Project MM - XNA Framework | Report

Daniel Loran & Stefan van de Kaa

2008 - 2009

Faculty of Computer Science

Vrije Universiteit Amsterdam

Contents

1 Introduction

XNA Framework was developed by Microsoft to provide a managed runtime environment and a set of tools that promotes rapid game development process.

In this project we explore the fundamentals of XNA Framework by discussing code examples organized in labs, providing tips and "knowabouts" that simplify development tasks and lead to efficient object oriented game architecture.

This report includes the following chapters:

XNA Game Development Environment - describes the tools and technological environment required to develop XNA applications for Windows and Xbox 360 game console.

Creating Windows Game Project with Visual Studio 2005 – describes creation of a basic XNA project.

Drawing – explores the theory of drawing XNA primitives.

Shaders – provides introduction into Graphics Pipeline, rendering and shader definition language.

Creating virtual world - describes the process of importing 3D models into Maya, exporting models from Maya to XNA, setting up the camera, loading models into XNA environment and simple animation.

Advanced Animation Techniques - describes how to animate all parts of a model separately from each other and how to implement elliptical motion path animation with focus on object oriented techniques.

Lighting and Special Effects – adds lighting and fog to the scene.

Appendix A: Maya to FBX Export – describes in detail the steps of exporting Maya models to FBX format that can be loaded into XNA environment.

Bibliography and References - provides references to tools, study material, books, relevant websites, free model archives and XNA tutorials.

2 XNA Game Development Environment

Microsoft XNA Game Studio 2.0 [XNA] is a free add-on to a commercial product Visual Studio 2005 or to a free Visual Studio C# 2005 Express Edition [C# IDE]. The .Net Framework version 2.0 is traditionally required for all this software to run and in most of the cases already present on any modern version of Windows.

The fact that both XNA Game Studio 2.0 and Visual Studio 2005 C# Express are provided for free makes it possible for anyone to develop games for Windows and Xbox 360 without any substantial investment. XNA is designed to be written once and run on both Windows and Game console. There are however some slight compiler differences between the two, e.g. Windows supports mouse input, while Xbox does not. With that being said, although XNA development for Windows platform can be made completely free, the same for Xbox requires purchasing the Xbox (priced around 350 USD) and if you would like to make your game available to the game players worldwide, also purchasing the subscription to the Xbox 360 Creators Club (priced 99 USD per year). Putting this together one can conclude that Microsoft did really make XNA Game Development very accessible.

In this project all the development and testing will be done with Vista. Since we don't have an Xbox and did not pay Xbox 360 Creators Club membership fee we are also not able to check potential Xbox compatibility issues. In general, if everything works with Vista, it is safe to assume that it can be easily recompiled for Xbox (perhaps with some minor changes in the code if Xbox compiler complains about variable declarations and similar. It may complain because it is known to be stricter in nature.).

There are 2 types of Game projects in Visual Studio 2005: Xbox Game project and Windows Game project. Debugging an Xbox Game project with Visual Studio 2005 as the name suggests literally requires you to have Xbox connected to your PC and having your membership active. This is because when you press the debug button in Visual Studio 2005 your project is deployed to Xbox for compilation. The process is completely automated and controlled by Visual Studio 2005. If you work with Windows Game project instead, it does not require Xbox or any membership fees, and your code will be compiled and executed on Windows.

Except of these minor differences, the code of Windows Game and Xbox Game projects will most likely be the same.

3 Creating Windows Game Project with Visual Studio 2005

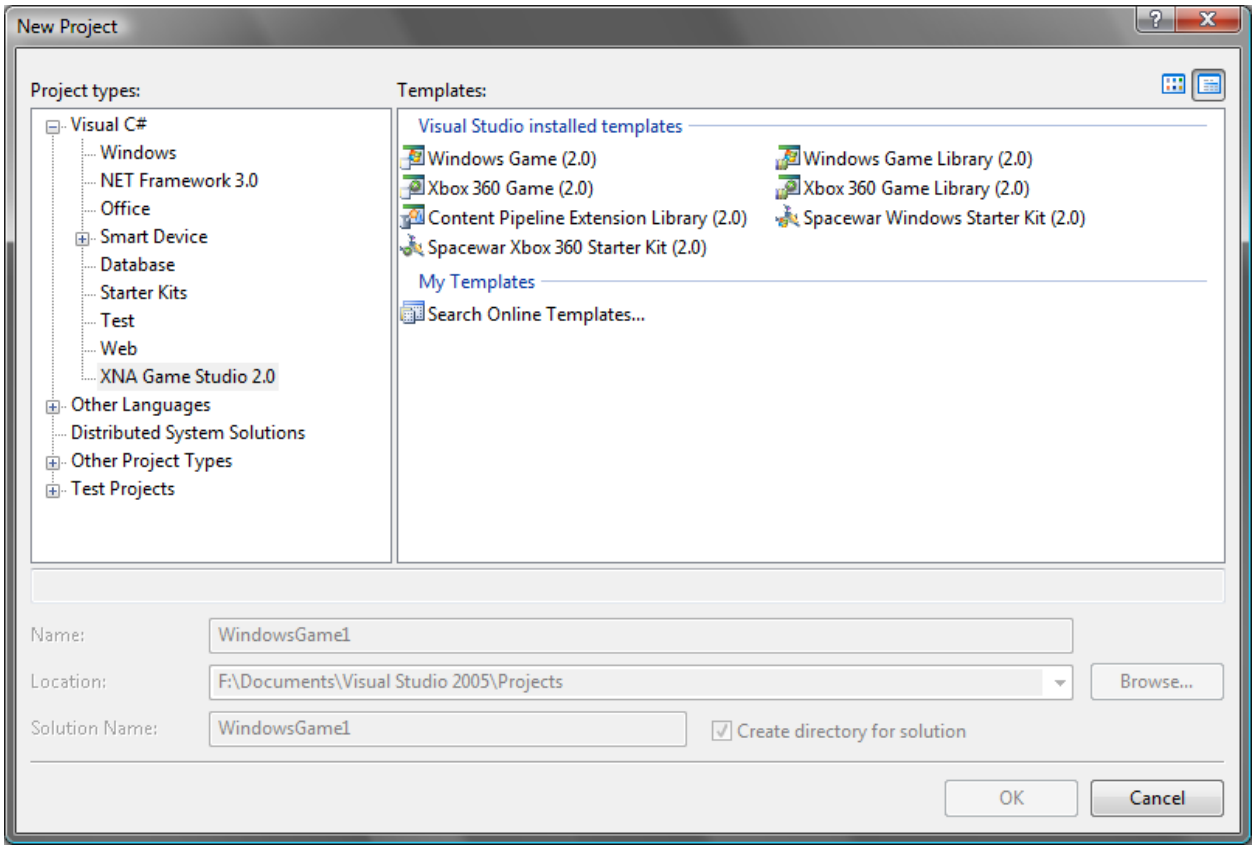
The source code for this chapter can be found at Labs/ WindowsGame1 folder.

Installing XNA Game Studio 2.0 on Vista after Visual Studio 2005 already has been installed adds new Game project definitions as shown in Figure 3.1 below.

As mentioned above, there are two different project types under the XNA Game Studio 2.0 project group: Windows Game (2.0) project and Xbox 360 Game (2.0) project. In the same place the corresponding Library projects can be found.

For the purpose of this lab we create a basic Windows Game (2.0) project which is by default called "WindowsGame1" as shown in Figure 3.1 below. The output of compiling this project is an empty window as shown in Figure 3.2 further in this text.

Figure 3.1 Creating New Game Project with Visual Studio 2005



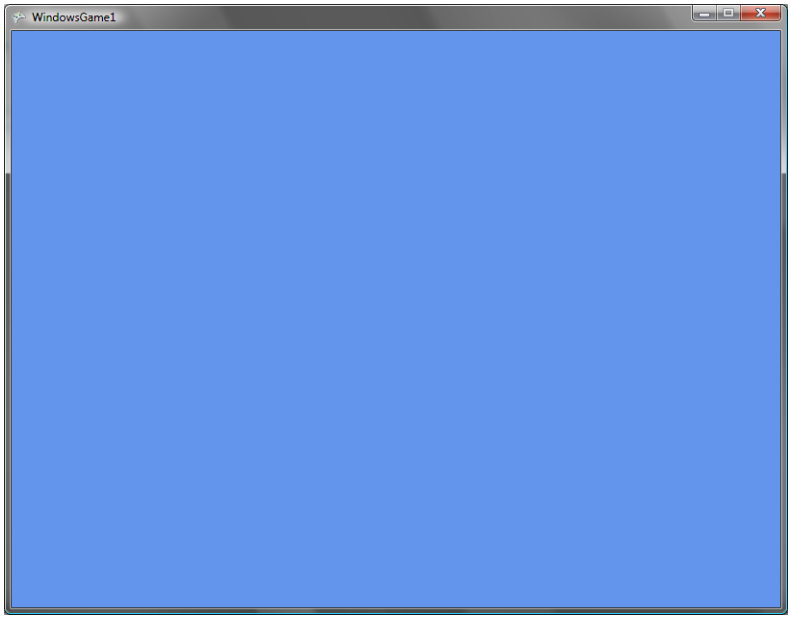
When new game project is created by default the following files are added automatically:

Game1.cs - contains the game loop

Program.cs – execution starts here in the main() function where Game1 object is instantiated

Game.ico – this is the small icon in the title bar of the default window generated as an output of this basic game project (see left-top corner of Figure 3.2 below).

Figure 3.2 WindowsGame1 project output



Our game application starting point is located at Program.cs in the Main() function as shown in Figure 3.3 below. This is where we create an instance of Game1 class that represents our game object. Game is started by executing the Run() method of game object.

Figure 3.3 Program.cs

```
using System;

namespace WindowsGame1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
// defines a scope at the end of which an object (in this case Game1) will be disposed
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
```

```
}
```

Our basic game object with all its methods is described in Figure 3.4. Game1 class inherits from the base Microsoft.Xna.Framework.Game class.

Constructor() - initializes GraphicsDevice and ContentManager object.

Initialize() - this is where all the initialization logic is located, in other words: window and application properties, views, loading and initializing resources such as shaders, vertices, audio and similar.

LoadContent() – is called once per game and therefore is the best place to load all the content necessary for the game before it starts.

UnloadContent() – similarly to the LoadContent() this function is called once per game only now all the content participating in the game will be unloaded so that the game can gracefully exit and release its resources.

Update() - allows the game to run logic such as updating the world, checking for collisions, gathering input, and playing audio.

Draw() – called each time when the game should (re)draw itself.

Figure 3.4 Game1.cs

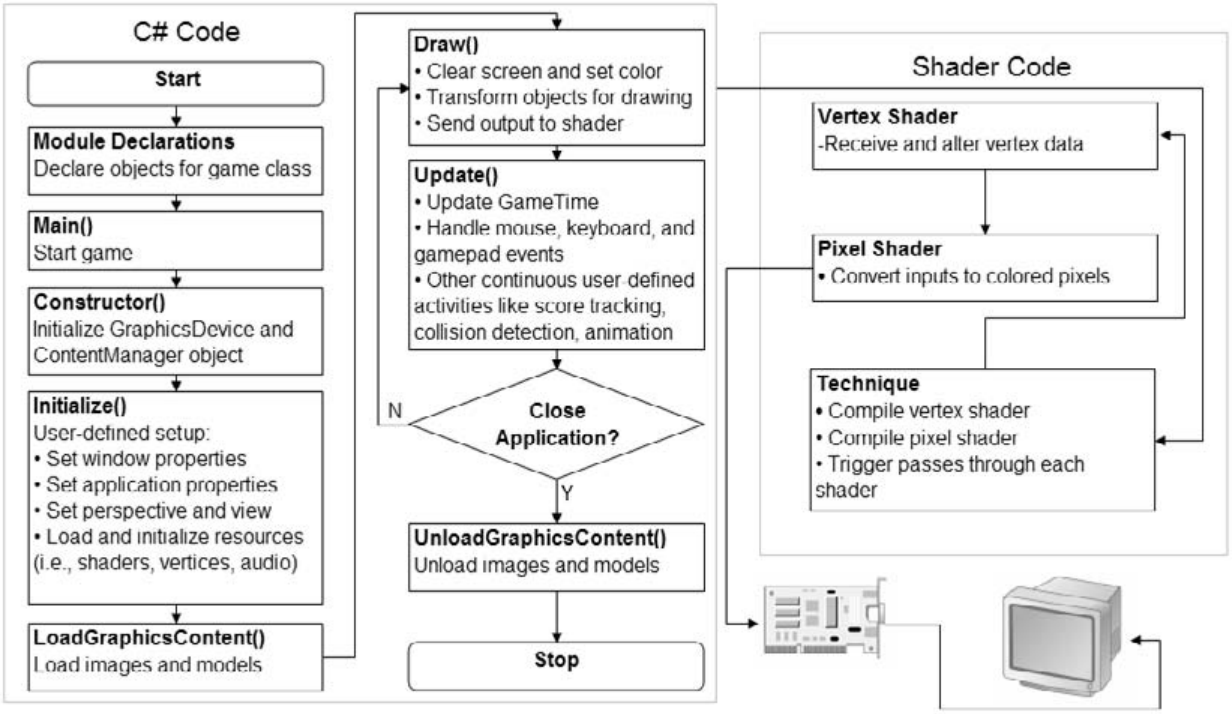
```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace WindowsGame1
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
```

```
        /// Allows the game to perform any initialization it needs to before starting
to run.
        /// This is where it can query for any required services and load any non-
graphic
        /// related content.  Calling base.Initialize will enumerate through any
components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
```

```
        }
    }
}
```

By now it should be clear how the game loop is implemented, but to make it more explicit we have included the flowchart in Figure 3.5 describing all the steps necessary to initialize the game, load the graphics and content, draw the scene, update scene elements, and finally unload the graphics and content before exiting the application.

Figure 3.5 Flowchart Game Loo

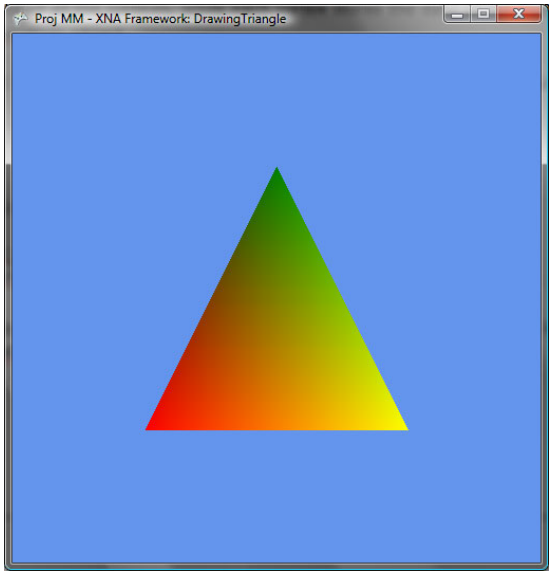


Source: [XNA Guide]

4 Drawing

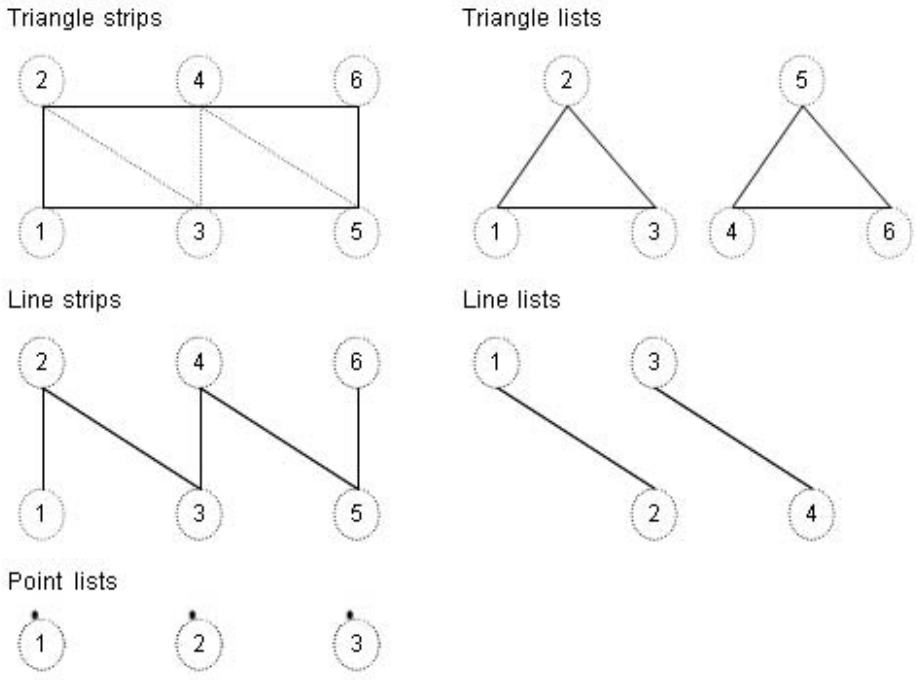
In this lab we will draw a primitive shape – a colored triangle as shown at Figure 4.1 below. Source code for this lab can be found at Labs/DrawingTriagle folder.

Figure 4.1 DrawingTriangle Lab



Shapes are built from series of points, lines, or triangles (see Figure 4.2).

Figure 4.2 Shapes



Source: [XNA Guide]

To draw primitive shapes we can use Lists or Strips. Lists can accommodate separate points, lines or triangles while Strips provide a more efficient way of storing data because vertices can be combined to create a complex shape. To keep it simple storing a complex 3D model in a Strip will most likely cut the memory requirements for vertex data in half than storing the same model in a List. XNA platform has several predefined Common Primitive Types: TriangleStrip, TriangleList, LineStrip, LineList and PointList.

To be able to draw any shape we will also need a Vertex Buffer, which is able to store the X, Y, Z coordinates, a normal vector and color. XNA platform has several predefined Storage Formats for Vertex Buffers: VertexPositionColor, VertexPositionTexture, VertexPositionNormal, VertexPositionNormalTexture.

To draw a shape we have to provide a list of all the vertices, while each vertex stores the data about its position in space and color attributes. We also need VertexDeclaration object to inform the device about the Storage Format of our Vertex Buffer so we could draw it later. In the case of triangle we use VertexPositionColor as a storage format.

The source code for this lab is shown at Figure 4.3 below. Code that was added for the purpose of this lab to the solution of the previous lab has a gray background.

We did add the following declarations:

```
        GraphicsDevice device;
        Effect effect;
        VertexPositionColor[] vertices;   // vertex buffer
        VertexDeclaration myVertexDeclaration;
```

Most of these declarations were already explained before. What's new here is the Effect object. In XNA in order to draw something we must define the Effect in a special file with .fx extension in which we specify the Techniques used by our effect. We will discuss the content of effects.fx file in the Shaders section further in this text.

Initialize() – code to set several graphic device properties was added.

init_triangle() – function that populates triangle vertices with data (coordinates and color of each vertex) and instantiates VertexDeclaration object.

LoadContent() - device is instantiated, content of effects.fx file is loaded and init_triangle() is executed.

What happens now inside the Draw() method deserves some special attention. As it was aready mentioned before, inside the effects.fx we define several Techniques we can use to influence the appearance of our Effect. In this case we use the "Pretransformed" technique. Since effect can be rendered in multiple passes through the Graphics Pipeline, each pass needs to be drawn separately. This is why for each Effect we will usually have a block of effect.Begin() statement closed by effect.End() statement inside which we traverse through the loop of passes we have defined in our Technique. Similarly inside the loop we have a block of pass.Begin() closed by pass.End(). Inside this block we will draw all the scene elements.

Figure 4.3 DrawingTriangle

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace DrawingTriangle
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GraphicsDevice device;
        Effect effect;
        VertexPositionColor[] vertices;
        VertexDeclaration myVertexDeclaration;


        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting
to run.
        /// This is where it can query for any required services and load any non-
graphic
        /// related content.  Calling base.Initialize will enumerate through any
components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here
            graphics.PreferredBackBufferWidth = 500;
            graphics.PreferredBackBufferHeight = 500;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "Proj MM - XNA Framework: DrawingTriangle";

            base.Initialize();
        }



        private void init_triangle()
        {
            vertices = new VertexPositionColor[3];
```

```
            vertices[0].Position = new Vector3(-0.5f, -0.5f, 0f);
            vertices[0].Color = Color.Red;
            vertices[1].Position = new Vector3(0, 0.5f, 0f);
            vertices[1].Color = Color.Green;
            vertices[2].Position = new Vector3(0.5f, -0.5f, 0f);
            vertices[2].Color = Color.Yellow;

            myVertexDeclaration = new VertexDeclaration(
              device, VertexPositionColor.VertexElements);
        }


        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            device = graphics.GraphicsDevice;

            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            effect = Content.Load<Effect>("effects");

            init_triangle();
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
```

```
            // TODO: Add your drawing code here
            effect.CurrentTechnique = effect.Techniques["Pretransformed"];
            effect.Begin();
            foreach (EffectPass pass in effect.CurrentTechnique.Passes) {
                pass.Begin();

                device.VertexDeclaration = myVertexDeclaration;
                device.DrawUserPrimitives(PrimitiveType.TriangleList, vertices, 0, 1);

                pass.End();
            }
            effect.End();


            base.Draw(gameTime);
        }
    }
}
```

5 Shaders

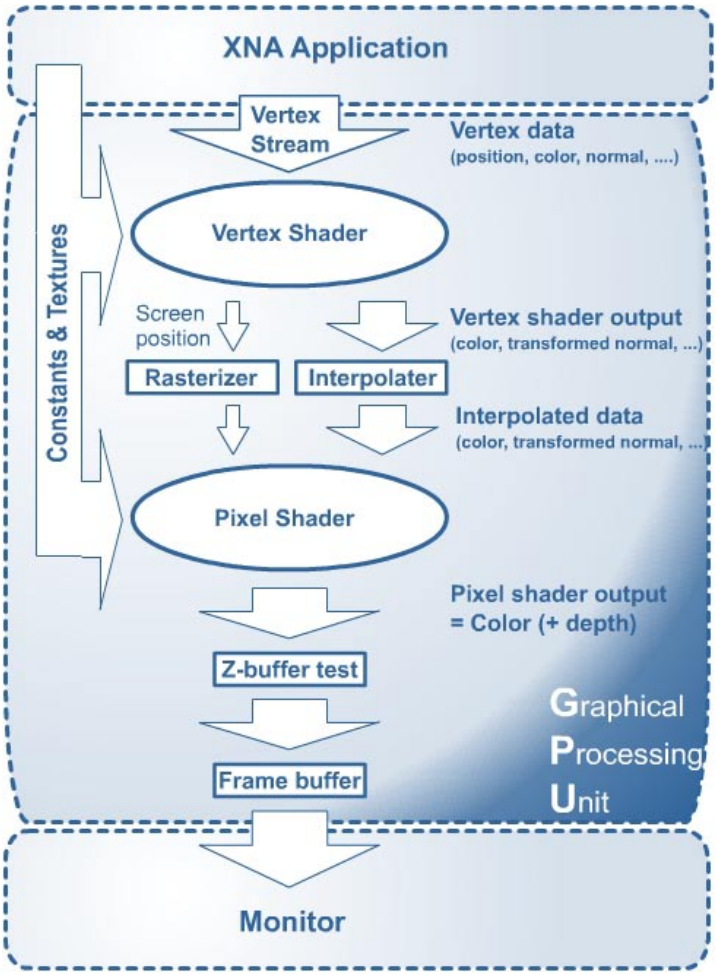
Shaders are responsible for all the colors we see and have full control on how the data passes through the Graphics Pipeline and what will be rendered as a result of each pass. There are 2 types of shaders: Vertex Shaders and Pixel Shaders. Pixel shaders are computationally more expensive than vertex shaders.

Vertex shader does two things:

- transforms 3D position of each vertex into 2D coordinates on screen
- calculates the amount of light the vertex receives

In other words shaders allow you to create any graphical effect you could think of and give you direct access to the Graphical Processing Unit. Shader programming is done with HLSL - the High Level Shader Language. If you use vertex shader every vertex on screen will first pass through your shader before being drawn. Hence pixel shaders should be avoided if it is possible to do the same with vertex shaders from performance reasons. However some effects can only be done with pixel shaders.

Figure 5.1 Graphics Pipeline



Source: [Riemers]

The flowchart at Figure 5.1 above represents the flow of vertex data from our XNA application to the vertex shader on the graphical card. This chain of events happens each time we need to draw something.

We are now ready to discuss the content of effects.fx file shown in Figure 5.2 below.

Shader has to pass to the graphical card the following: a structure that holds vertex data and a data definition. To satisfy those requirements we add VertexToPixel structure that holds 3D position and Color data. We then define the Pretransformed technique which has 1 pass via the Graphics Pipeline and uses PretransformedVS() and PretransformedPS() functions to output position and color data to the graphical card.

Figure 5.2 effects.fx

```
struct VertexToPixel
{
    float4 Position        : POSITION;
    float4 Color           : COLOR0;
    float LightingFactor: TEXCOORD0;
    float2 TextureCoords: TEXCOORD1;
};

struct PixelToFrame
{
    float4 Color : COLOR0;
};



//------- Technique: Pretransformed --------

VertexToPixel PretransformedVS( float4 inPos : POSITION, float4 inColor: COLOR)
{
       VertexToPixel Output = (VertexToPixel)0;

       Output.Position = inPos;
       Output.Color = inColor;

       return Output;
}

PixelToFrame PretransformedPS(VertexToPixel PSIn)
{
       PixelToFrame Output = (PixelToFrame)0;

       Output.Color = PSIn.Color;

       return Output;
}

technique Pretransformed
{
       pass Pass0
       {
              VertexShader = compile vs_1_1 PretransformedVS();
              PixelShader  = compile ps_1_1 PretransformedPS();
       }
}
```

6 Creating virtual world

Populating XNA universe with believable 3D models usually requires using one of the 3D modeling software packages that has the ability to export models into the FBX format supported by the XNA. For our project we did choose Autodesk Maya 2008. Instead of creating models ourselves we have decided to spare some time by importing freely available models into Maya and then exporting them into FBX as described in the following sections.

6.1 Importing 3D Models into Maya in 3DS Format

In our world we did use various 3D models available online for free in 3ds format. In order to import 3ds models into Maya 2008 we have downloaded the corresponding version of Bonus Tools for Maya 2008 available as a free download from Autodesk website [Bonus Tools]. Then using the Plug-in Manager in Maya we did load 3dsImport.mll that comes with Bonus Tools to enable 3ds import. For free 3D models please refer to [Klicker] and [Turbosquid].

Figure 6.1 represents the initial world setup in Maya. Important at this point to make sure that all the imported models first brought to their intended size (while staying at the origin) and that those changes are made final by selecting "Freeze Transformations" for each model in Maya. So for instance if a tree was originally 10 times bigger than the intended size, we would want Maya to remember the size of the tree after it was resized and before we start to move and rotate it. This is because we need to remember the translation from the origin and local rotation of each model in order to correctly position them in the XNA environment. We can still resize trees in XNA to allow for some variation in size of the duplicated trees, so some of them will be higher than the others.

Figure 6.1 World setup in Maya

6.2 Exporting 3D Models from Maya to XNA in FBX Format

XNA platform supports loading 3D models made with 3D applications in .x or .fbx formats. It is theoretically possible to load other formats as well but it requires writing a custom model loader. Before exporting the models to fbx some preparations have to be made. First of all we did position all the

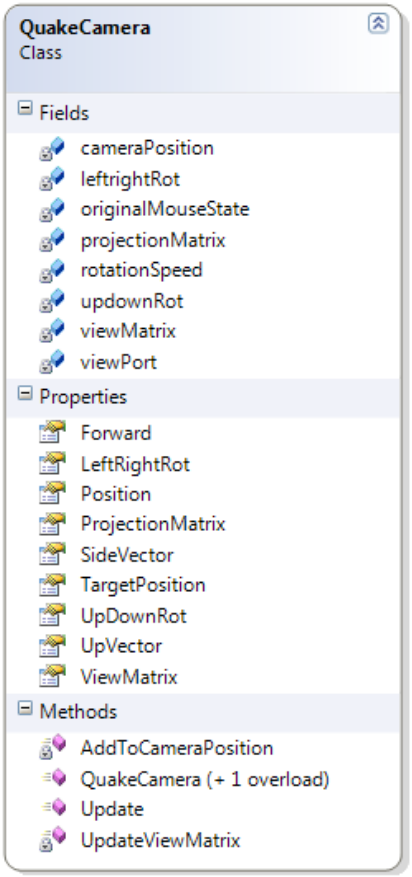
models on separate layers in Maya. This makes our job easier because we will need to bring each model to be exported back to the origin (insure that its pivot point is exactly at the origin) but before we do that we will write down its intended size, location and rotation attributes. Bringing model to the origin insures that we will have all the control we need when it comes to positioning of models inside the XNA environment without unwanted transform anomalies that will be introduced if we leave models (and their pivot points) at their desired locations as shown in Figure 6.1 above. The advantage of this approach is that in our XNA universe each model will have exactly the same coordinates as in our Maya universe. See also "Appendix A: Maya to FBX Export" for technical details.

6.3 Camera

Camera defines what part of the virtual world we see on a screen. That part is called a View. Camera view of the world defines what will be actually rendered. Before rendering takes place we have to specify the position and the view of our camera. In other worlds in order to render we need the View and Projection matrices of the camera. View matrix is defined by Position, Target and Up vectors of the camera, while Projection matrix holds only the part of the world that is actually seen by the camera.

Lab QuakeCamera presents a Quake-Style camera as seen in First-Person Shooter games. The code for the QuakeCamera class is too large to be included in this text, that's why we will only provide the Class Diagram shown at Figure 6.2 below and briefly discuss this class's functionality.

Figure 6.2 QuakeCamera Class Diagram



Private properties and methods are indicated with a lock, while everything unlocked is public.

As expected from a first-person shooter camera, this QuakeCamera has the ability to respond to user input (from both kyaboard and mouse). Pressing the Up and Down arrows will move camera forward and back, pressing the Left and Right arrows will move camera to the left and to the right. Moving the mouse will rotate the camera. Notice that mouse pointer is hidden at all times, so in chance you will wander how to close the screen if you have no mouse pointer – it will react on ESC button as well as the traditional ALT + F4 or similar facilities to close any window without a mouse.

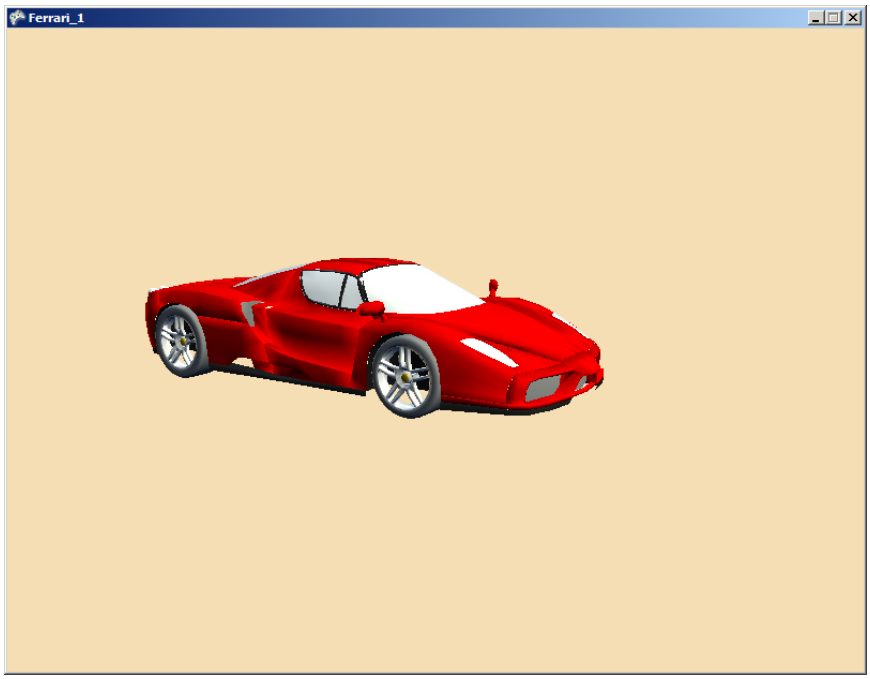
6.4 Loading 3D Model and Simple Animation

XNA framework has a built-in Content Pipeline which makes it relatively simple to load a 3D model into the scene. The Content Pipeline is described in detail on Microsoft website. For more information please refer to the [Content Pipeline].

The source code for this lab can be found at Labs/Ferrari_1 folder. For the purpose of this lab we will need a model that we can load. We will use the Ferrari Enzo model from [Turbosquid].

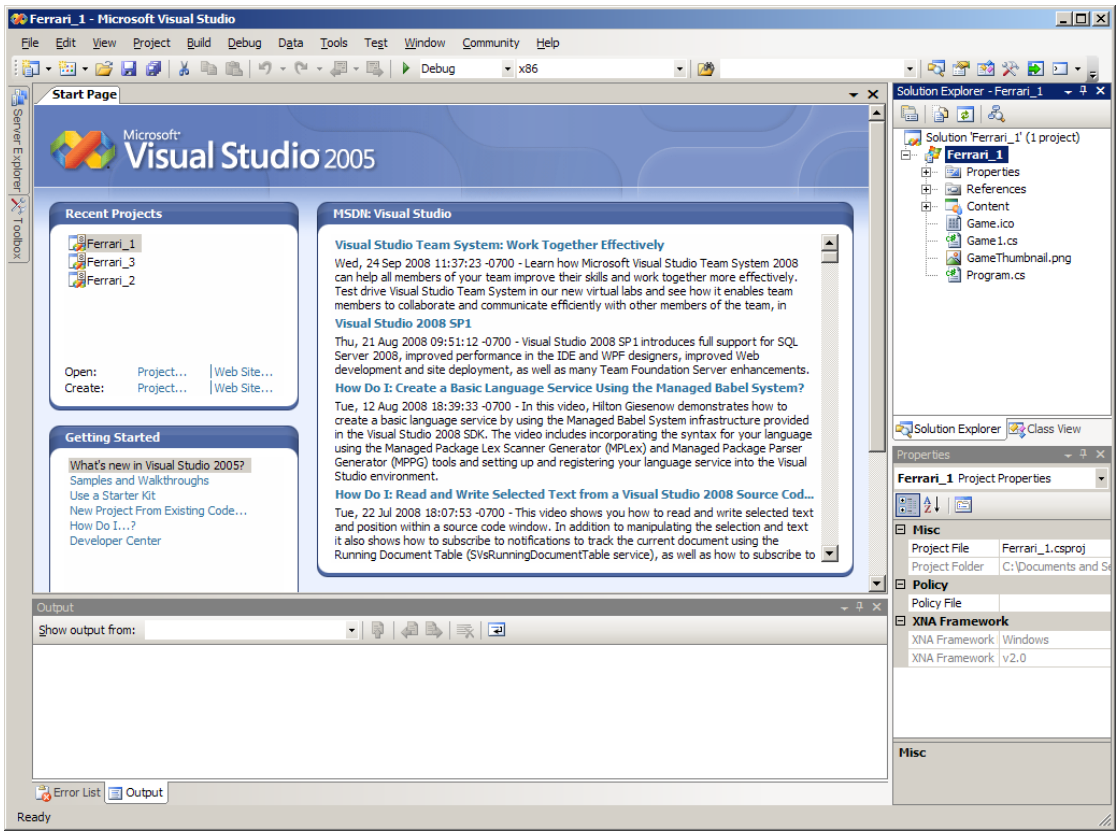
The final result of this Lab is a FerrariEnzo car rotating around the Y-axis on the screen as shown at Figure 6.3 below.

Figure 6.3 FerrariEnzo car rotating around the Y-axis



After creating a new project called Ferrari_1 in Visual Studio, the directory structure inside Solution Explorer will look as presented at Figure 6.4 below.

Figure 6.4 Solution Explorer



One of the folders that has been generated by default is the Content folder. Inside the Content folder we will add a new folder Models to store our Ferrari model.

Since we are going to use an existing model, all we have to do to make it part of our project, is selecting Add > Existing Item while right-mouse clicking the Models folder. The Content Pipeline will then import the file and convert it into a generic format which can be loaded dynamically. The advantage of this is that all the imported files (e.g. images, 3D models, 2D models) can be loaded in the same way. This gives us the ability to disregard file extension when we load a model into the game resulting in source code that needs less maintenance when modifications take place.

If we than click on the Ferrari_Enzo.fbx file we see the properties of this file in the Properties panel. The only thing left to do is to change the Asset Name to the intended name which in our case is FerrariEnzo. This way we can create a reference to the model in our source code.

The source code of the Game1 class in which we load FerrariEnzo model is shown at Figure 6.5 below.

Figure 6.5 Game1.cs loading the FerrariEnzo model

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
```

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace Ferrari_1
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        // We added a reference to the Enzo Model
        // Also the position of the model and camera should be defined.

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Model Enzo;
        float aspectRatio;
        float modelRotation = 0.0f;
        Vector3 modelPosition = Vector3.Zero;
        Vector3 cameraPosition = new Vector3(0.0f, 50.0f, 250.0f);

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting
to run.
        /// This is where it can query for any required services and load any non-
graphic
        /// related content.  Calling base.Initialize will enumerate through any
components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here

           // Here we load the real model with the reference name we defined.
            Enzo = Content.Load<Model>("Models\\FerrariEnzo");
            aspectRatio = (float)graphics.GraphicsDevice.Viewport.Width /
(float)graphics.GraphicsDevice.Viewport.Height;
        }

        /// <summary>
```

```
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
            // TODO: Unload any ResourceManagementMode.Automatic content
            Content.Unload();
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here
            // To do more than only loading the model we want to do something with it.
            // We added the modelRotation variable with which we will rotate the model
            modelRotation += (float)gameTime.ElapsedGameTime.TotalMilliseconds *
MathHelper.ToRadians(0.1f);

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.Wheat);
            Matrix[] transforms = new Matrix[Enzo.Bones.Count];
            Enzo.CopyAbsoluteBoneTransformsTo(transforms);

            foreach (ModelMesh mesh in Enzo.Meshes)
            {
  // This is where the mesh orientation is set, as well as our camera and projection.
                foreach (BasicEffect effect in mesh.Effects)
                {
                    effect.EnableDefaultLighting();
                    effect.World = transforms[mesh.ParentBone.Index]
                        * Matrix.CreateRotationY(modelRotation)
                        * Matrix.CreateTranslation(modelPosition);
                    effect.View = Matrix.CreateLookAt(cameraPosition,
                        Vector3.Zero, Vector3.Up);
                    effect.Projection = Matrix.CreatePerspectiveFieldOfView(
                        MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 10000.0f);
                }
                // Draw the mesh, using the effects set above.
                mesh.Draw();
            }
            base.Draw(gameTime);
        }
    }
}
```

Inside LoadContent() function we load the model using its reference name defined earlier with the following code:

```
Enzo = Content.Load<Model>("Models\\FerrariEnzo");
```

Then we define the animation inside Update() function which is the place to modify parameters that are updated on each game clock tick before the drawing of a game window takes place. In this case we increase the rotation angle stored in global modelRotation variable each time as shown below:

```
modelRotation += (float)gameTime.ElapsedGameTime.TotalMilliseconds *
MathHelper.ToRadians(0.1f);
```

Last thing left to do is to bind this rotation variable to the world rotation matrix inside the Draw() method's ModelMesh loop as shown below:

```
effect.World = transforms[mesh.ParentBone.Index]
                        * Matrix.CreateRotationY(modelRotation)
                        * Matrix.CreateTranslation(modelPosition);
```

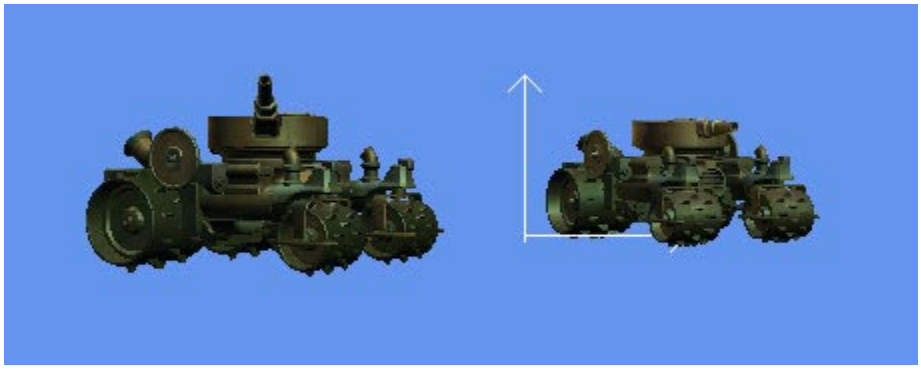
By observing all the operations that we need to do to animate one model we can soon come to the conclusion that this will not be the way to program a game. The more models we use the more complicated the code becomes. This is why a more object oriented approach is needed which we will discuss in the next section where we introduce advanced animation techniques.

7 Advanced Animation Techniques

In this chapter we are going to investigate advanced animation techniques that build up on the simple animation method introduced in section 6.4 (which results in extensive amount of source code per model scattered all over the game class). To improve the readability of code we will use object oriented techniques. We will then discuss how to animate parts of a model separately from each other and how to animate the entire model that already has partial animations.

The source code for this lab can be found at Labs/Tanks folder.
The final result of this lab is presented in the figure below.

Figure 7.1 Tanks Lab



For the purpose of this lab we have selected a very complex freely available tank model that consists of multiple parts which will be all animated separately and also as a group. Tank model can be found in Content/tank.fbx file. In order to animate parts of the tank we first need to examine the model source file (which can be opened with any text editor). Sins we did not build this tank, we are also not aware of what names the artist has given to all the parts. In XNA terms those parts are ModelBone objects and can be referenced in a program as long as their names are known to the programmer.

Tank model file is very long, it contains 37762 lines of code. We are particularly interested in "Object relations" section located very close to the end of file, to be precise between lines 37616-37672. The source code is shown in Figure 7.2 below. Luckily enough the artist has given very comprehensive names to all parts of the tank (in the context of the model source file called meshes). For instance we can clearly identify 3rd mesh as the right back wheel of the tank given name "r_back_wheel_geo" and similarly identify all other parts of the tank. In the absence of comprehensive names our task could become extremely difficult when dealing with models that are built by somebody else.

Figure 7.2 Content/tank.fbx

```
; Object relations
;------------------------------------------------------------------

Relations:  {
    Model: "Model::tank_geo", "Mesh" {
    }
    Model: "Model::r_engine_geo", "Mesh" {
```

```
    }
    Model: "Model::r_back_wheel_geo", "Mesh" {
    }
    Model: "Model::r_steer_geo", "Mesh" {
    }
    Model: "Model::r_front_wheel_geo", "Mesh" {
    }
    Model: "Model::l_engine_geo", "Mesh" {
    }
    Model: "Model::l_back_wheel_geo", "Mesh" {
    }
    Model: "Model::l_steer_geo", "Mesh" {
    }
    Model: "Model::l_front_wheel_geo", "Mesh" {
    }
    Model: "Model::turret_geo", "Mesh" {
    }
    Model: "Model::canon_geo", "Mesh" {
    }
    Model: "Model::hatch_geo", "Mesh" {
    }
    Model: "Model::Producer Perspective", "Camera" {
    }
    Model: "Model::Producer Top", "Camera" {
    }
    Model: "Model::Producer Bottom", "Camera" {
    }
    Model: "Model::Producer Front", "Camera" {
    }
    Model: "Model::Producer Back", "Camera" {
    }
    Model: "Model::Producer Right", "Camera" {
    }
    Model: "Model::Producer Left", "Camera" {
    }
    Model: "Model::Camera_Switcher", "CameraSwitcher" {
    }
    Material: "Material::turret_phong", "" {
    }
    Material: "Material::engine_phong", "" {
    }
    Texture: "Texture::file1", "TextureVideoClip" {
    }
    Texture: "Texture::steamroller_tank61_file3", "TextureVideoClip" {
    }
    Video: "Video::file1", "Clip" {
    }
    Video: "Video::steamroller_tank61_file3", "Clip" {
    }
}
```

Instead of loading the model in its entirety as we did earlier with FerrariEnzo in section 6.4 we will have to load all meshes from which the model is composed. We can then reference each mesh (each ModelBone) individually in order to animate it. This suggests that we need to create a generic class that will be responsible for loading all models this way, in other words a class that will treat a model as a set of bones rather than a single peace. We did call this class WorldModel and its source code is presented in Figure 7.3 below.

Figure 7.3 WorldModel.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace World1
{
    class WorldModel
    {
        public Model model;
        public Matrix[] boneTransforms;

        public void Initialize(Model in_model)
        {
            model = in_model;
            boneTransforms = new Matrix[model.Bones.Count];
            model.CopyAbsoluteBoneTransformsTo(boneTransforms);
            foreach (ModelMesh mesh in model.Meshes) {
                foreach (BasicEffect effect in mesh.Effects) {
                    effect.EnableDefaultLighting();
                }
            }
        }


        public void Update(GameTime gameTime)
        {

        }


        public void Draw(GameTime gameTime, Matrix projectionMatrix,
            Matrix viewMatrix, Matrix worldMatrix)
        {
            foreach (ModelMesh mesh in model.Meshes) {
                foreach (BasicEffect effect in mesh.Effects) {
                    effect.EnableDefaultLighting();
                    effect.World = boneTransforms[mesh.ParentBone.Index]
                                    * worldMatrix;
                    effect.View = viewMatrix;
                    effect.Projection = projectionMatrix;
                }
                mesh.Draw();
            }
        }
    }
}
```

Ech model needs to be initialized, then its parameters can be optionally updated before the model is drawn.

Initialize() - loads all meshes and stores them in bonTransforms Matrix array.

Update() - intentionally left empty, represents optional parameter updates (e.g. animation parameters) that will vary per model.

Draw() - draws all meshes.

For the purpose of this lab we are going to load two tanks, each in a different way. Tank #1 is going to be static and loaded using the WorldModel class while tank #2 will be animated and loaded with a special class Tank that derives from the WorldModel class. This way we can use the code already available in our base class to initialize and draw the meshes with the addition of class properties unique to our tank model so we could animate each ModelBone of tank #2 separately inside the overriden Update() method as shown in Figure 7.4 below.

Figure 7.4 Tank.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System.Diagnostics;


namespace World1
{
    class Tank : WorldModel
    {
        // constants
        public const float ELLIPSE_W = 14.0f;   // elliptical animation path width
        public const float ELLIPSE_H = 10.0f;   // elliptical animation path height
        public const float TANK_TIME = 1000.0f; // time it takes to complete the
animation path

        // Animation parameters
        float time;
        float tank_time;
        public float rotation_y; // entire model
        public float position_x; // entire model
        public float position_z; // entire model
        public float wheelRotation;
        //public float steerRotation;
        public float turretRotation;
        public float cannonRotation;
        public float hatchRotation;

        // Bones
        ModelBone leftBackWheelBone;
        ModelBone rightBackWheelBone;
        ModelBone leftFrontWheelBone;
        ModelBone rightFrontWheelBone;
        ModelBone leftSteerBone;
        ModelBone rightSteerBone;
        ModelBone turretBone;
        ModelBone cannonBone;
        ModelBone hatchBone;

        // Bone transforms
```

```
        Matrix leftBackWheelTransform;
        Matrix rightBackWheelTransform;
        Matrix leftFrontWheelTransform;
        Matrix rightFrontWheelTransform;
        Matrix leftSteerTransform;
        Matrix rightSteerTransform;
        Matrix turretTransform;
        Matrix cannonTransform;
        Matrix hatchTransform;

        public new void Initialize(Model model)
        {
            rotation_y = 0.0f;
            position_x = 0.0f;
            position_z = 0.0f;
            time = 0.0f;
            tank_time = 0.0f;

            // Set bones (their names are taken from tank.fbx file)
            leftBackWheelBone = model.Bones["l_back_wheel_geo"];
            rightBackWheelBone = model.Bones["r_back_wheel_geo"];
            leftFrontWheelBone = model.Bones["l_front_wheel_geo"];
            rightFrontWheelBone = model.Bones["r_front_wheel_geo"];
            leftSteerBone = model.Bones["l_steer_geo"];
            rightSteerBone = model.Bones["r_steer_geo"];
            turretBone = model.Bones["turret_geo"];
            cannonBone = model.Bones["canon_geo"];
            hatchBone = model.Bones["hatch_geo"];

            // Set bone transforms
            leftBackWheelTransform = leftBackWheelBone.Transform;
            rightBackWheelTransform = rightBackWheelBone.Transform;
            leftFrontWheelTransform = leftFrontWheelBone.Transform;
            rightFrontWheelTransform = rightFrontWheelBone.Transform;
            leftSteerTransform = leftSteerBone.Transform;
            rightSteerTransform = rightSteerBone.Transform;
            turretTransform = turretBone.Transform;
            cannonTransform = cannonBone.Transform;
            hatchTransform = hatchBone.Transform;

            base.Initialize(model);
        }


        public new void Update(GameTime gameTime)
        {
            time = (float)gameTime.TotalGameTime.TotalSeconds;

            // Animation parameters
            wheelRotation = time * 5;
            //steerRotation = (float)Math.Sin(time * 0.75f) * 0.5f;
            turretRotation = (float)Math.Sin(time * 0.333f) * 1.25f;
            cannonRotation = (float)Math.Sin(time * 0.25f) * 0.333f - 0.333f;
            hatchRotation = MathHelper.Clamp((float)Math.Sin(time * 2) * 2, -1, 0);

            float temp;
            float t;
            float tank_theta;
            float tank_theta_degrees;
            float tank_theta_radians;

            t = time;
            temp = t - tank_time;
```

```
            if (temp > TANK_TIME) {
                tank_time = t;
                while (temp > TANK_TIME) {
                    temp -= TANK_TIME;
                }
            }

            // elliptical tank motion
            tank_theta = (temp / TANK_TIME) * 360.0f;
            tank_theta_degrees = (tank_theta * 360.0f) / (2.0f * (float)Math.PI);
            tank_theta_degrees = tank_theta_degrees % 360.0f; // float modulus

            tank_theta_degrees += 90.0f; // this is because tank is originally not in
the right angle
            tank_theta_radians = ((float)Math.PI / 180.0f) * tank_theta_degrees;

            rotation_y = tank_theta_radians;
            position_x = ELLIPSE_W * (float)Math.Sin(tank_theta);
            position_z = ELLIPSE_H * (float)Math.Cos(tank_theta);

            /*
            Debug.Print(
                "time=" + time +
                " tank_time=" + tank_time +
                " tank_theta=" + tank_theta +
                " tank_theta_degrees=" + tank_theta_degrees +
                " tank_theta_radians=" + tank_theta_radians +
                " x=" + position_x +
                " z=" + position_z
                );
            */
        }


        public new void Draw(GameTime gameTime, Matrix projectionMatrix, Matrix
viewMatrix, Matrix worldMatrix)
        {
            // Modify bones matrices
            leftBackWheelBone.Transform = Matrix.CreateRotationX(wheelRotation) *
leftBackWheelTransform;
            rightBackWheelBone.Transform = Matrix.CreateRotationX(wheelRotation) *
rightBackWheelTransform;
            leftFrontWheelBone.Transform = Matrix.CreateRotationX(wheelRotation) *
leftFrontWheelTransform;
            rightFrontWheelBone.Transform = Matrix.CreateRotationX(wheelRotation) *
rightFrontWheelTransform;
            //leftSteerBone.Transform = Matrix.CreateRotationY(steerRotation) *
leftSteerTransform;
            //rightSteerBone.Transform = Matrix.CreateRotationY(steerRotation) *
rightSteerTransform;
            turretBone.Transform = Matrix.CreateRotationY(turretRotation) *
turretTransform;
            cannonBone.Transform = Matrix.CreateRotationX(cannonRotation) *
cannonTransform;
            hatchBone.Transform = Matrix.CreateRotationX(hatchRotation) *
hatchTransform;

            // Look up combined bone matrices for the entire model.
            base.model.CopyAbsoluteBoneTransformsTo(base.boneTransforms);
            // Draw the model
            base.Draw(gameTime, projectionMatrix, viewMatrix, worldMatrix);
        }
```

```
    }
}
```

Tank #2 is going to follow elliptical path (the size of wich is defined by ELLIPSE_W and ELLIPSE_H). It is a time based animation to make sure that on different machines it will take approximately the same time (TANK_TIME) for the tank to complete 1 cycle of following its elliptical path. While tank is orbiting according to its unique elliptical path all of its meshes (ModelBones) will be animated: the left, right, back and front wheels, the turret, the cannon and the hatch.

Notice that inside the Initialize() method we perform all the necessary initialization of tank animation parameters (such as tank position, rotation and time). We then initialize each bone reference to its corresponding mesh using the names defined in tank.fbx file, store the original bone transforms so we could modify them later in a Draw() method and call the base.Initialize() method to load the meshes.

Update() method contains the animation of all tank parts as well as the calculation of elliptical tank motion in terms of rotation_y, position_x and position_z animation parameters.

Finally inside the Draw() method the bone transforms are modified to their updated values and base.Draw() method is invoked.

We are now ready to see what effect this object oriented approach has on our code in Game1 class.

All the initialization, loading, animation and drawing code was encapsulated into WorldModel and Tank classes resulting in a very clean Game1 class that hides all the implementation details. In fact the only visible difference between two tanks in this class is the following two lines of code:

```
WorldModel tank1 = new WorldModel();
Tank tank2 = new Tank();
```

This is because both tank1 and tank2 have their Initialize(), Update() and Draw() methods. Even the fact that tank2 is animated and tank1 is static, is completely hidden from the Game1 class and can be only observed by checking the instance of the class.

Figure 7.5 Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
```

```
using Microsoft.Xna.Framework.Storage;

namespace World1
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;
        BasicEffect basicEffect;
        QuakeCamera fpsCam;
        CoordCross cCross;

        WorldModel tank1 = new WorldModel();
        Tank tank2 = new Tank();


        public Game1()
        {

            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            fpsCam = new QuakeCamera(GraphicsDevice.Viewport, new Vector3(0,
3, 45), 0, 0);
            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = graphics.GraphicsDevice;
            basicEffect = new BasicEffect(device, null);
            cCross = new CoordCross(device);

            Model model;

            model = Content.Load<Model>("tank");

            tank1.Initialize(model);
            tank2.Initialize(model);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            GamePadState gamePadState = GamePad.GetState(PlayerIndex.One);
            MouseState mouseState = Mouse.GetState();
            KeyboardState keyState = Keyboard.GetState();

            if (gamePadState.Buttons.Back == ButtonState.Pressed
                || keyState.IsKeyDown(Keys.Escape))
                this.Exit();
```

```
            fpsCam.Update(mouseState, keyState, gamePadState);

            tank1.Update(gameTime);
            tank2.Update(gameTime);
            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.CornflowerBlue, 1, 0);

            Matrix worldMatrix;

            // Coordinate system
            cCross.Draw(fpsCam.ViewMatrix, fpsCam.ProjectionMatrix);

            // tank1 :
            worldMatrix = Matrix.CreateScale(0.01f, 0.01f, 0.01f)
                * Matrix.CreateRotationY(0.5f)
                * Matrix.CreateTranslation(5, 0, 0);
            tank1.Draw(gameTime, fpsCam.ProjectionMatrix, fpsCam.ViewMatrix,
worldMatrix);


            // tank2 :
            worldMatrix = Matrix.CreateScale(0.01f, 0.01f, 0.01f)
                * Matrix.CreateRotationY(tank2.rotation_y)
                * Matrix.CreateTranslation(tank2.position_x, 0,
tank2.position_z);
            tank2.Draw(gameTime, fpsCam.ProjectionMatrix, fpsCam.ViewMatrix,
worldMatrix);


            base.Draw(gameTime);
        }
    }
}
```

8 Lighting and Special Effects

In this chapter we will add lighting and special effect - fog to the scene.

The source code for this lab can be found at Labs/World3 folder.

Since all objects in our scene, except the floor, are meshes and based on the WorldModel class or inherit from it like the Tank class, we can use the WorldModel class to add lighting and special effect processing to affect all those models when they are rendered inside the Draw() method.

The new code of WorldModel class is presented in Figure 8.1 below.

The important new methods in this class are lights() and fog(). Both are using the BasicEffect class to render the desirable effect.

lights() - disables default lighting and adds directional light to the scene;

fog() - adds fog effect to the scene, with fog color slightly darker than the background "sky" color to achieve more realistic foggy look and feel;

draw() - instead of effect.EnableDefaultLighting() we now have calls to lights() and fog() methods.

Notice that floor is not affected by fog. This is because floor is a vector based structure which was manually built in source code. In other words it was not built with Maya and loaded into XNA later. It is therefore not based on the WorldModel class like all the other models in the scene and passes its own

unique rendering routine. If we would like to create a floor which is also affected by fog and custom lighting we have defined earlier, we will have to load it in the same way we did load all other models.

Figure 8.1 WorldModel.cs with lights and fog

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace World1
{
    class WorldModel
    {
        public Model model;
        public Matrix[] boneTransforms;

        public void Initialize(Model in_model)
        {
            model = in_model;
            boneTransforms = new Matrix[model.Bones.Count];
            model.CopyAbsoluteBoneTransformsTo(boneTransforms);
            foreach (ModelMesh mesh in model.Meshes) {
                foreach (BasicEffect effect in mesh.Effects) {
                    effect.EnableDefaultLighting();
                }
            }
        }


        public void Update(GameTime gameTime)
        {

        }


        protected void lights(BasicEffect effect)
        {
            effect.LightingEnabled = true;

            effect.AmbientLightColor = new Vector3(0.5f, 0.5f, 0.5f);

            effect.DirectionalLight0.Enabled = true;
            effect.DirectionalLight0.DiffuseColor = new Vector3(1, 1, 1);
            effect.DirectionalLight0.SpecularColor = new Vector3(1, 1, 1);
            effect.DirectionalLight0.Direction = new Vector3(1, 1, 1);
        }


        protected void fog(BasicEffect effect)
        {
            Color fog_color = new Color(67, 102, 164);
            effect.FogEnabled = true;
            effect.FogColor = fog_color.ToVector3();
            effect.FogStart = 120;
            effect.FogEnd = 180;
        }
```

```
        public void Draw(GameTime gameTime, Matrix projectionMatrix, Matrix
viewMatrix, Matrix worldMatrix)
        {
            foreach (ModelMesh mesh in model.Meshes) {
                foreach (BasicEffect effect in mesh.Effects) {
                    //effect.EnableDefaultLighting();
                  lights(effect);
                  fog(effect);
                    effect.World = boneTransforms[mesh.ParentBone.Index] *
worldMatrix;
                    effect.View = viewMatrix;
                    effect.Projection = projectionMatrix;
                }
                mesh.Draw();
            }
        }
    }
}
```

9 Appendix A: Maya to FBX Export

This appendix describes the procedure of exporting a Maya model into the FBX file ready to be loaded into XNA environment.

It is recommended to **keep each model on a separate layer** in Maya, so it will be possible to easily isolate it (by hiding all the other layers) when given model needs to be exported.

All textures used in a model must have **size (width and height) which is power of 2**.
Example A: 128 x 512 – this is a good example because everything is power of 2.
Example B: 200 x 512 – this is a wrong example, this texture will not load into the XNA environment because 200 is not a power of 2.

If you use an existing model that comes with original textures that do not comply with "power of 2" requirements, use Photoshop to resize each texture until it does comply. You will usually enlarge width and height to the closest number which is power of 2. In many cases it does not create visible problems and does not require texture remapping. To be sure you can reload textures in Maya after you modify their sizes to see the results.

Before exporting always optimize your model by **removing invisible faces**. If you leave them they will only take additional rendering time consuming valuable resources.

Make sure your **model is at the origin** (X, Y, Z = 0) and its **scaling ratio is equal to 1**.

Finally group all the parts of your model into a single group. Select the group and then do File > Export Selection. Give your model a name and select the file type FBX. When you hit Export button the window (presented in Figure 9.1 below) will open where you can specify additional parameters.

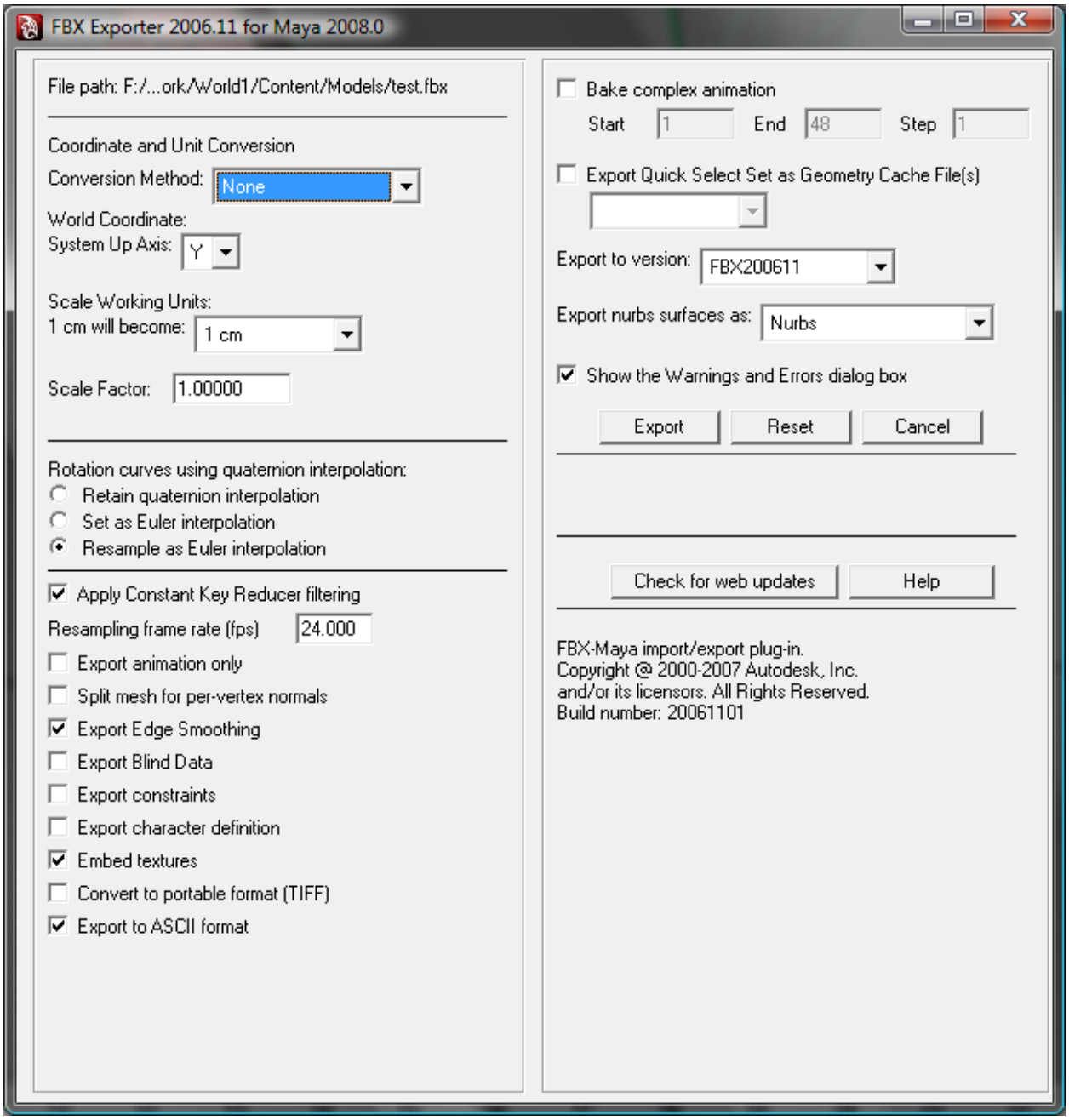
Make sure that "**Embed textures**" and "**Export to ASCII format**" are both checked. Also select the appropriate Conversion Method (if you don't have any animation defined for your model, select here the None option).

Press the Export button to start exporting.

After FBX file has been created there are still a few issues to keep in mind. Since we exported to the ASCII format we can open FBX file as a regular text file inside Visual Studio. The "Embed textures" option we have checked earlier only means that inside your ASCII FBX file the paths (both absolute and relative) to your textures will be included. You will have to modify the **RelativeFilename** path of each texture inside the ASCII FBX file to point to the Textures folder inside the Content folder of your Visual Studio Game Project.

If your FBX file is located at **Content\Models** and your textures are located at **Content\Textures**, then you would like to make sure that you have: **RelativeFilename: "..\textures\mytexturename.bmp"**.

Figure 9.1 Export Options



10 Bibliography and References

Software

[Bonus Tools] Bonus Tools for Maya is a free download from:
http://area.autodesk.com

[C# IDE] Free Microsoft Visual C# 2005 Express Edition
http://www.microsoft.com/express/2005/

[XNA] Microsoft XNA Game Studio 2.0:
http://www.microsoft.com/downloads/details.aspx?FamilyId=DF80D533-BA87-40B4-ABE2-1EF12EA506B7&displaylang=en

[XNA 2.0 Project Upgrade Wizard] This utility converts and upgrades an existing XNA Game Studio 1.0 game project and saves it as a new XNA Game Studio 2.0 game project.
http://creators.xna.com/en-us/utilities/project_upgrade_wizard

[XNA 2.0 Changes] XNA Framework Changes in XNA Game Studio 2.0
http://msdn.microsoft.com/en-us/library/bb975648.aspx

3D Models

[Klicker] Free 3D Plants in 3DS format
http://www.klicker.de/plants.html

[Turbosquid] Free 3D Models
http://www.turbosquid.com/

XNA Books

[XNA Guide] Microsoft® XNA™ Game Studio Creator's Guide - An Introduction to XNA Game Programming, by Stephen Cawood, Pat McGee, 2007

[XNA Recipes] XNA 2.0 Game Programming Recipes A Problem-Solution Approach, by Riemer Grootjans, 2008

XNA Tutorials

[Riemers] HLSL Introduction Tutorial
http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/HLSL_introduction.php

[Ziggyware] Terrain generation
http://www.ziggyware.com/readarticle.php?article_id=132&rowstart=0