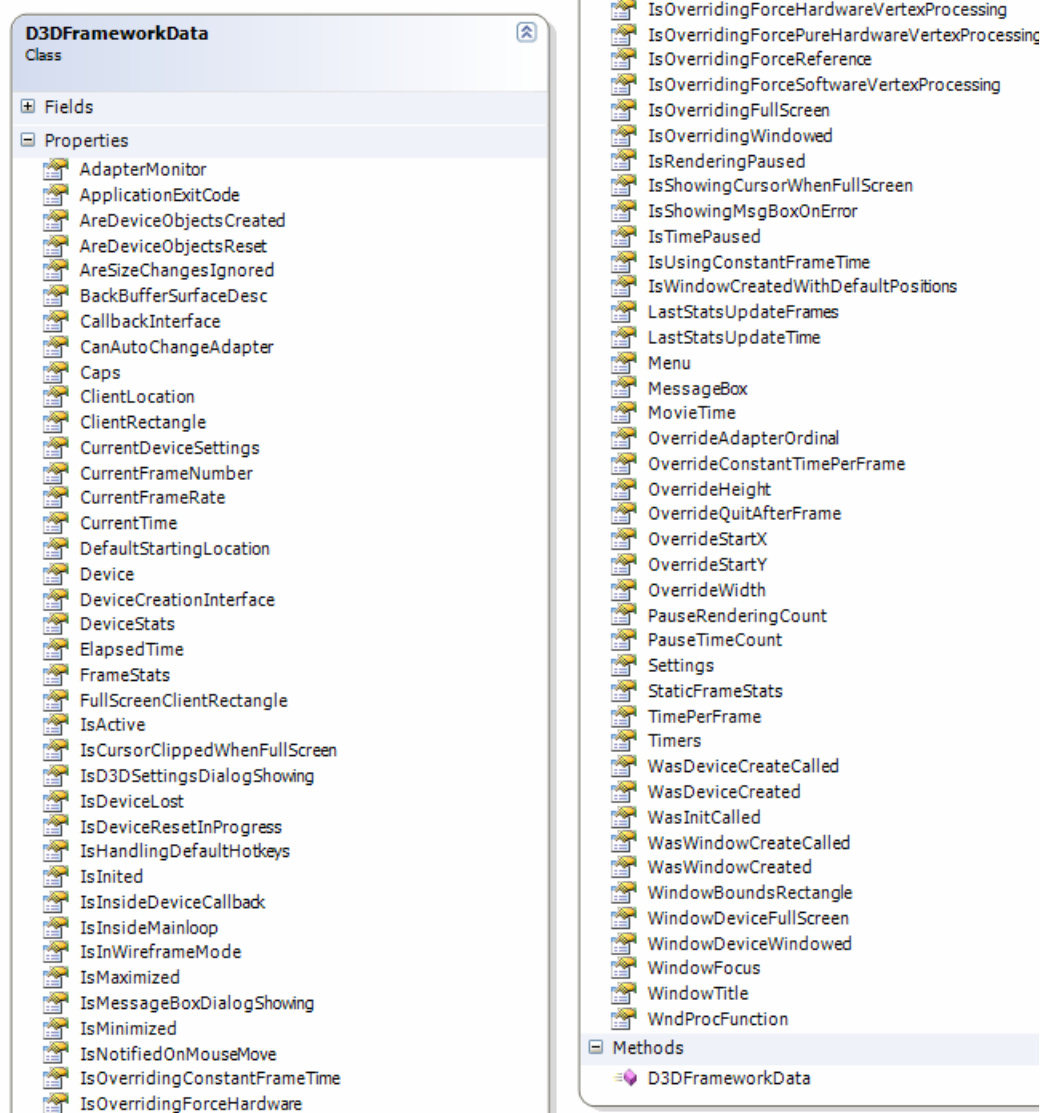


Figure 6.7 D3DFrameworkData class interface (continued)

Main operations are done in D3DFramework sealed class. Also some statistical data is stored in this class and available for use Weendigo IDE like frame stats, FPS and movie stats. Class details are given below for clarity:

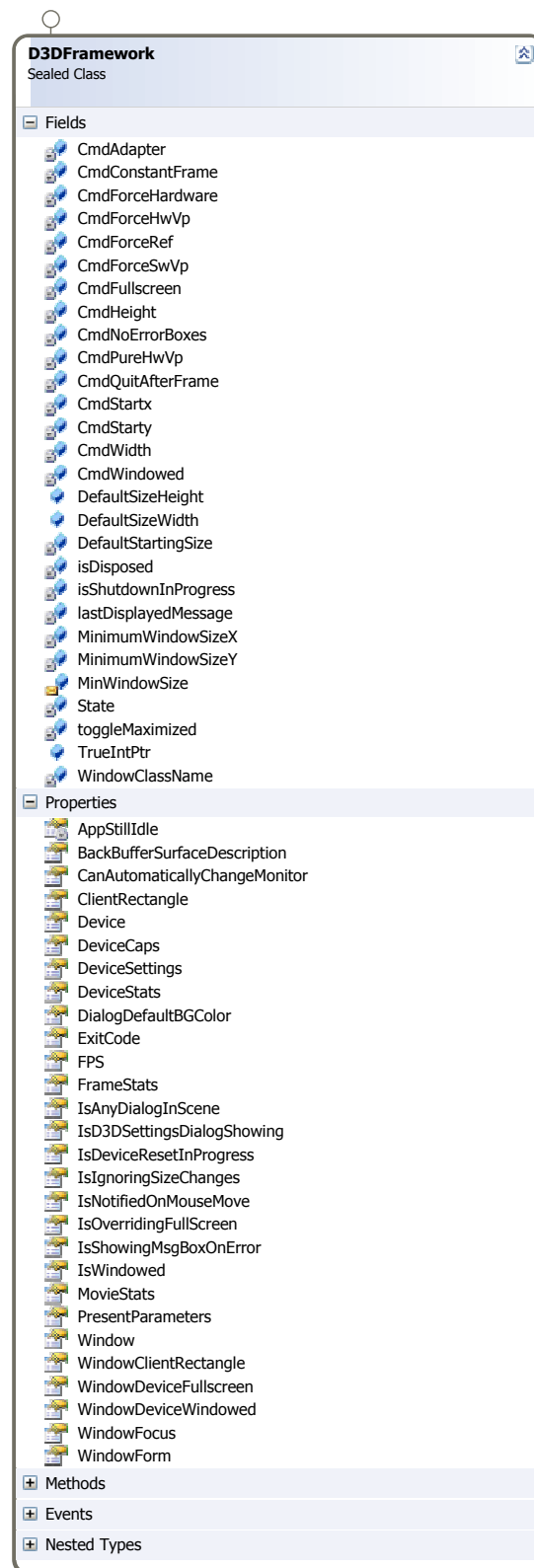


Figure 6.8 D3DFramework class interface

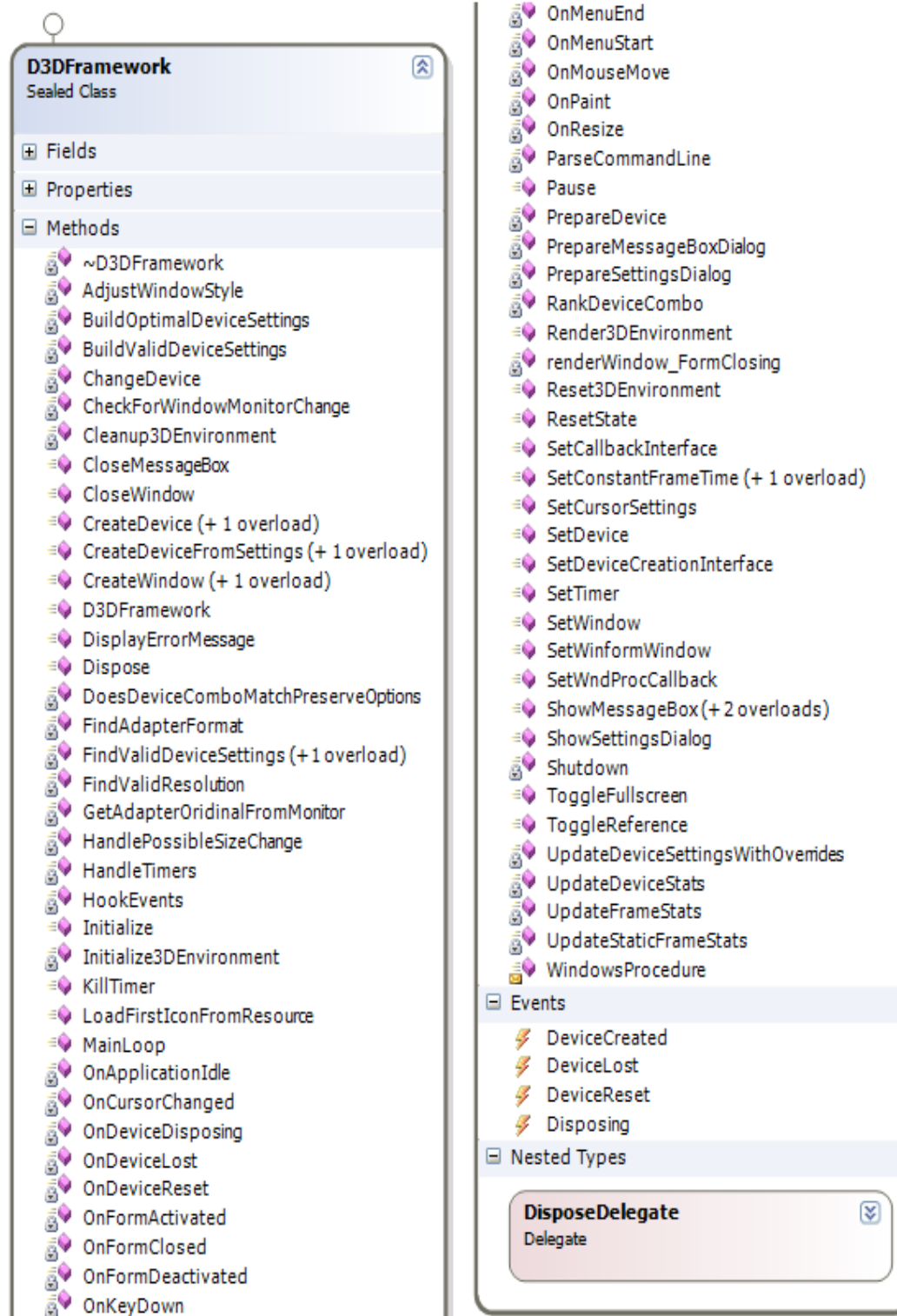


Figure 6.9 D3DFramework class interface

Inside `Render3DEnvironment` method, some preconditioned checks are done. Here is a list of the preconditions:

- Device resetting is in progress,
- Device is lost or rendering is paused
- Render target window is active (if window is minimized or paused yields CPU time to other processes)

There might be several exceptions raised in this method. One of the important one is `DriverInternalErrorException` which reports graphics driver has an internal error. This exception can be raised when `Present` method of device called which forces graphics card to flush memory to screen. When this exception is thrown the application can do one of the following:

- End, with the pop-up window saying that the application cannot continue because of problems in the display adapter and that the user should contact the adapter manufacturer.
- Attempt to restart by calling `Device.Reset`, which is essentially the same path as recovering from a lost device. If `Device.Reset` throws the `DriverInternalErrorException`, the application should end immediately with the message that the user should contact the adapter manufacturer.

Pneuma attempts the path of resetting the device.

## 6.2 Hardware Enumeration

A 3D application's performance is measured mostly with its performance and reality. Underlying hardware specification directly affects application performance. By the way, using hardware in a best way will lead application to gather best performance and view. Microsoft DirectX provides a hardware abstraction layer, all interaction between applications and hardware is performed via this layer. Nevertheless, detecting underlying hardware and deciding best configuration must be done by application as a preliminary obligation due to mentioned performance reasons.



Hardware enumeration is a time consuming and asynchronous process. Enumerating and retrieving capabilities of hardware is implemented with using asynchronous callbacks. This callback method calls uses Microsoft DirectX to query device information. Following hardware properties and capabilities can be queried using DirectX API:

- **Adapter Count:** number of display adapters installed on the system.
- **Adapter Identifier:** description of the physical display adapters present in the system.
- **Adapter Display Mode:** the currently in use display mode of the adapter.
- **Adapter Mode Count:** number of display modes available on this device.
- **Adapter Modes:** query the device to determine whether the specified adapter supports the requested format and display modes
- **Adapter Monitor:** the handle of the monitor associated with device.
- **Device Multisampling Capability:** if a multisampling technique is available with device.
- **Device Capabilities:** device specific additional capabilities.

Detecting these capabilities are representing these capabilities are implemented using several classes. Sorting available display modes is done with DisplayModeSorter class. This class implements IComparer interface using templates in C# 2.0. Simply compares two distinct display modes by using width and height information. If the first one is a larger display mode returns 1, if the second on is a larger display mode return -1, if both are equal returns 0. Simple class definition is given below;

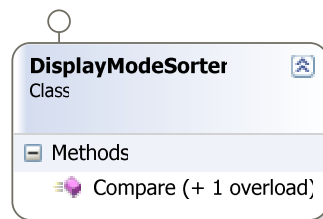


Figure 6.10 Display Mode Sorter class interface

And here is the class implementation:

```
public class DisplayModeSorter : IComparer<DisplayMode>, IComparer
{

    #region IComparer<DisplayMode> Members
    /// <summary>
    /// Compare two display modes
    /// </summary>
    public int Compare(DisplayMode d1, DisplayMode d2)
    {
        if (d1.Width > d2.Width)           return +1;
        if (d1.Width < d2.Width)           return -1;
        if (d1.Height > d2.Height)         return +1;
        if (d1.Height < d2.Height)         return -1;
        if (d1.Format > d2.Format)          return +1;
        if (d1.Format < d2.Format)          return -1;
        if (d1.RefreshRate > d2.RefreshRate) return +1;
        if (d1.RefreshRate < d2.RefreshRate) return -1;
        // They must be the same, return 0
        return 0;
    }
    #endregion

    #region IComparer Members
    public int Compare(object x, object y)
    {
        if (x is DisplayMode && y is DisplayMode)
        {
            return Compare((DisplayMode)x, (DisplayMode)y);
        }
        throw new WeendigoException("Only DisplayMode is supported");
    }
    #endregion
}
```

Adapter information is defined with the class AdapterInformation. Any instance of this class describes an adapter which contains a unique adapter ordinal that is installed in the system.

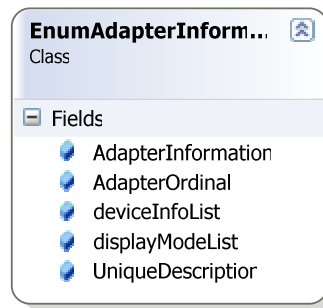


Figure 6.11 EnumAdapterInformation  
class interface

Device information is defined with the class DeviceInformaiton. Any instance of this class describes a Direct3D device that contains a unique support device type.

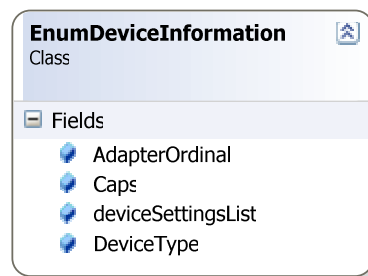


Figure 4.12 EnumDeviceInformation  
class interface

A depth/stencil buffer format that is incompatible with a multisampling type is defined with the class EnumDepthStencilMultisampleConflict.

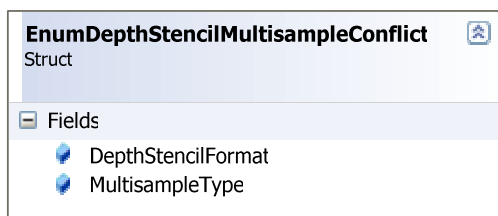


Figure 4.13 EnumDepthStencilMultisampleConflict  
class interface

EnumDeviceSettingsCombo is the class describing device settings that contain a unique combination of adapter format, back buffer format, and windowed that is compatible with a particular Direct3D device and the application.

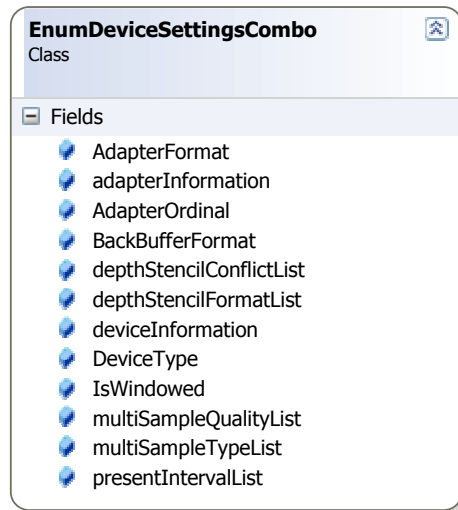


Figure 6.14 EnumDeviceSettingsCombo class interface

Pure hardware enumeration is done with the class `HardwareEnumeration` which is implemented using singleton design pattern. This class enumerates available Direct3D adapters, devices, modes, etc. This class definition is given below for clarity:

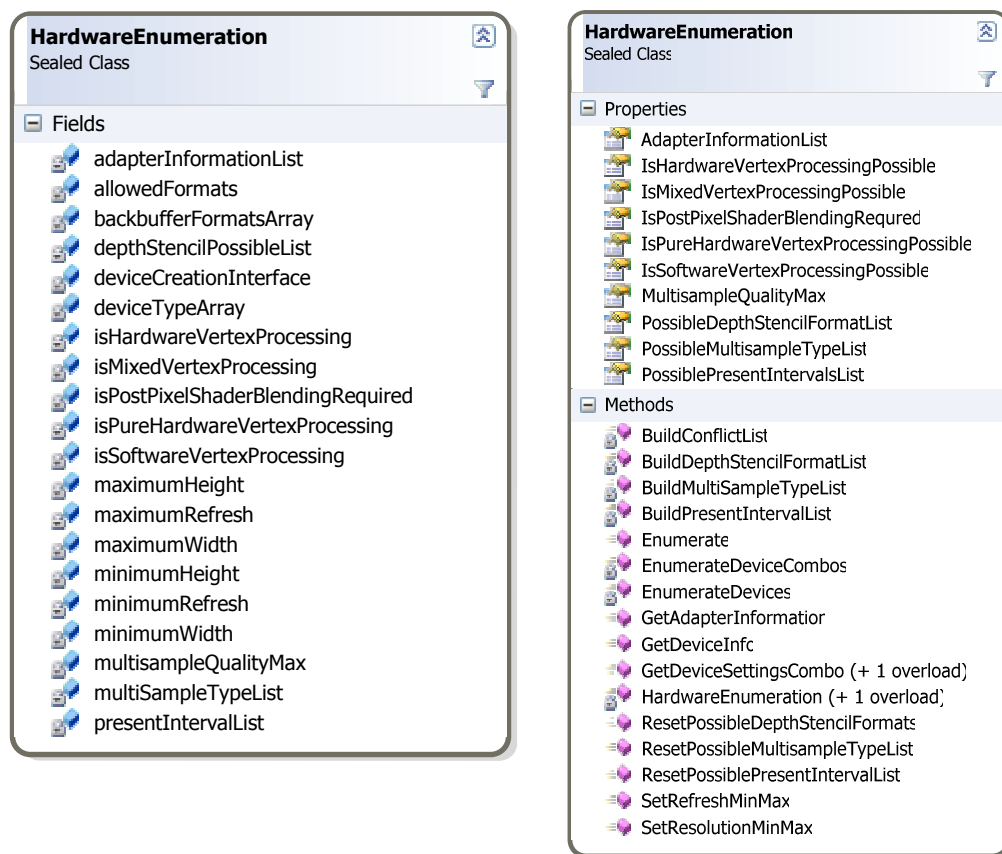


Figure 6.15 Hardware Enumeration class interface

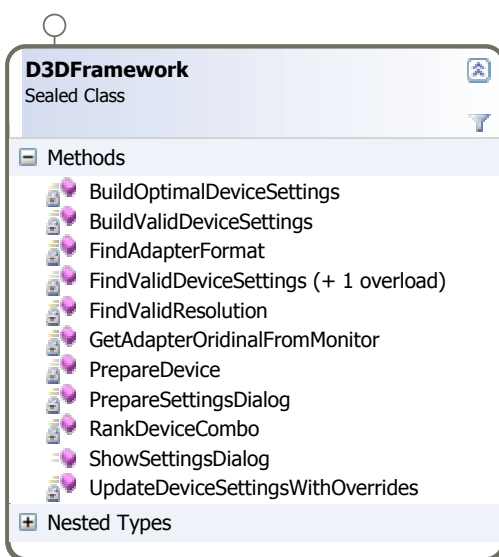


Figure 6.16 D3Dframework class interface

Pneuma framework directly uses this enumeration classes and types to find valid device settings. Building optimal device settings is the process to find out the minimum settings to view the scene with the possible best view. For instance, display mode is set 640x480 for full screen mode, and for the windowed mode this setting is set to active windows display mode.

Finding valid device settings is the process of interpreting current device settings whether that can be visible on the underlying hardware such as display modes and refresh rates. This method finds the best combination of the followings:

- Adapter Ordinal
- Device Type
- Adapter Format
- Back Buffer Format
- Windowed

Given what is available on the system and the match options combined with the device settings input. This combination of settings is encapsulated by the EnumDeviceSettingsCombo class.

Settings Dialog is prepared using these classes and types. If user changes any of the settings displayed to the user, and clicks “OK”; first of all these settings are tested and applied then.

Following settings are displayed to the user and available to set by user at runtime:

- A list of display adapters available to use
- Render device (Hardware, Reference, Software)
- Windowed mode
  - Clip to device when window spans across multiple monitors
- Full screen mode

- Adapter Format
- Resolution
- Refresh Rate
- Back Buffer Format
- Depth Stencil Format
- Multisample Type
- Multisample Quality
- Vertex Processing
- Present Interval

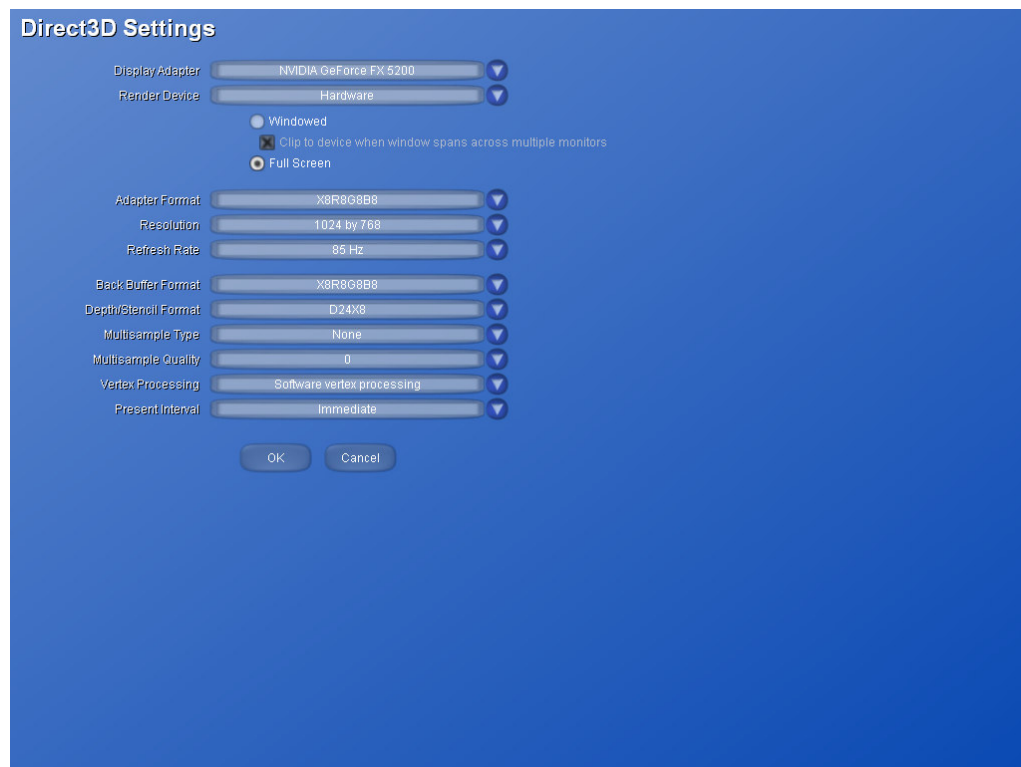


Figure 6.17 Direct 3D Setting at the time of playing movie

### 6.3 Common Controls in 3D Environment

One of the common problems of game and 3D application development is using user controls in a 3D environment. Hence 3D application requires too much CPU and considering performance, most of the 3D applications are developed using low level

programming languages like C, C++ and assembly. It is hard to implement a user control like drop down menu with this low level programming languages. There are several pseudo remedies for this problem. One of the commonly used one is developing user interface with a high level programming language and interoperating these two programming language. Thus requires knowledge of two distinct technologies and also interoperation cost can not be ignored.

In Pneuma implementation, it is a requirement to serve a framework for user interaction. Perhaps, there is no need to have a user control in a movie but it is important to have a preview of movie and controlling 3D graphics card settings and facilitate these settings to gather best view.

Weendigo 3D user controls are implemented in C# under “Weendigo.PNeuma” namespace. BaseControl is the name of the abstract base class which has the common properties of a user control. This class is the base class of all Weendigo controls and has the following properties:

- **CanHaveFocus**, ability to gather focus
- **ControlType**, One of the supported control types
- **Height**, height of the control in the units of pixels.
- **Hotkey**, a key-combination to activate, user, or identify this control can be different for each instance of same user control type.
- **ID**, unique identifier of the control inside a parent control
- **IsEnabled**, specifies or determines whether this control is functional. When set to false, the control appears dimmed, preventing any input from being entered in this control to be processed.
- **IsVisible**, visibility of this control, effects child controls recursively
- **Left**, left position of the control inside parent control in the units of pixels
- **Parent**, reference to parent control which hosts this control inside.
- **Top**, top position of the control inside parent control in the units of pixels
- **UserData**, control specific user data
- **Width**, width of the control in the units of pixels.



Class details are given below for clarity:

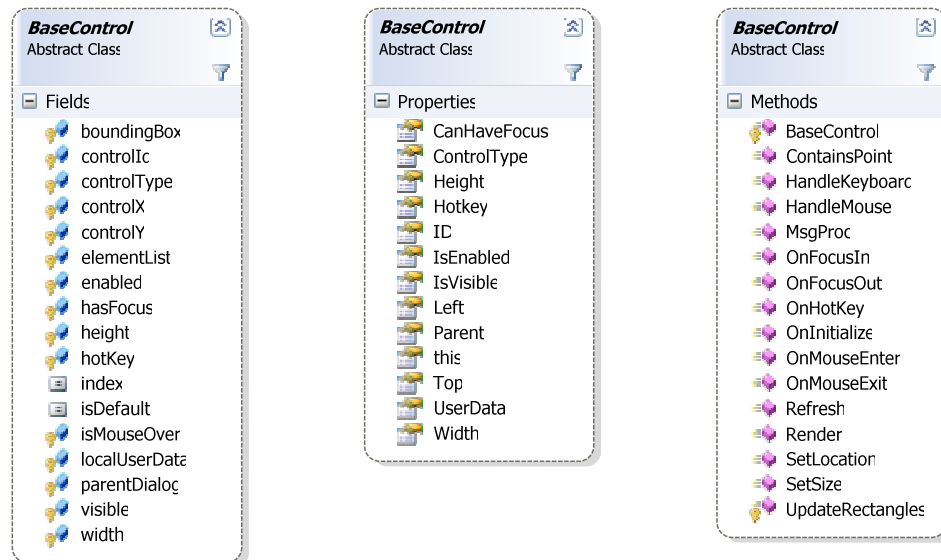


Figure 6.18 Base Control abstract class interface

A list of built in supported user controls are given below:

- Slider has similar interface with track bar control in Win32 environment allows user to seek some value at specified range.
- List box has similar interface with list box control in Win32 environment. A list of name value collection is held and rendered, and allows user to choose one of the listed values.
- Textbox has similar interface with edit box control in Win32 environment, allows user to edit content with full support mouse and keyboard.
- Label has similar interface with label control in Win32 environment, has a read only text.
- Button has similar interface with button in Win32 environment, has a read only text and handles click and submit events, allows user to trigger some event.
- Checkbox has similar interface with checkbox in Win32 environment has a read only associated text and a square which handles click events, allows user to enable/disable some option.

- Radio Button has similar interface with radio button in Win32 environment, has a read only associated text and a square which handles click event, allows user to choose one of the option from grouped multiple radio buttons.
- Combo box has a similar interface with combo box control in Win32 environment. A list of name value collection is held and rendered, and allows user to choose one of the listed values.
- Scrollbar has similar interface with vertical/horizontal scroll bar in Win32 environment, only interact with mouse.

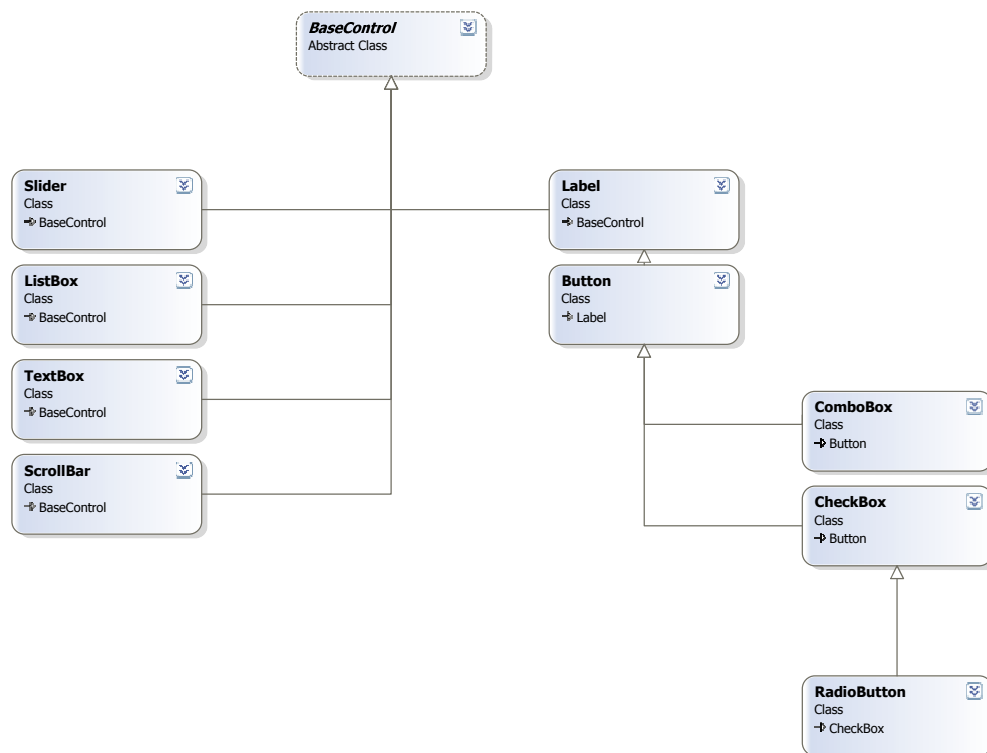


Figure 6.19 Class hierarchy between specified controls

Each class details are given below for clarity:

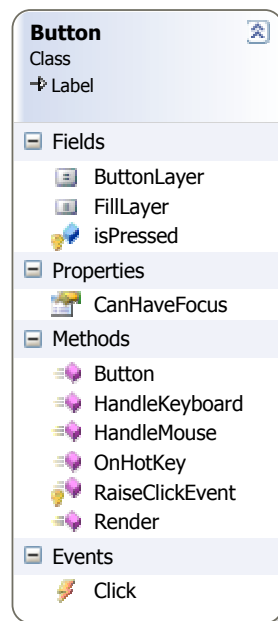


Figure 6.20 Button  
class interface

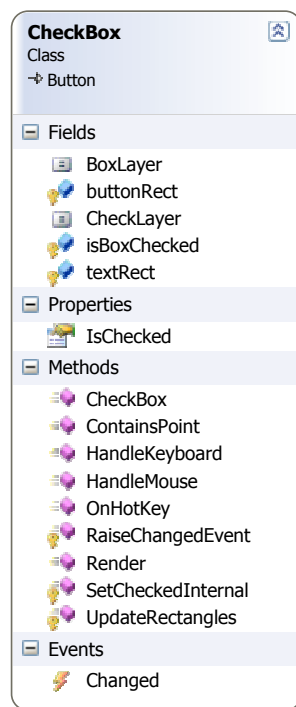


Figure 6.21 CheckBox class  
interface

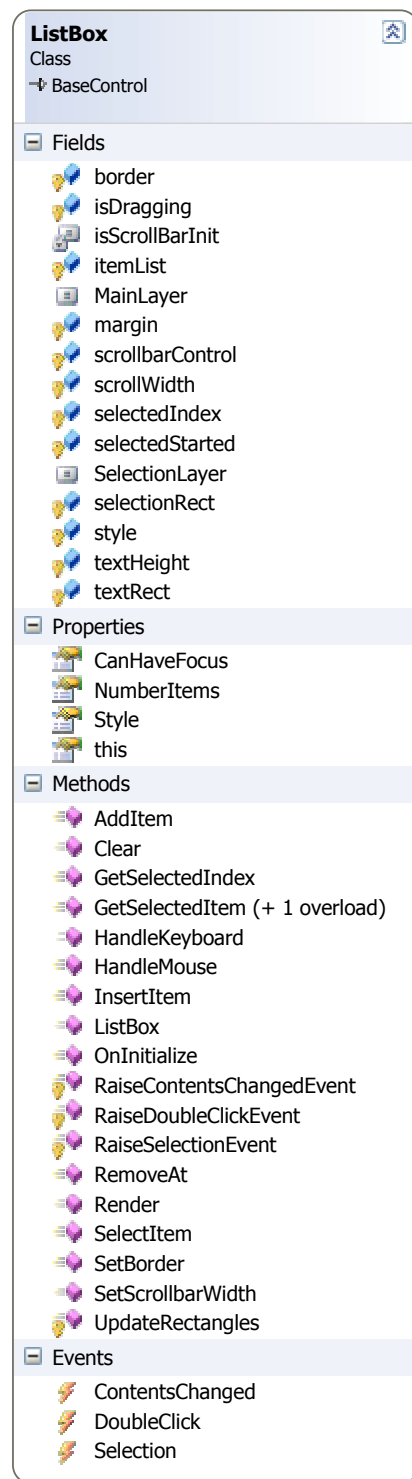


Figure 6.22 ListBox class interface

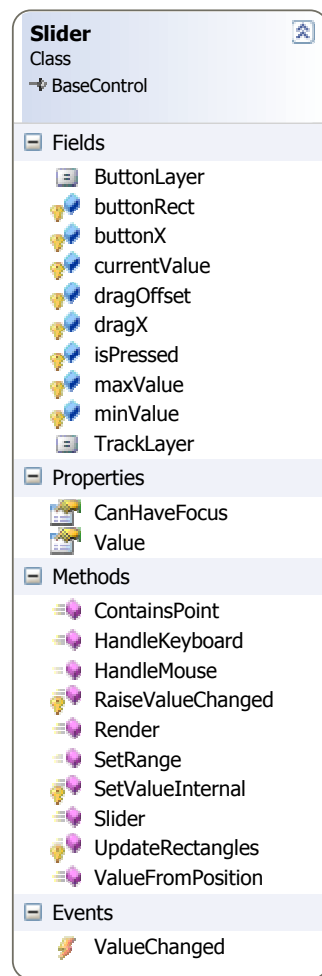


Figure 6.23 Slider class interface

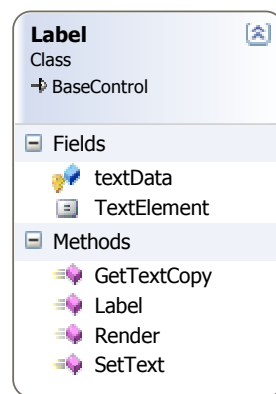
Figure 6.24 Label  
class interface



Figure 6.25 Textbox class interface

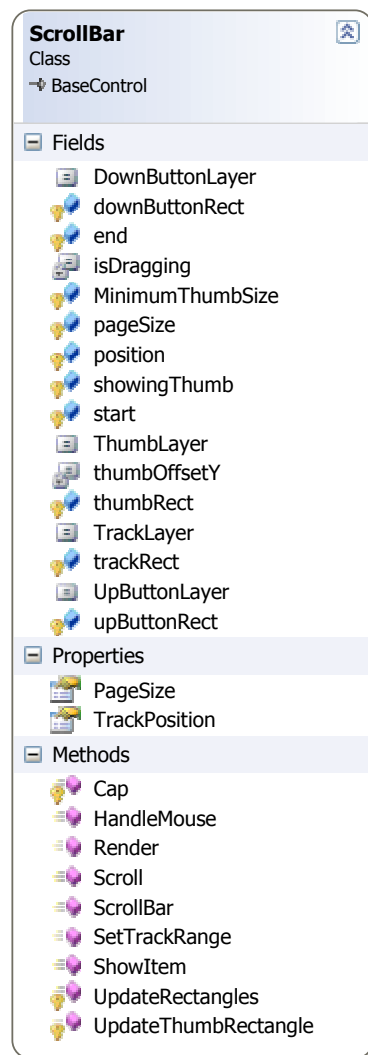


Figure 6.26 Scrollbar class interface

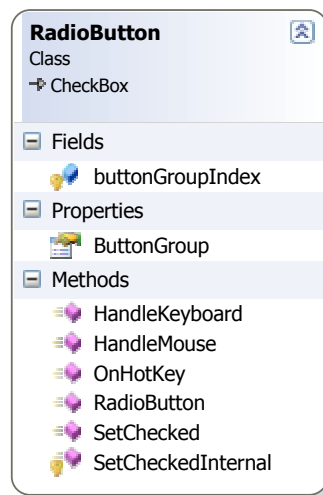


Figure 6.27 Radio Button class interface

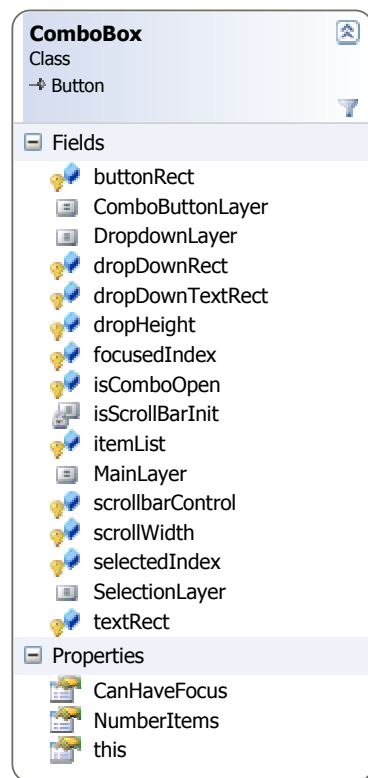


Figure 6.28 Combobox class interface



These user controls are implemented as a requirement of Pneuma (graphics engine). In Weendigo implementation these controls are used while playing prepared movie, and graphic card settings window, and message boxes used in 3D mode.

First snapshot (Figure 4.29) shows button usage while playing movie. When user moves mouse over a button, button's background color is changed to have a 3D effect. As this snapshot taken, mouse was over Toggle Full Screen button.

Second snapshot (Figure 4.30) shows most of the control's usage in a 3D environment. There are eleven combo boxes, two radio buttons, one check box, two buttons, and twelve labels in this screen. This screen is intended to allow user to modify Direct3D settings currently in use. If user clicks button labeled as "OK" all changes are submitted and applied without any confirmation. If user clicks button labeled as "Cancel" all changes are reverted and nothing done. Third snapshot (Figure 4.31) shows combo box usage as a drop down menu.

Fourth snapshot (Figure 4.32) shows a confirmation dialog box like Win32 message box which asks a question to user and perform some action according to user response.



Figure 6.29 Button usage in 3D environment

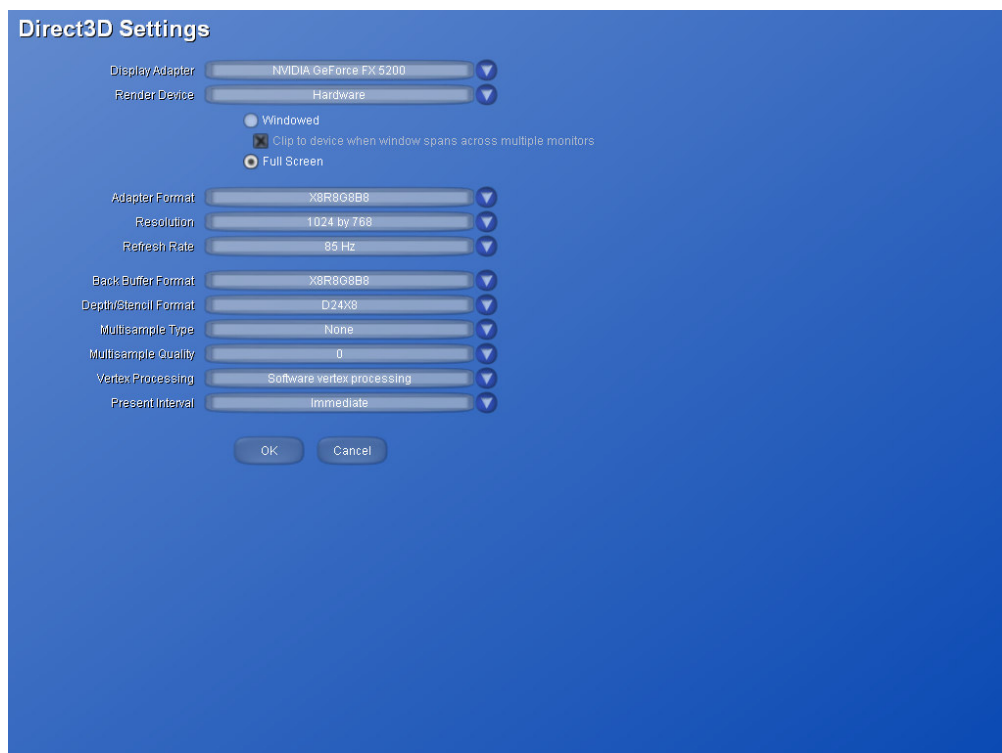


Figure 6.30 Most of the common controls used in a single scene

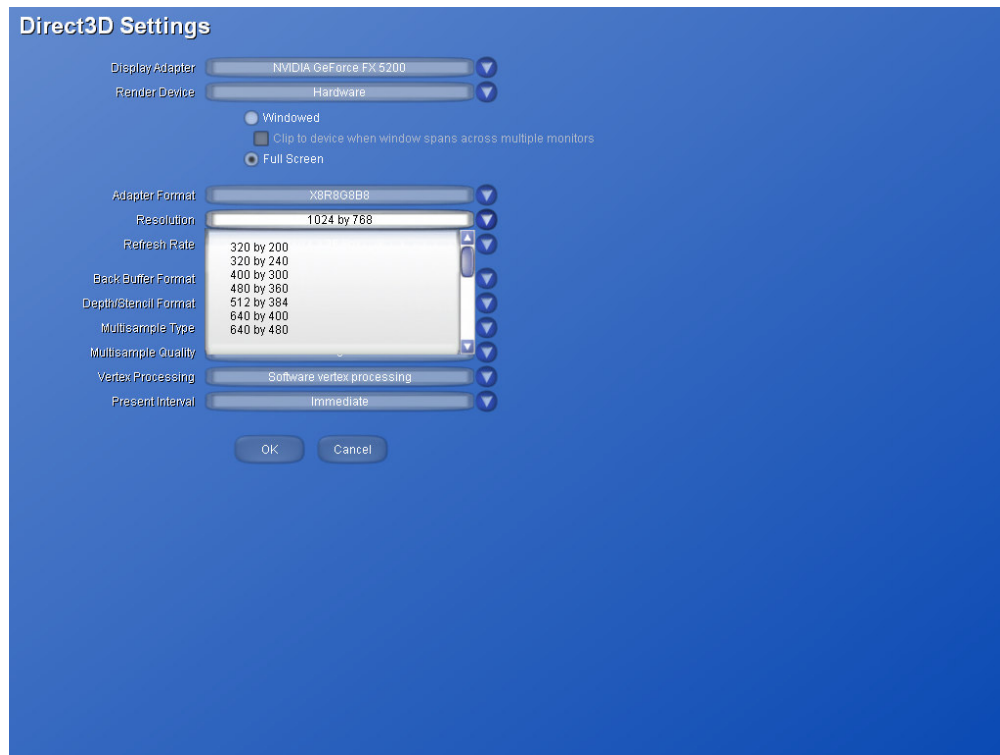


Figure 6.31 Combobox usage

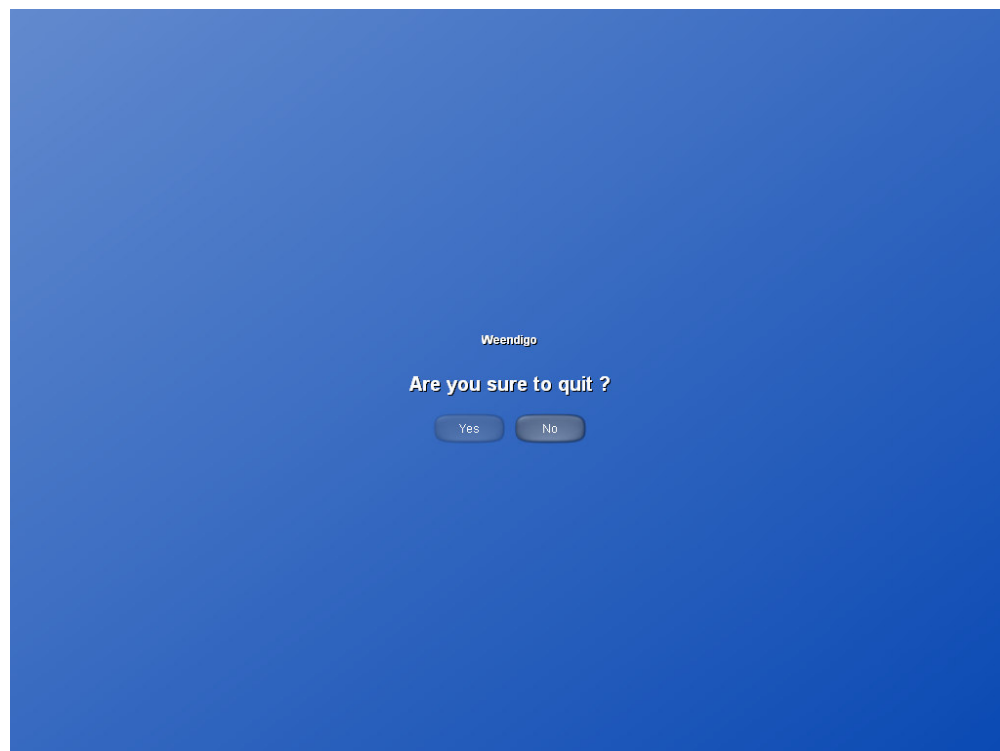


Figure 6.32 A sample confirmation dialog box like Win32 message box

## 6.4 Darken Scene Algorithm

A film consists of scenes added sequentially. Between scene passes there are different tricks to make this pass as a smooth pass. Common and easiest one of these passes is darkening old scene and enlightening new scene. Weendigo, accomplish this smooth pass by using color linear interpolation.

Linear Interpolation is a method that can be used for predicting. Very often something changes over a period of time: an object might change its position; a computer graphic image might change its shape; a population might increase. Linear interpolation allows you to predict an unknown value (position, shape, population, etc.) if you know any two particular values and assume that the rate of change is constant.

Linear interpolation assumes:

- that you know two particular values
- that the process is changing at a constant rate
- that you desire to find an unknown data point

Linear interpolation is a continuous process but it has a start and end point. Interpolation is processed in between these start and end time intervals. We only know the initial state and final state. Since, state transition is a prediction all intermediate states are calculated with a linear formula like following:

$$\text{CurrentState} = \text{InitialState} + (\text{FinalState} - \text{InitialState}) * (\text{Delta})$$

Delta is percentage of processing status. Assume that, as an initial state we have 5, and target is 20. We want to interpolate 5 through 20 in 100 units of time. It is certain that after 100 units of time we will have 20. But in any intermediate state we have different values. Sensitivity is up to your decision. In this example we have sensitivity in 100 units of time. For this instance, we will have a formula like following:

$$\text{CurrentState} = 5 + (20-5) * (\text{Time}/100)$$

Thus formula means that after 20 units of time from the beginning, we will have **8** ( $5 + (20-5) * (20/100)$ ) for this intermediate state. This interpolation process is shown in the following figure:

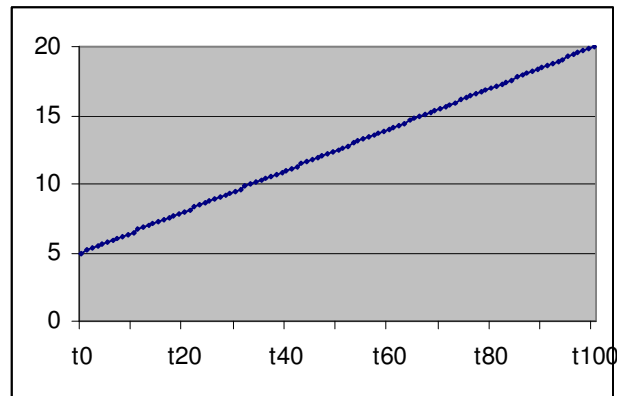


Figure 6.33 Linear Interpolation Graph






















Weendigo includes one second for scene pass delay. First half of this second is consumed for darken scene, and the second half is consumed for enlightening scene. Background colors for subsequent scenes may differ. Therefore, darken scene and enlighten scene operation takes place independently. But both have same algorithm. Linear interpolation is used for calculating new color like following way:

0.5 second is consumed for darken scene, and 0.5 second is consumed for enlighten scene. Current scene color is interpolated linearly to black. For instance, let current scene background color be Red (Alpha=255, Red=255, Green=0, Blue=0). This color will be replaced with color black (Alpha=255, Red=0, Green=0, Blue=0) in 0.5 seconds. Each color element (Red, Green, and Blue) is interpolated independently to new color (black) elements.

After 0.5 seconds, current scene background color (black) is interpolated linearly to new scene color. Let new scene background color be yellow (Alpha=255,

Red=255, Green=255, Blue=0). This time interpolation will be performed in 0.5 seconds to new color. Following table shows interpolated color values at 0.05 seconds sensitivity.

Table 6.1 Color Interpolation along time

Time	Red	Green	Blue	Color Value	Color
0,00	255	0	0	(255; 0; 0)	
0,05	229	0	0	(229; 0; 0)	
0,10	204	0	0	(204; 0; 0)	
0,15	178	0	0	(178; 0; 0)	
0,20	153	0	0	(153; 0; 0)	
0,25	128	0	0	(128; 0; 0)	
0,30	102	0	0	(102; 0; 0)	
0,35	77	0	0	(77; 0; 0)	
0,40	51	0	0	(51; 0; 0)	
0,45	26	0	0	(26; 0; 0)	
0,50	0	0	0	(0; 0; 0)	
0,55	26	26	0	(26; 26; 0)	
0,60	51	51	0	(51; 51; 0)	
0,65	77	77	0	(77; 77; 0)	
0,70	102	102	0	(102; 102; 0)	
0,75	128	128	0	(128; 128; 0)	
0,80	153	153	0	(153; 153; 0)	
0,85	178	178	0	(178; 178; 0)	
0,90	204	204	0	(204; 204; 0)	
0,95	229	229	0	(229; 229; 0)	
1,00	255	255	0	(255; 255; 0)	

Color interpolation has a simple algorithm to implement and use. By the way, smoothing scene passes requires skills on art and hard to implement. Darken scene is a simple and generic solution for this complex problem. A more complex and more artistic choice for scene passes might be blending scenes in a constant time. But this

algorithm will increase hardware requirements. But further versions of Weendigo should allow designer to choose this algorithm as an option.

## 6.5 Rendering Static Meshes

A static mesh file refers to a non-animated mesh object stored in an X file. X Files were introduced in Microsoft DirectX 2.0. But methods and interfaces are available for reading and writing X files since Microsoft DirectX 6.0. As Weendigo using Microsoft DirectX 9.0 using X files is included by design. X files provide a template-driven format that enables the storage of meshes, textures, animations, and user-definable objects. Support for animation sets enables you to store predefined paths for playback in real time. Instancing and hierarchies are also supported. Instancing enables multiple references to an object, such as a mesh, while storing its data only once per file. Hierarchies are used to express relationships between data records.

The X file format provides low-level data primitives on which applications define higher-level primitives through templates. Pneuma provides an abstraction on low-level operations on X files and hides implementation details from Weendigo designer and Weendigo component developer.

Any object demanded to be compatible with Pneuma must inherit BaseDisplayObject abstract class. BaseDisplayObject has a reference for Pneuma framework to gather a valid 3D device when needed. Also BaseDisplayObject class contains some additional pure virtual method which leaves implementation details to inheritors. Following figure shows BaseDisplayObject class in a bit details:

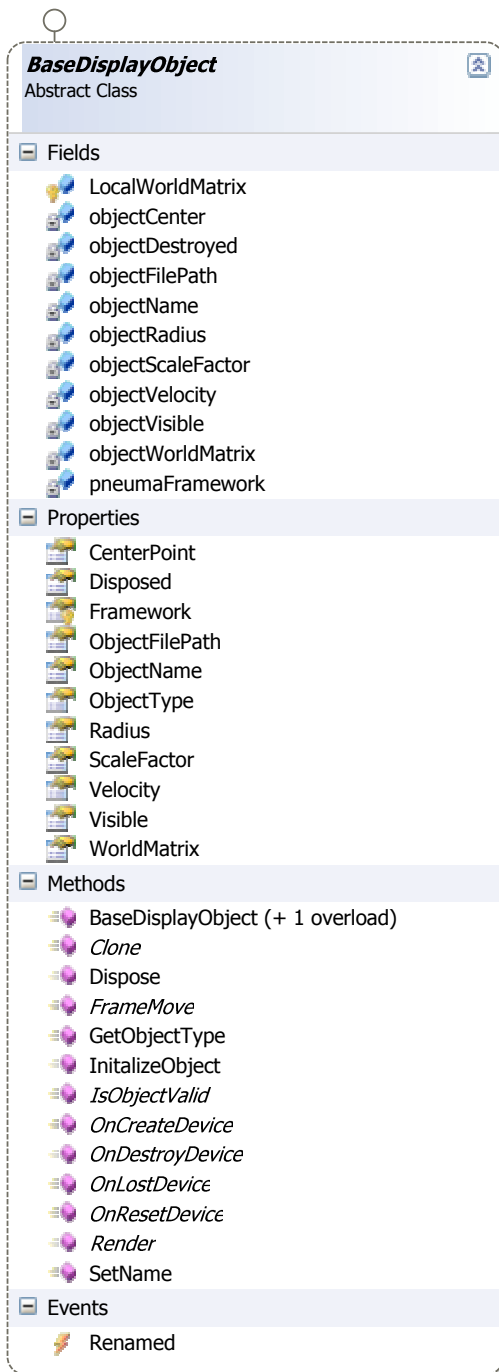


Figure 6.34 Base Display Object class interface



As seen in the figure above BaseDisplayObject has an event named as Renamed. This event is raised only in design time. When an object is renamed in design time, this event is raised to resolve name conflicts. *IsValid* method is used for Weendigo compilation and not used at run time.

StaticObject class inherits BaseDisplayObject and accepts a file path of mesh file for rendering at run time.

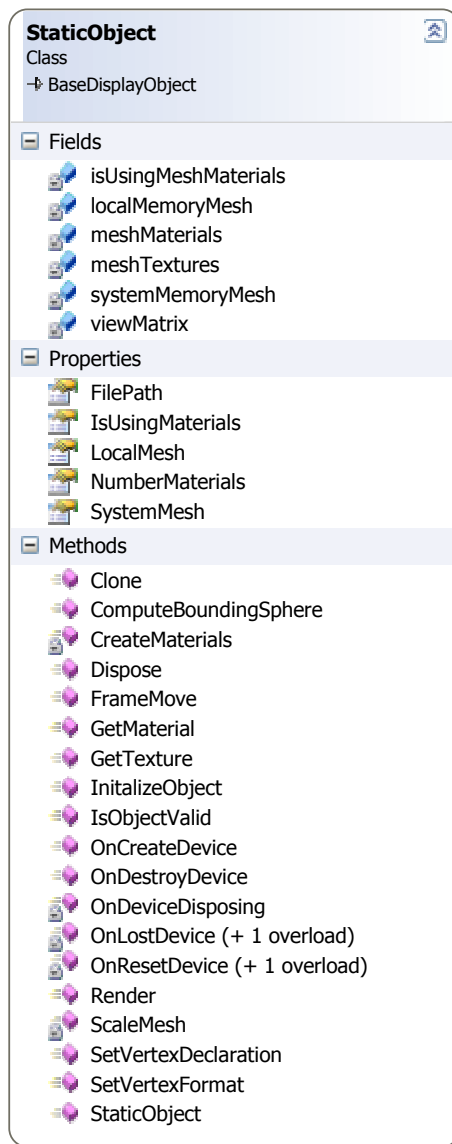


Figure 6.35 StaticObject class interface

Any instance of this class at runtime loads a mesh file (only X files supported). The mesh file is loaded to system memory initially. Any texture and materials are loaded in system memory. Each time when device is reset, these objects are copied to graphic card's memory to get a better performance. Due to graphics cards has a more volatile memory than system memory, original copies of these object reside in system memory during lifetime of the application. After loading mesh file, if any scale is specified, scale operation is performed using a simple interpolation algorithm. Modifying contents of vertex buffer in graphics card memory is not a recommended operation. So all scale operation is performed on system memory. Also at the time of loading mesh, a bounding sphere is calculated. Radius of this bounding sphere is used by graphic engine to have a common scene view.

On each frame move, local and global world matrices are calculated. Any translation and rotation operation is done at this time. These operations allow us to reposition of objects.

On each frame render, a static object renders itself by using reference pointer of Pneuma framework. Pneuma framework exposes a valid rendering device. Rendering of static objects contains two phases. In first phase of rendering operation, mesh object's opaque parts which have subsets without alpha are rendered. Non-opaque parts are rendered at second phase to have alpha blending support. Any exception raised during rendering mesh object is caught and reported Pneuma framework. It is Pneuma framework duty to handle and decide what to do with this exception.

Following figures shows screenshots of static objects rendered by Weendigo.



Figure 6.36 Microsoft logo is loaded from a mesh file and rendered according to this mesh file.

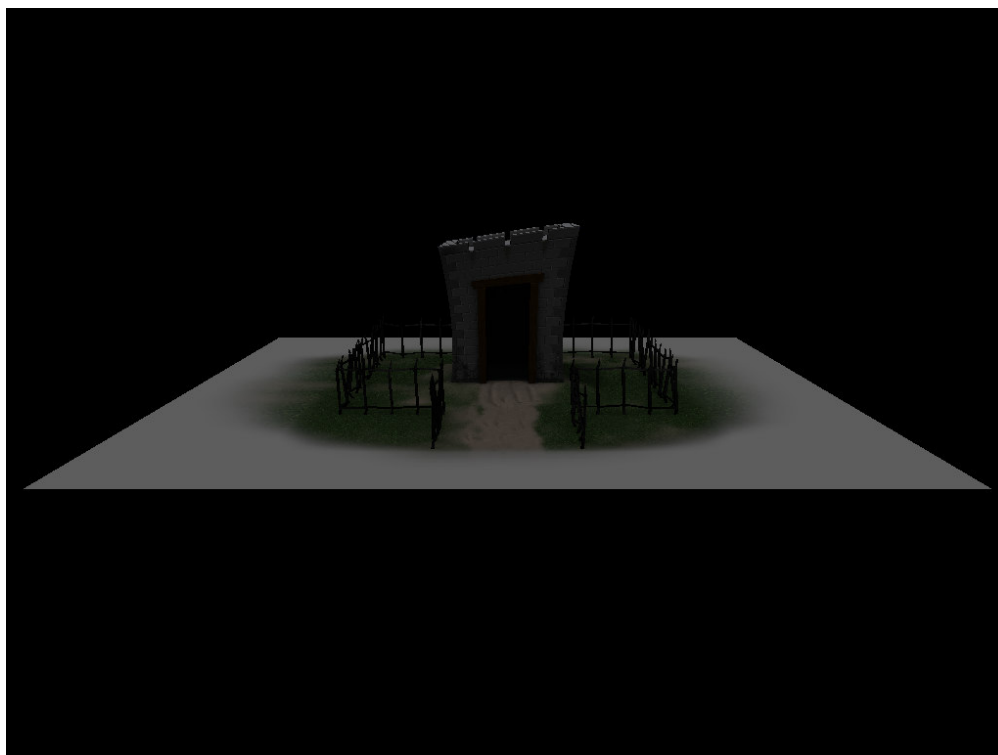


Figure 6.37 More complex mesh file is rendered by loading from an X file.

As static objects do not have a mesh hierarchy, any animation can only be done by performing a linear interpolation. Positioning of objects to be rendered in run time is designed at design time by using Weendigo IDE. A static object has the following properties that are enumerated by Weendigo IDE:

- **Appearance**
  - **Center Point:** Position of mesh object
  - **Scale Factor:** Default 1, If modified scales object by extending vertices with this scale factor
  - **Velocity:** Not used by default. Available for inheritors
- **Behavior**
  - **Visible:** Render this static object at any frame.
- **Data**
  - **File Name:** Mesh object file name to be loaded and rendered
  - **Radius (read only):** radius of mesh object
- **Design**
  - **Name:** Unique object name inside current scene.
  - **Type (read only):** Shows the class name “StaticObject” unless this class inherited.

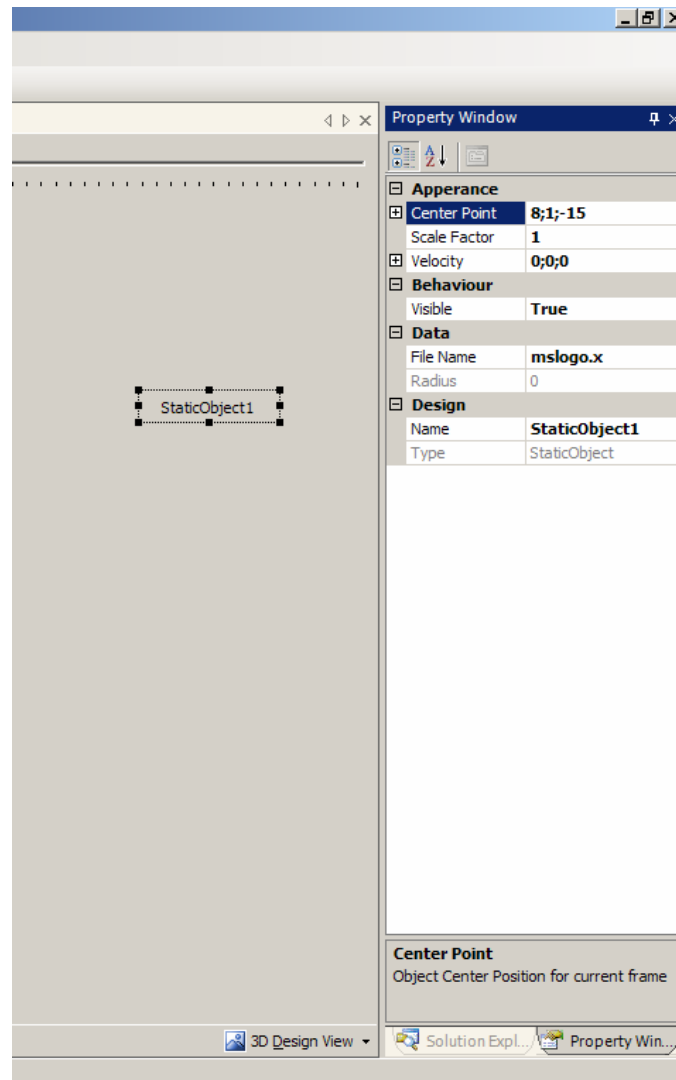


Figure 6.38 Static Object Properties enumerated by Weendigo

The following screen shot is taken by putting two distinct objects on same scene. Number of objects can be rendered inside a scene is not limited. But practically it is limited with the mesh object data amount and your graphics card memory limit since all objects are loaded to graphics card.



Figure 6.39 Crypt is scaled twice to have a scene like above.

## 6.6 Rendering Animated Meshes

Pneuma includes support to animate a 3D model that has a skeletal structure. Skinning is an animation technique that takes data organized in a skeletal-mesh hierarchy and applies geometry blending to transform mesh vertices. The geometry blending generates smooth surfaces with fewer artifacts.

Direct3D and D3DX can be used to animate meshes. Applications can load X files containing animation data, then control and render the animated meshes. There are five skinning techniques:

- fixed-function non-indexed,
- fixed-function indexed,
- software,
- assembly shader,
- HLSL shader

Each technique has its advantages and disadvantages, and application developers should weigh these and choose the appropriate techniques for the needs of their application. Animation Object in Pneuma framework prefers fixed-function indexed.

Skinning is a popular animation technique that is modeled like a human body. Skinning uses a set of interconnected bones (or frames) that form a hierarchy. Moving or rotating the bones causes the mesh surface to move or rotate. The mesh surface is analogous to the skin of the human body. Each point (or vertex) on the mesh surface is associated with a number of bones. Its position is determined by the position of the associated bones. For instance, consider the hierarchy in given below which describes a human limb:

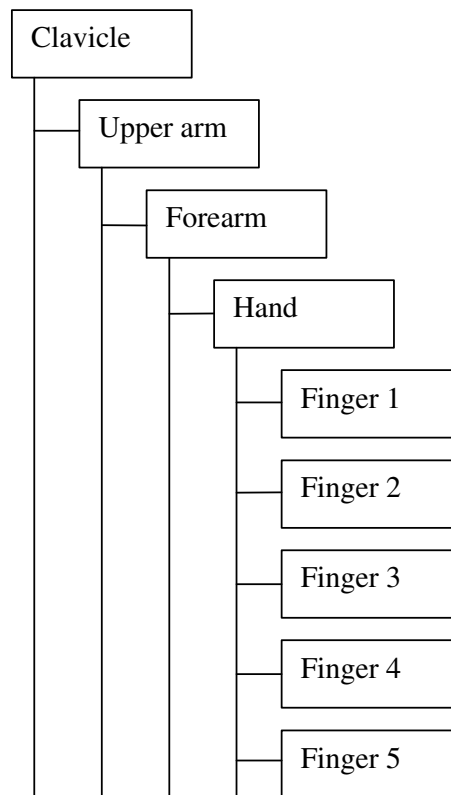


Figure 6.40 Human limb hierarchy

The skin at the elbow is influenced by the upper arm bone and the forearm bone. If the upper arm remains stationary and the forearm moves, the elbow skin position is

affected; the same is true when holding the forearm stationary and moving the upper arm. It can be concluded that the skin at the elbow is associated with both the upper arm and forearm bones. Transforming vertices with multiple matrices are done with geometry blending. Each matrix has a blending weight ranging from 0 to 1, with the sum of all blending weights equal to 1. The blending operation is carried out by first transforming the vertex with each matrix, yielding several transformed vertex coordinates. Then, these coordinates are interpolated based on the corresponding blending weights. Vertices can have different blending weights even if they are influenced by the same bones. This geometry blending allows surfaces to stay smooth when animated.

Furthermore, any movement by an ancestor bone may affect the skin position. For instance, if both the upper arm and forearm bones remain stationary (that is, no rotation), and the clavicle bone moved, the skin between the upper arm bone and the forearm bone should change position. This explains why the child bones are modeled in a hierarchy to generate this dependency.

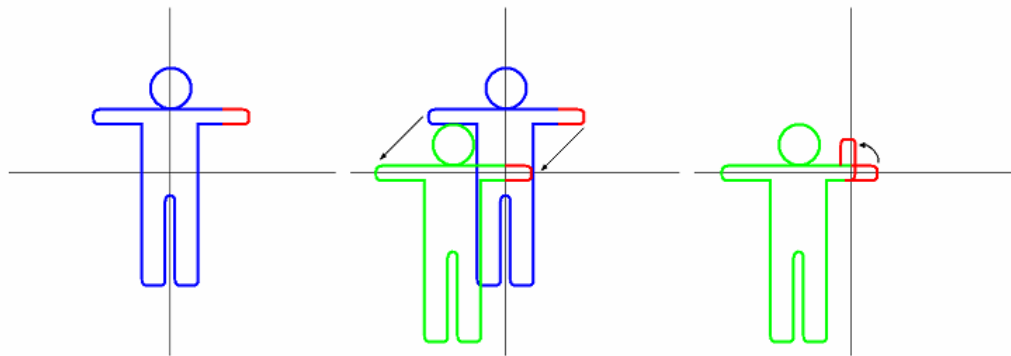


Figure 6.41 any movement by ancestor bone might affect the skin position

- Left: The character as it is stored on the disk.
- Middle: The bone offset transformation changes the vertex coordinates from the mesh space to bone space.
- Right: Now that vertices are in bone space, animation transformation can be performed. In this case, a rotation is applied to the forearm



The X file format provides low-level data primitives on which applications define higher level primitives through templates. Pneuma provides an abstraction on low-level operations on X files and hides implementation details from Weendigo designer and Weendigo component developer.

Any object demanded to be compatible with Pneuma must inherit `BaseDisplayObject` abstract class. `BaseDisplayObject` has a reference for Pneuma framework to gather a valid 3D device when needed. Also `BaseDisplayObject` class contains some additional pure virtual method which leaves implementation details to inheritors.

`BaseDisplayObject` has an event named as `Renamed`. This event is raised only in design time. When an object is renamed in design time, this event is raised to resolve name conflicts. *`IsValidObject`* method is used for Weendigo compilation and not used at run time.

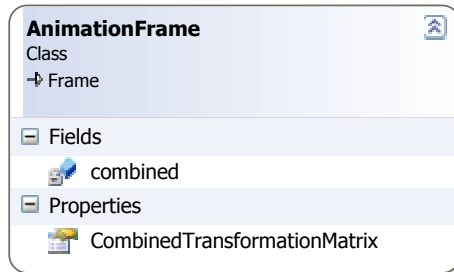


Figure 6.42 Animation Frame class interface

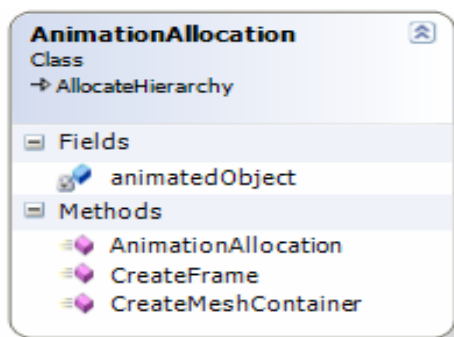


Figure 6.43 Animation Allocation class interface

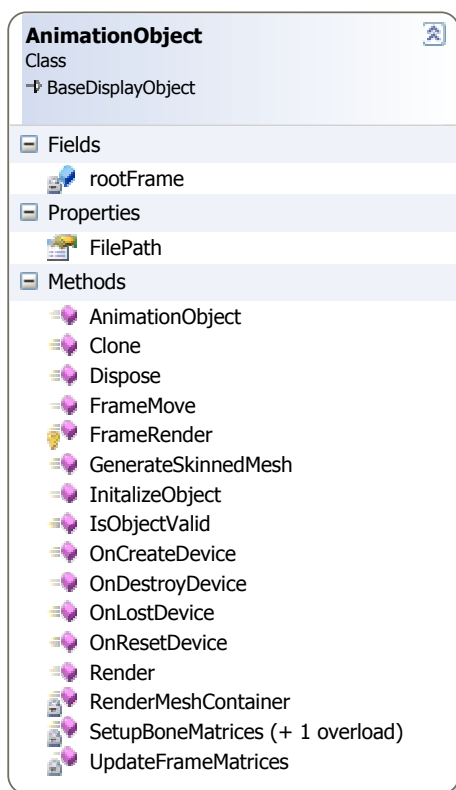


Figure 6.44 Animation Object class interface

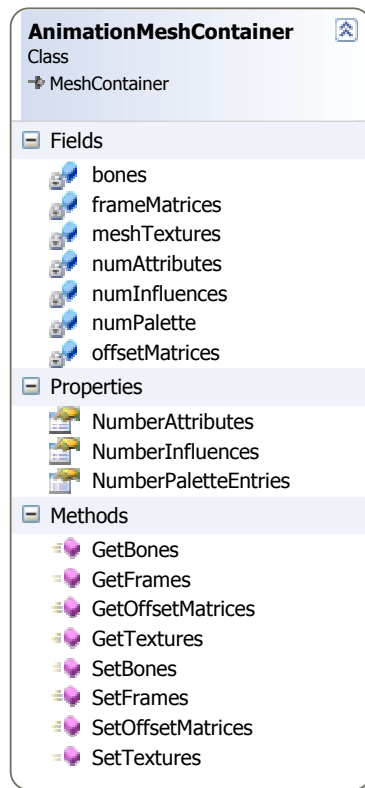


Figure 6.45 Animation mesh container class interface

D3DX, an animated mesh is composed from a frame hierarchy. At the very top, there is a root frame. The root frame has one or more child frames; each child frame has its own child frames, and so forth.

A frame in the hierarchy is represented by an instance of “AnimationFrame”. In a hierarchy, some frames will have valid container values, which are the mesh geometry data. When rendering the mesh, the container is drawn regardless of its location in the mesh hierarchy. Similar to a frame, an application should define its own mesh container type like “AnimationMeshContainer”.

Because the two fundamental structures of a frame hierarchy are often derived and defined by the application itself, the application has to define functions that handle the allocation and deallocation of the frames and mesh containers. “AnimationAllocation” is the name of the class intended to provide this.

Rendering the animated mesh is a two-step process: setting up the matrices and the actual rendering. To set up the matrices, `AdvanceTime` is called first. This method takes a `TimeDelta` parameter that indicates how much to advance since the last call, then it updates the frame hierarchy's transformation matrices with matrices that correspond to the bone positions at that instance of time. These matrices contain transformation with respect to their parent frames, consisting of a scaling plus rotation plus translation transformation. Next step is calling `UpdateFrameMatrices`. This function updates the frame hierarchy's combined transformation matrix. The combined transformation matrix holds the product of all of the ancestor frames' matrices, including the frame's own matrix. Recall that bones are influenced by their parent bones, and the parent bones are influenced by the grandparent bones, and so forth. Combining all influencing matrices makes the combined transformation matrix absolute, or relative to the world, which is more suitable for rendering. `UpdateFrameMatrices` achieves this by recursively traversing the hierarchy tree. For each frame, it writes the product of the transformation matrix and the parent's matrix to the combined transformation matrix. Then it calls `UpdateFrameMatrices` on its sibling and first child nodes, passing its parent matrix to the sibling and passing its own combined transformation matrix as the parent matrix for the children. All of these operations are performed in `FrameMove`.

The actual rendering, like all other `BaseDisplayObject` types, starts in `Render`. The rendering of the mesh is done by the `FrameRender`. This function takes a pointer to a frame node, draws the frame's mesh if one exists, then recursively calls itself again with the frame's sibling and children. The result is that all mesh containers in the frame hierarchy will be drawn when the top `DrawFrame` returns. When a frame holds a valid mesh container, `FrameRender` calls the `RenderMeshContainer`, which is where the entire mesh rendering takes place.

An animation object has the following properties that can be enumerated by Weendigo IDE.

- **Appearance**
  - **Center Point:** Position of mesh object
  - **Scale Factor:** Default 1, If modified scales object by extending vertices with this scale factor
  - **Velocity:** Not used by default. Available for inheritors
- **Behavior**
  - **Visible:** Render this static object at any frame.
- **Data**
  - **File Name:** Mesh object file name to be loaded and rendered
  - **Radius (read only):** radius of mesh object
- **Design**
  - **Name:** Unique object name inside current scene.
  - **Type (read only):** Shows the class name “StaticObject” unless this class inherited.

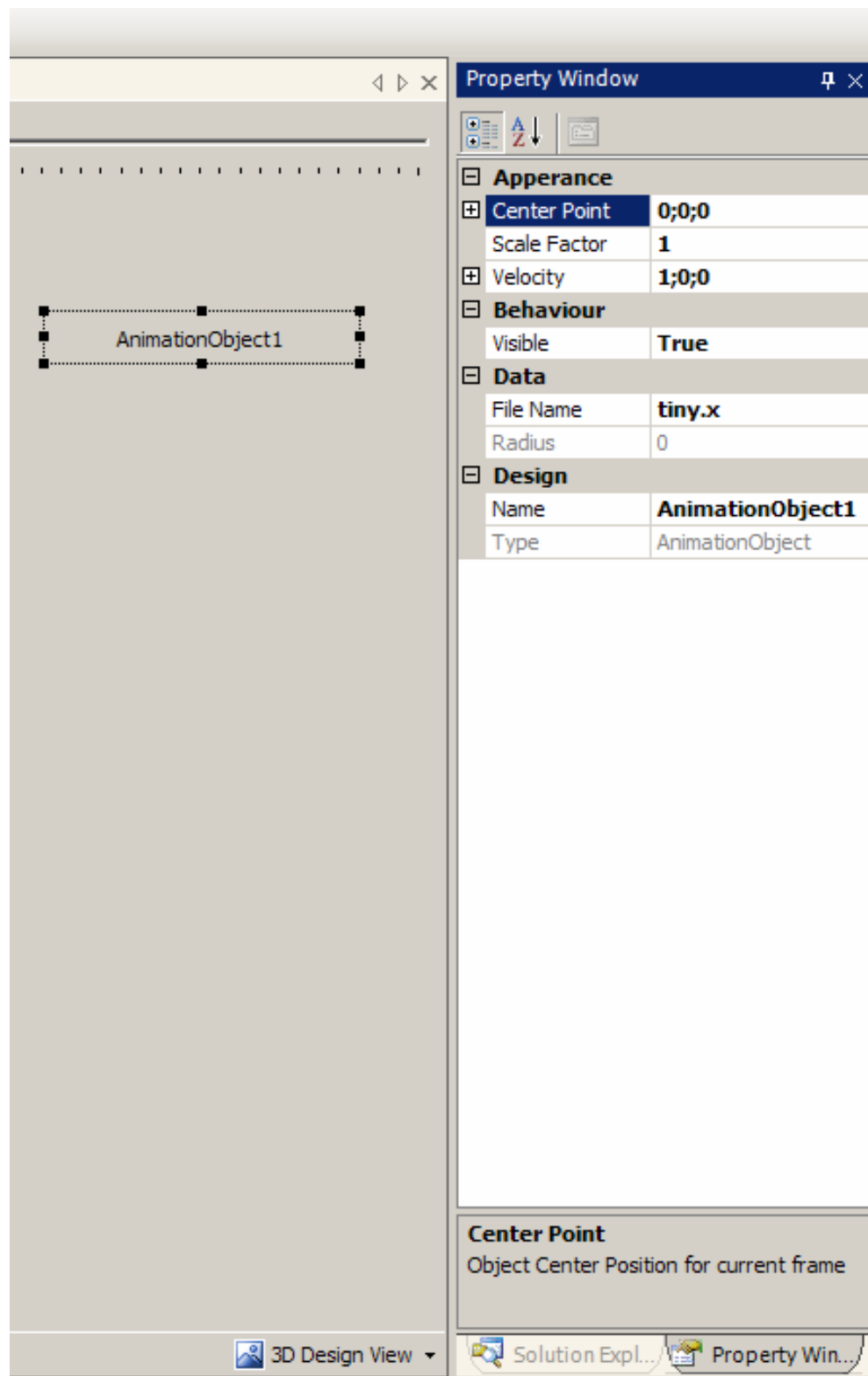


Figure 6.46 Animation Object properties are enumerated by Weeendigo

Wired and solid rendered of two instances of animation objects is given below. Note that, model used in these examples is taken from Microsoft DirectX 9.0 SDK Samples (tiny.x). Each instance of an animated object has a valid property called Velocity. Velocity determines the animation allocation hierarchy advancing time.



Figure 6.47 Wired rendered of two instances of animation objects



Figure 6.48 Solid rendered of two instances of animation objects

## 6.7 Rendering Text

Rendering a two-dimensional image or animation in a three dimensional environment is a common problem. In computer graphics world, a common remedy is using Sprites. A sprite is a two-dimensional image or animation that is integrated into a larger scene. Displaying a text front of a scene is a common need for a three dimensional animation film. Weendigo provides a class named as “TextObject” for this functionality.

Sprites were originally invented as a method of quickly compositing several images together in two-dimensional video games using special hardware. As computer performance improved, this optimization became unnecessary and the term evolved to refer specifically to the two dimensional images themselves that were integrated into a scene. That is, figures generated by either custom hardware or by software alone were all referred to as sprites. As three-dimensional graphics became



more prevalent, the term was used to describe a technique whereby flat images are seamlessly integrated into complicated three-dimensional scenes.

More often sprite now refers to a partially transparent two dimensional animation that is mapped onto a special plane in a three dimensional scene. Unlike a texture map, the sprite plane is always perpendicular to the axis emanating from the camera. The image can be scaled to simulate perspective, it can be rotated two dimensionally, it can overlap other objects and be occluded, but it can only ever be viewed from the same angle. This rendering method is also referred to as “billboarding”.

Sprites create an effective illusion when:

- the image inside the sprite already depicts a three dimensional object
- the animation is constantly changing or depicts rotation
- the sprite exists only for a short period of time
- the depicted object has a similar appearance from many common viewing angles (such as something spherical)
- The viewer accepts that the depicted object only has one perspective. (such as small plants or leaves)

When the illusion works viewers will not notice that the sprite is flat and always faces them. Often sprites are used to depict phenomena such as fire, smoke, small objects, small plants (like blades of grass), or special symbols (like "1-Up"). The sprite illusion can be exposed in video games by quickly changing the position of the camera while keeping the sprite in the center of the view.

Sprites have also occasionally been used as a special effects tool in movies. Explosion effects are commonly done by using sprites. By the way, Pneuma exposes a set of functionality for rendering text in a three dimensional scene using sprites.

As Pneuma has only the ability to render objects derived by Base Display Object, Text Object inherits BaseDisplayObject. BaseDisplayObject has a reference for Pneuma framework to gather a valid 3D device when needed. Also

BaseDisplayObject class contains some additional pure virtual method which leaves implementation details to inheritors.

BaseDisplayObject has an event named as Renamed. This event is raised only in design time. When an object is renamed in design time, this event is raised to resolve name conflicts. *IsValid* method is used for Weendigo compilation and not used at run time.

TextObject class implements BaseDisplayObject abstract class and accepts a string to display at run time.

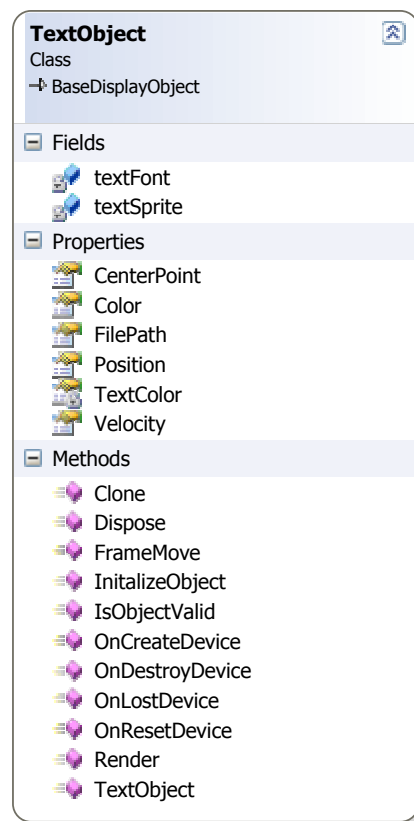


Figure 6.49 Text Object class interface

Any instance of this class, initiates a new instance of Microsoft.DirectX.Direct3D.Font class and uses this font object to render specified text. Each frame render, Render method is called. Inside render method a new

instance of TextHelper class is initiated and string is drawn specified position. Velocity property is intended to determine forecolor of the text. Thus enables us to have a different forecolor at each render.

“TextHelper” class uses specified device and font objects and renders string. By default, text rendering is done with alpha blending is enabled. There is no need to use a background for a text object. It automatically blends background by obeying three dimensional rules. This class implements IDisposable interface. After each frame render, the resources consumed by this class is collected by garbage collector. Class details are given below:

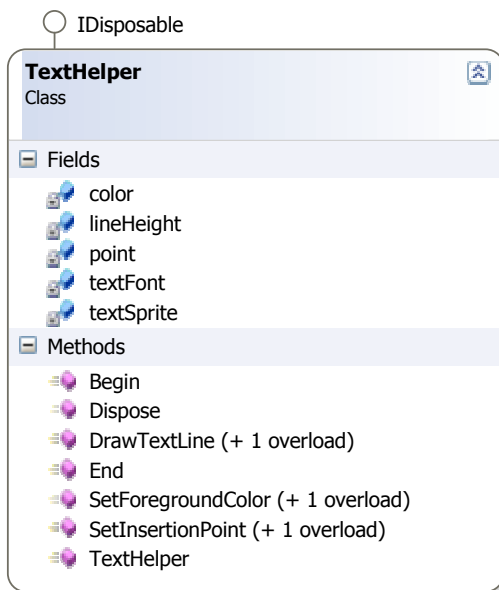


Figure 6.50 Text Helper class interface

The following figures show screenshots of text objects rendered by Weendigo using Pneuma.

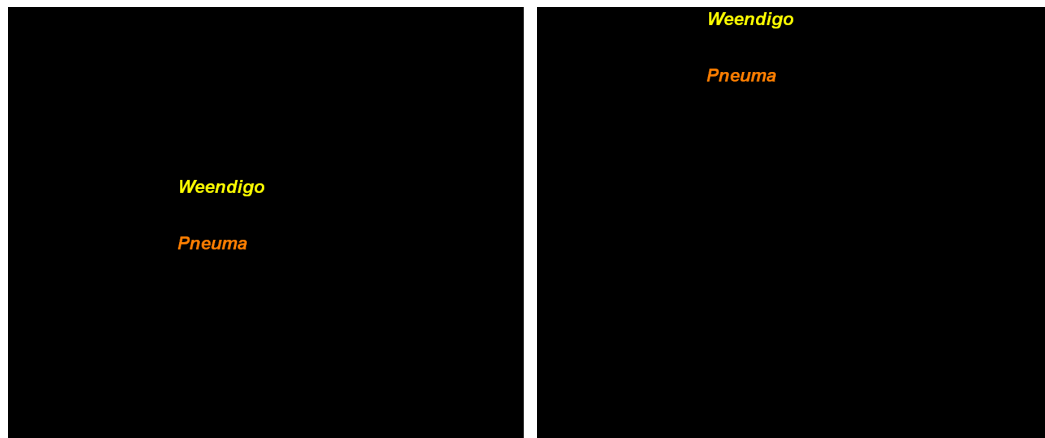


Figure 6.51 Weendigo also performs interpolation on text objects

In the figure at left side, shows “Weendigo” and “Pneuma” strings’ initial state. After a while these text objects start to move like casting titles in a movie. This simply done by interpolating positions of each text object independently. Fore color of each text object can independently have different color at each frame. Also, interpolating fore color is possible. Following figures also exchanges colors between “Weendigo” and “Pnuema” strings.



Figure 6.52 Weendigo also performs color interpolation on text objects colors

A text object has the following properties that are enumerated by Weendigo IDE:

- **Appearance**
  - **Color:** Text fore color
  - **Scale Factor:** Default 1, If modified scales object by extending vertices with this scale factor
  - **Position:** a two dimensional point of text to be rendered
- **Behavior**
  - **Visible:** Render this text object at any frame.
- **Data**
  - **Text:** text to be rendered. It is possible to render a distinct text for each time unit.
  - **Radius (read only):** radius of mesh object
- **Design**
  - **Name:** Unique object name inside current scene.
  - **Type (read only):** Shows the class name “TextObject” unless this class inherited.

## 6.8 Scene Background

A 3D film scene is built up with objects including players and other things in scenery. By the way there is another missing point: environment. In real films, there is already atmosphere as environment. By the way, in a 3D film scene it is not possible to fill up scene with objects. In weendigo approach, there is a background object used in the scene. This background object determines the clear color at runtime rendering.

As in the rendering process scene is prepared in a back buffer and finally when the processing is done this back buffer is swapped with the current one. Next frame started to be prepared then goes on. In Microsoft Direct 3D this times are controlled using `BeginScene()`, `EndScene()`, and `Present()` methods. As a common approach a flow like following is done:

```

Begin Scene Statement Block
..
Rendering is done here!
..
End Scene Statement Block
Present Scene Statement Block

```

Begin Scene statement locks back buffer, and end scene unlocks back buffer. Since multithread access on same back buffer is not possible. Usually this is not a requirement on rendering. Present Scene statement flushes back buffer rendering and forces to swap chain.

Scene object rendering in a 3D environment must be done consecutively from back to front. And the first thing is done is clearing the scene with a color. Afterwards scene is rendering process starts. Background object in Weendigo is subject to control this color and clear parameters and give user the ability to customize these parameters for each second in the scene. Background object is implemented in Pneuma with “BackgroundObject” class. Class details are given below for clarity:

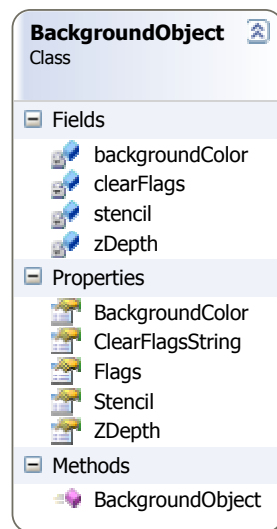


Figure 6.53 BackgroundObject  
class interface

A background object instance is mapped to only one scene. A background object has the following properties that can be enumerated by Weendigo IDE.

- **Appearance**

- **Background Color:** Clear color of the back buffer
- **Clear Flags:** back buffer clear flags. Target and Z-Buffer is set to default.
- **Stencil:** Clear the stencil buffer to this new value which ranges from 0 to  $2^{n-1}$  (n is the bit depth of the stencil buffer).
- **Z-Depth:** Clear the depth buffer to this new z value which ranges from 0 to 1.

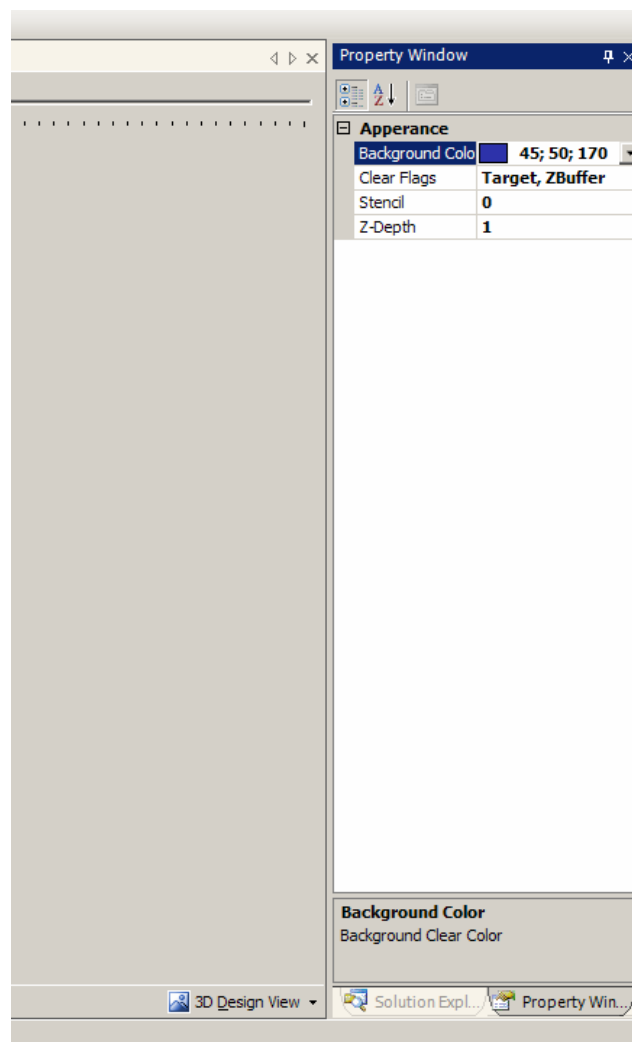


Figure 6.54 Background object properties can be enumerated

When user changes the scene background color for a specific time, this change is not done suddenly. A color interpolation is done by Weendigo automatically.

## 6.9 Camera Usage

Camera in a 3D environment is an abstract object unlike real world. Camera object is simulated with 3D projection in a 3D environment. A 3D projection is a mathematical transformation used to project three dimensional points onto a two dimensional plane. 3D projection is often the first step in the process of representing three dimensional shapes two dimensionally in computer graphics, a process known as *rendering*. In computer graphics success is scaled how much scene is closed to real life. Mostly objects used in rendering process determine realism, even though realistic rendering requires good use of camera.

Data about the objects to render is usually stored as a collection of points, linked together in triangles. Each point is a set of three numbers, representing its X, Y, Z coordinates from an origin relative to the object they belong to. Each triangle is a set of three such points. In addition, the object has three coordinates X, Y, Z and some kind of rotation, for example, three angles alpha, beta and gamma, describing its position and orientation relative to a "world" reference frame. Also the observer (or camera) has a set of three X, Y, Z coordinates and three alpha, beta and gamma angles, describing the observer's position and the direction in which it is pointing. The first step is to transform the point's coordinates, taking into account the position and orientation of the object they belong to. Following four matrices is used for transformation.

- Object Translation
- Rotation about the x-axis
- Rotation about the y-axis
- Rotation about the z-axis



These four matrices are multiplied together, and the result is the world transform matrix: a matrix that, if a point's coordinates were multiplied by it, would result in the point's coordinates being expressed in the "world" reference frame.

Note that unlike multiplication between numbers, the order used to multiply the matrices is significant; changing the order will change the results too. When dealing with the three rotation matrices, a fixed order is good for the necessity of the moment that must be chosen. The object should be rotated before it is translated, since otherwise the position of the object in the world would get rotated around the centre of the world, wherever that happens to be.

$$\text{World transform} = \text{Translation} \times \text{Rotation}$$

To complete the transform in the most general way possible, another matrix called the scaling matrix is used to scale the model along the axes. This matrix is multiplied to the four given above to yield the complete world transform.

The second step is virtually identical to the first one, except for the fact that it uses the six coordinates of the observer instead of the object, and the inverses of the matrixes should be used, and they should be multiplied in the opposite order. The resulting matrix can transform coordinates from the world reference frame to the observer's one.

The camera typically looks in its z direction, the x direction is typically left, and the y direction is typically up. The two matrices obtained from the first two steps can be multiplied together to get a matrix capable of transforming a point's coordinates from the object's reference frame to the observer's reference frame.

$$\text{Camera transform} = \text{inverse rotation} \times \text{inverse translation}$$

$$\text{Transform so far} = \text{camera transform} \times \text{world transform.}$$

The resulting coordinates would be good for an isometric projection or something similar, but realistic rendering requires an additional step to simulate perspective distortion. Indeed, this simulated perspective is the main aid for the viewer to judge distances in the simulated view.

Perspective distortion describes the appearance of a part of the subject as abnormally large, relative to the rest of the scene. This is especially noticeable when that part of the scene extends towards the camera. It is affected solely by the distance between the camera and subject, and the smaller this distance the greater the perspective distortion. Perspective projection distortion is an error introduced in drawing when images are created by the use of "projectors". Projectors are imaginary constructs which aid in the production of real images. These projectors create an imaginary image where they intersect an imaginary plane of projection. This image represents the object as seen by the eye from the point of concurrence. The image is then reproduced onto a planar surface (e.g. paper) by geometric protocols.

There are different types of camera used in computer graphics. Commonly used ones are described below:

- **First Person Camera:** is a camera moves and rotates it allows yaw and pitch but not roll.
- **Model Viewer Camera:** is a camera that rotates around an object.
- **Third Person Camera:** is a camera that does not occupy any character, but typically floats behind and above the main character. It is like a third person observing the scene.

Although third person camera looks like a third person observing the scene which seems that it meets the requirements of a 3D animation film. By the way, determining the main character in each scene is not easy. So this decision could not be automated and must be done manually. Camera in a 3D animation film needed to be controlled via person for each frame. By the way, this fact conflicts with the Weendigo definitions. Weendigo is not a professional tool to create 3D animation

films. It makes easier to produce a 3D animation film, but allowing a user to edit camera position and look position conflicts weendigo easy to use approach.

Fortunately, Weendigo offers a new style of camera which is called as “Scene Camera”. A Scene Camera is a camera similar to a third person camera and model viewer camera.

A model viewer camera directly looks to the center point of the model. While it rotating around the object, model position never changes, only camera position changes. Following figure illustrates model viewing at a time:

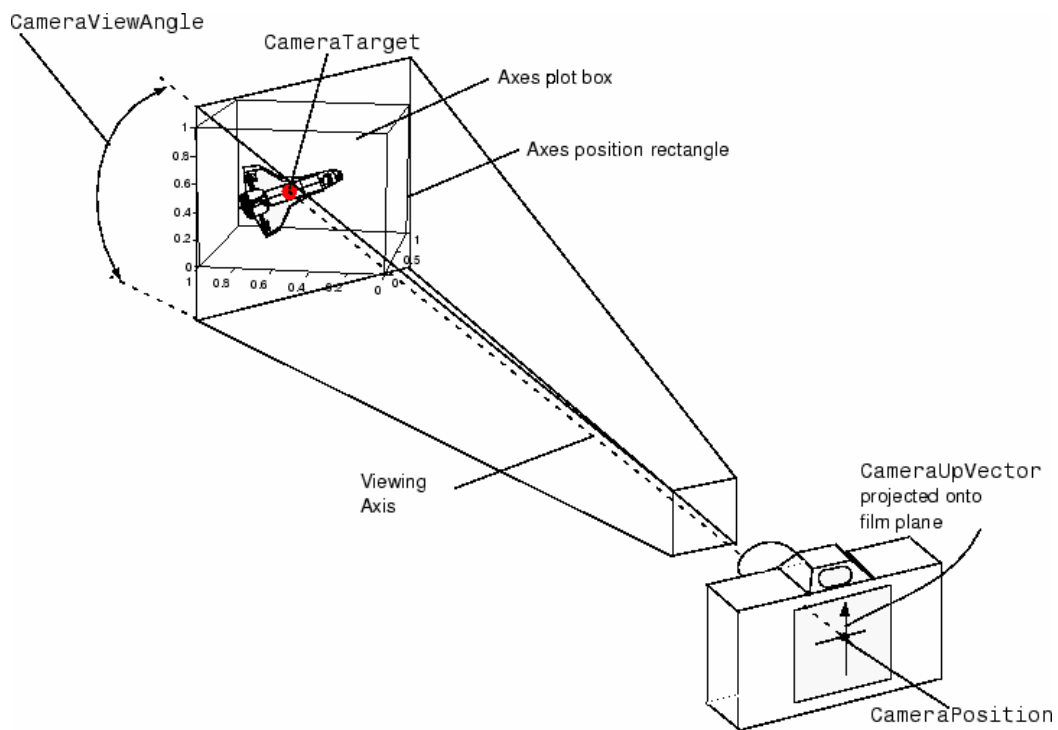


Figure 6.55 Model viewing camera

A first person camera rotates around its own position; a third person camera rotates around a focus point. A third person cameras orientation is the direction from its position to the focus point. The focus point is moved with the focus subject; the game hero. The cameras position is set to be a certain distance from the focus point. This distance is the viewing distance.

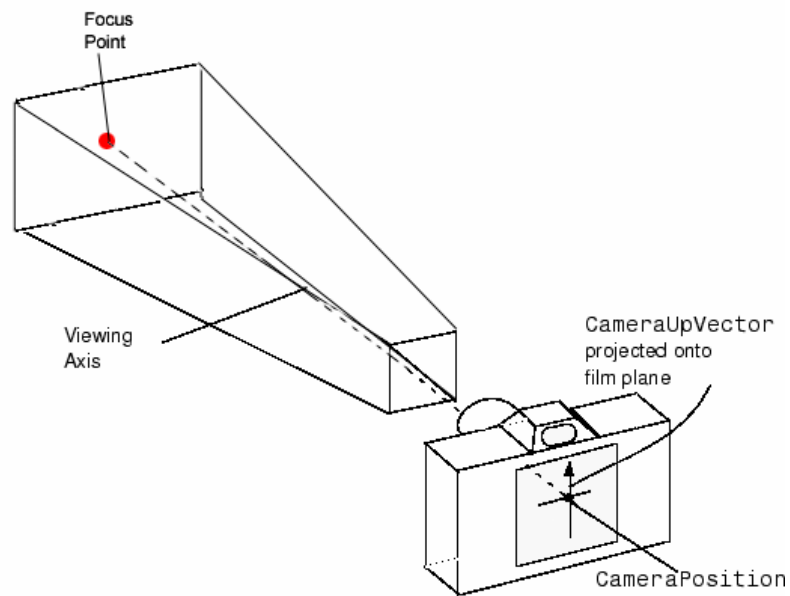


Figure 6.56 Camera position has to be set a certain distance from focus point

Focus point is calculated by Weendigo using center point and radius of each visible object in active scene. To accomplish this, a bounding algorithm is built up. First of all the space needed for each object is calculated. This is done by using bounding sphere approach. A bounding sphere is a hypothetical sphere that completely encompasses an object. It is defined by a 3D coordinate representing the center of the sphere, and a scalar radius that defines the maximum distance from the center of the sphere to any point in the object.

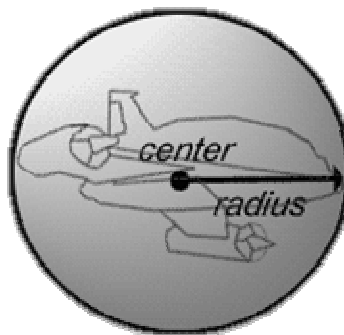


Figure 6.57 Bounding Sphere is calculated for each instance of objects in scene

Each object's bounding sphere is calculated independently. Then each object is put in a logical bounding box with their bounding spheres. Nearest and furthest object positions are determined and a logical bounding box is found. The center of gravity of this bounding box is the focus point. If there is only one object in scene, The center of gravity of bounding box equals to the center of gravity of the object.

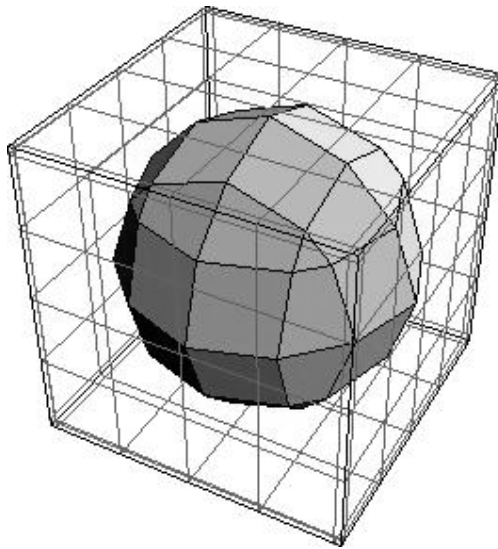


Figure 6.58 center of gravity of bounding box is calculated

In Weendigo implementation a base camera class is defined to use for any purposes. The “SceneCamera” class inherits this base class and implements this bounding scene is generation. “BaseCamera” and “SceneCamera” class details are given below for clarity in figure 4.59 .

In further versions of Weendigo, setting camera position and focus point at design time feature might be added. For future compatibility a design view is defined in Weendigo, but not implemented yet (CameraDesignView).

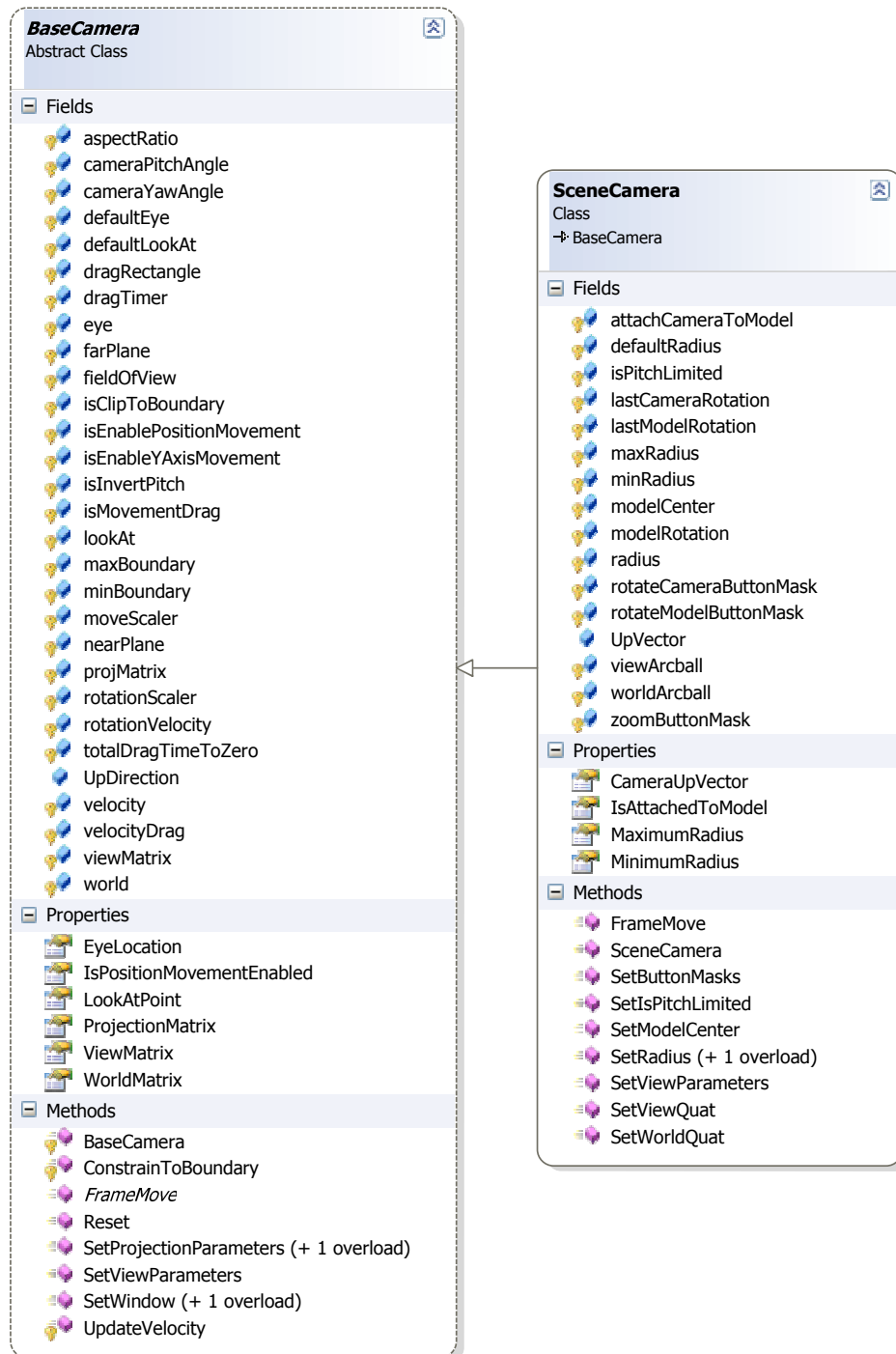


Figure 6.59 Base Camera and Scene Camera class interfaces

## **CHAPTER SEVEN**

### **POST IMPLEMENTATION REVIEW**

This chapter includes post implementation review of the project including benchmark, and security tests. Also there exists several common secure software development issues and tricks implemented in this project.

#### **7.1 Cross Threading Issue**

Cross threading is an issue for mechanical engineering. Cross-threading occurs when the spark plug enters the spark plug hole at an angle. This mangles the existing threads, pushing them out of alignment with the deeper, intact threads. Since the cross-threaded threads no longer point the spark plug down the correct axis of the hole, the spark plug cannot be fully seated. Even if the spark plug is withdrawn and reinserted, it now engages on the cross-threaded threads – still the same problem. For the programming perspective, cross threading is a preliminary security issue in windows forms. For the programming perspective, cross threading is a preliminary security issue in windows forms. UI thread continues to server user, when another thread performing a time consuming operation. Since this is an asynchronous operation, guessing the completion time is not possible in many cases. After finalization of the time consuming operation, this thread wants to update user interface and inform user on output of the operation. But allowing this operation can be abused and makes our application vulnerable Trojan horse attacks.

##### ***7.1.1 Cross-threading Vulnerability***

For instance, “Troj/Torpig-AE” is an information stealing Trojan for the Windows platform. The Trojan attempts to steal passwords, as well as logging key-presses and open window titles to text files and periodically sends the collected information to a remote user via HTTP. This Trojan attaches itself to explorer.exe. By doing this, it becomes invisible to user. It is possible to intercept all SSL traffic with trespassing

SSL encryption. This Trojan horse is the ancestor of the upcoming security threats. Trojan performs this interception by modifying the following registry key:

“HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows  
NT\CurrentVersion\Winlogon”

As a default this registry key contains “explorer.exe”. Windows automatically executes value of these key when any user logs in. Trojan appends a new executable name on this key which is the malicious code for stealing usernames and password.

By default, Microsoft .NET 2.0 performs checks on calling code and calling thread. Any call from different thread triggers CLR to raise a security exception. This is an important feature, but it also prevents accessing from different threads in same application domain and process.

As developing windows applications, there is user interface between your application and user. Much of the users want, application should perform the task they requested immediately. But in computers world, it is not possible every time. Users’ demand might require too much processing or connecting to another computer using any transmission channel. By the way, users would not accept persuaded with any reason. After clicking a button in user interface, if our application is blocked till the operation succeeded, user might think our application is not responding and try to kill our application. Here is a sample code which will block user interface for twenty seconds. This operation does not consume CPU, it yields CPU task switching for twenty seconds. But blocks user interface for three seconds. This sample code demonstrates blocking call:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
```



```
using System.Windows.Forms;
namespace CrossThreading
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            private void Form1_Load(object sender, EventArgs e)
            {
            }

            private void sampleButton_Click(object sender, EventArgs e)
            {
                System.Threading.Thread.Sleep(20000);
            }
        }
    }
}
```

This sample code has a simple interface like following:

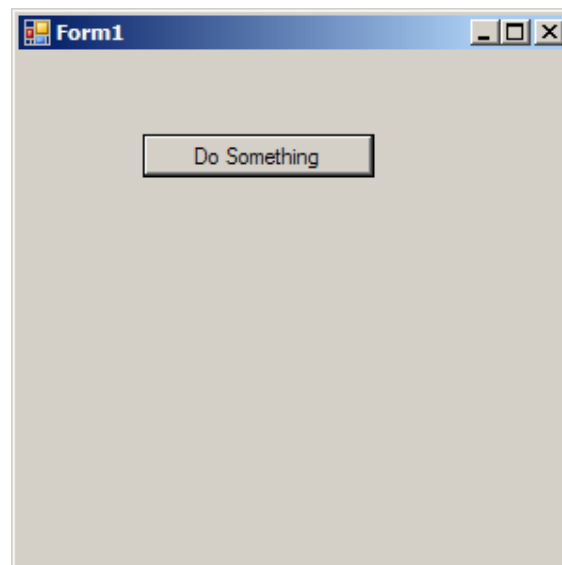


Figure 7.1 Cross Threading is a security issue

By the way, after clicking button labeled “Do Something”, whole user interface would be blocked. If user attempts to click elsewhere in the form, our application would be in a not responding state. Because, main thread of our application is busy on performing some operation, and will not respond any message passed on this thread until the operation completion. User should meet a screen like following.

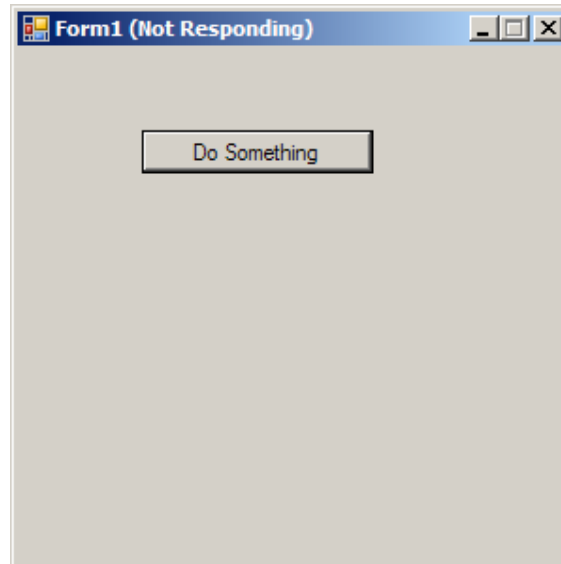


Figure 7.2 Bad programming practice. User interface is waiting the operation to be completed to respond user

To overcome this issue, developers prefer creating a new thread for this time consuming operation and allowing our user interface interact with user. As a best practice on developing windows based applications creating a thread behind an event is a reasonable developer behavior. Modifying event handler code and adding a simple method as shown in the following code snippet will fix this problem:

```
private void sampleButton_Click(object sender, EventArgs e)
{
    System.Threading.Thread th = new System.Threading.Thread(
        new System.Threading.ParameterizedThreadStart(DoSomething));
    th.IsBackground = true;
    th.Start();
}
```

```
private void DoSomething()  
{  
    System.Threading.Thread.Sleep(20000);  
}
```

By the way, this remedy will introduce cross threading problem if you want to access any element in the form. As mentioned above, Microsoft .NET Framework 2.0 will block these calls.

Access to Windows Forms controls is not inherently thread safe. If you have two or more threads manipulating the state of a control, it is possible to force the control into an inconsistent state. Other thread-related bugs are possible as well, including race conditions and deadlocks. It is important to ensure that access to your controls is done in a thread-safe way.

The .NET Framework helps you detect when you are accessing your controls in a manner that is not thread safe. When you are running your application in the debugger, and a thread other than the one which created a control attempts to call that control, the debugger raises an `InvalidOperationException` with the message, "Control control name accessed from a thread other than the thread it was created on."

This exception occurs reliably during debugging and, under some circumstances, at run time. You are strongly advised to fix this problem when you see it. You might see this exception when you debug applications that you wrote with the .NET Framework prior to .NET Framework version 2.0.

Following code, will introduce this problem with using a splash screen which is intended to inform user that the application loading is in progress. This exception is not a runtime exception, it only raised at debugging time. But as pointed out before, this might be abused and your application would be vulnerable any security threat like Trojan horses.

Create a new form named as “SplashForm” which has no control box and any control except a docked picture box filled with a splash image. Then, create a new class named “DialogManager” to perform dialog operations. And here is a sample implementation for DialogManager displaying and closing splash screen:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CrossThreading
{
    public sealed class DialogManager
    {
        #region Singleton Implementation
        #region Private Members
        private static DialogManager current;
        private static object synObject = new object();
        #endregion
        #region Property Declarations
        public static DialogManager Current
        {
            get
            {
                if (current == null)
                {
                    lock (synObject)
                    { // Double Checked Null Design Pattern
                        if (current == null)
                            current = new DialogManager();
                    }
                }
                return current;
            }
        }
        #endregion
        #region Internal Constructor
        internal DialogManager() { }
        #endregion
    }
}
```

```

#region Instance Data
private SplashForm mySplashForm = null;
#endregion

#region Public Methods
public void ShowSplashScreen()
{
    mySplashForm = new SplashForm();
    System.Threading.Thread thDoSplash =
        new System.Threading.Thread(
            new System.Threading.ThreadStart(DoSplash)
        );
    thDoSplash.IsBackground = true;
    thDoSplash.Start();
}

public void CloseSplashScreen()
{
    mySplashForm.Close();
}
#endregion

#region Private Helpers
private void DoSplash()
{
    mySplashForm.ShowDialog();
}
#endregion
}
}

```

Modify the main application form constructor and application load events like following:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

```

```

namespace CrossThreading
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            DialogManager.Current.ShowSplashScreen();
            InitializeComponent();
            DoSomething();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            DialogManager.Current.CloseSplashScreen();
            Activate();
        }

        private void DoSomething()
        {
            System.Threading.Thread.Sleep(5000);
        }
    }
}

```

When you build and run this code, everything will work fine. Your splash screen would be shown till your application raise form load event. After this event, splash screen will be closed automatically. But when you debug your application, you will get an “Invalid Operation Exception” complaining on illegal thread calls. Current windows versions do not raise this exception at run time but upcoming windows versions starting with Windows Vista (codenamed longhorn), will raise this exception at run time due to security reasons explained above.

There are two choices you can do, by pass cross threading checks, or fix this problem. By pass this check can be done by setting “CheckForIllegalCrossThreadCalls” property to false inside “SplashForm.cs”. But it is not recommended. And here is the solution for this cumbersome problem:

1. Add a new public method to “SplashForm.cs” like following:

```
public void ShowModal(ref bool running)
{
    this.Show();
    while (running)
    {
        Update();
        Application.DoEvents();
    }
    this.Close();
}
```

2. Modify “DialogManager.cs” to use newly added method like following:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace CrossThreading
{
    public sealed class DialogManager {
        #region Singleton Implementation
        #region Private Members
        private static DialogManager current;
        private static object synObject = new object();
        #endregion
        #region Property Declarations
        public static DialogManager Current{
            get{
                if (current == null) {
                    lock (synObject){ // Double Checked Null Design Pattern
                        if (current == null)
                            current = new DialogManager();
                    }
                }
                return current;
            }
        }
        #endregion
    }
}
```

```

#region Internal Constructor
internal DialogManager()
{ }
#endregion
#endregion
#region Instance Data
private bool isShowingSplashScreen = false;
private SplashForm mySplashForm = null;
#endregion
#region Public Methods
public void ShowSplashScreen()
{
    mySplashForm = new SplashForm();
    System.Threading.Thread thDoSplash =
        new System.Threading.Thread(
            new System.Threading.ThreadStart(DoSplash)
        );
    isShowingSplashScreen = true;
    thDoSplash.IsBackground = true;
    thDoSplash.Start();
}
public void CloseSplashScreen()
{
    isShowingSplashScreen = false;
}
#endregion
#region Private Helpers
private void DoSplash()
{
    mySplashForm.ShowDialog(ref isShowingSplashScreen);
}
#endregion
}
}

```

As given in the code, passing a reference between threads in a shared memory (it is only possible if both threads are in same process domain) and simply modifying this reference variable when needed is a resident solution. Also this can be achieved



by using delegates and asynchronous method calls. But this is the simplest solution for cross threading problem. And here is the splash screen of Weendigo:

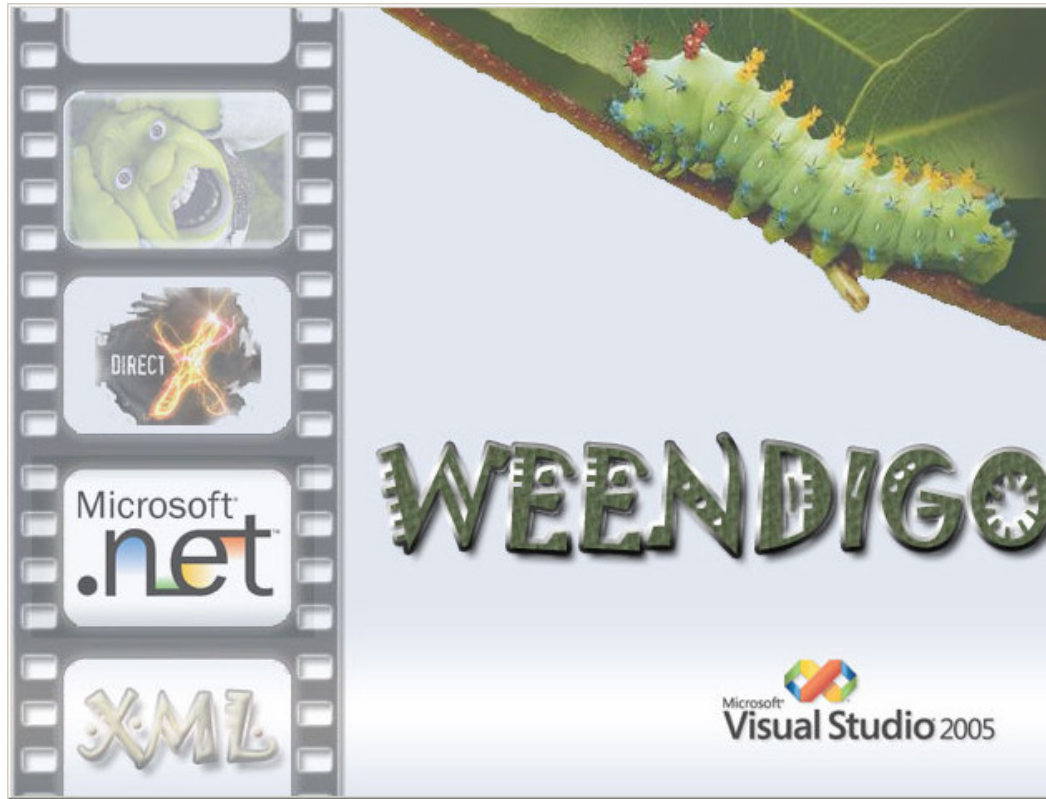


Figure 7.3 Weendigo splash screen

Weendigo is not just a tool to creating 3D animation films. Additionally, it is a secure application. All of the security issues are investigated and solved at design time of the project not after starting development. It is possible to say, Weendigo is secure by design.

## 7.2 Dynamic Code Injection

A programming language is a stylized communication technique intended to be used for controlling the behavior of a machine (often a computer). Like human languages programming languages have syntactic and semantic rules used to define meaning. There are different types of classification for programming languages. A common approach is classifying them either a scripting language or a compiled

programming language. Scripting languages are intended to be used for batch and job control operations. A compiled language is processed once and reduced to a simpler form that allows it to run faster than a script that has to be reprocessed every time. Scripting languages run inside another program, and interpreted unlike compiled programming languages. As these differences pointed out, there is a sharpen line between both, and have different advantages to each. In some cases we need to simulate scripting language features in a compiled programming language, which requires on-the-fly dynamic code generation. As scripting languages are interpreted, it is easy to generate and execute code for specific cases. Thus, allows an application to have different attitudes which are determined at run time.

JavaScript is a scripting programming language based on the concept of prototypes. The language is best known for its use in websites, but is also used to enable scripting access to objects embedded in other applications. There is a well known method named “eval” (short term of evaluate) intended to build dynamic codes. Here is an example usage of “eval” method:

```
<html>
<script language=javascript>
<!--
function useEval()
{
    alert(eval(sampleForm.textfield.value))
}
//-->
</script>
<body>
<form id="sampleForm" name="sampleForm" >
<input type="text" name="textfield" id="textfield">
<input type="button" name="calculateButton" id="calculateButton" value="Calculate"
onclick="javascript:{ useEval()}" />
</form>
</body>
</html>
```

When you browse this html code in a browser, you will get a simple user interface like following:



Figure 7.4 User interface will be shown when the prior instance code is browsed in a browser

When you type “new Date()” into input box and click calculate button you will see a message box showing current date and time information which means you will get a date time data as a return value of “eval” method. When you type “3\*5” into input box and click calculate button you will see a message box showing “15” which means you will get an integer data as a return value of “eval” method. During development process, we do not know anything about return value and implementation details of eval method. Therefore, we have no idea what eval method does inside.

### ***7.2.1 Why Weendigo Needs On-the-fly Code Generation?***

Weendigo is not just a tool to create 3D animation films also it is a Robust IDE. It is easy to integrate your own components with Weendigo. Weendigo supports objects that have inherited from “BaseDisplayObject” abstract class. Inheritors may have different constructors since they have different needs. It is possible to query constructor method information using “System.Reflection” namespace. But instantiating of a new object requires the knowledge of these parameter’s values at runtime. This is a common problem that you will face if you are designing and developing an Integrated Development Environment. Therefore, developers must inform Weendigo with the default parameters which are required to operate on their objects properly. Weendigo overcomes this problem via the following way:

Weendigo includes an attribute for class declaration, called `ToolboxDescriptorAttribute` that enables developers specify the runtime values for

their objects. Thus enforce weendigo to have a support for scripting in a compiled programming language. As this is not a build-in feature in any programming language, Weendigo simulates this need, by code generation, compiling, linking and executing specified code snippet. ToolboxDescriptorAttribute has an implementation like following:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace SystemFramework
{
    [AttributeUsage(AttributeTargets.Class)]
    public sealed class ToolboxDescriptorAttribute : Attribute
    {
        #region Instance Data
        private string descriptorText;
        private string defaultParameterString;
        private char defaultParameterDelimiter;
        private Type thisType;
        #endregion
        #region Property Declarations
        public string Descriptor
        {
            get
            {
                return descriptorText;
            }
            set
            {
                descriptorText = value;
            }
        }
        public object[] ParameterArray
        {
            get
            {
                string[] strArray= defaultParameterString.
```

```

        Replace("\", "").
        Split(defaultParameterDelimiter);
object[] parameters = new object[strArray.Length];
int i=0;
foreach (string str in strArray)
{
    if (str == "null")
    {
        parameters[i++] = null;
        continue;
    }
    parameters[i++] = DynamicCode.Current.Eval(str);
}
return parameters;
}
}
public Type ThisType
{
    get { return thisType; }
    set { thisType = value; }
}
#endregion
#region Constructor Declarations
public ToolboxDescriptorAttribute(string pDescriptorText, string pDefaultParameterString)
{
    descriptorText = pDescriptorText;
    defaultParameterString = pDefaultParameterString;
    defaultParameterDelimiter = ';';
}
public ToolboxDescriptorAttribute(string pDescriptorText, string pDefaultParameterString,
char pDefaultParameterDelimiter)
{
    descriptorText = pDescriptorText;
    defaultParameterString = pDefaultParameterString;
    defaultParameterDelimiter = pDefaultParameterDelimiter;
}
}
#endregion
}
}

```

It is easy to use this attribute on a class declaration and here is a sample usage:

```
using SystemFramework;
```

```
namespace Weendigo.PNeuma
{
    [ToolboxDescriptor("Weendigo Static Object", "", 1.0f)]
    public class StaticObject: BaseDisplayObject
    {
        //Implementation details are not given for clarity
    }
}
```

Toolbox enumerates all objects which are inherited from BaseDisplayObject and have these attribute on class declaration. Constructor parameters are evaluated and assigned by using this attribute declaration at run time.

### ***7.2.2 What is Code Dom?***

Microsoft .NET Framework includes a mechanism called the Code Document Object Model (CodeDOM) that enables developers of programs that emit source code to generate source code in multiple programming languages at run time, based on a single model that represents the code to render.

The System.CodeDom namespace defines types that can represent the logical structure of source code, independent of a specific programming language. The System.CodeDom.Compiler namespace defines types for generating source code from CodeDOM graphs and managing the compilation of source code in supported languages. Compiler vendors or developers can extend the set of supported languages.

Language-independent source code modeling can be valuable when a program needs to generate source code for a program model in multiple languages or for an uncertain target language. For example, some designers use the CodeDOM as a

language abstraction interface to produce source code in the correct programming language, if CodeDOM support for the language is available.

Microsoft .NET Framework includes code generators and code compilers for C#, JScript, and Visual Basic.

### ***7.2.3 Weendigo On-the-fly Code Generation***

Weendigo includes a sealed class implemented with Singleton design pattern named as “DynamicCode”. This class has an Eval method JavaScript like, gets a string parameter and returns evaluate value inside an object. This class implementation details are given below:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.CodeDom.Compiler;
using System.Reflection;

namespace SystemFramework
{
    public sealed class DynamicCode
    {
        #region Singleton Implementation
        #region Private Members
        private static DynamicCode current;
        private CompilerParameters defaultCompilerParams = null;
        private static StringBuilder sbStart = new StringBuilder("");
        private static StringBuilder sbEnd = new StringBuilder("");

        private static object synObject = new object();
        #endregion
        #region Property Declarations
        public static DynamicCode Current
        {
```

```

get
{
    if (current == null)
    {
        lock (synObject)
        { // Double Checked Null Design Pattern
            if (current == null)
                current = new DynamicCode();
        }
    }
    return current;
}
}
#endregion
#region Internal Constructor
internal DynamicCode()
{
    sbStart.AppendLine("using System;");
    sbStart.AppendLine("namespace Weendigo.SystemFramework.DynamicCode{");
    sbStart.AppendLine("public class CSCodeEvaluator{");
    sbStart.AppendLine("public object Eval(){");
    sbStart.Append("return ");
    sbEnd.AppendLine(";");
    sbEnd.AppendLine("}");
    sbEnd.AppendLine("}");
    sbEnd.AppendLine("}");
    defaultCompilerParams = new CompilerParameters();
    defaultCompilerParams.CompilerOptions = "/t:library";
    defaultCompilerParams.GenerateInMemory = true;
    defaultCompilerParams.ReferencedAssemblies.Add("system.dll");
}
#endregion
#endregion
#region Public Methods
public object Eval(string pEvalString)
{
    CodeDomProvider compiler = null;
    StringBuilder sbSourceCode = new StringBuilder();
    try

```



```

    {
        compiler = CodeDomProvider.CreateProvider("C#");
        sbSourceCode.Append(sbStart.ToString());
        sbSourceCode.Append(pEvalString);
        sbSourceCode.Append(sbEnd.ToString());
        CompilerResults cr = compiler.CompileAssemblyFromSource(
            defaultCompilerParams,
            sbSourceCode.ToString());
        if (cr.Errors.Count > 0)
            return null;
        Assembly a = cr.CompiledAssembly;
        object dynamicObject =
a.CreateInstance("Weendigo.SystemFramework.DynamicCode.CSCEvaluator");
        Type t = dynamicObject.GetType();
        MethodInfo mInfo = t.GetMethod("Eval");
        return mInfo.Invoke(dynamicObject, null);
    }
    finally
    {
        if (compiler != null)
            compiler.Dispose();
    }
}
#endregion
}
}

```

Eval method simply creates an instance of C# compiler as a first step of the evaluation process. Then, it merges predefined code required for a standard eval methods and specified string parameter. It compiles this merged string as a source code. If compilation succeeded, a dynamic object is created and Eval method is called by using MethodInfo.Invoke method. Return value of this function is interpreted as return value of eval method. Here is a sample code generated in memory by weendigo returning “1.0f”.

```

using System;
namespace Weendigo.SystemFramework.DynamicCode

```

```

{
    public class CSEvaluator
    {
        public object Eval()
        {
            return 1.0f;
        }
    }
}

```

This code is compiled as a binary assembly which has a reference “System.dll”. By this implementation, Weendigo also achieves scripting in a compiled language problem which is a common software engineering problem.

### 7.3 Exception Handling

Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of some condition that changes the normal flow of execution. The condition is called an exception. There are a numerous type of exceptions, but all can be categorized in two main subsets: First chance exceptions and second chance exceptions. Handle and /or raise exceptions are developer choice but must be determined at design time.

In general, current state will be saved in a predefined location and execution will switch to a predefined handler. Depending on the situation, the handler may later resume the execution at the original location, using the saved information to restore the original state. For example, an exception which will usually be resumed is a page fault, while a division by zero usually cannot be resolved transparently.

A piece of code is said to be exception-safe if run-time failures within the code will not produce ill-effects, such as memory leaks, garbled data or invalid output. Exception-safe code must satisfy invariants placed on the code even if exceptions occur. There are several levels of exception safety:

- Failure transparency, operations are guaranteed to succeed and satisfy all requirements even in presence of exceptional situations. (best)
- Commit or rollback semantics, operations can fail, but failed operations are guaranteed to have no side effects.
- Basic exception safety, partial execution of failed operations can cause side effects, but invariants on the state are preserved (that is, any stored data will contain valid values).
- Minimal exception safety, partial execution of failed operations may store invalid data but will not cause a crash.
- No exception safety, no guarantees are made. (worst)

Usually at least basic exception safety is required. Failure transparency is difficult to implement, and is usually not possible in libraries where complete knowledge of the application is not available.

Exceptions can be categorized into two main subsets: First chance exceptions and second chance exceptions. A first chance exception is non-fatal unless it is handled correctly by using an error handler. If this problem occurs, the exception is raised again as a second chance exception. The application is either not in debug mode or no attached debugger handles a second chance exception, underlying operating system handles this exception but the application quits.

It is a common practice to use structured exception handling as a signaling mechanism. Some application programming interfaces register an exception handler in anticipation of a failure condition that is expected to occur in a lower layer. When an exception is raised, the handler may correct or ignore the condition rather than allow a failure to propagate up through intervening layers. This is very useful in complex environments such as networks where partial failures are expected and it is not desirable to fail an entire operation just because one of several optional parts failed. In this case, the exception can be handled so that the application does not recognize that an exception has occurred.

However, if the application is being debugged, the debugger sees all exceptions before the program does. This is the distinction between the first and second chance exception: the debugger gets the first chance to see the exception (hence the name). If the debugger allows the program execution to continue and does not handle the exception, the program will see the exception as usual. If the program does not handle the exception, the debugger gets a second chance to see the exception. In this latter case, the program normally would crash if the debugger were not present.

If you do not want to see the first chance exception in the debugger, you should disable first chance exception handling for the specific exception code. Otherwise, when the first chance exception occurs, you may need to instruct the debugger to pass on the exception to the program to be handled as usual.

Hence Weendigo is a tool to create 3D animation films, raising an exception at last frame of a two hours length movie is not a desired situation. Weendigo tries to handle all types of exceptions in a common manner. According to exception importance Weendigo decides what to do. For instance, when an exception is caught at design time, weendigo will report this exception to user and prepares a dump that contains detailed information which will be helpful for Weendigo developer to overcome this exception.

By the way, if same exception is caught at render time, weendigo will only log exception details and will not report exception to user unless this exception is a block to complete rendering operation. Exception details are logged to Windows Event Log and/or a text file.

Nevertheless, Weendigo is tested with debugging tools to monitor first chance and second chance exceptions. This is an important requirement for Weendigo, because rendering a 3D scene needs more memory and in this case a memory leak could be costly. “Microsoft debugging tools for Windows” is used to monitor first chance and second chance exceptions. Here is a list of used tools:

- ADPlus (adplus.vbs), also known as Autodump+, is a console-based Microsoft Visual Basic script. It automates the CDB debugger to produce memory dumps and log files that contain debug output from one or more processes.
- Application Verifier is a tool for testing user-mode applications. It can monitor their actions, put them through a variety of stresses and tests, and report on possible problems or errors in their design.
- Dr. Watson for Windows, drwtsn32.exe, creates memory dump files from user-mode programs which have crashed
- WinDbg is a user-mode and kernel-mode debugger with a graphical interface.

### 7.3.1 ADPlus

ADPlus has two modes of operation:

- **Hang mode** is used to troubleshoot process that hangs, 100 percent CPU utilization, and other problems that do not involve a crash. When you use ADPlus in hang mode, you must wait until the process hangs before you run the script.
- **Crash mode** is used to troubleshoot crashes that would normally activate a postmortem debugger, or any other type of error that causes a program or service to terminate unexpectedly. When you use ADPlus in crash mode, you must start ADPlus before the crash occurs. ADPlus can be configured to notify an administrator or a computer of a crash through the -notify option.

In hang mode, ADPlus immediately produces full memory dumps for all of the processes that are specified on the command line after the script is finished running. Each .dmp file created goes into a folder that contains the date/time stamp from when ADPlus was run, and each file name contains the process name, the process ID, and the date/time stamp from when ADPlus was run. While the process memory is being dumped to a file, the process is frozen, with the debugger non-invasively attached to it. After the memory dump is finished, the debugger detaches and the process resumes.

Each kind of exception (access violation, stack overflow, and so on) can be raised to a debugger as either a "first chance" or "second chance" exception. First chance exceptions are, by definition, non-fatal unless they are not handled properly with an error handler, at which point they are raised again as a second chance exception (which only a debugger can handle). If no debugger handles a second chance exception, the application is shut down.

By default, when ADPlus detects a first chance (non-fatal) exception for all types of exceptions except unknown and EH exceptions, it takes the following actions:

1. Pauses the process to log the date and time that the exception occurred in the log file for the process that is being monitored.
2. Logs the thread ID and call stack for the thread that raised the exception in the log file for the process that is being monitored.
3. Produces a uniquely-named minidump file of the process at the time the exception occurred and then resumes the process.

By default, ADPlus does not produce a unique minidump file for first chance EH and unknown exceptions because it is quite common for a process to encounter a significant number of these exceptions, which are usually handled by error handling code within a process or DLL. In other words, these exceptions are usually handled exceptions, and they do not become second chance (unhandled) exceptions, which terminate the process.

When ADPlus detects a second chance (fatal) exception for all types of exceptions (including EH and unknown exceptions), it takes the following actions:

1. Pauses the process.
2. Logs the date and time that the exception occurred.
3. Logs the thread ID and the call stack for the thread that raised the exception.

4. Produces a full memory dump of the process at the time the fatal exception occurred.
5. Exits the debugger. This terminates the process.

### ***7.3.2 Application Verifier***

Application Verifier requires a utility named AppVerif.exe. This tool can be used alone or in conjunction with a user-mode debugger.

Application Verifier reports information about the target process in several different ways:

- By recording information in logs that can be viewed through AppVerif.exe
- By recording information in logs that can be viewed through the debugger
- By raising an exception, causing the application to break into the debugger and display information about the exception
- By issuing an Application Verifier Stop, causing the application to break into the debugger and display a status message

Only exception logs are recorded for Weendigo tests. The exception log records all exceptions that have occurred in the target process.

### ***7.3.3 Dr. Watson***

The Dr. Watson for Windows program (drwtsn32.exe) is preinstalled in your system directory (typically c:\winnt\system32) when Windows is set up. The default options are set the first time Dr. Watson for Windows runs, which can be either when an application error occurs or when you run it from the command prompt or the Run dialog box. To run Dr. Watson, the following command line is generally used:

*drwtsn32 -p ProcessID*

By default, Dr. Watson is set up to save a memory dump file immediately upon the failure of a user-mode component. By default, this file is named `user.dmp`. Its default location depends on the version of Windows that is being used:

- In Windows NT, the default location is the `%UserProfile%\Local Settings` folder.
- In Windows 2000, the default location is the `%AllUsersProfile%\Documents\DrWatson` folder.
- In Windows XP and later versions of Windows, the default location is the `%AllUsersProfile%\Application Data\Microsoft\Dr Watson` folder.

This dump file is as large as the amount of RAM dedicated to the user-mode process on the system that failed. It includes the contents of the failed user-mode's memory segment at the time that the error occurred.

As with any user-mode debugging session, symbols for the appropriate executables and DLLs must exist on the local computer. For non-existent symbol files, each of them are downloaded from the registered microsoft symbol file web site which is located at the URL <http://msdl.microsoft.com/download/symbols>.

#### ***7.3.4 WinDbg***

WinDbg has ten kinds of debugging information windows. Eight of these are individual windows that can be visible or invisible: the Debugger Command window, the Watch window, the Locals window, the Registers window, the Calls window, the Disassembly window, the Processes and Threads window, and the Scratch Pad. In addition to these eight individual windows, WinDbg can display any number of Source windows and Memory windows.

All problems found by these debugging tools are investigated and all found bugs in Weendigo implementation are fixed. Weendigo guarantees that underlying implementation is done by considering exception safety rules and it is very close to



failure transparency. But the words of “Weendigo is unbreakable” are very pretentious words. But weendigo is very close to be unbreakable software.

## 7.4 Performance Issues

The vast majority of graphics applications are CPU-limited on video hardware with high-performance graphics processing units (GPU). This is sometimes due to poor use of batching for draw submissions, but more typically this is due to other graphics systems consuming a large portion of the available CPU cycles. In cases, it is due to very high fill rate or pixel shader demand in high-resolution settings or on video cards with lower vertex-shader performance. Because most titles are CPU-limited, the biggest performance wins come from optimizations made to CPU-intensive graphics applications. Typically, the AI or physics systems and the associated collision detection logic are the primary consumer of CPU cycles in well-behaving Microsoft Direct3D applications especially in game systems. Any work to improve these systems can improve the overall performance.

Achieving good parallelism with the GPU requires that draw batches contain enough geometry - and the shader have the right complexity - to keep the video card busy, while not using so many batches that the command buffer is flooded. On current generation hardware, it is recommended approximately three hundred or less draw batch submissions per frame (fewer on lower-end CPUs) to prevent the driver's processing of the command-buffer from becoming a performance bottleneck. Some other application programming interface (API) state calls and driver combinations can result in high CPU cost (driver compiling of shader for example), so it is a highly recommend routine performance analysis with PIX and other tools.

“PIX” is a tool for analyzing, optimizing, and debugging Direct3D applications. “PIX” is designed to capture detailed information from an application while it is running. “PIX” can be configured to gather data such as the Direct3D APIs called, timing information, and mesh vertices before and after transformations, screenshots,

and various statistics. PIX can also be used for vertex and pixel shader debugging including setting breakpoints and stepping through shader code.

During the development of most PC titles, developers use convenient data structures and strings for content management. The CPU work required for string comparison, copying, and other manipulations often has a measurable overhead, particularly when taking into account the associated cache and memory subsystem hit. Plans should be made when developing these systems for removing or minimizing the reliance on string processing once the product enters into the primary testing and release phases.

Even with the widespread availability of accelerated graphics port (AGP), Modern video hardware performs well when dealing with static data. High-end cards often have very large video memories, but this memory cannot be effectively utilized by dynamic data. While reasonably efficient usage patterns of dynamic vertex buffers/index buffers can be implemented for dynamic content, many titles overuse this idiom for what is otherwise static content. Putting as much content into static resources as possible can greatly reduce the bandwidth overhead of transferring data to the video card, makes better use of on-board VRAM, and reduces the CPU/cache overhead involved in processing this content.

PC games have gotten a reputation for long load times, particular when compared with console titles with strict loading-time requirements. The overhead of opening a file is usually much higher than developers typically think. With on-demand virus scanners in widespread use, and the additional functionality of NTFS, opening a file is a fairly expensive operation. Opening many files at once or opening and closing the same file repeatedly is therefore a poor method of dealing with file management. Some graphics applications have attempted to mitigate this performance cost by doing "file exist" tests before opening a file. The reality is that the "file exists" test on NTFS requires doing a file open, so this results in paying the cost twice.

For games that allow mods (an expression for the act of changing a piece of software or hardware to do a function that was not designed or authorized by the original manufacturer) or still include development scaffolding to check for override data files, the additional checks for these files can add a significant delay to the load when they are not present. It is recommended that graphics applications only check for these files when run with a special command-line switch or other mode indicator so that only those making use of this functionality actually pay the performance cost of these (often extensive) checks.

Additional performance can be obtained from the file system by appropriate use of the file system `FILE_FLAG_RANDOM_ACCESS` and `FILE_FLAG_SEQUENTIAL_SCAN` hints, reasonable sized buffers to avoid large numbers of calls to the OS read/write APIs, asynchronous file I/O, and background loading threads. It is also recommended using offline (build or install time) data conversion processes rather than relying on on-load data conversion, as these typically remain in use even for the final release and impose a significant performance tax for every user.

A number of issues related to the 32-bit limit on process virtual memory space. While 2 GB of user virtual address space has been more than adequate historically, the increased use of large memory-mapped files, custom memory allocations, and increasing VRAM size (which must be mapped into process space) has started to cause situations where virtual memory space allocations are failing. These problems most often appear when the game makes use of a custom memory allocation scheme that attempts to allocate a large continuous chunk of virtual memory space. Recommendation is to write allocations such that they request more reasonable sized portions of the virtual address space (for example, 64 or 256 MB at a time, but not 1 GB) as needed, although care should be taken to not cause further fragmentation. The advent of 64-bit operating systems and hardware will greatly help these issues in the long term, but care must be taken on current generation 32-bit systems.

As a debugging aid, some developers have been enabling exceptions on the floating-point unit (FPU) via manipulations of the floating-point control word. Doing this is highly problematic and will likely cause the process to crash. Just like the calling convention requires the “ebx” register be preserved, the majority of the system assumes that the FPU is in a default state, will give reasonable results, and won't generate exceptions. Drivers and other system components will often compute results assuming that standard error values will appear in the registers for bad conditions, and if exceptions are enabled, these will go unhandled and result in crashes.

Direct3D will set the floating-point unit to "single-precision, round-to-nearest" as part of initialization, unless the `D3DCREATE_FPU_PRESERVE` flag is used, in which case the FP control word is untouched. In any libraries where we need to have different rounding rules or other behavior - such as dealing with software vertex shaders or compilation - we save and restore the control word. If the game needs to make use of non-standard rounding or FPU exceptions, it should save and restore the FP control word and be sure to not call any external code not proven to be safe from these problems, including system APIs.

## CHAPTER EIGHT

### CONCLUSION

Weendigo idea has been talked at the year 2003. Weendigo project is planned and first version is ready to use at the year 2006. Weendigo provides a set functionality to prepare 3D animation films. An animation film is definitely differs from a computer game. A comparison between them from a development perspective is given in the following table.

Table 8.1 Comparison between computer games and animation films

A 3D Game	A 3D Film
Designated to execute properly in a PC	Output should be a video file
User interaction is important	No user interaction is required
Camera can be controlled by user	Camera control must be done in design time
Requires more complex game engine	Requires more robust graphics engine
Graphics engine must test underlying hardware capabilities (user computer capabilities are not known at design time)	Underlying hardware capabilities is important at rendering process

Please, note that only main differences are given.

Most of the roles in game development and animation film development intersect but additionally an animation film has to have a scenario management and camera management. An animation film generation project should follow the steps given below:

- **Determination of film subject:** definitions of focus points should be enough whole story definition is not needed.
- **Defining requirements:** film requirements and main characters should be defined. A draft scenario should be defined at this step.

- **Scenario Management:** Entire scenario should be completed main character's draft appearances must be included.
- **Project Planning:** Project plan can be defined at this step. Project manager leads the project team to develop a detail and comprehensive project plan based on the preliminary project plan and requirements document
- **Character Preparation:** Entire animated and non animated characters, titles, colors, lights, camera positions should be defined at this step.
- **Execution:** Integrating prepared characters and 3D objects and embedding whole in a single scene.
- **Draft movie generation:** After finalizing scene preparation project team should generate movie and checks to see whether the output movie meets project requirements and movie is mapping with scenario.
- **User Acceptance:** a draft version of movie is presented to the project owner. Minor adjustment can be performed at this step.
- **Final movie generation:** animation film is generated and ready to distribute.

As you can see, these steps are covers must of the SDLC process steps. Movie generation process can be disciplined and controlled by using SDLC. But there is a missing part of this process. It is need for a tool to manage steps after character preparation. Weendigo is implemented to meet this need. Current version of Weendigo might be insufficient to produce an animation like movies listed above. By the way, Weendigo serves an important Integrated Development Environment to manage and produce an animated film.

## REFERENCES

Adams J. (2003). *Advanced Animation with DirectX*. Boston: Premier Press

IMDB (2006), *3D Animation Movie List*, (n.d.), <http://www.imdb.com>

Kovach, P.J. (2000). *Inside Direct 3D*, Washington: Microsoft Press

Microsoft Corporation (2005), *Best Security Practices in Game Development*, December 2005, <http://msdn.microsoft.com/security/>

Microsoft Corporation (2005), *Crash Dump Analysis*, December 2005, <http://www.microsoft.com/whdc/devtools/debugging/default.msp>

Microsoft Corporation (1998), *The Direct3D Transformation Pipeline*, April 1998, <http://msdn.microsoft.com/directX>

Microsoft Corporation (2006), *Microsoft DirectX SDK Library*, (n.d), <http://msdn.microsoft.com/directX/SDK>

Microsoft Corporation (2006), *Microsoft Media Format SDK Library*, (n.d), <http://msdn.microsoft.com/windowsmedia/techpages/wmformat>

Pesce, M. D. (2003). *Programming Microsoft DirectShow For Digital Video and Television*, Washington: Microsoft Press

## APPENDICES

List of 3D animation films (including upcoming) in chronological order

- 1995
  - Toy Story
- 1998
  - Antz
  - A Bug's Life
- 1999
  - Toy Story 2
- 2000
  - Dinosaur
- 2001
  - Shrek
  - Final Fantasy: The Spirits Within
  - Monsters, Inc.
  - Jimmy Neutron: Boy Genius
- 2002
  - Ice Age
  - Jonah: A VeggieTales Movie
- 2003
  - Finding Nemo
- 2004
  - Ark
  - Homeland (film)
  - Shrek 2
  - Shark Tale
  - Terkel in Trouble (Denmark, "Terkel i knibe")
  - The Incredibles
  - The Polar Express
- 2005
  - The Magic Roundabout (aka Doogal)



- Robots
  - Valiant
  - Madagascar
  - Midsummer Dream
  - Chicken Little
  - Hoodwinked
- 2006
  - Doogal
  - Ice Age: The Meltdown
  - The Wild
  - Over the Hedge
  - Cars
  - Monster House
  - Barnyard
  - Khan Kluay
  - The Ant Bully
  - Open Season (Sep. 2006)
  - Everyone's Hero (Sept. 2006)
  - Flushed Away (Nov. 2006)
  - Happy Feet (Nov. 2006)
- 2007
  - Meet the Robinsons (est Mar. 2007)
  - Teenage Mutant Ninja Turtles (Mar. 2007)
  - Foodfight! (est. April 2007)
  - Shrek the Third (est. May 2007)
  - Surf's Up (est. June 2007)
  - Ratatouille (est. June 2007)
  - Hood vs. Evil (est. 2007)
  - Bee Movie (est. November 2007)
  - Happily N'Ever After
- 2008
  - 9 (est. 2008)
  - Cat Tale (est. 2008)

- Kung Fu Panda (est. May 2008)
  - Where the Wild Things Are (est. May 2008)
  - American Dog (est. Sum 2008)
  - Madagascar 2 (est. Fall 2008)
  - The Smurfs (est. November 2008)
  - Punk Farm (film) (est. 2008)
- 2009
  - Rapunzel (est. 2009)
- 20??
  - W.A.L.-E
  - Toy Story 3
  - Shrek 4
  - Puss In Boots