TUPACHI-3D ENGINE

1st Person Multi Level Action –Adventure RPG (Role Playing Game) with RTS (Real-time Strategy) elements

By

Mohsin Ali Afzal

Najam Ul Hassan

Fawad Asghar

Haseeb Shakoor Paracha

SUBMITTED TO THE FACULTY OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY, RAWALPINDI IN PARTIAL FULFILLMENT FOR THE

REQUIREMENTS OF A B.E. DEGREE IN COMPUTER SOFTWARE ENGINEERING

APRIL 2004

ABSTRACT

TUPACHI-3D ENGINE

By

MOHSIN
NAJAM
FAWAD
HASEEB

This report focuses on development of a complete 3D engine “TUPACHI” using the Microsoft DirectX API. It details the various phases of development of the engine. The initial parts of the report focus on the DirectX API. DirectX, along with OpenGL, is the standard API for 3D development. The engine has been developed keeping in mind the latest trends of the gaming industry and is completely extensible and supports many features such as character loading and animation, map loading, an AI module, a physics module and a complete sound module. The engine also supports peer-to-peer multiplayer gaming using DirectPlay. The engine design is fully modular and can be easily extended to support more features. Currently three maps have been provided with the engine but it can load any number of maps provided to it in the .X format. The engine uses the .wav format to load sounds. The content for the demo presentation has been created using MAYA 3D though it can be used with any popular 3D program like 3D Studio Max, Milkshape 3d, etc. This document will be useful for anyone wanting to develop a fully functional 3d game engine or to people who want to use TUPACHI for game development.

DECLARATION

No portion of the work presented in this dissertation has been submitted in support of another award or qualification either at this institute or elsewhere.

DEDICATION

This document is dedicated to our beloved parents who have been a source of constant encouragement for us.

ACKNOWLEDGEMENTS

All acclamation to Almighty Allah Who has empowered and enabled us to accomplish this task successfully

We are extremely thankful to our project supervisor Dr. Saeed Murtaza, for his guidance and instructive supervision even with his highly busy schedule.

Acknowledgements are due to the HOD of CS Deptt Col. Raja Iqbal, Lt. Col Nadeem, MCS faculty members and administration who in spite of their busy schedule provided guidance and support not only during this work but also throughout the course of the degree.

Love and gratitude to our families for providing the help and moral support that we required throughout.

PERSPECTIVE

Game development is currently enjoying the largest share out of computer industry (more than 60%). It is the biggest industry in Silicon Valley but largely ignored in Pakistan. So our main objective will be to learn advance game programming techniques and to familiarize ourselves with animation, character building, graphic designing, music composition and coding using DirectX, OpenGL. We hope to enhance our knowledge of AI (Artificial Intelligence) and Computer Graphics while learning about the industry standards and research ways to improve the various AI and rendering algorithms used in computer games. Game development is an ever changing and evolving field as the need to capture the biggest market share drives companies to develop better games with better graphics, artificial intelligence and real world kinematics. During the project we will try to develop a game that would be competitive in the international market.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

2D – Two dimensional.

3D – Three dimensional

4D – four dimensional. Usually refers to the three spatial dimensions plus time. Also infers animated or moving 3D images.

AGP – Accelerated Graphics Port. A bus used to connect image generators in personal computers.

BSP – Binary Space Partitioning.

CG/CGI – Computer Graphics/Computer Graphics Image.

COM - Computer Operation Manual.

CPM - Computer Programming Manual.

CPU – Central Processing Unit.

DIS - Distributed Interactive Simulation. Communication protocol for military simulations.

DSP - Digital Signal Processor. Used to process sound and other signal data.

FFB - Force Feedback.

FLOPS - Floating Point Operations Per Second. A measure of computing power.

FOV – Field of View. A characteristic of display systems.

FPS – First Person Shooter.

FPS - Frames Per Second. A measure of computing and display performance.

GL – Graphics Language. A standard for specifying 3D objects for computer display.

GUI – Graphical User Interface.

MIPS - Millions of Instructions Per Second. A measure of computing power.

MPEG – Motion Pictures Expert Group, A standards setting organization. Deals with the coding of multimedia information.

MTP - Master Test Plan.

NPC- Non Player Characters.

Pixel - Picture Element. The basic building block of a graphic display, and the unit in which display resolution is usually expressed.

PVS – Potentially Visible Sets.

R&D – Research and development.

SPE - Software Product Evaluation.

SPM - Software Project Manager.

SRS -Software Requirements Specification.

STD -Software Test Description.

T&L – Transform and Lightning.

TFB – Tactile Feedback.

USB – Universal Serial Bus.

VGA – Video Graphics Array.

WBS -Work Breakdown Structure

CHAPTER 1

<u>INTRODUCTION</u>

Tupachi Engine is a complete game development framework for next-generation consoles and DirectX9-equipped PC's, providing the vast array of core technologies, content creation tools, and support infrastructure required by top game developers. Every aspect of the Tupachi Engine has been designed with ease of content creation and programming in mind, with the goal of putting as much power as possible in the hands of artists to develop assets in a visual environment with minimal programmer assistance; and to give programmers a highly modular and extensible framework for building, testing, and shipping games in a wide range of genres.

1.1 AIM

The aim of this project is to develop a scalable and fully functional 3d game engine that can be used to develop games, simulators or other 3d applications using the Microsoft DirectX 9.0 API.

1.2 LANGUAGE AND PLATFORM

The development of this 3d engine has been done using Microsoft Visual C++ 6.0 and the Microsoft DirectX API ver 9.0. The engine has been developed for Microsoft Windows 9.x, Windows 2000, Windows NT and Windows XP. The reason for using DirectX and Windows was the popularity of the platform as most gamers use the Windows operating system.

1.3 SOFTWARE MODEL

The complexity of the project and the unfamiliarity with the technology required that we adopt the incremental model so as to iteratively develop on the functioning of the engine.

1.4 REAL-TIME-3D GAME-ENGINE TAXONOMY [1]

Many computer games create virtual worlds, and the more recent ones can create and display remarkably detailed ones. Most interestingly, many of them can display virtual worlds in real time, that is, that display these worlds with essentially instant response to inputs and with automatic updating. This distinguishes them from typical 3D-modeling software, which generally only does preview modes in real time.

In particular, we will be focusing on game engines that do real-time 3D rendering; this is rendering that uses three-dimensional geometrical information.

1.4.1 Overall Environment

The highest-level classification is overall environment; this is the overall type of geometry, and it determines much of the rest of the world geometry.

- Indoor Engines
- Outdoor Engines
- Outer-Space Engines

These names describe what sort of scenes the engines are best adapted to. Indoor engines have floors and walls and ceilings, outdoor engines have essentially one big floor, and outer-space engines have no boundary surfaces. However, these are not absolutely fixed distinctions, since one type of engine can have features of another.

In particular, indoor and outdoor engines can have entities that fly or swim (or both!), making their physics much like that of entities in outer-space games. Also, indoor engines can do outdoor scenes by making some of their surfaces look like distant landscapes, while outdoor engines can do indoor scenes with appropriately-placed cliffs. And some recent games appear to have hybrid indoor/outdoor engines with indoor-engine segments added to outdoor-engine ones.

We will be discussing little about outer-space engines here, since they have very little by way of world geometry, and because the rendering of their inhabitants parallels that of other kinds of game engines.

Outdoor engines, with the exception of flight simulators and the like, generally feature a top-down or a slanted view; these engines are essentially 2D, with only recent ones having some 3D features. Bungie's Myth series was the first to have such real 3D features such as terrain elevations and 3D projectile physics.

Indoor engines are the ones with the most advanced 3D rendering, and it is these that I will discuss in the most detail. Since this is a fairly big field, I will subdivide it into generations, each with characteristic rendering features.

1.4.1.1 Generation Zero

We are including this generation for completeness; it is the side scroller, a kind of 2D engine where the view direction is horizontal. All the inhabitants are sprites (2D pictures), and landscapes may be scrolled at some fractional speed to create the appearance of perspective. Their outdoor-engine counterparts are 2D engines with top-down or slanted views.

1.4.1.2 First Generation

These were the first of the indoor-engine real-time-3D games to appear; they include such notable examples as id's Wolfenstein 3D and Bungie's Pathways into Darkness in the early 1990's. They share the features like single-floor map, orthogonal walls, constant-height floors and ceilings, textures being aligned horizontally or vertically, all inhabitants being sprites, 2D game physics and view the direction always being horizontal.

1.4.1.3 Second Generation

This generation started to appear in the mid-1990's; it includes id's Doom series, Bungie's Marathon series, LucasArts's Dark Forces, 3D Realms's Duke Nukem, etc. They generally share the features like stacked floors either being impossible or possible only by creating portals, walls having arbitrary horizontal orientation, variable-height floors and ceilings, textures being aligned horizontally or vertically, though shift able, all inhabitants being sprites, game physics with varying amounts of 3D and view direction having a limited vertical range.

The first one happens because several of the examples (Doom, Dark Forces, Duke Nukem) have their horizontal geometry specified by Binary Space Partitions (BSP's), which do not allow stacked floors. However, stacked floors can be produced by dividing the level into sub maps, some of whose surfaces act as portals to other sub maps. Marathon is an exceptional case, because it does not use BSP's, thus allowing stacked floors to be created in a very natural manner. However, each Marathon map sector/polygon could be interpreted as a sub map with portals comparable to those in the other engines mentioned.

1.4.1.4 Third Generation

This generation started to appear in 1996, with the release of id's Quake. It has since been followed by Tomb Raider, Quake 2, Unreal, and several others; it is now the dominant sort of indoor game engine. They generally share the features such as stacked floors being easy, arbitrary orientation of surfaces, no engine distinction between floors, walls, and ceilings, textures being arbitrarily aligned, most inhabitants being 3D models instead of sprites, fully 3D game physics and view direction capable of being nearly vertical

As noted, sprites are generally not used for very much; mostly explosion effects, flames, and lens flare. This is partly because 3D models automatically have the correct appearance in all directions; with sprites, one has to create sets of them representing some entity viewed from different directions, and not surprisingly, some of the more recent sprite makers have been known to use 3D-modeling software for that task.

3D-model character animation is more complicated than doing animation of sprites, which is to make a simple series of them. The most common way of doing that is the Quake approach, which features using a sequence of vertex set in a single continuous model. An alternative is to use the Tomb Raider approach, which is to break the models up into segments, and animate by moving those segments relative to each other; this is a form of skeletal animation.

CHAPTER 2

THE COMPONENTS OF A GAME

A video game is a new way of programming that’s more conducive to real-time applications and simulation, rather than the single-line, event-driven, or sequential logic programs that you may be used to. A video game is basically a continuous loop that performs logic and draws an image on the screen, usually at a rate of 30 frames per second (fps) or more. This is similar to how a movie is displayed, except that they are creating the movie as they go. [2]

2.1 GAME LOOP ARCHITECTURE

Figure 2-1 shows the general game loop architecture. The basic architecture is divided into various sections the detail of which is explained below.

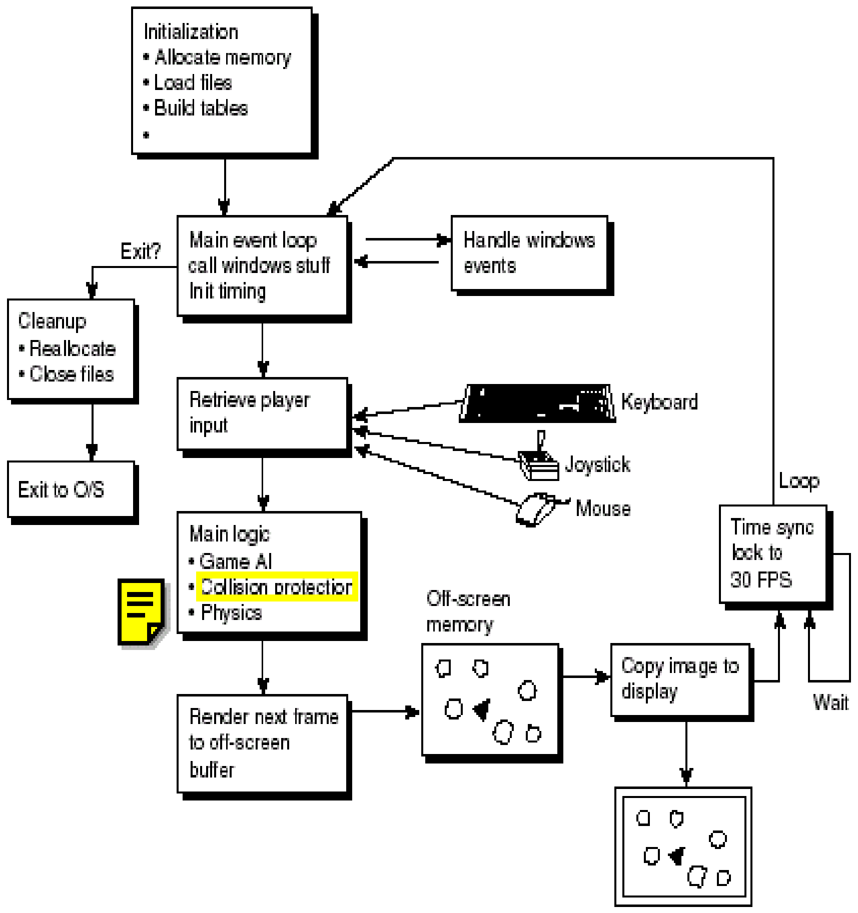

FIGURE 2-1 *General Game Loop Architecture*

The description of each section: is given below:

Section 1: Initialization

In this section, you perform the standard operations you would for any program, such as memory allocation, resource acquisition, loading data from disk, and so forth.

Section 2: Enter Game Loop

In this section, the code execution enters into the main game loop. This is where the action begins and continues until the user exits out of the main loop.

Section 3: Retrieve Player Input

In this section, the player's input is processed and/or buffered for later use in the AI and logic section.

Section 4: Perform AI and Game Logic

This section contains the majority of the game code. The artificial intelligence, physics systems, and general game logic are executed, and the results are used to draw the next frame on the screen.

Section 5: Render Next Frame

In this section, the results of the player's input and the execution of game AI and logic are used to generate the next frame of animation for the game. This image is usually drawn on an off-screen buffer area, so you can't see it being rendered. Then it is copied very quickly to the visible display.

Section 6: Synchronize Display

Many computers will speed up or slow down due to the game's level of complexity. For example, if there are 1,000 objects running on the screen, the CPU is going to have a higher load than if there were only 10 objects. The frame rate of the game will vary, which isn't acceptable. Hence, you must synchronize the game to some maximum frame rate and try to hold it there using timing and/or wait functions. Usually, 30fps is considered to be optimal.

Section 7: Loop

This section is fairly simple—just go back to the beginning of the game loop and do it all again.

Section 8: Shutdown

This is the end of the game, meaning that the user has exited the main body or game loop and wants to return to the operating system. However, before the user does this, you must release all resources and clean up the system, just as you would for any other piece of software.

2.2 STATE TRANSITION DIAGRAM

Figure 2-2 is the state transition diagram of the game loop. First the game is initialized n then the lop goes into the game menu. From the game menu one can either start the game or exit the game. From the starting loop one goes to the running loop of the game where the game is being played. One remains in the same loop unless and until one opts for the restart of the game. From the restart state one goes to the main menu of the game.[3]

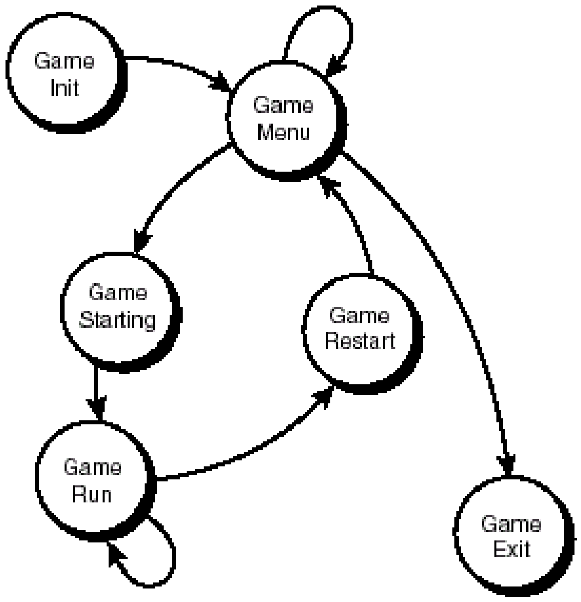

FIGURE 2-2: *State Transition Diagram of the Game Loop*

CHAPTER 3

RENDERING 3D PRIMITIVES

3.1 SETTING RENDER STATES

A Direct3D device has dozens of settings that you can change to affect how primitives are rendered. These settings are called *render states*. [3]

The section below describes the render states that you'll most likely want to use.

3.1.1 Alpha Blending States

Alpha blending (which allows the rendering of semitransparent objects) can be activated through the setting of a few render states.

3.1.2 Alpha Testing State

Alpha testing controls whether pixels are written to the render-target surface that is, it verifies whether the pixels are accepted or rejected.

3.1.3 Ambient Lighting State

Ambient light is the light that surrounds the object and emanates from all directions. This lighting is used as the background lighting for the RoadRage application.

3.1.4 Antialiasing State

Antialiasing makes lines and edges look as smooth as possible on the screen.

3.1.5 Clipping State

Primitives being rendered partially outside the viewport can be clipped.

3.1.6 Color Keying State

You can set a color key to treat the key color as transparent. Once set, whenever a texture is applied to one of the primitives, all the texels that match the key color won't be rendered on the primitive

3.1.7 Culling State

When you set up a triangle that you want to see both sides of, you usually should make sure that you create *two* triangles instead of one—one that represents the front of the triangle and one that represents the back. You need to do this because Direct3D culls any primitives that are facing away from the camera during rendering. By rendering both the front and back triangles, the triangle is visible from both sides because you've rendered two triangles instead of just one.

3.1.8 Depth-Buffering State

Depth buffering removes hidden lines and surfaces. By default, Direct3D doesn't perform depth buffering.

3.1.9 Fill State

By default, Direct3D fills in the contents of the triangles that you specify. But it can also be configured to draw just the "wireframe" outline of the triangle or render just a single pixel at each vertex of the triangle.

3.1.10 Fog State

You can use fog effects to simulate fog or to decrease the clarity of a scene with distance. The latter technique causes objects to become hazy as they become more distant from the viewer, as happens in real life.

3.1.11 Lighting State

You can enable or disable lighting calculations. (They are enabled by default.) Vertices containing a vertex normal are the only ones that will be properly lit. Any others will use a dot product of 0 in all lighting computations, so they will end up receiving no lighting.

3.1.12 Outline State

Direct3D devices default to using a solid outline for primitives. You can easily change the outline pattern by using the D3DLINEPATTERN structure.

3.1.13 Per-Vertex Color States

The flexible vertex format allows vertices to contain both vertex color and vertex normal information (though the D3DVERTEX, D3DLVERTEX, and

D3DTLVERTEX vertex types can't contain both color and normal information). The color and normal are used for lighting computation.

3.1.14 Shading State

Although Direct3D defaults to Gouraud shading, you can use flat shading.

3.1.15 Stencil Buffering States

You can use the stencil buffer to decide whether a pixel is written to the rendering target surface.

3.1.16 Texture Perspective State

You can apply perspective correction to textures to make them fit properly onto primitives that diminish in size as they get farther away from the viewer. You must enable perspective correction to use w-based fog and w-buffers.

3.1.17 Texture Wrapping State

The D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 render states are used to enable and disable *u*-wrapping and *v*-wrapping for various textures in the device's multitexture cascade.

3.1.18 Vertex Color Lighting State

The flexible vertex format allows vertices to contain both vertex color and vertex normal information. Direct3D defaults to using this information when it calculates lighting.

CHAPTER 4

ADDITIONAL DRAWINGS & CAMERA

4.1 ADDITIONAL DRAWINGS

4.1.1 Volumetric Fog Generator

Fog generator is specifically designed to attain the additional environmental effects of fog in the custom made environment. Another purpose of the fog generator is to introduce the noise in environment so that the tracker's efficiency can be tested in noisy environments as well.

Fog generator employees a different approach than the conventional fog calculations in DirectX. This deviation is the per vertex calculation of fog component for each vertex. The volumetric fog is generated according to the height of each vertex (Y component of each vertex) using the intensity specified.

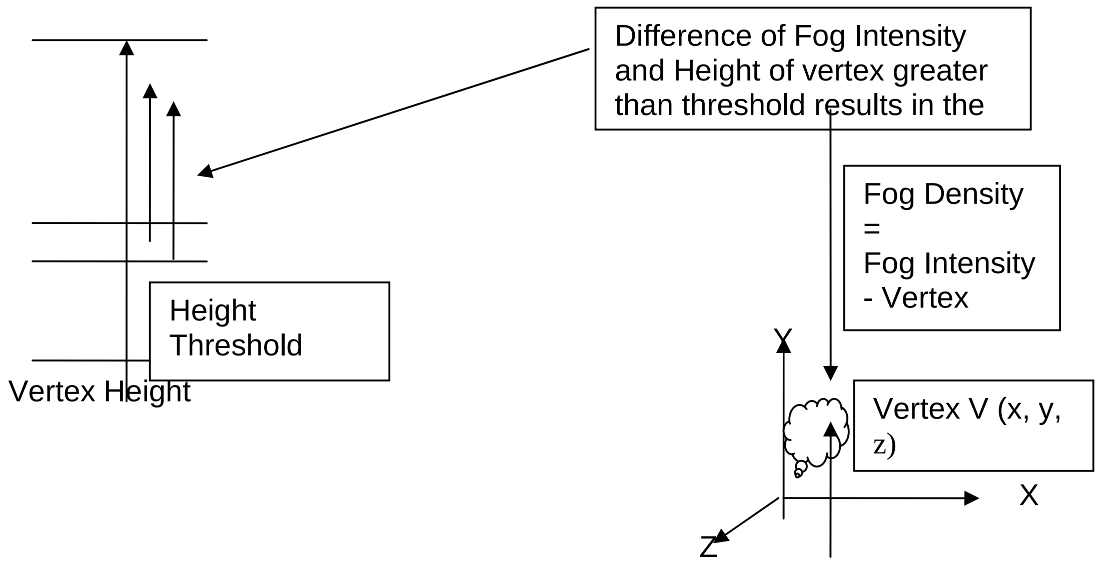
Difference of Fog Intensity and Height of vertex greater than threshold results in the
Fog Density = Fog Intensity - Vertex
Height Threshold
Vertex Height
Y
Vertex V (x, y, z)
X
Z

FIGURE 4-1 *Generation of Volumetric Fog*

{For additional study on fog generation please refer Appendix B}

4.1.2 Fog Intensity Specifier

The user specified needs are considered in each and every part of this application. So the fog intensity specifier is also designed to assign the fog intensity as per user requirements.

4.1.3 Dust Generator

Dust generator follows the same phenomenon as the Volumetric Fog Generator and its purpose is the same: to introduce *noise* in the environment generated.

Dust is generated according to the height of each vertex using the intensity specified.

4.1.4 Sunlight Generator

The Sunlight Generator gives the additional illuminating effect of sunlight along with the environment lighting. The sunlight effects are important because the additional illumination will affect the Target Object's color and shading models. It will also introduce noise by deviating the clear view of the target object.

4.2 CAMERA

The camera is the foremost important part in any graphics application.

The camera employed in our application allows the free floating view of the environment in all aspects.

The basics of the camera are given as the three vectors associated with it Fig 4-2.

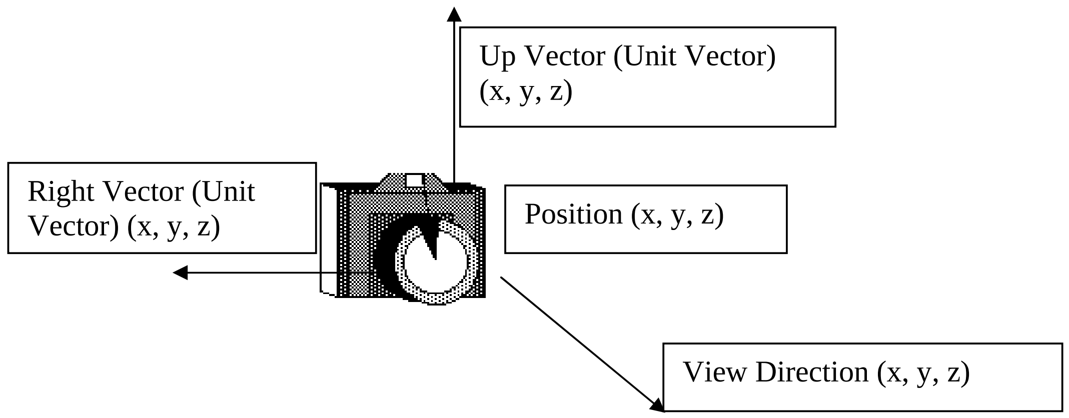
Up Vector (Unit Vector) (x, y, z)
Right Vector (Unit Vector) (x, y, z)
Position (x, y, z)
View Direction (x, y, z)

FIGURE 4-2 *Basics of the Camera*

The necessary movements allowed by the free-floating architecture of the camera are listed below:

1- Forward:

The forward movement adds the forward intercept along View Direction in the camera position.

2- Backward:

The backward movement adds the forward intercept along the negative View Direction in the position.

3- Strafe Right:

The Strafe right movement adds the right intercept along the Right Vector in the position of the camera.

4- Strafe Left:

The Strafe left movement adds the right intercept along the negative Right Vector in the position of the camera.

5- Rotate X:

The rotation in X-axis (Right Vector) follows the set of calculations given below in the figure 4-3.

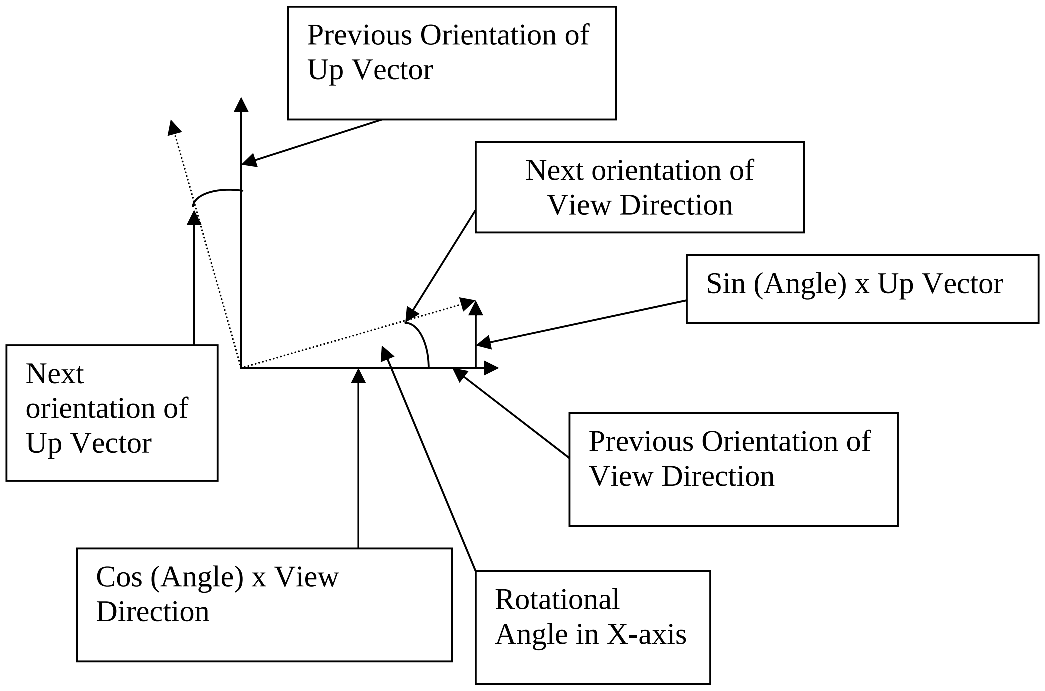
Previous Orientation of Up Vector
Next orientation of View Direction
Sin (Angle) x Up Vector
Next orientation of Up Vector
Previous Orientation of View Direction
Cos (Angle) x View Direction
Rotational Angle in X-axis

FIGURE 4-3 – *Calculations for Rotation in X-axis*

So the next orientation of View Direction:

Equation 1: View Direction (Next) = Sin (Angle) x Up Vector + Cos (Angle) x View Direction.

6- Rotate Y:

The rotation in Y-axis (Up Vector) follows the same set of calculations but the difference is that instead of Up Vector and View Direction, Right Vector and View Direction are used.

7- Rotate Z:

The rotation in Z-axis (View Direction) follows the same set of calculations but the difference is that Up Vector and Right Vector are used.

The Camera module consists of a

4.2.1 <u>Renderer</u>

The Renderer calculates the orientation of the camera making use of the user inputs at runtime. The Up Vector, View Direction and position are the essential primitives in the camera orientation, location and viewing direction.

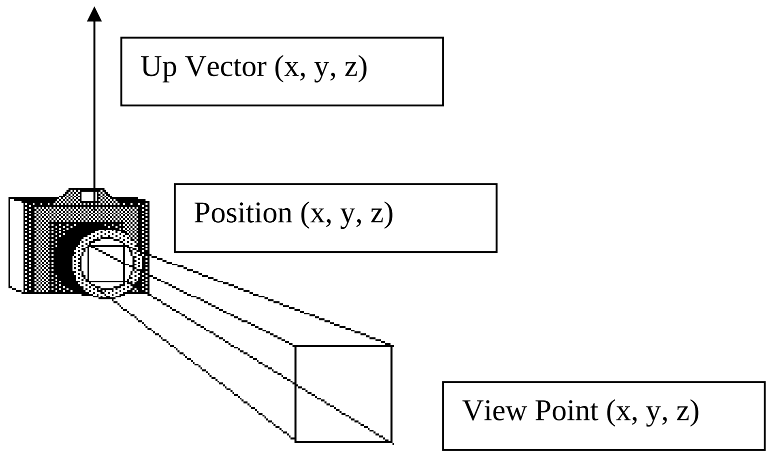

FIGURE 4-4 - *Camera vectors*

The figure clearly specifies that Up Vector orients the camera in X, Y and Z coordinates. Position specifies the location X, Y and Z coordinates, and the View Point is point where the camera should be looking at.

CHAPTER 5

DESIGN OVERVIEW

5.1 OVERVIEW

Figure 5-1 shows the basic overview of our engine. Our engine is basically consists of eight modules as shown in the figure. The engine module basically runs the main exe of the program. The dll libraries of each module are listed in the figure along with its respective module along with the abbreviation of each module.

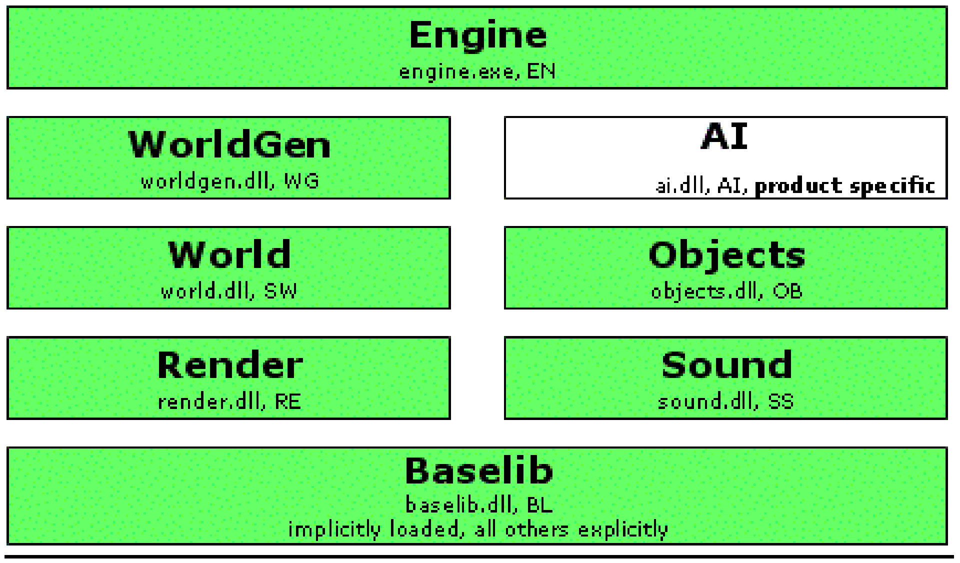

FIGURE 5-1 *Design Overview*

5.2 LEVEL 1 DIAGRAM OF THE ENGINE

Level 1 diagram of the Tupachi engine is shown in the figure 5-2. The figure below clearly shows that each and every module is dependent upon the world module which is the main module of the engine. This is the world module which calls the other modules on its requirements and initializes the other module as soon as it receives the input. This is the world module which is controlling the other module of the engine.

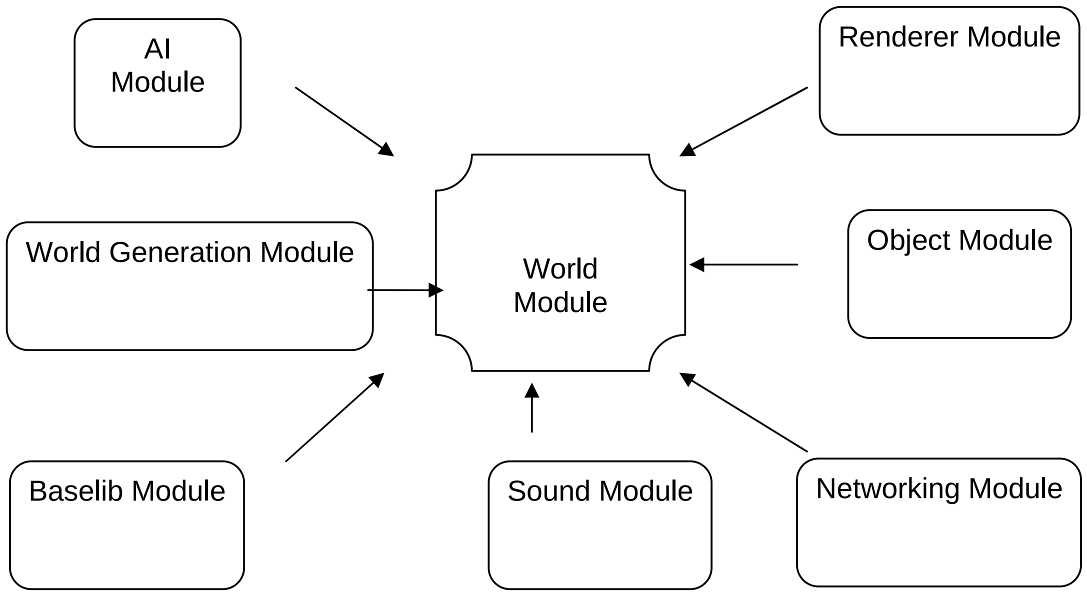

FIGURE 5-2 *Level 1 Diagram*

5.3 ENGINE MODULE

5.3.1 Class Diagram

Class diagram of engine module is shown in the figure 5-3. the main loops calls the class ENWinStart class which in turns calls the ENEngine class which is linked with the engine module the engine module then interacts with the AI module if it is required and at the same time calls the three classes named ENObjectsManager, ENObjectsLoader and ENWorldLoader for objects coordinates, object loading and world loading respectively.

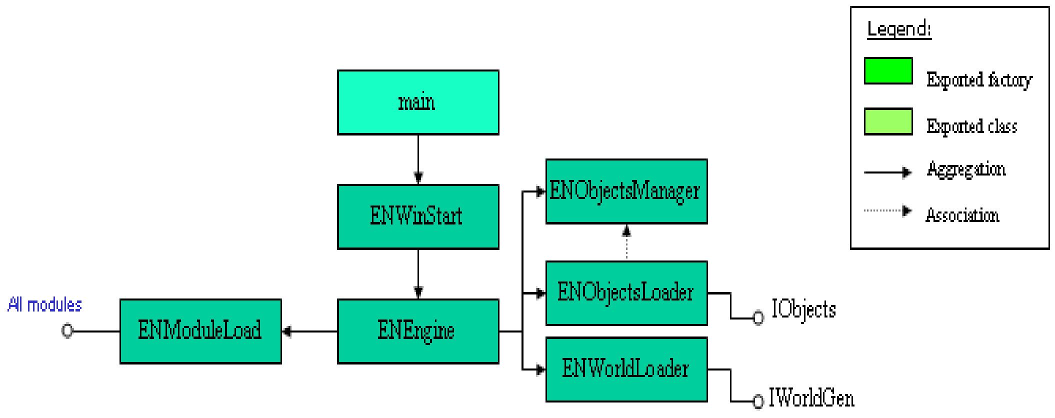

FIGURE 5-3 *Class Diagram of the Engine Module*

5.3.2 Brief Description

As shown in the class diagram, the engine module owns most other modules. It serves three main purposes of providing a windows framework for the application, holding the main loop that instructs other modules to contribute their part on each frame and calling the product specific AI Module on designated occasions Basically, the Engine Module just cares on ownerships and data flow, it is the top level layer and main executable**.**

5.4 ARTIFICIAL INTELLIGENCE MODULE

5.4.1 Introduction

The AI module is different from all other modules in that it is not part of the engine. This module contains product specific implementations.

5.4.2 Interface

The module gets calls from the engine, one at startup to initialize, three during main loop. These callbacks can be used to perform product specific tasks (object controls, HUD, etc). On its initialization call, the AI module receives all other module instances from the engine so that it can use their APIs directly (Sound, Rendering etc.)

5.4.3 Sequence Diagram

Figure 5-4 is the sequence diagram and shows the dll functions called by the engine. IAI class is called by the engine to get he dll instance. IAI class is also called once to allow the initializations of the game engine. At the beginning of the mail loop, the main loop calls the IAI class to begin the main loop. The function CB_PreRender is called after the rendering pipelines have been filled and animation is been done and before the rendering takes place. The function CB_PostRendering is called after the redering is done.

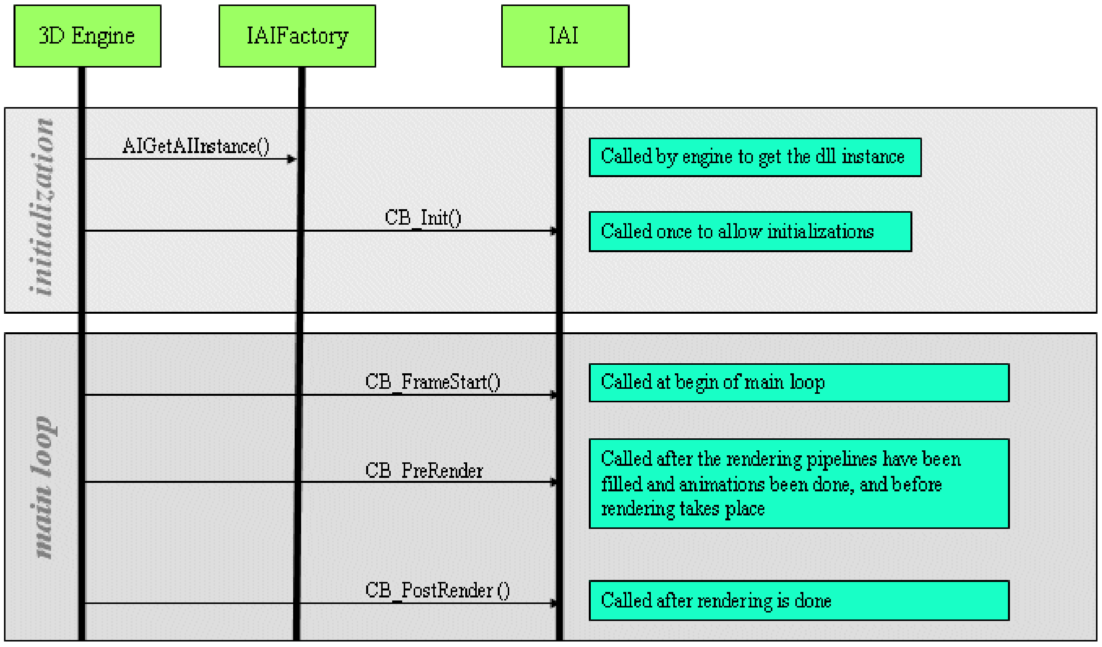
3D Engine
IAIFactory
IAI
initialization
AIGetAIInstance()
CB_Init()
Called by engine to get the dll instance
Called once to allow initializations
main loop
CB_FrameStart()
CB_PreRender
CB_PostRender ()
Called at begin of main loop
Called after the rendering pipelines have been filled and animations been done, and before rendering takes place
Called after rendering is done

FIGURE 5-4 *Sequence Diagram of the AI Module*

5.5 RENDER MODULE

5.5.1 Introduction

Rendering of dynamic shadow (the shadow of moving objects) can be quite time consuming, and it is important to balance the necessary time penalty against the visual benefit.

The kind of shadow matching developers need should have the features like providing clearly visible shadow matching the object shape to help users get a sense for the object height above ground (e.g. for helicopter landing) for directional light, avoiding double shadows and aliasing artifacts and shadow casting object should also cast shadow on other objects and ground

Finally, the shadow improves a scene visibly, but should not take up more than some percent of rendering time.

5.5.2 Class Diagram

Figure 5-5 shows the class diagram of the rendering module. In this module the IRenderFactory calls for the particle and the shadow rendering where it is being done. The 2D and the texture rendering is also done in the same class. Rendering of the 3D objects, rendering of the material, the pixel and the vertex shading is being performed by the RERenderManager where the objects being rendered calls their respective buffers for loading the objects and for their further information about the next frame which is to be rendered.

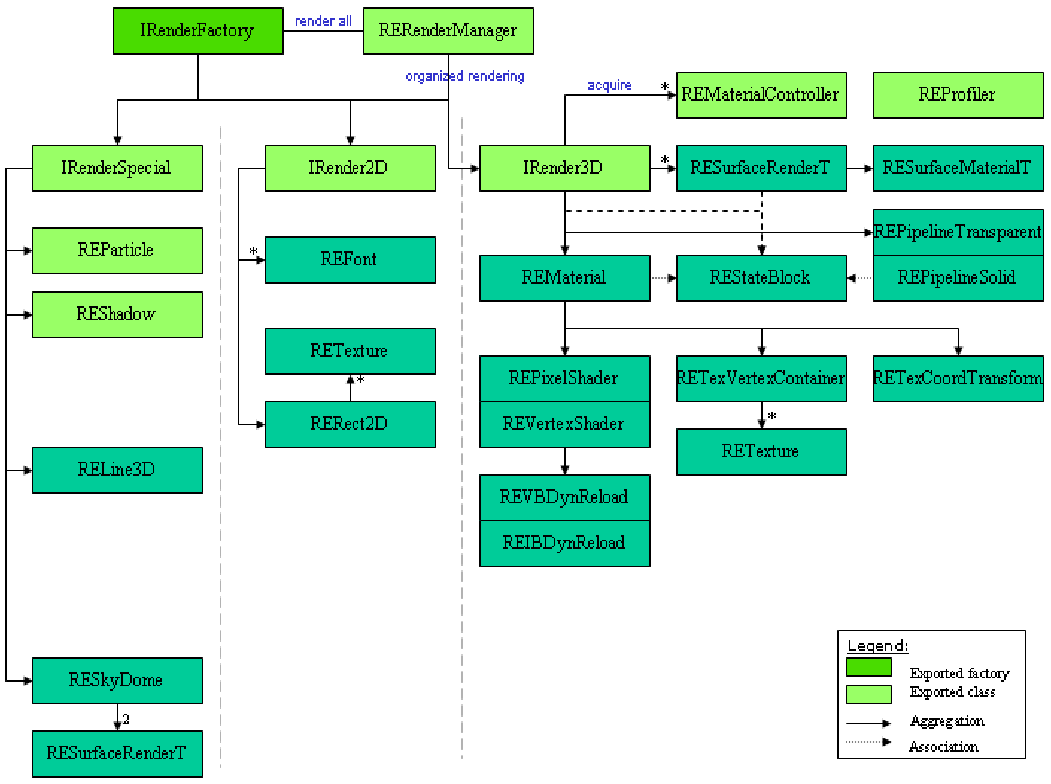

FIGURE 5-5 *Class Diagram of the Render Module*

5.5.3 Sequence Diagram

Figure 5-6 shows the sequence diagram of the render module. The sequence diagram of the render module has five entities including user, IRenderFactory, IRender3D, IRender2D and IRenderSpecial. At the initialization phase class IRenderFactory is called for the user interface. IRender3D is called to check the vertex,texture and controller requirements for a material. In the main loop IRender 3d is called to check which surfaces came insight and went out of sight since the last frame. IRenderSpecial is called by the initialization loop by the function BackBufferToMapTexture to check the back buffer for the coordinates of the next frame. The class IRenderFactory is called from the main loop for rendering all the 2D and 3D surfaces in the correct order.

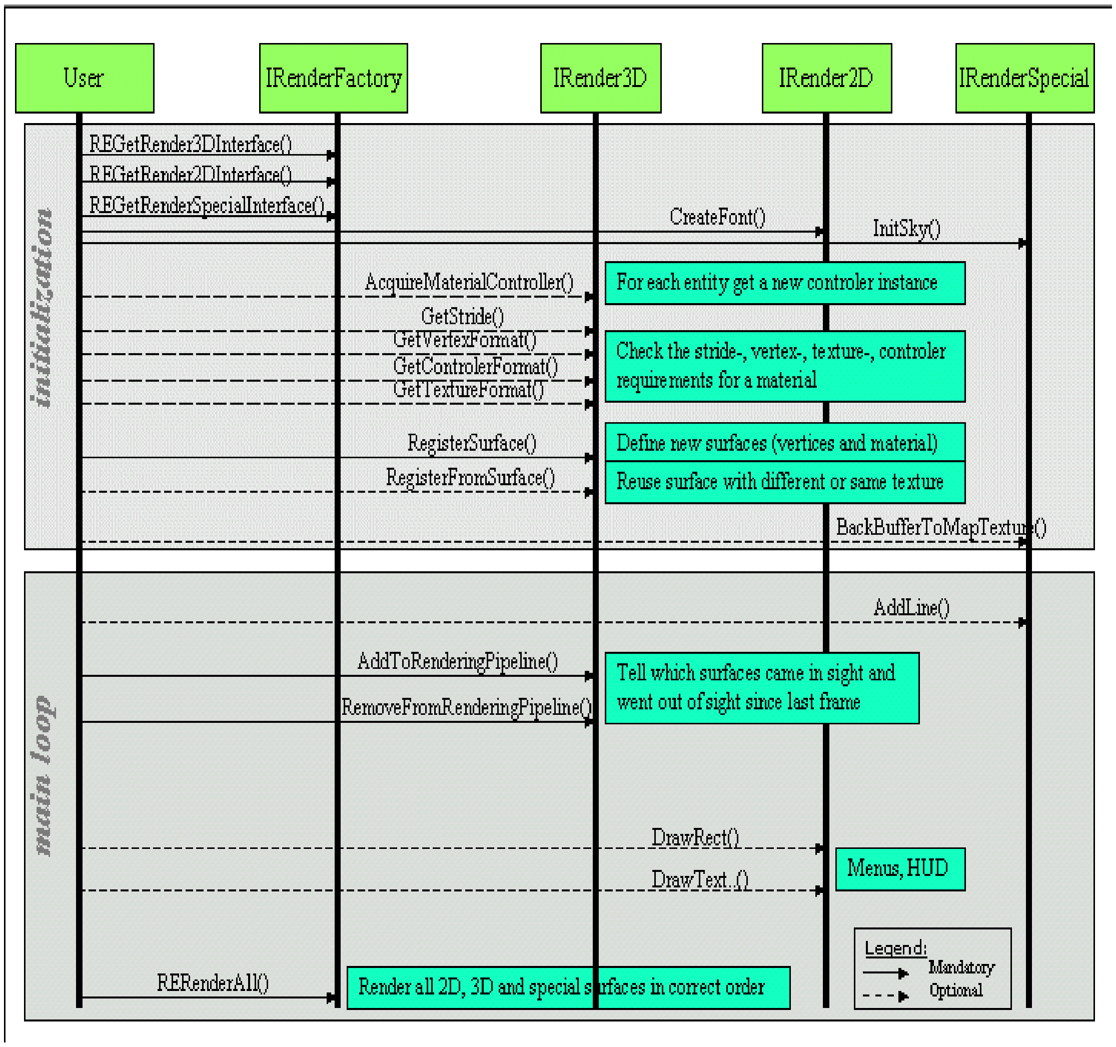
User
IRenderFactory
IRender3D
IRender2D
IRenderSpecial
initialization
REGetRender3DInterface()
REGetRender2DInterface()
REGetRenderSpecialInterface()
CreateFont()
InitSky()
AcquireMaterialController()
For each entity get a new controler instance
GetStride()
GetVertexFormat()
GetControlerFormat()
GetTextureFormat()
Check the stride-, vertex-, texture-, controler requirements for a material
RegisterSurface()
Define new surfaces (vertices and material)
RegisterFromSurface()
Reuse surface with different or same texture
BackBufferToMapTexture()
main loop
AddLine()
AddToRenderingPipeline()
RemoveFromRenderingPipeline()
Tell which surfaces came in sight and went out of sight since last frame
DrawRect()
DrawText ()
Menus, HUD
RERenderAll()
Render all 2D, 3D and special surfaces in correct order
Legend:
Mandatory
Optional

FIGURE 5-6 *Sequence Diagram of the Render Module*

5.6 THE WORLDGEN MODULE

5.6.1 Brief Description

The module takes the **model files** and **the images** as input, together with positional data. It then builds static geometry out of the given input and registers the created surfaces with the world module.

5.6.1.1 Model File Input

The specified model is shown in the materials it comprises at a specified location (scale and translation only). Reuse of models is properly detected and adequately processed (reuse of vertex buffers etc). Shadowing through directional light etc is done as the used materials are properly rendered. Supported model formats include .3ds and .x, further formats are added on demand.

5.6.1.2 Image Input

2D Images are converted into a height map where height relates to pixel brightness. A texture can be applied, together with a blending weight that blends each vertex between texture and the original image color at that vertex position. The height map can then be shaded and low pass filtered (one or many 3x3 passes). To alter heightmap heights, a second texture can optionally be provided where each pixel bright is multiplied with the heightmap to allow for "forced" valleys or equally heightened boundaries.

5.6.2 Sequence Diagram of World Generation Module

Figure 5-7 shows the sequence diagram of the world generation module. The sequence diagram has the three main classes named user class,the IWorldFactory and the IWorld. At the initialization phase user calls the interface function for accessing the IWorldFactory class. The user class calls the IWorld to load the number of world’s available for loading. This is done by taking the coordinates from the files which are saved at a location. UpLoadAllSurfaces function is used to upload all the worlds in the engine. At the main loop IWorld is called for checking the object position for the object module and also checks whether the object remains inside the world.

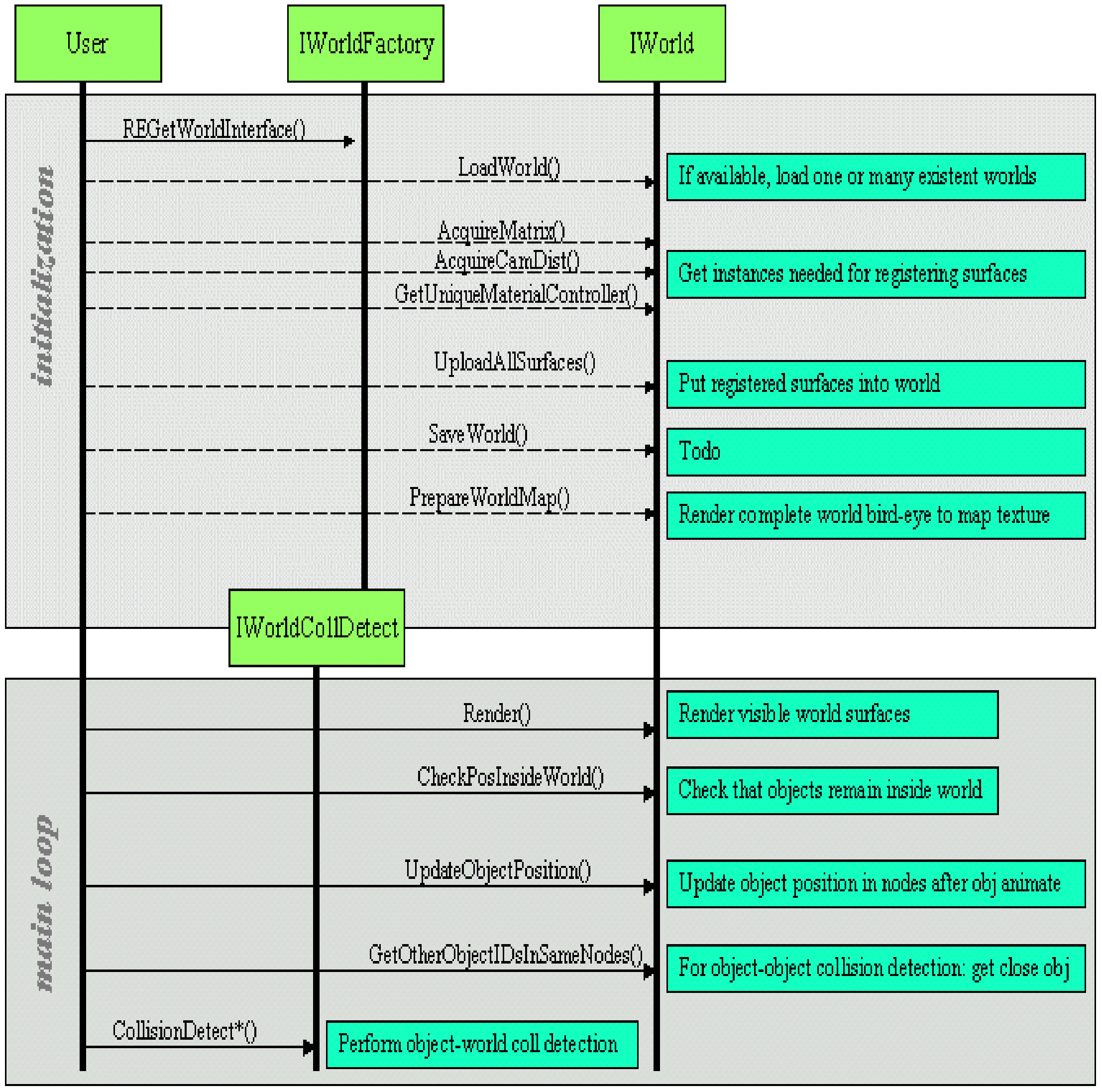
User
IWorldFactory
IWorld
initialization
REGetWorldInterface()
LoadWorld()
If available, load one or many existent worlds
AcquireMatrix()
AcquireCamDist()
GetUniqueMaterialController()
Get instances needed for registering surfaces
UploadAllSurfaces()
Put registered surfaces into world
SaveWorld()
Todo
PrepareWorldMap()
Render complete world bird-eye to map texture
IWorldCollDetect
main loop
Render()
Render visible world surfaces
CheckPosInsideWorld()
Check that objects remain inside world
UpdateObjectPosition()
Update object position in nodes after obj animate
GetOtherObjectIDsInSameNodes()
For object-object collision detection: get close obj
CollisionDetect*()
Perform object-world coll detection

FIGURE 5-7 *Sequence Diagram of the World Generation Module*

5.7 THE SOUND MODULE

5.7.1 Brief Description

The Sound Module provides the interface for playing either stereo or 3D sound. It builds directly on the capabilities of the DirectX Sound interface. Hence, all sound formats (e.g. Dolby Digital) are supported.

5.7.1.1 2D and 3D Sounds

Sounds can be played 2D or 3D. 2D sounds need manual control of volume, frequency, panning. 3D sounds need to specify their position. They are played with correct balancing and physical altering depending on the camera position and speed.

5.8 THE WORLD MODULE

5.8.1 Brief Description

At initialization phase, the world module loads 0..n worlds, and may receive additional surfaces from e.g. a level editor. World surfaces are surfaces whose transformation matrix is NULL, they cannot be transformed during rendering time (except transformation into world space). A house and a street are built out of world surfaces as well as a rigid streetlight, but a street light that bends on a crash does not belong to the world, it is an object.
The world can be saved at any time; visibility information and optimizations have then been applied.

The World Module fulfills two tasks during rendering. The first is given the camera position, find all surfaces in view and pass them to the Rendering Module. The second one is object-world collision detection: answer requests if a bounding object shape collides with world.

5.8.2 Class Diagram of World Module

Figure 5-8 shows the class diagram of the world module. The IWorld class calls the SWRender class for checking he camera position and the orientation of the camera. IWorldCollDetect is called for checking the collision and detection. The SWRender gets the vertex data of the surfaces from the IRender. IWorld passes all the registered surfaces to the SWConstruct and SWMaterialControler where SWConstruct renders all the visible surfaces.

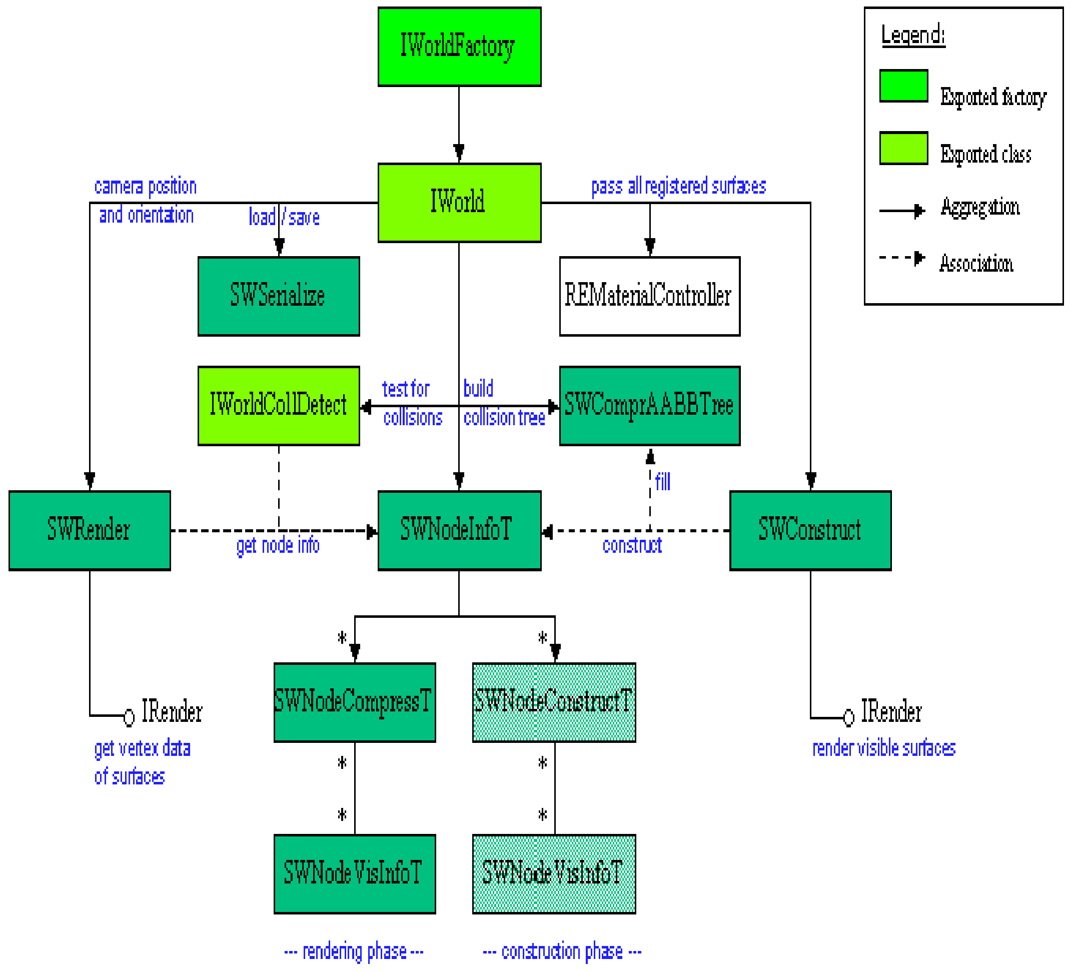

FIGURE 5-8 *Class Diagram of the World Module*

CHAPTER 6

ADDED TECHNICAL DEVELOPMENT

(SYSTEM EFFICIENCY)

6.1 PERFORMANCE AND ITS IMPORTANCE

Performance is an aspect of software design that is often overlooked until it becomes a serious problem. If the developer waits until the end of your development cycle to do performance tuning, it may be too late to achieve any significant improvements. Performance is something to include early in the design phase and continue improving all throughout the development cycle.

Efficiency is the cost to complete an operation, measured by the usage of system resources. It is the measurement aspect of efficiency that makes it useful for software design. Improving the performance of a piece of software requires measuring the efficiency of the software and then improving the software based on those measurements. [4] For example, the developer might measure some of the aspects anyone’s software like how much memory does the software use? How many instructions does it take to perform a given task? How many files does the software open on launch? How much code must be in memory at any given time to perform a task?

These are all aspects of performance that can be measured. Improving on these measurements increases the efficiency of the software. For example, one might use a different algorithm to perform a task and reduce the number of instructions needed to perform that task, thereby improving the efficiency of his code. One might allocate less memory or open fewer files, which improves the efficiency of his software by reducing its consumption of resources. Or one might reorganize his code to minimize his application footprint during specific operations, once again reducing his memory consumption.

Therefore, to improve the efficiency of ones application, one should first understand the basic metrics of system efficiency like memory usage—how much memory does your application consume, cpu time—for a given span of time, how much of that time is spent running your application's code, as opposed to being idle or running other processes and file-system usage—how often does your application access the disk?

6.2 CPU TIME

CPU time is the time a computer spends executing your application's code. An efficient application uses as little CPU time as possible to perform the tasks it needs to perform. When your application is idle, it should consume no CPU time at all. [5]

6.3 MEMORY USAGE AND RUNTIME GARBAGE COLLECTION

Memory on modern computing hardware is typically composed of progressively slower (but larger) types of memory. The fastest memory available to the CPU is the CPU's own registers. The next fastest is the L1 cache, followed by the L2 and L3 caches when they are available. The next fastest memory is the main memory. The slowest memory of all consists of virtual memory pages that reside on disk and must be paged in before they can be used.

Unfortunately, most of an application's code and data resides either in main memory or paged out to disk. Therefore, it is important that the application's code and data is organized in a way that minimizes the time spent in these slower mediums. Reducing the memory footprint of your application can significantly improve its performance.[6] A small memory footprint usually has two advantages. First, the smaller your application, the fewer memory pages it occupies. Fewer memory pages, typically means less paging. Second, code is usually smaller as a result of being more heavily optimized and better organized. Thus, fewer instructions are needed to perform a given task and all of the code for that task is gathered on the same set of memory pages.

For this purpose people implement the guidelines such as all variables are preferred to have a local scope and are destroyed when the function exits. All memory allocations are freed and their handles destroyed so that there are no memory leaks. Global arrays if any are reused so that the same contiguous memory locations can be used to store different data. The global arrays are freed as soon as their utility is over. After writing the environment "binaries" in 3D ENGENT the memory used up by the object models is freed. Linked lists are preferred to arrays wherever there is a large memory requirement as they don't take up contiguous blocks of memory and hence are more efficient to allocate and de-allocate. Function arguments are passed by reference in order to avoid duplication except where necessary.

CHAPTER 7

SCREEN SHOTS

7.1 SCREEN SHOT LEVEL 1

Figure 7-1 is the first person view of the player of level 1. In the top left corner in the white is the frame rate of the game. Player score shows the number of times the player has killed the opponent. The opponent score is the number of times the opponent has killed the player. The number 54 is showing the number of bullets the player is lift with and the 55 on the screen along with the golden bar shows the player health. When the player health becomes zero, the player dies.

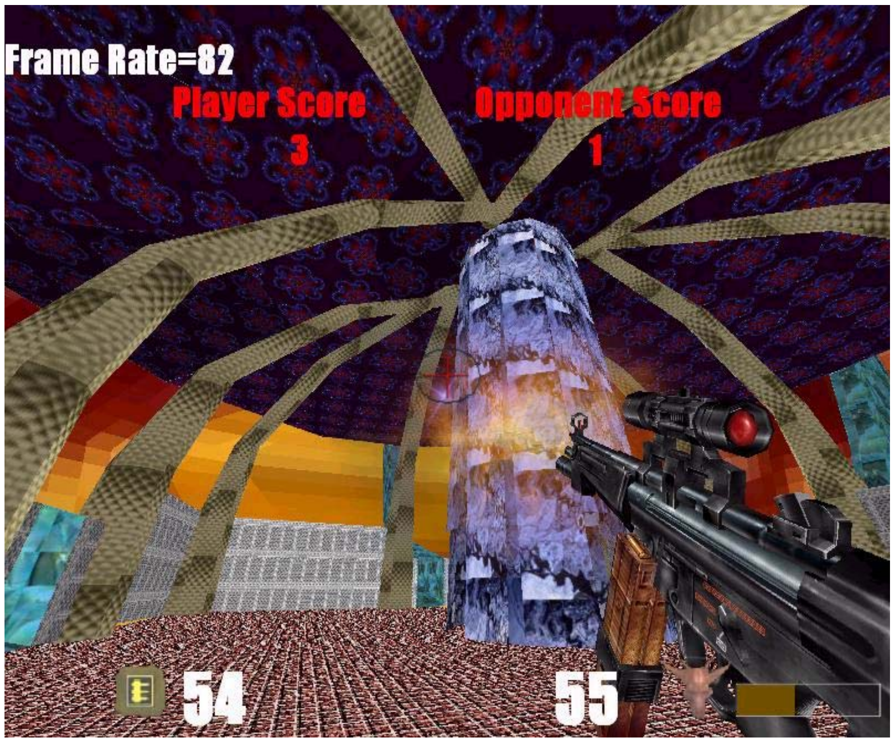

FIGURE 7-1: *Screen Shot Level 1*

7.2 SCREEN SHOT LEVEL 2

Figure 7-2 is the first person view of the player of level 1. In the top left corner in the white is the frame rate of the game. Player score shows the number of times the player has killed the opponent. The opponent score is the number of times the opponent has killed the player. The number 9981 is showing the number of bullets the player is lift with and the 98 on the screen along with the green bar shows the player health. When the player health becomes zero, the player dies.

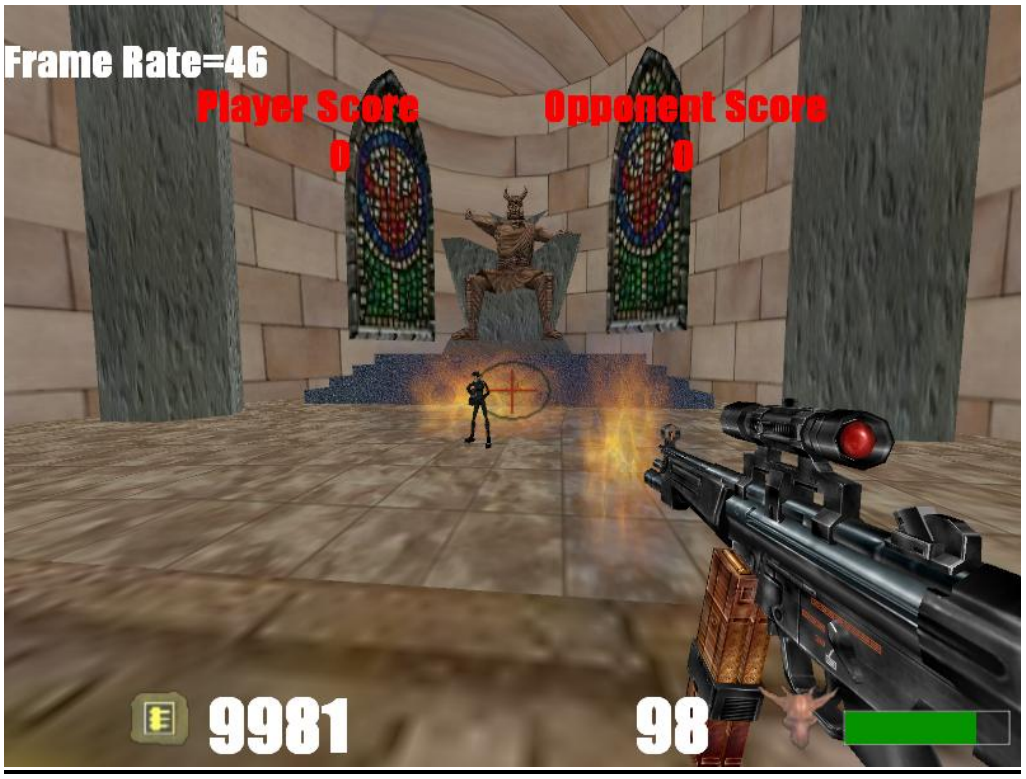

FIGURE 7-2: *Screen Shot Level 2*

7.3 SCREEN SHOT LEVEL 3

Figure 7-3 is the first person view of the player of level 1. In the top left corner in the white is the frame rate of the game. Player score shows the number of times the player has killed the opponent. The opponent score is the number of times the opponent has killed the player. The number 9971 is showing the number of bullets the player is lift with and the 71 on the screen along with the green bar shows the player health. When the player health becomes zero, the player dies.

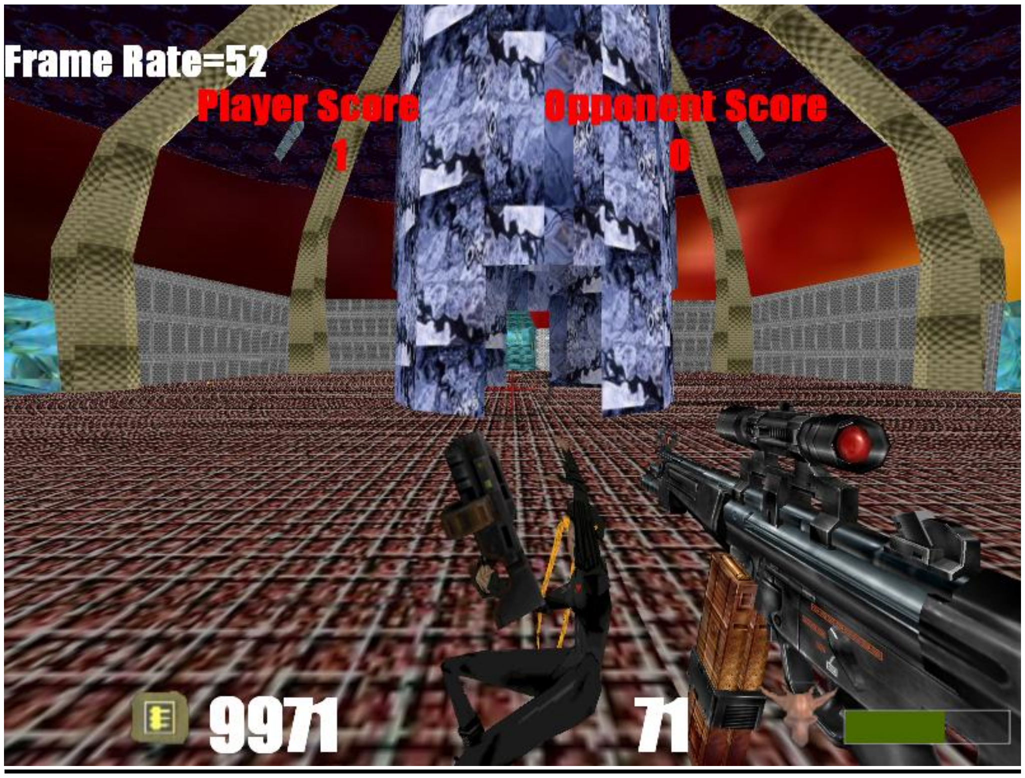

FIGURE 7-3: *Screen Shot Level 3*

CHAPTER 8

RESULTS & CONCLUSION

8.1 RESULTS

Basically our game is platform dependent to only windows operating system. It doesn’t work on the linux or any other operating system. The code was executed on different windows version the summary of which is shown in the table 9.1. From the table it is clear that the game runs fasters and more smoother with the graphics card of higher quality. The frame rate is the indication of the speed at which the game runs on the computer. When the frame rate is lesser, the game stucks while somebody is playing the game. It is easily shown from the results that the frame rate vary in accordance with the graphics card and the memory available. The games developed by specialists are also designed in such a away that the games running on higher resolution and higher Rams runs smoother, so this is not a big issue.

Windows Version	Graphics Card	RAM	Frame Rate
Windows 98	128 MB 9200 Radeon	512 MB RD	45
	128 MB 9800 Radeon	1 GB SD	78
Windows 2000	128 MB 9200 Radeon	512 MB RD	42
	128 MB 9800 Radeon	1 GB SD	67
Windows XP	128 MB 9200 Radeon	512 MB RD	41
	128 MB 9800 Radeon	1 GB SD	61

TABLE 9-1 *Results*

8.2 ANALYSIS

We have developed a 3D engine with the features including capable of dynamic rendering. It has a support for vertex shaders and a full support for pixel shaders (from high end graphics cards). It has the feature of lightning and shadows using per-pixel lightning and multi-level texturing. The engine has the geo-mod technology along with the real world collision and other physics. The engine includes the features of particle effects and explosions. It has a support for 3D sound FX and is capable of interfacing with a variety of input devices. We have used the md2 models of the characters already available and have used md2 animation for the characters.

We have also demonstrated the capabilities of the 3D engine by using it to develop an Action Adventure RPG (Role Playing Game) with RTS (Real-time Strategy) elements. All technical aspects of the engine are highlighted in the game. Game displays a fully rendered and changeable 3D world. State of the art AI is also implemented in the game. Game is produced using production values of the highest quality.

8.3 PROBLEMS FACED

As can be expected, any project is never free from impediments to progress. We had our share of obstacles laid out to test our patience as well. A few are described below:

8.3.1 Non Availability of Equipment

The availability of equipment was a major headache. The equipment required for the rendering of the scene was a major headache. Infact rendering requires a high quality graphics card along with the Rams of 512MB. This was done so that the process of redering can be made fast

8.3.2 Shortage of Time

The time available was too short for us to develop a 3D Engine to the extent that we wanted. Had more time been available, we could have worked on the networking module if we were given a month more for the project.

8.3.3 Availability of Help Regarding 3D Engine

There is hardly any open source available for the 3D Engine in the DirectX. Help is scarcely available to undergraduates in this important field of engineering. Not many people work in this field and are willing to guide new students especially in Pakistan.

8.3.4 Model Loading

Model loading was difficult as the format of the 3D program used (Maya) wasn't compatible with the API's being used. Therefore we had to write our own converter routine so that the models made in Maya could be opened in our own application.

8.4 FUTURE WORK

The features such as 64-bit color High Dynamic Range rendering pipeline can be added to the engine. The gamma-correct, linear color space renderer provides for immaculate color precision while supporting a wide range of post processing effects such as light blooms, lenticular halos, and depth-of-field. Support for all modern per-pixel lighting and rendering techniques including normal mapped, parameterized Phong lighting; virtual displacement mapping; light attenuation functions; pre-computed shadow masks; and pre-computed bump-granularity self-shadowing using spherical harmonic maps. Advanced Dynamic Shadowing. Tupachi Engine provides full support for three shadow techniques. The first one is dynamic stencil buffered shadow volumes supporting fully dynamic, moving light sources casting accurate shadows on all objects in the scene. The second one is dynamic characters casting dynamic soft, fuzzy shadows on the scene using 16X-oversampled shadow buffers. The third one is ultra high quality and high performance pre-computed shadow masks allow offline processing of static light interactions, while retaining fully dynamic specular lighting and reflections.

All of the supported shadow techniques are visually compatible and may be mixed freely at the artist's discretion, and may be combined with colored attenuation functions enabling properly shadowed directional, spotlight, and projector lighting effects. Powerful material system, enabling artists to create arbitrarily complex real time shaders on-the-fly in a visual interface that is comparable in power to the non-real time functionality provided by Maya. The material framework is modular, so programmers can add not just new shader programs, but shader components which artists can connect with other components on-the-fly, resulting in dynamic composition and compilation of shader code. Full support for seamlessly interconnected indoor and outdoor environments with dynamic per-pixel lighting and shadowing supported everywhere. Artists can build terrain using a dynamically-deformable base height map extended by multiple layers of smoothly-blended materials including displacement maps, normal maps and arbitrarily complex materials, dynamic LOD-based tessellation, and vegetation layers with procedurally-placed meshes. Further, the terrain system supports artist-controlled layers of procedural weathering, for example, grass and vegetation on the flat areas of terrain, rock on high slopes, and snow at the peaks. Volumetric environmental effects including height fog and physically accurate distance fog can also be added.

8.5 CONCLUSION

Game development is currently enjoying the largest share out of computer industry (more than 60%). It is the biggest industry in Silicon Valley but largely ignored in Pakistan. So the group's main objective was to learn advance game programming techniques and to familiarize us with animation, character building, graphic designing, music composition and coding using DirectX, OpenGL. We tried to enhance our knowledge of AI (Artificial Intelligence) and Computer Graphics while learning about the industry standards and research ways to improve the various AI and rendering algorithms used in computer games. Game development is an ever changing and evolving field as the need to capture the biggest market share drives companies to develop better games with better graphics, artificial intelligence and real world kinematics. During the project we tried to develop a game that would be competitive in the international market.

Tupachi is a complete 3d engine that supports all the essential features required for game development for various genres. Its modular framework ensures extensibility and scalability so that tupachi's feature set can be enhanced to meet next-generation game demands. Since it is developed in DirectX it is windows dependent.

The engine has been developed keeping in mind the latest trends of the gaming industry and is completely extensible and supports many features such as character loading and animation, map loading, an AI module, a physics module and a complete sound module. The engine design is fully modular and can be easily extended to support more features. For example due to the lack of time, the group members weren't able to add multiplayer aspect in the game. But the engine has been designed so that multiplayer aspect can easily be added in the engine. Currently three maps have been provided with the engine but it can load any number of maps provided to it in the .X format. The engine uses the .wav format to load sounds. The content for the demo presentation has been created using MAYA 3D though it can be used with any popular 3D program like 3D Studio Max, Milkshape 3d, etc. This document will be useful for anyone wanting to develop a fully functional 3d game engine or to people who want to use TUPACHI for game development.

APPENDIX A- WORK BREAKDOWN STRUCTURE

Table A-1 on the next page shows the work breakdown structure, which means that how, the work was done and how the different tasks were broken down into smaller ones. The chart is made in the Microsoft project. The chart also shows the details and the number of days taken to complete each task. The chart also shows the dependencies of different tasks on one another.

APPENDIX B- FOG

You can use fog to achieve a number of effects in Microsoft Direct3D Immediate Mode applications. By adding fog to a scene, you can simulate the real world in a powerful way. Combined with the right sounds and music, fog can help you create worlds that convey a range of atmospheres, from mysterious or creepy to fantastic or other-worldly to pastoral or humorous. Even more important for real-time applications, in which you need to eke out the last possible bit of performance, you can use fog to hide the bizarre and distracting effects of objects popping into existence as they cross into the viewing frustum. [6] To prevent popping, you just set up fog so that users can't see beyond the far clipping plane.

Direct3D implements fog by blending the color of each object in a scene with the fog color you select. The amount of blending that occurs is based on the object's distance from the viewpoint. Direct3D blends the colors of distant objects so that the object's final color approximates the color of the fog. The colors of objects that are near the viewpoint change slightly or not at all. For example, if you use a color such as blue or white as your fog color, your objects will become increasingly obscured the farther away from the viewpoint they are, producing the illusion of fog. If you use black as your fog color, objects will appear to fade into the darkness in a night scene. If the scene has a solid background color (that is, if rendered objects don't cover every screen pixel), you should set the fog color to that background color. If objects are rendered over every screen pixel, however, you can pick any fog color you like. Then, as polygons recede from the camera, they will smoothly fade into the background. In this case, white will give you a realistically fogged scene.

Direct3D supplies two different forms of fog you can use in a scene: vertex fog and pixel fog.

Fog Formulas

Fog is a measure of visibility; the lower the value produced by the fog equations, the less visible the object is. You control fog by using the D3DFOGMODE enumerated type, whose members identify the three available fog formulas:

```
typedef enum _D3DFOGMODE {
D3DFOG_NONE   = 0,
D3DFOG_EXP    = 1,
D3DFOG_EXP2   = 2,
D3DFOG_LINEAR = 3
D3DFOG_FORCE_DWORD   = 0x7fffffff,
} D3DFOGMODE;
```

These members are defined as follows:

- **D3DFOG_NONE** No fog is used.
- **D3DFOG_LINEAR** The fog increases linearly between the start and end points, using the following formula:

$$f = \frac{end - d}{end - start}$$

- **D3DFOG_EXP** The fog increases exponentially, using the following formula:

$$f = 1/e^{d \times density}$$

- **D3DFOG_EXP2** The fog increases exponentially with the square of the distance, using the following formula:

$$f = 1/e^{(d \times density)^2}$$

- **D3DFOG_FORCE_DWORD** This member forces this enumerated type to be 32 bits.

Each of the three formulas in the preceding list calculates a fog factor as a function of distance by using the parameters you pass it. How Direct3D computes distance varies depending on the projection matrix you use and on whether you've enabled range-based fog.

You use the first formula for computing linear fog. For linear fog, the *start* value defines the distance at which fog effects begin, *end* defines the distance at which fog effects no longer increase, and *d* specifies the distance from the scene's viewpoint. For all these values, 0.0 corresponds to the near plane and 1.0 corresponds to the far plane. Both pixel fog and vertex fog support linear fog.

The other two formulas Direct3D provides are D3DFOG_EXP and D3DFOG_EXP2. Only pixel fog supports these exponential fog formulas. In these formulas, *e* is the base of natural logarithms (~2.71828); *density* is an arbitrary fog density, which can range from 0.0 through 1.0; and *d* is the distance from the scene's viewpoint.

Figure B-1 shows a graph of the three fog formulas. Densities of 0.33 and 0.66 are the formula parameters for both exponential formulas.

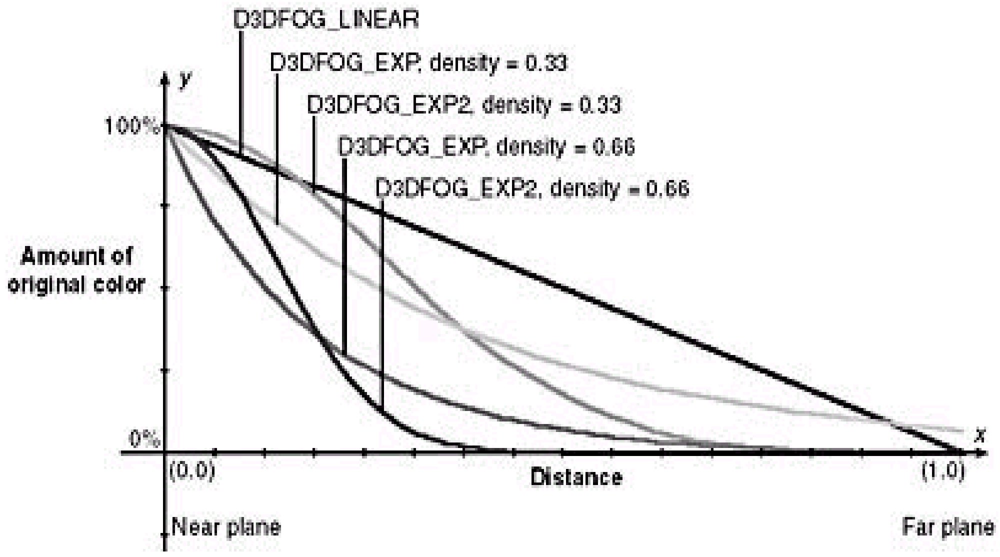

Figure B-1 *Comparison of fog formulas*

The fog factors for each of the three fog effects, which are computed using the equations shown in Figure B-1, are used in the Direct3D blending formula, which is computed as follows:

$$C = f \cdot C_i + (1 - f) \cdot C_f$$

This formula (used for all DirectX devices) scales the color of the current polygon, C_i, by the fog factor, f, and then adds the product to the fog color, C_f, scaled by the inverse of the fog factor. The color value that is computed is a blend of the fog color and the original color, with more of the fog color and less of the original color being blended as the distance increases.

APPENDIX C- MATH AND TRIGONOMETRY REVIEW

Trigonometry

Trigonometry is the study of angles, shapes, and their relationships. [7] Most trigonometries are based on the analysis of a right triangle, as shown in Figure C.1.

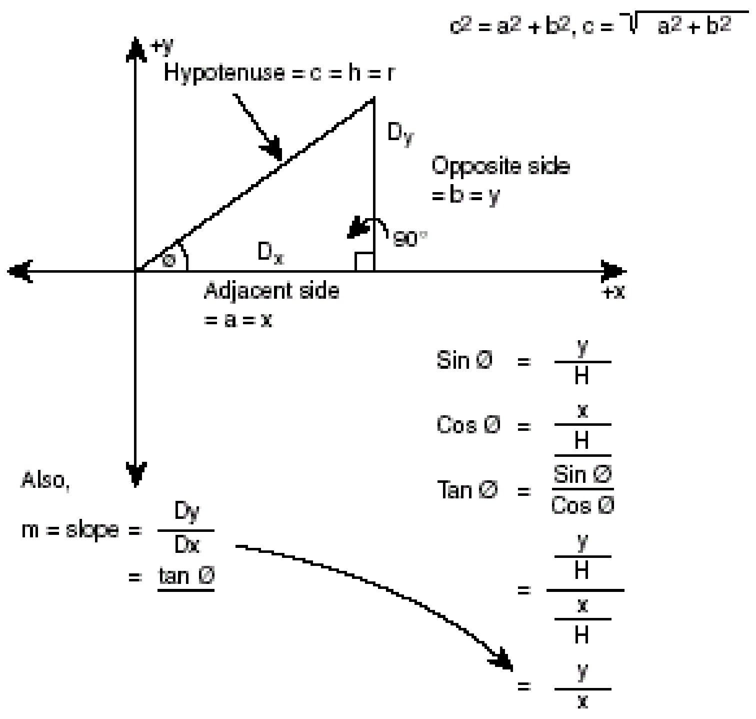

FIGURE C.1- *The Right Triangle*

Table C.1 lists the radian/degree values.

360 degrees = 2*PI radians is approx. 6.28 radians
180 degrees = PI radians is approx. 3.14159 radians
360 degrees
2*PI radians
2*PI radians
360 degrees
= 1 radian is approx. 57.296 degrees
= 1 degree is approx. 0.0175 radians

TABLE C.1 -*Radians vs. Degrees*

Here are some trigonometric facts:

Fact 1: There are 360 degrees in a complete circle, or 2*PI radians. Hence, there are PI radians in 180 degrees. The computer functions sin() and cos() work in radians, *not* degrees—remember that! Table C.1 lists the values.

Fact 2: The sum of the interior angles *theta1* + *theta2* + *theta3* = 180 degrees or PI radians.

Fact 3: Referring to the right triangle in Figure C.1, the side opposite *theta1* is called the *opposite side*, the side below it is called the *adjacent side*, and the long side is called the *hypotenuse*.

Fact 4: The sum of the squares of the sides of a right triangle equals the square of the hypotenuse. This is called the *Pythagorean theorem*. Mathematically, it's written like this:

hypotenuse2 = adjacent2 + opposite2

or sometimes using a, b, and c for dummy variables:

c2 = a2 + b2

Therefore, if you know two sides of a triangle, you can find the third.

Fact 5: There are three main trigonometric ratios that mathematicians like to use:

sine, *cosine*, and *tangent*. They are defined as

$$\text{cos(theta)} = \frac{\text{adjacent side}}{\text{hypotenuse}} = \frac{x}{r}$$

DOMAIN: 0 <= theta <= 2*PI
RANGE: -1 to 1

$$\text{sin(theta)} = \frac{\text{opposite side}}{\text{hypotenuse}} = \frac{y}{r}$$

DOMAIN: 0 <= theta <= 2*PI
RANGE: -1 to 1

$$\text{tan(theta)} = \frac{\text{sin(theta)}}{\text{cos(theta)}} = \frac{\text{opposite/hypotenuse}}{\text{adjacent/hypotenuse}}$$

$$= \frac{\text{opposite}}{\text{adjacent}} = \frac{y}{x} = \text{slope} = M$$

DOMAIN: -PI/2 <= theta <= PI/2
RANGE: -infinity to +infinity

Figure C.2 shows graphs of all the functions. Notice that they're all periodic (repeating) and that *sin(theta)* and *cos(theta)* have periodicity of 2*PI, while *tangent* has periodicity of PI. Also, notice that *tan(theta)* goes to +-infinity whenever *theta mod PI* is PI/2.

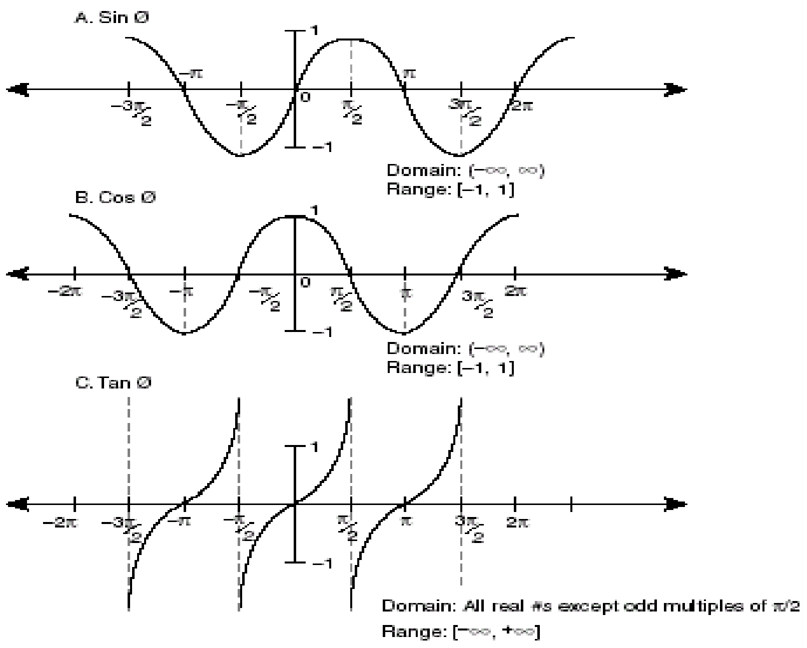

FIGURE C.2- *Graphs of Basic Trigonometric Functions*

Now, there are about a gazillion trigonometric identities and tricks, and it would take a math book to prove them all. I'm just going to show you the ones that a game programmer should know. Table C.2 lists some trigonometric ratios as well as some neat identities.

Cosecant: csc(theta) = 1/sin(theta)

Secant: sec(theta) = 1/cos(theta)

Cotangent: cot(theta) = 1/tan(theta)

Pythagorean Theorem in terms of trig functions:

sin(theta)2 + cos(theta)2 = 1

Conversion identity:

sin(theta1) = cos(theta1 – PI/2)

Reflection identities:

sin(-theta) = -sin(theta)

cos(-theta) = cos(theta)

Addition identities:

sin(theta1 + theta2) = sin(theta1)*cos(theta2) + cos(theta1)*sin(theta2)

cos(theta1 + theta2) = cos(theta1)*cos(theta2) - sin(theta1)*sin(theta2)

sin(theta1 - theta2) = sin(theta1)*cos(theta2) - cos(theta1)*sin(theta2)

cos(theta1 - theta2) = cos(theta1)*cos(theta2) + sin(theta1)*sin(theta2)

TABLE C.2- *Useful Trigonometric Identities*

Of course, you could derive identities until you turned many shades of green. In general, identities help you simplify complex trigonometric formulas into simpler ones so you don't have to do the math. Hence, when you come up with an algorithm based on sin, cos, tan, and so on, always take a look in a trigonometry book to see if you can simplify your math so that fewer computations are needed to get to the result.

APPENDIX D

Modeling Behavioral State Systems

To create a truly robust FSM (Finite State Machines), you need two properties:

First one is reasonable number of states, each of which represents a different goal or motive. Second one is lots of input to the FSM, such as the state of the environment and the other objects within the environment. [8]

The premise of "a reasonable number of states" is easy enough to understand and appreciate. We humans have hundreds, if not thousands, of emotional states, and within each of these we may have further substates. The point is that a game character should be able to move around in a free manner, at the very least. For example, you may set up the following states:

State 1: Move forward.

State 2: Move backward.

State 3: Turn.

State 4: Stop.

State 5: Fire weapon.

State 6: Chase player.

State 7: Evade player.

States 1 to 4 are straightforward, but states 5, 6, and 7 might need substates to be properly modeled. This means that there may be more than one phase to states 5, 6, and 7. For example, chasing the player might involve turning and then moving forward.

Take a look at Figure D-1 to see the concept of substates illustrated. However, don't assume that substates must be based on states that actually exist—they may be totally artificial for the state in question.

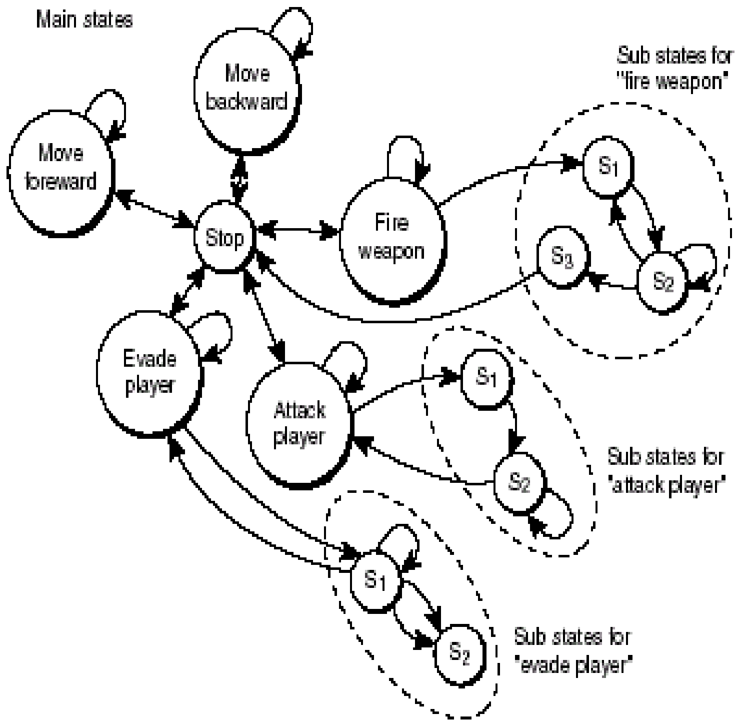

FIGURE D-1: *A Master FSM with Finite Substates*

The point of this discussion of states is that the game object needs to have enough variety to do "intelligent" things. If the only two states are stop and forward, there isn't going to be much action! Remember those stupid remote-control cars that went forward and then reversed in a left turn? What fun was that?

Moving on to the second property of robust FSM AIs, you need to have feedback or input from the other objects in the game world and from the player and environment. If you simply enter a state and run it until completion, that's pretty dumb. The state may have been selected intelligently, but that was 100 milliseconds ago. Now things have changed, and the player just did something that the AI needs to respond to. The FSM needs to track the game state and, if needed, be preempted from its current state into another one.

Elementary State Machines

At this point, you should be seeing a lot of overlap in the various AI techniques. For example, the pattern techniques are based on finite state machines at the lowest level which perform the actual motions or effects. What I want to do now is take finite state machines to another level and talk about high-level states that can be implemented with simple conditional logic, randomness, and patterns. In essence, I want to create a virtual brain that directs and dictates to the creature.

To better understand what I'm talking about, let's take a few behaviors and model them with the aforementioned techniques. On top of these behaviors, we'll place a master FSM to run the show and set the general direction of events and goals.

Most games are based on conflict. Whether conflict is the main idea of the game or it's just an underlying theme, the bottom line is that most the time the player is running around destroying the enemies and/or blowing things up. As a result, we can arrive at a few behaviors that a game creature might need to survive given the constant onslaught of the human opponent. Take a look at Figure D-2, which illustrates the relationships between the following states:

Master State 1: Attack.

Master State 2: Retreat.

Master State 3: Move randomly.

Master State 4: Stop or pause for a moment.

Master State 5: Look for something—food, energy, light, dark, other computer-controlled creatures.

Master State 6: Select a pattern and follow it.

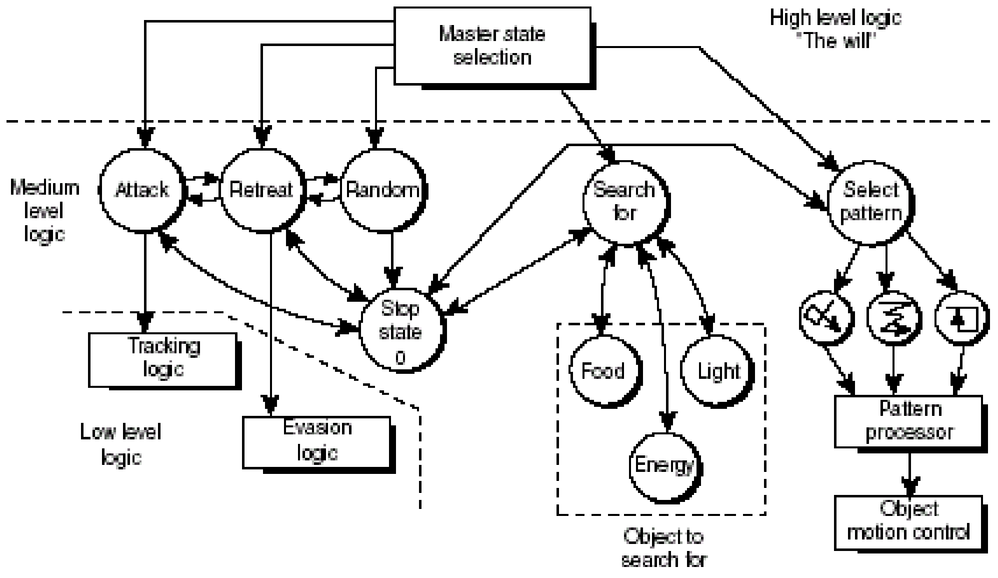

FIGURE D-2: *Building a Better Brain*

You should be able to see the difference between these states and the previous examples. These states function at a much higher level, and they definitely contain possible substates or further logic to generate. For example, states 1 and 2 can be accomplished using a deterministic algorithm, while states 3 and 4 are nothing more than a couple of lines of code. On the other hand, state 6 is very complex because it dictates that the creature must be able to perform complex patterns controlled by the Master FSM. [9]

As you can see, your AI is getting fairly sophisticated. State 5 could be yet another deterministic algorithm or even a mix of deterministic algorithms and preprogrammed search patterns. The point is that you want to model a creature from the top down; that is, first think of how complex you want the AI of the creature to be, and then implement each state and algorithm.

BIBLIOGRAPHY

[1] Kelly Dempski, ***Real time rendering tricks and techniques in DirectX***, Premier Press, 2002.

[2] Kris Gray, ***The Microsoft DirectX 9 Programmable Graphics Pipeline***, Microsoft Press, 30 July, 2003.

[3] Mason McCuskey, Andre LaMothe, ***Special Effects Game Programming with DirectX*** , Premier Press, 01 December, 2001.

[4] Robert Dunlop, DALE Shepard, Mark Martin, ***Teach Yourself DirectX 7 in 24 Hours***, SAMS, December, 1999.

[5] Andre Lamothe, ***Tricks of the Windows Game Programming Gurus fundamentals of 2D and 3D Game Programming***, SAMS, October 1999.

[6] Peter Walsh, ***The Zen of Direct3D Game Programming***, Premier Press, 01 June, 2002.