Department of Computer Science and Engineering
The University of Texas at Arlington

Detailed Design Specification

Project: Virtual Reality Xplorer

Team Members:
Osuvaldo Ramos
Joseph Onwuchekwa
Sukuya Nakhaima
Chris Otterbine

Last Updated: 2/28/2014 1:18 PM

Table of Contents

TABLE OF CONTENTS 2

DOCUMENT REVISION HISTORY 4

LIST OF FIGURES 5

LIST OF TABLES 6

1. INTRODUCTION 7

1.1 PRODUCT CONCEPT 7
1.2 PRODUCT SCOPE 7

2. ARCHITECTURE OVERVIEW 8

2.1 UNREAL DEVELOPMENT KIT 8
2.2 MODULE DECOMPOSITION 8
2.3 DECOMPOSITION DIAGRAM 9
2.4 PRODUCER-CONSUMER RELATIONSHIP MATRIX 10

3. INPUT LAYER: USER INPUT SUBSYSTEM 11

3.1 PLAYER INPUT MODULE 11
3.2 PLAYER CONTROLLER MODULE 13

4. PROCESSING LAYER: GAME MAP SUBSYSTEM 15

4.1 PAWN MODULE 15
4.2 ACTORS MODULE 17
4.3 GAME EVENTS MODULE 19
4.4 PAUSE MENU MODULE 22
4.5 HUD MODULE 24

5. PROCESSING LAYER: GAME SUBSYSTEM 26

5.1 GAME INFO MODULE 26
5.2 SAVE GAME STATE MODULE 28

5.3 ACADEMIC REPORT MODULE 30

6. PROCESSING LAYER: USER INTERFACE MAPS 32

6.1 LOAD SCREEN MODULE 32
6.2 MAIN MENU MODULE 34

7. STORAGE LAYER: CONTENT SUBSYSTEM 36

7.1 TEXTURES MODULE 36
7.2 MATERIALS MODULE 37
7.3 STATIC MESH MODULE 38
7.4 AUDIO MODULE 39

8. QUALITY ASSURANCE 40

8.1 TEST PLANS AND PROCEDURES 40
8.2 MODULE TEST 40
8.3 COMPONENT TESTING 42
8.4 INTEGRATION TESTING 43
8.5 REQUIREMENT TESTING 43

9. REQUIREMENTS TRACEABILITY MATRIX 44

10. ACCEPTANCE PLAN 46

10.1 OVERVIEW 46
10.2 INSTALLATION AND PACKAGING 46
10.3 ACCEPTANCE TESTING 46
10.4 ACCEPTANCE CRITERIA 47

Document Revision History

Revision Number	Revision Date	Description	Rationale
0.1	2/10/14	Initial Integration	
1.0	2/10/14	Review-ready final version	Added QA, Acceptance Plan, and Traceability matrices
1.1	2/27/14	Re-review version	Completely reworked modules and revised Quality Assurance section. Corrected list of figures and tables. Added description of UDK and tools.
2.0	2/28/14	Baseline version	Baseline submission

List of Figures

Figure #	Title	Page #
2-1	Detailed Design Decomposition Diagram	9
3-1	Player Input Module	11
3-2	Player Controller Module	13
4-1	Pawn Module	15
4-2	Actors Module	17
4-3	Game Events Module	19
4-4	Kismet Visual Script	20
4-5	Pause Menu Module	22
4-6	HUD Module	24
5-1	Game Info Module	25
5-2	Save Game State Module	28
5-3	Academic Report Module	30
6-1	Load Screen Module	32
6-2	Main Menu Module	34
7-1	Textures Module	36
7-2	Materials Module	37
7-3	Static Mesh Module	38
7-4	Audio Module	39

List of Tables

Table #	Title	Page #
2-1	Producer-Consumer Relationship Matrix	10
3-1	Input Module Interfaces	12
3-2	Player Controller Module Interfaces	13
4-1	Pawn Module Interfaces	15
4-2	Actors Module Interfaces	17
4-3	Game Events Module Interfaces	19
4-4	Pause Menu Module Interfaces	22
4-5	HUD Module Interfaces	24
5-1	Game Info Module Interfaces	26
5-2	Save Game State Module Interfaces	29
5-3	Academic Report Module Interfaces	30
6-1	Main Menu Module Interfaces	34
9-1	Requirements Traceability Matrix	44
10-1	Acceptance Criteria	47

1. Introduction

1.1 Product Concept

The Virtual Reality Xplorer is an educational video game that will employ the Oculus Rift virtual reality device and an Xbox controller to immerse students in a virtual environment where they can learn and explore different topics from their curriculum. Students will be presented different topics and learn how to apply the knowledge they are learning in the virtual environment. Teachers will be able to see how each student performed in the different virtual environments.

The Virtual Reality Xplorer is a product designed to simulate an environment while simultaneously providing the user an entertaining and educational experience. The Virtual Reality Xplorer will be installed on a PC. The program will be launched from the operating system and display the main menu. Once the user starts a new game, they will be allowed to explore an open environment while the Virtual Reality Xplorer displays information using the heads up display. The user will also encounter intermittent puzzles and challenges.

The Virtual Reality Xplorer's intended users will be 5th and 6th grade science students. The intended consumer will be 5th and 6th grade science teachers that want an alternative method of teaching a certain topic. Other audiences may consist of school districts or educational programs.

1.2 Product Scope

The Virtual Reality Xplorer sets out to provide an educational and entertaining experience through virtual reality as to allow the user to gain more "sensory" knowledge.

The hardware component of this system accommodates the Oculus Rift, a piece of hardware needed for virtual reality, an Xbox 360 controller for lateral movement, and a wireless headset for user freedom.

The user interface will feature a menu for selecting options such as new, save, quit, etc. While navigating through the virtual environment, the user will see the environment as well as the HUD (Heads-up Display). The HUD is a user interface that provides a crosshair, tool tips when you can interact with objects, and statistics. The user sees the HUD in addition to the virtual environment.

This product is designed for 5th and 6th graders, although the system is informative to all ages.

2. Architecture Overview

2.1 Unreal Development Kit

The Unreal Development Kit (or UDK) is a development environment specifically for game development. The UDK acts as an operating environment for our product. Every module is built on top of the UDK. The UDK handles all input and outputs from the Oculus Rift and Xbox Controller as well as rendering graphics and audio. The UDK also comes with a set of INI files that can be modified to customize the functionality of the UDK. The UDK provides tools to make the game development process easier. The Unreal Editor is a software application for building maps and managing content such as textures, materials, static meshes, animations, lighting, and audio. UnrealScript is an object oriented programming language for the UDK environment. The framework provides hundreds of utility and base classes as a starting point. Kismet is a visual scripting language with built-in triggers, events, and actions that can be connected to the map and actors in the game. Matinee is an animation scripting tool that allows users to create custom animations easily. An actor is any object that can interact with the map world. Actors can be cameras, sound actors that trigger sound events, objects that the player can interact with, or static objects like walls and floors. Actors can be placed using the Unreal Editor or dynamically with UnrealScript. A Pawn is a specific actor that represents the player's physics presence in the map. A heads-up display (or HUD) is the display drawn over the player's view that displays information about their gameplay.

2.2 Module Decomposition

A. User Input Subsystem
    1. Player Input Module
    2. Player Controller Module
B. Game Map Subsystem
    1. Pawn Module
    2. Actors Module
    3. Game Events Module
    4. Pause Menu Module
    5. HUD Module
C. Game Subsystem
    1. Game Info Module
    2. Save Game State Module
    3. Academic Report Module
D. User Interface Maps Subsystem
    1. Load Screen Module
    2. Main Menu Module
E. Content Subsystem
    1. Textures Module

2. Materials Module
3. Static Mesh Module
4. Audio Module

2.3 Decomposition Diagram

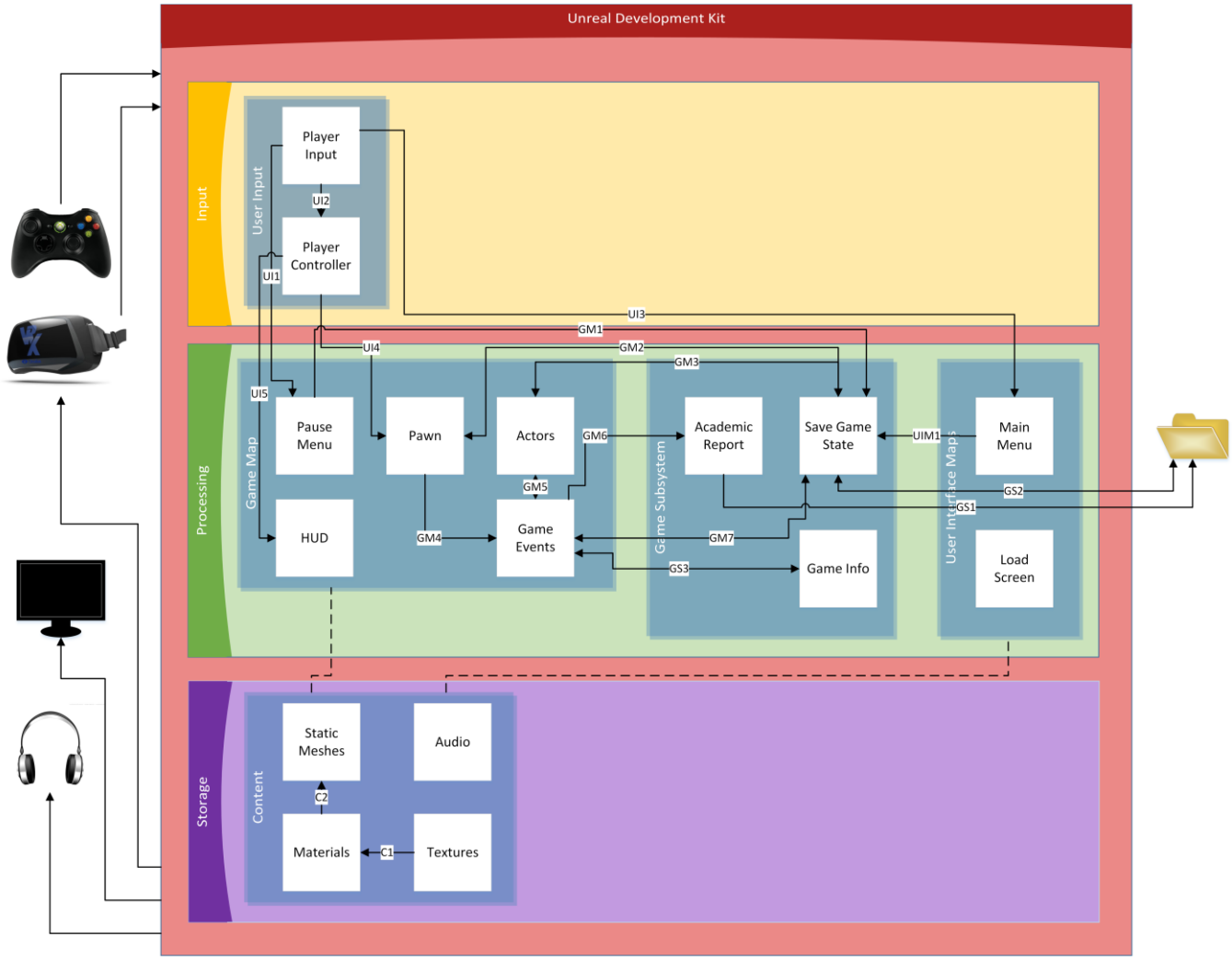

Figure 2.1 Detailed Design Module Decomposition Diagram

In this diagram the UDK encompasses the entire system. Every module interacts with the UDK in one way or another so it would not make sense to depict it as a subsystem/modules. The way it is depicted in this diagram is meant to show how the UDK acts as an operating environment for the subsystems and modules.

2.4 Producer-Consumer Relationship Matrix

Producer	Player Input	Player Controller	Pause Menu	HUD	Pawn	Actor	Game Events	Save Game State	Game Info	Academic Report	Main Menu	Load Screen	Static Meshes	Materials	Audio	Textures
Player Input		UI2	UI1		UI3						UI5					
Player Controller					UI4											
Pause Menu								GM1								
HUD																
Pawn							GM4									
Actor							GM5									
Game Events						GM5		GM7		GM6						
Save Game State					GM2	GM3	GM7									
Game Info							GS3									
Academic Report																
Main Menu								UIM1								
Load Screen																
Static Meshes																
Materials													C2			
Audio																
Textures														C1		

Table 2.1 Producer-Consumer Relationship Matrix

3. Input Layer: User Input Subsystem

3.1 Player Input Module

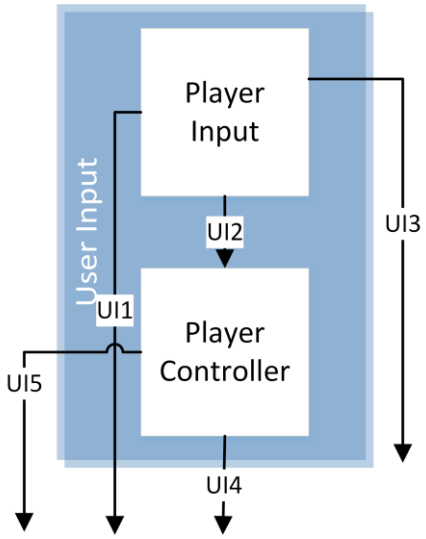

Figure 3.1 Player Input Module

3.1.1 Description

Stores the player's input in variables that are updated by the UDK. The UDK reads the UDKInput.ini file and loads bindings from keyboard, mouse, and gamepad (Xbox Controller in this case) to variables defined inside of the Player Input module. This module is a class that extends the default UDK's PlayerInput class.

3.1.2 Interfaces

Data	Source	Destination	Description
-	UDK	Player Input	The state of the buttons and axes on Xbox controller, mouse buttons and movement, and keyboard key presses.
UI2	Player Input	Player Controller	A floating point number between -1.0 and 1.0 for each joystick axis. A byte with either 0 or 1 for not pressed or pressed for each button.
UI3	Player Input	Main Menu	A floating point number between -1.0 and 1.0 for each joystick axis. A byte with either 0 or 1 for not pressed or pressed for each button.
UI1	Player Input	Pause Menu	A floating point number between -1.0 and 1.0 for each joystick axis. A byte with either 0 or 1 for not pressed or pressed for each button.

Table 3.1 Player Input Module Interfaces

3.1.3 Physical Data Structures and Data File Descriptions

```
/**
Defines the variables the UDK updates
**/
class VRXPlayerInput extends UDKPlayerInput

var float gForward;
var float gBack;
var float gLeft;
var float gRight;
var input byte  aRun;
var input byte  bInteract;
var input byte  yJump;
var input byte  xActivate;
```

3.1.4 Processing

```
/**
Acquires and binds buttons to certain actions
**/
function initializeInput(){
    InitializeBindings = (Name=" XboxTypeS_A",Command="Button aRun");
    InitializeBindings = (Name=" XboxTypeS_LeftX ",Command="Axis gLeft");
   //Additional bindings go here
}
```

3.2 Player Controller Module

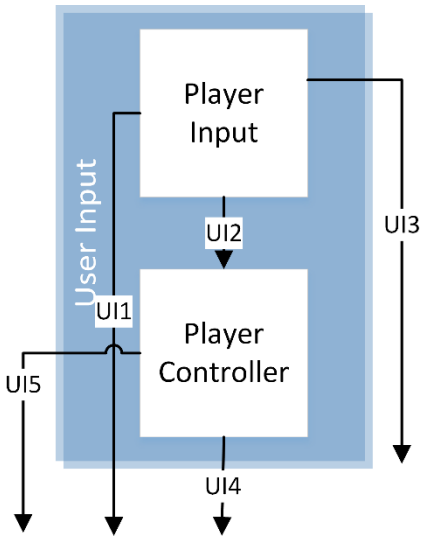

Figure 3.2 Player Controller Module

3.2.1 Description

Reads the input values from the Player Input class and updates the player's Pawn speed and direction. The UDK reads the Oculus Rift settings from the UDKEngine.ini file and updates the game camera

3.2.2 Interfaces

Data	Source Module	Destination Module	Description
UI2	Player Input	Player Controller	A floating point number between -1.0 and 1.0 for each joystick axis. A byte with either 0 or 1 for not pressed or pressed for each button.
UI4	Player Controller	Pawn	Speed and direction of the player's Pawn.
UI5	Player Controller	HUD	Player's status to be displayed on the HUD.

Table 3.2 Player Controller Module Interfaces

3.2.3 Physical Data Structures and Data File Descriptions

None.

3.2.4 Processing

```
class VRXPlayerController extends UDKPlayerController{

        /**
        Translates certain values of the Xbox Controller into values that are  easily calculated.
        **/
        function convertValues(float accelerationInput, float rotationRate,
                               float tilt, float gravityInput){

                float sensitivity = VectorValue(accelerationInput, rotationRate,
                                                tilt, gravityInput);
                                    //VectorValue represents the amount of
                                    sensitivity generated by these four inputs via
                                    the control sticks

                float rotation = RotationValue(rotationRate,tilt);
                                    //FloatValue represents the rotation generated
                                    by rotating the control sticks.  0 represents
                                    no change while 65535 represents a full
                                    revolution

                }
}
```

4. Processing Layer: Game Map Subsystem

4.1 Pawn Module

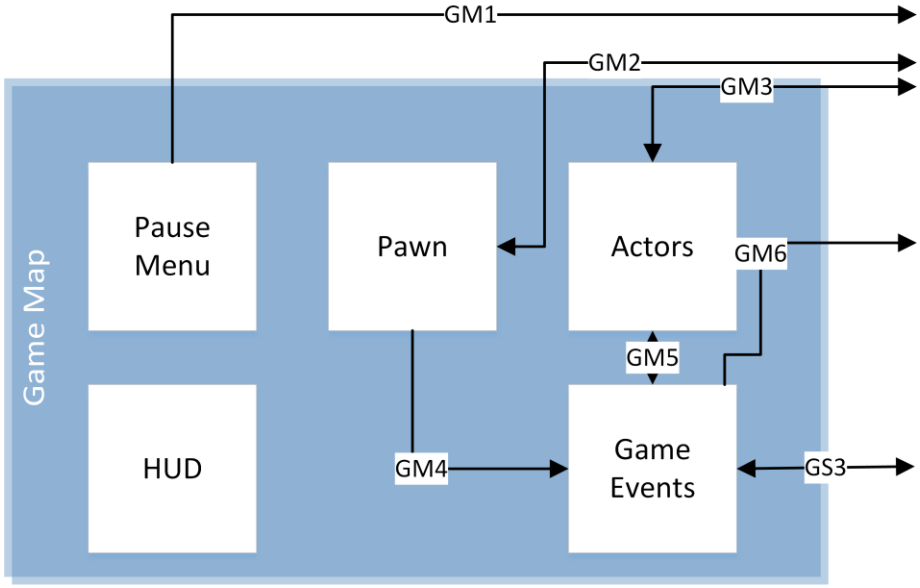

Figure 4.1 Pawn Module

4.1.1 Description

The Pawn is an Actor that represents the player's physical presence in the game. The pawn class is specified in the Game Info module which is loaded when the map is loaded. The pawn is assigned a static mesh which the UDK uses to calculate collisions and render the pawn.

4.1.2 Interfaces

Data	Source Module	Destination Module	Description
UI4	Player Controller	Pawn	Speed and direction of the player's Pawn.
GM2	Save Game State	Pawn	Serialized data to be de-serialized by the Pawn.
GM2	Pawn	Save Game State	Serialized Pawn data to be saved to save file.
GM4	Pawn	Game Events	Pawn triggers an event in a Kismet script.

Table 4.1 Pawn Module Interfaces

4.1.3 Physical Data Structures and Data File Descriptions

None

4.1.4 Processing

```
A Pawn is the player.  This class stores and initializes all properties of the player.
**/
class VRXPawn extends Pawn{

	/**
	Takes in parameters representing the properties of the player and initializes them
	**/
	function initializeProperties(UTPawn pwn, Swimming swm, Camera cmr, ....){
		var UTPawn height = pwn.EyeHeight();
		var Swimming swimSpeed = swm.zSwimming();
		var Camera victoryCamera = cmr.HeroCamera();
		//More initializations
	}
}
```

4.2 Actors Module

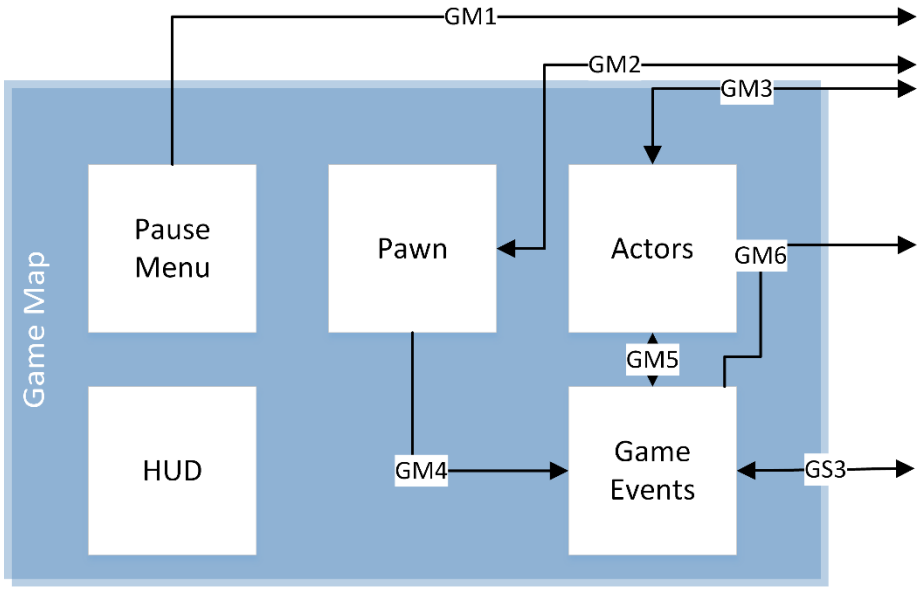

Figure 4.2 Actors Module

4.2.1 Description

A map is made up of a collection of actors. The Actors module represents this collection of actors. Objects in the environment are instances of the Actor class. Actors can be cameras, sound actors that trigger sound events, objects that the player can interact with, or static objects like walls and floors. Actors are placed in the map using the UDK Editor.

4.2.2 Interfaces

Data	Source Module	Destination Module	Description
GM3	Save Game State	Actors	Serialized data to be de-serialized by the Actor.
GM3	Actors	Save Game State	Serialized Actor data to be saved to save file.
GM5	Actors	Game Events	Actors triggers an event in a Kismet script.
GM5	Game Events	Actors	Game Events can cause an Actor's state to change.

Table 4.2 Actors Module Interfaces

4.2.3 Physical Data Structures and Data File Descriptions

None

4.2.4 Processing

```
/**
Actors are game related objects that make up the overall structure of a game.  Examples would be an object
that determines where a player spawns or adding a lighting effect.
This class initializes and spawns all actors that were created
**/
class VRXActor extends Actor{

	/**
	The function initializeActors takes in an array of Actors as well as a vector of their
	location and spawns them respectively in the world.
	**/
	function initializeActors(class <Actor> arrAct, class <Actor>(Vector) location){

		Spawn(arrAct,location);
	}

}
```

4.3 Game Events Module

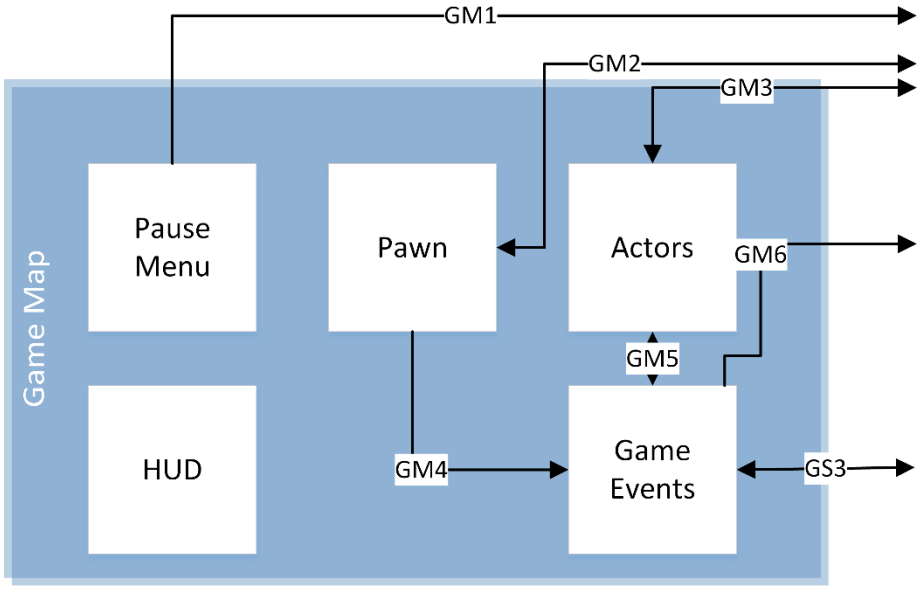

Figure 4.3 Game Events Module

4.3.1 Description

The Game Events module consists of Kismet scripts that control specific game events and Matinee scripts that control game animations. Kismet scripts are triggered by the player Pawn and Kismet scripts can play and stop Matinee animations.

4.3.2 Interfaces

Data	Source Module	Destination Module	Description
GM4	Pawn	Game Events	Pawn triggers an event in a Kismet script.
GS3	Game Info	Game Events	Game state data
GS3	Game Events	Game Info	Updates game state data
GM7	Game Events	Save Game State	Serialized Kismet and Matinee script data that captures the state of each script
GM7	Save Game State	Game Events	Serialized Kismet and Matinee script data to restore the state of each script

GM6	Game Events	Academic Report	Sends academic progress information to be written to the report file

Table 4.3 Game Events Module Interfaces

4.3.3 Physical Data Structures and Data File Descriptions

None

4.3.4 Processing

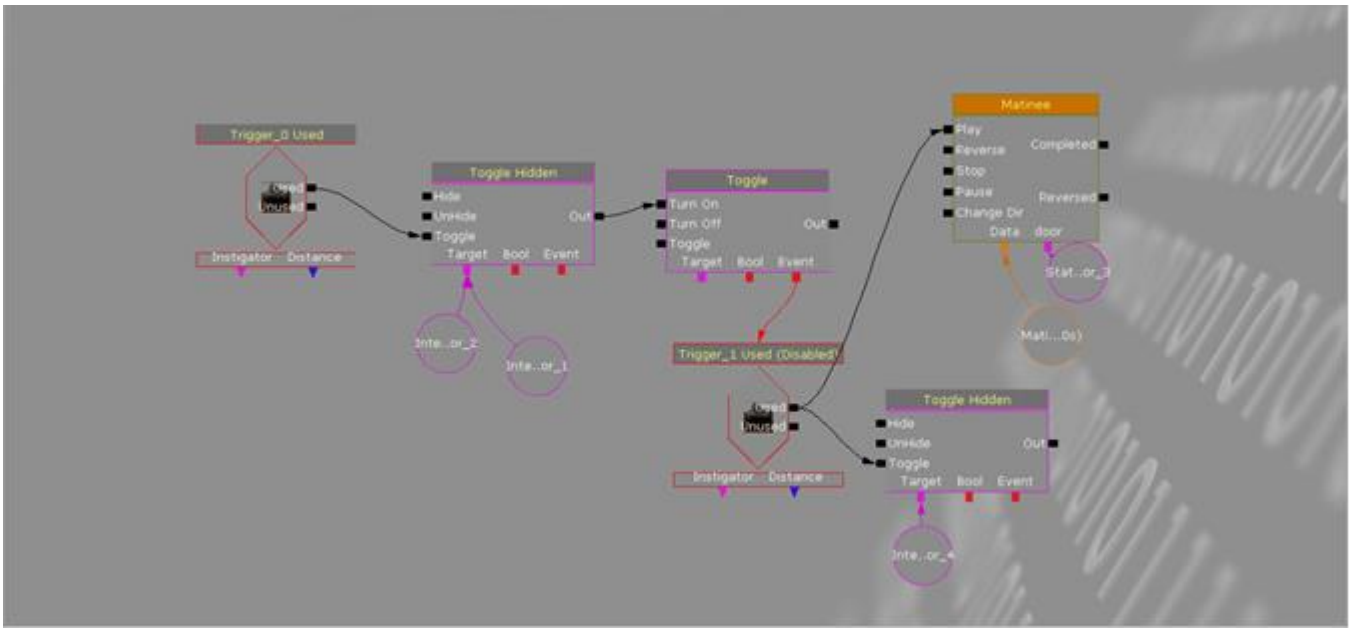

Figure 4.4 Kismet Visual Script

The visual script above demonstrates a simple event. A trigger (Ex: switch) is assigned to an object and sends the data to the Action Object "Toggle Hidden". This object sends data to the "Toggle" object in which upon success, triggers one more event which sends this data to the Matinee object. The Matinee object functions as an animation and upon feedback from Trigger_1, opens a door.

```
Begin Object Class=SeqVar_Object Name=SeqVar_Object_6
   ObjValue=InterpActor'InterpActor_4'
   ObjInstanceVersion=1
   ParentSequence=Sequence'Open_Door_Switch'
   ObjPosX=392
   ObjPosY=648
   DrawWidth=32
   DrawHeight=32
   Name="SeqVar_Object_6"
   ObjectArchetype=SeqVar_Object'Engine.Default__SeqVar_Object'
End Object
Begin Object Class=SeqAct_ToggleHidden Name=SeqAct_ToggleHidden_1
```

```
   InputLinks(0)=(DrawY=541,OverrideDelta=14)
   InputLinks(1)=(DrawY=562,OverrideDelta=35)
   InputLinks(2)=(DrawY=583,OverrideDelta=56)
   OutputLinks(0)=(DrawY=562,OverrideDelta=35)

VariableLinks(0)=(LinkedVariables=(SeqVar_Object'SeqVar_Object_6'),DrawX=422,Override
Delta=16)
   VariableLinks(1)=(DrawX=473,OverrideDelta=76)
   EventLinks(0)=(DrawX=522,OverrideDelta=119)
   ObjInstanceVersion=1
   ParentSequence=Sequence'Open_Door_Switch'
   ObjPosX=384
   ObjPosY=504
   DrawWidth=173
   DrawHeight=109
   Name="SeqAct_ToggleHidden_1"
   ObjectArchetype=SeqAct_ToggleHidden'Engine.Default__SeqAct_ToggleHidden'
End Object
```

4.4 Pause Menu Module

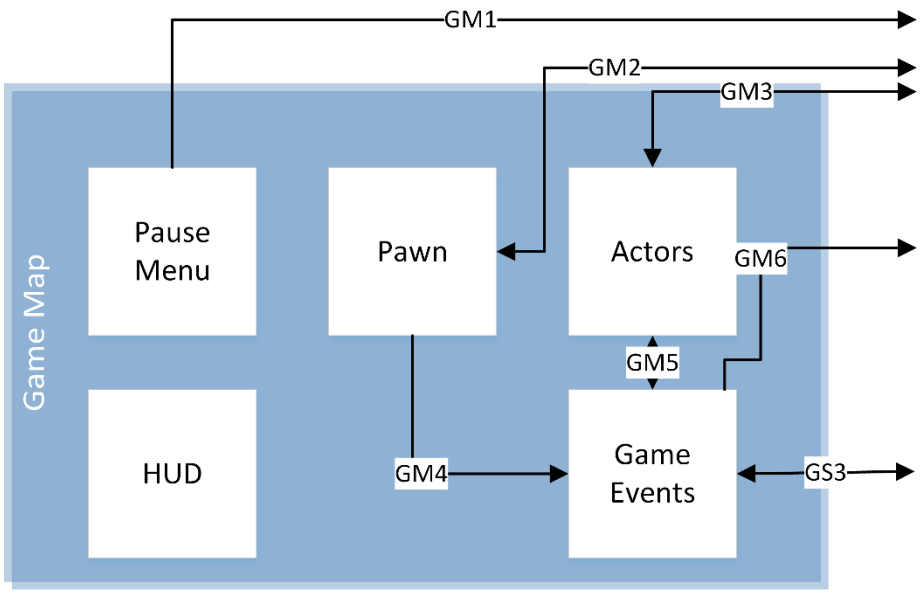

Figure 4.5 Pause Menu Module

4.4.1 Description

The Pause Menu is responsible for displaying the pause menu and invoking the process for saving the game and exiting to the main menu. The UDK must be provided with a class to display the pause menu. By default, the Escape key on the keyboard and the Start button on the Xbox Controller are bound to toggle the pause menu.

4.4.2 Interfaces

Data	Source Module	Destination Module	Description
UI1	Player Input	Pause Menu	Reads the input from the controller to navigate the menu.
GM1	Pause Menu	Save Game State	Invokes the Save Game State object to begin serializing all the game state data and gives it the file name to save to.

Table 4.4 Pause Menu Module Interfaces

4.4.3 Physical Data Structures and Data File Descriptions

The Pause Menu Module implements Scaleform GFx, an external API that utilizes Flash in order to create graphical user interfaces.

4.4.4 Processing

Processing is handled by a Kismet script that displays the menu items and responds to user input.

4.5 HUD Module

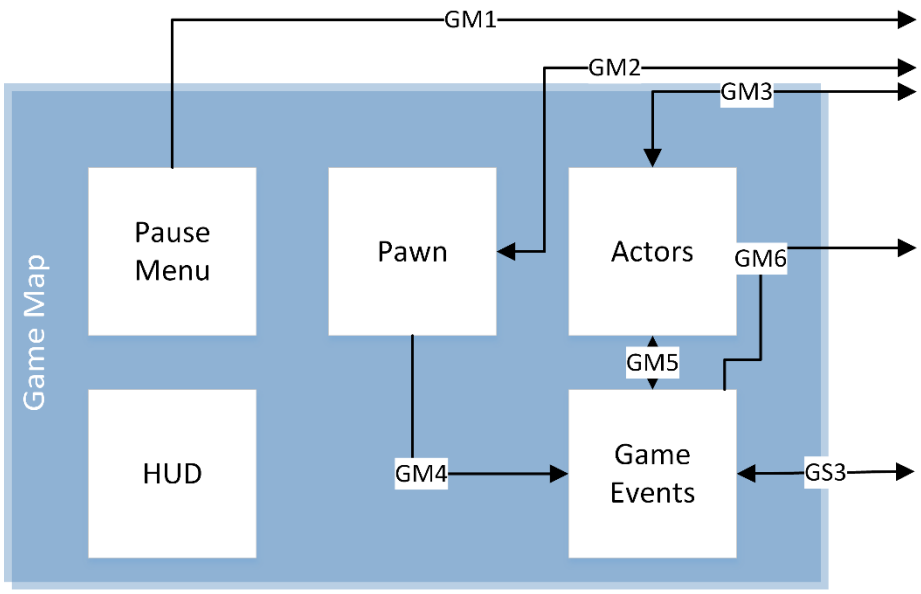

Figure 4.6 HUD Module

4.5.1 Description

The HUD is responsible for displaying the heads-up display over the camera view. The UDK must also be provided with a class to draw the HUD and toggles between the Pause Menu and the HUD automatically when the key bound to the pause menu is pressed.

4.5.2 Interfaces

Data	Source Module	Destination Module	Description
UI5	Player Controller	HUD	Reads the player's state to display on the HUD

Table 4.5 HUD Module Interfaces

4.5.3 Physical Data Structures and Data File Descriptions

The HUD Module implements Scaleform GFx, an external API that utilizes Flash in order to create graphical user interfaces.

4.5.4 Processing

```
#import "GFx.lib"

/**
This class governs the Heads Up Display (HUD) that players will see during the actual gameplay.  The GFx
library is needed as well as the Canvas for displaying images
**/
class UIHeadsUpDisplay extends GFxMoviePlayer{
        local GFxObject TimeImage;
        local GFxObject CompassImage;
        local GFxObject SpecialBarImage;
        local GFxObject CrosshairImage;

        /**
        Initializes and displays the HUD
        **/
        function drawHUD(){

                Canvas.ShowHUD =  true;
                Canvas.HUDCanvasScale(1);                //Adjusts the scale of the TV ratio
                assembleHUD();
                Super.drawHUD();
}
        /**
        The function assembleHUD places the images loaded into positions on a
        256 * 256 screen.  Precision can be increased if necessary.
        **/
        function assembleHUD(){
                getImages();
                Canvas.SetPosition(TimeImage,256,256);
                Canvas.SetPosition(CompassImage, 128,256);
                Canvas.SetPosition(SpecialBarImage, 0,256);
                Canvas.SetPosition(CrosshairImage,128,128);
        }

        /**
        This function obtains images from the package specified
        **/
        function getImages(){

                TimeImage = GetObject("TimeImage"); //Gets an object that references itself
                TimeImage.SetImage("source","img://MyPackage.Interface.HUD.TimeClock");
                CompassImage = GetObject("CompassImage");
                CompassImage.SetImage("source","img://MyPackage.Interface.HUD.Compass");
                SpecialBarImage = GetObject("SpecialImage");
                SpecialBarImage.SetImage("source","img://MyPackage.Interface.HUD.SpecialMeterBar");
                CrosshairImage = GetObject("CrosshairImage");
                CrosshairImage.SetImage("source","img://MyPackage.Interface.HUD.CrosshairDefault");
        }
}
```

5. Processing Layer: Game Subsystem

5.1 Game Info Module

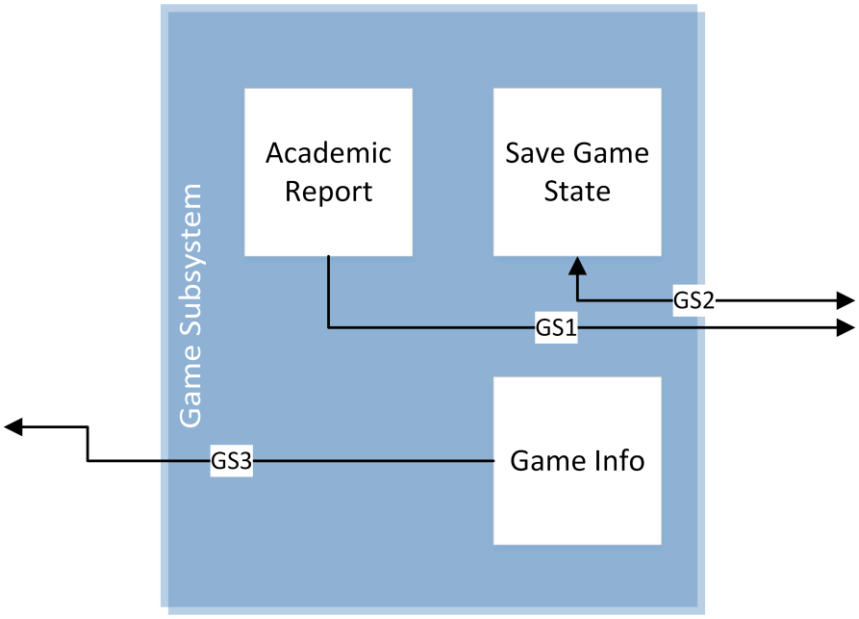

Figure 5.1 Game Info Module

5.1.1 Description

The Game Info module consists of a class named VRXGame. This class is instantiated when the game map is loaded. The Game Info module sets the Player Controller and Pawn to be used in the map and tells the UDK what class to use for the HUD and Pause Menu.

5.1.2 Interfaces

Data	Source Module	Destination Module	Description
GS3	Game Info	Game Events	Game state data
GS3	Game Events	Game Info	Updates game state data

Table 5.1 Game Info Module Interfaces

5.1.3 Physical Data Structures and Data File Descriptions

None

5.1.4 Processing

```
/**
This class initializes variables needed for the game to start
**/
class VRXGame extends UDKGame{

	struct InitSettings{
		var int setActors;
		var int time;
		var float progress;
		var bool playerSpawn;
		var Vector playerSpawnLocation;
		//Additional initializations are listed here
	}

	/**
	This function loads the game with proper values from the struct allowing the game to be played
	at a logical level
	**/
	function InitGame(InitSettings init){
		// initialize game
	}
}
```

5.2 Save Game State Module

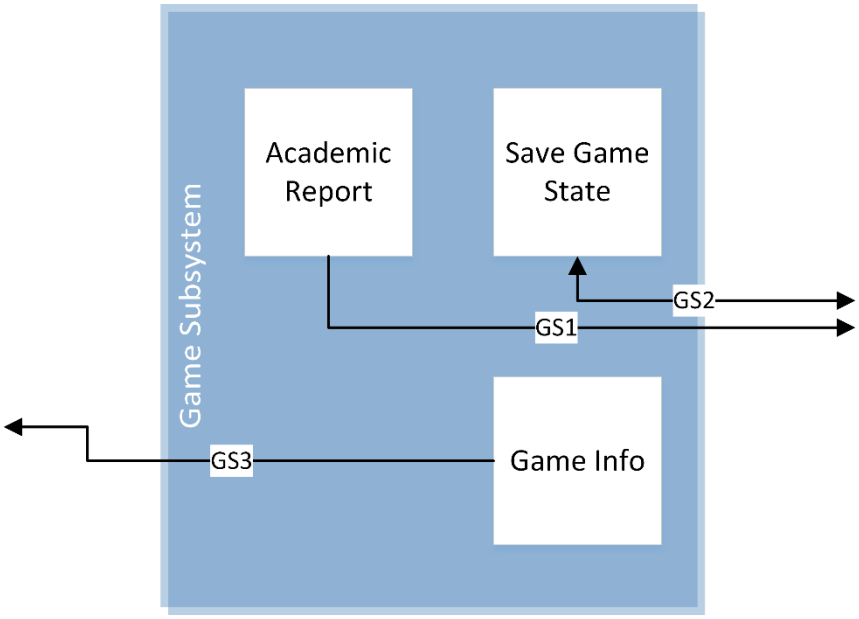

Figure 5.2 Save Game State Module

5.2.1 Description

The Save Game State module reads and writes the serialized actor and Kismet/Matinee scripts from and to the save file. It also requests serialized data from all actors and Kismet and Matinee scripts and sends the serialized data to be restored to all actors and scripts. This module uses the UDK BasicSaveObject and BasicLoadObject to write and read to the save files. These objects require a file name and the data to be saved.

5.2.2 Interfaces

Data	Source Module	Destination Module	Description
GM2	Save Game State	Pawn	Serialized data to be de-serialized by the Pawn.
GM2	Pawn	Save Game State	Serialized Pawn data to be saved to save file.
GM3	Save Game State	Actors	Serialized data to be de-serialized by the Actor.
GM3	Actors	Save Game State	Serialized Actor data to be saved to save file.
GM1	Pause Menu	Save Game State	Invokes the Save Game State object to begin serializing all the game state data and gives it the file name to save to.
GS2	Save Game State	File	Writes the serialized save data to a file on the hard drive.
GS2	File	Save Game State	Reads the serialized save data from a file on the hard drive.

Table 5.2 Save Game State Module Interfaces

5.2.3 Physical Data Structures and Data File Descriptions

None

5.2.4 Processing

```
//This function takes a filename as input, serializes the data, and stores it on the disk.

function SaveGameState(String fileName) {
        WorldData = SerializeWorldData();
        ActorData = SerializeActors();
        KismetData = SerializeKismet();
        MatineeData = SerializeMatinee();
}
//This function takes a filename, reverses the serialization process, and loads the saved file.
function LoadGameState(String fileName) {
        WorldData = UnSerializeWorldData();
        ActorData = UnSerializeActors();
        KismetData = UnSerializeKismet();
        MatineeData = UnSerializeMatinee();
}
```

5.3 Academic Report Module

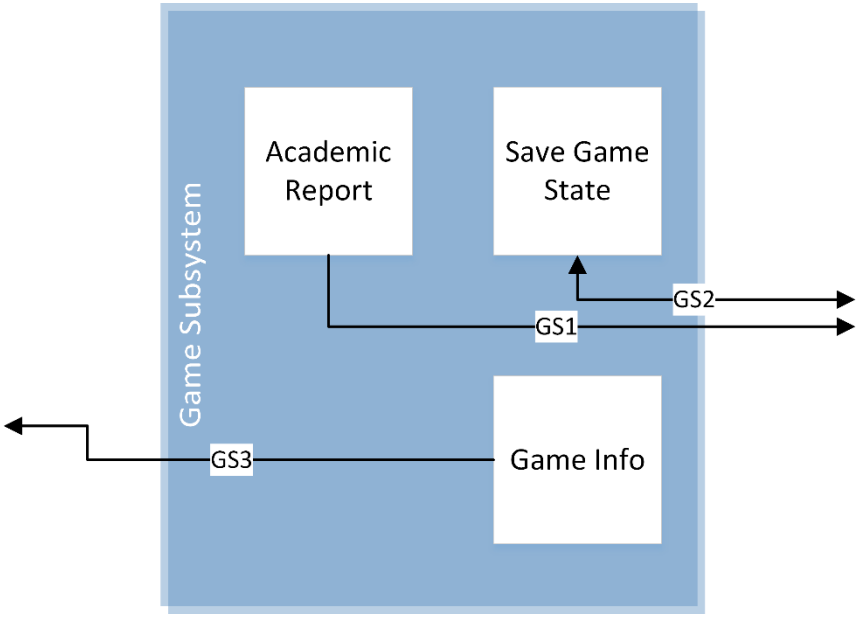

Figure 5.3 Academic Report Module

5.3.1 Description

The Academic Report module is responsible for writing the player's academic progress to a plain text and human readable file. This module uses the UDK FileWriter which provides an easy interface to write lines of text to a file.

5.3.2 Interfaces

Data	Source Module	Destination Module	Description
GS3	Game Info	Game Events	Game state data
GS3	Game Events	Game Info	Updates game state data
GS1	Academic Report	File	Writes the academic progress data to a file on the hard drive.

Table 5.3 Academic Report Module Interfaces

5.3.3 Physical Data Structures and Data File Descriptions

None

5.3.4 Processing

```
/**
This function gets passed parameters from a level and stores the statistics in a .ini file.
**/
function createAcademicReport(float timeTaken, int retries,
        int grade){

        var FWFileVar cTimeTaken = timeTaken;
        var FWFileVar cRetries = retries;
        var FWFileVar cGrade = grade;
        var FWFileType acadReport;

        CreateFile(acadReport, FWFT_Files, .txt );  //Creates a file with these statistics and stores
                                                    it in the Files folder.
}
```

6. Processing Layer: User Interface Maps

6.1 Load Screen Module

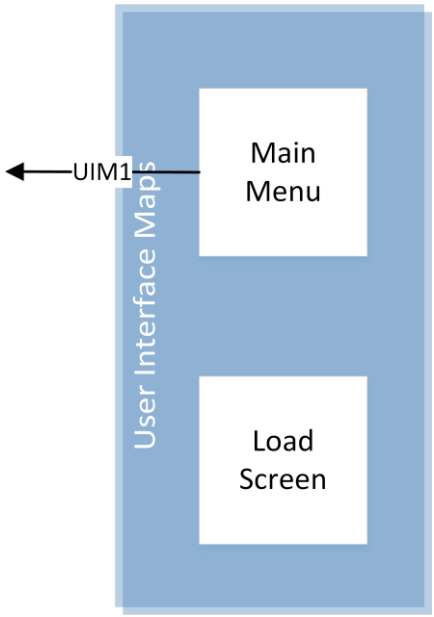

Figure 6.1 Load Screen Module

6.1.1 Description

The Load Screen module is the map that is loaded when transitioning from the main menu to a new game or a saved game. This module does not interact with any other module but it is automatically loaded by the UDK when another map is loaded. The map file to load is specified in UDKEngine.ini.

6.1.2 Interfaces

None.

6.1.3 Physical Data Structures and Data File Descriptions

The Load Screen Module implements Scaleform GFx, an external API that utilizes Flash in order to create graphical user interfaces.

6.1.4 Processing

```
#import “GFx.lib”

/**
When a user triggers a certain event, this class loads the respective movie with the loadScreen function.
**/
class LoadingScreenMovie extends GFxMoviePlayer{

	/**
	Loads the screen given the proper index of the screen located in the array screen
	**/
	event bool loadScreen(LoadMap screen, index i){

		switch(i){
			case(0):
			  LoadMapMovies(“C:\UDK\UDK-2013-09\UDKGame\Content\Screen0.bik”,i,screen);
			  break;
			case(1):
			  LoadMapMovies(“C:\UDK\UDK-2013-09\UDKGame\Content\Screen1.bik”,i,screen);
			  break;
			  //And so on
			default:
			   LoadMapMovies(“C:\UDK\UDK-2013-09\UDKGame\Content\Default.bik”);
		}
	}
}
```

6.2 Main Menu Module

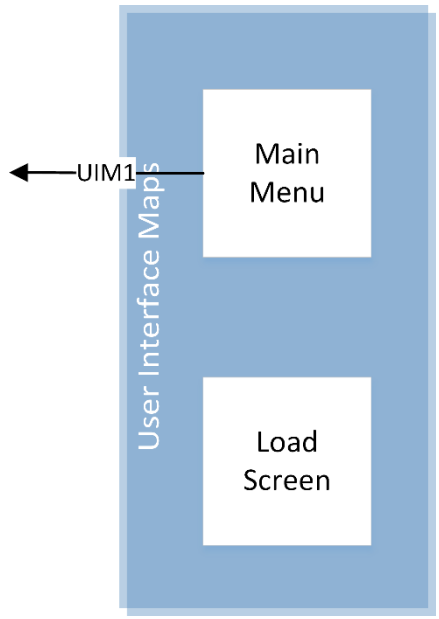

Figure 6.2 Main Menu Module

6.2.1 Description

The Main Menu module is the default map loaded by the UDK. This is specified in the UDKEngine.ini file.

6.2.2 Interfaces

Data	Source Module	Destination Module	Description
UIM1	Player Input	Main Menu	A floating point number between -1.0 and 1.0 for each joystick axis. A byte with either 0 or 1 for not pressed or pressed for each button.

Table 6.1 Main Menu Module Interfaces

6.2.3 Physical Data Structures and Data File Descriptions

The Main Menu Module implements Scaleform GFx, an external API that utilizes Flash in order to create graphical user interfaces.

6.2.4 Processing

```
#import "GFx.lib"

/**
This class contains code from the GFx library that implements the main menu.
**/
class UIMainMenu extends GFxMoviePlayer{
        var GFxCLIKWidget NewGameButton;
        var GFxCLIKWidget LoadGameButton;
        var GFxCLIKWidget OptionsButton;
        var GFxCLIKWidget QuitButton;

        function bool StartGame(){          //Starts the game
                Super.start();
                Advance();                  //Advances objects initialized (Ex: Load up Movie)
                return true;
        }

        /**
        This event features a switch statement that allows the program to
        execute an event when a certain button is pressed
        **/
        event bool UICommandWidget(WidgetName name, Widget wdgt){

            switch(name){
                  case 'newGameButton':
                    //Equivalent to creating a new button and adding it
                    NewGameButton = GFxCLIKWidget(wdgt);
                    //Performs the action if the button is pressed
                    NewGameButton.AddActionListener('CLIK_clik',CreateNewGame);
                    break;
                  case 'loadGameButton':
                    LoadGameButton = GFxCLIKWidget(wdgt);
                    LoadGameButton.AddActionListener('CLIK_clik',LoadSavedGame);
                    break;
                  case 'optionsButton':
                    OptionsButton = GFxCLIKWidget(wdgt);
                    OptionsButton.AddActionListener('CLIK_clik',LoadOptionMovie);
                    break;
                  case 'quitButton':
                    QuitButton = GFxCLIKWidget(wdgt);
                    QuitButton.AddActionListener('CLIK_clik',CloseMovie);
                    break;
                  default:
                    return false;
                }
        }
}
```

7. Storage Layer: Content Subsystem

The storage layer is a special layer because no other module directly interacts with it. The modules in the processing layer tell the UDK what assets from the content subsystem and the UDK manages loading and rendering the graphics and audio.

7.1 Textures Module

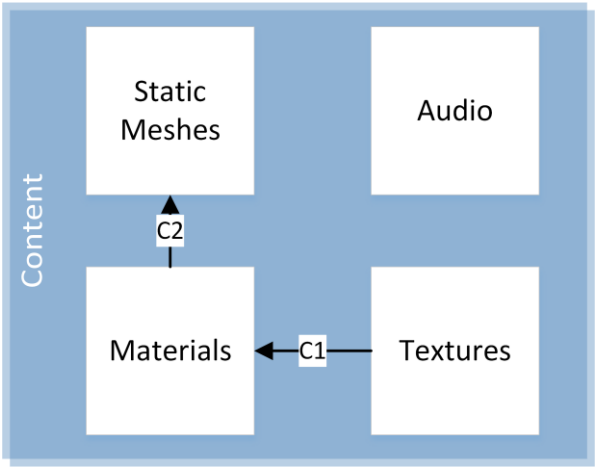

Figure 7.1 Textures Module

7.1.1 Description

Textures are 2D images that are stored and managed by the UDK Editor. Textures can be rendered directly to the screen using the UDK Canvas interface or applied to materials in the material editor of the UDK Editor. Most common image formats are supported.

7.2 Materials Module

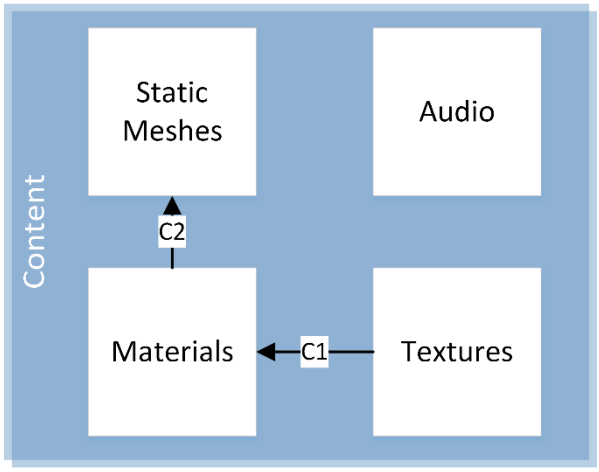

Figure 7.2 Materials Module

7.2.1 Description

Materials are made up of layers of textures and can be applied to static meshes. Materials are also stored in the UDK Editor and can be created and edited in the material editor.

7.3 Static Mesh Module

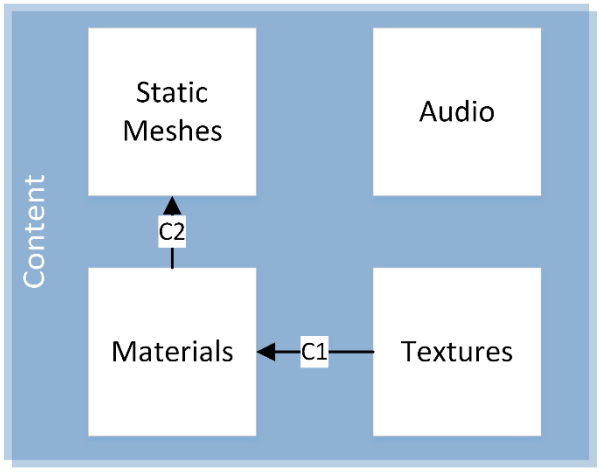

Figure 7.3 Static Mesh Module

7.3.1 Description

A Static Mesh is made up of a wireframe and a material. Static meshes can be applied to actors in the world to be rendered. Static meshes are also stored in the UDK Editor and managed and edited with the static mesh editor.

7.4 Audio Module

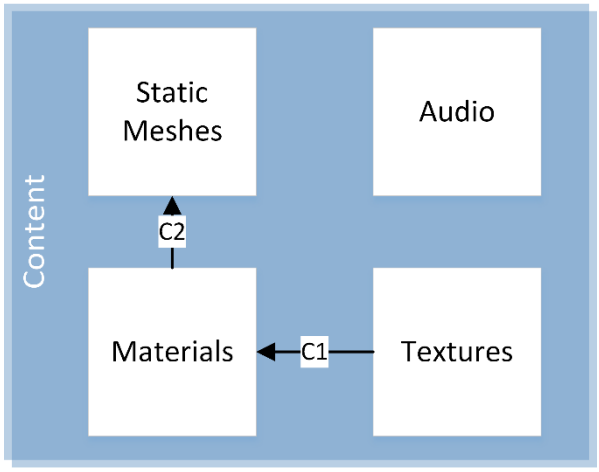

Figure 7.4 Audio Module

7.4.1 Description

The Audio files are also stored and managed by the UDK Editor. The UDK only supports uncompressed little endian 16 bit wave files.

8. Quality Assurance

8.1 Test Plans and Procedures

Team VR-X will perform comprehensive testing in order to verify that the Virtual Reality Xplorer meets all specified requirements detailed in the SRS, and depicted in the ADS. This comprehensive testing will consist of module testing for each individual module, component testing for any hardware that will be used, and integration testing for the overall functionality of the Virtual Reality Xplorer system.

8.2 Module Test

Since there is no Unit testing framework for testing Unreal script (UDK programming language), Team VRX will write small programs to test the functionality of each individual module.

- **Input Layer**
    - **Player Input**

      Team VRX will write small programs to check if the key binding specified in the "Default.ini file" are adequately read by the Player input subsystem.

    - **Player Controller**

      This subsystem will be tested by verifying if the buttons/keys mapped to a specific action in the "Default.ini file" are actually initiating its corresponding action.

- **Processing Layer**
    - **Pawn**

      During game play, specific actions like jump, picking up objects will be performed to ensure that the Pawn is receiving the appropriate commands from the Player Controller module.

    - **Pause Menu**
      During game play the Pause Menu module will be tested by hitting a button bound to pause the current game play, which will then display the Pause menu.

    - **Actors**

      Team VRX will write scripts to trigger an Actor (object) in a scene to perform a specific action. Based on this action, Team VRX will verify if the key binding

specified in Default.ini file are corresponding to the action performed by the Actor.

- **Game Events**

  This module contains all the script needed to control an object in order to perform an action. Small programs or scripts will be written in this module to make sure that Pawn and Actor are performing specific action based on the logic specified in the Kismet script.

- **HUD**

  The HUD provide most of the information and contents the player needs to interact with objects during the game, so this module will be tested by writing small programs to trigger the HUD during game play. The test must verify that the information being passed to the HUD are being displayed at the appropriate time.

- **Academic Report**

  Modular programs will be written to check if XML or text files can be written to within the Academic report class. Then after a game play, the Academic Files folder will be checked to see if text files contain the data from Academic Report class.

- **Save Game State**

  When a previous saved game is loaded, the Game State should be able to pull all data relating to that stage from the Scene folder. In order to test the Save Game state, the player's characters (Pawn) and actors (other stage objects) must start from the last position before the game was saved.

- **Game Info**

  Since this subsystem contains all variables related to the game state such as players positions, calculated velocity, and scores. Functions will be written to check if the values of the game variables are being updated based on the game play. In addition to testing the variable, boundary checks will also be conducted to avoid buffer overflow or infinite loops that could result in inaccurate behavior of objects. The subsystem will also be tested to ensure that the Pawn and Player Controller are given control to interact with objects in the scene.

- **Main Menu**

  This subsystem will be tested after the game is started, it shall display a menu containing the options to load game, save game.

- **Load Screen**
  This subsystem will be tested during the loading of a stage; it is generally supposed to appear during a loading phase. Adequate testing will be conducted to ensure that Kismet scripts are proper mapped in the pipeline to produce the Load Screen.

- **Storage Layer**

Most of the modules in the storage layer will be tested by verifying that the appropriate stored files are easily viewable in their respective folders. The Storage layer should contain all the Static meshes and Audio, materials and texture.

8.3 Component Testing

This section will define the procedures that will be performed to ensure adequate functionality of all the hardware required for the use of the system.

- **Oculus Rift**

  Whenever the Oculus Rift is being used, readings showing the angles of the player's head movement will be displayed in the configuration utility window. The configuration utility window must be checked to make sure corresponding angles are generated during head movements. The Oculus Rift will also be tested based on the player's immersive feel during game play.

- **Xbox Controller**

  A basic key pressed event on the Xbox controller must be recognized by the Xbox controller Driver which should be accessible from the Player Input module.

- **Headphones**

  Before the game is loaded or started, the headphones should be properly tested by making sure that the Operating System detects it. During game play, the headphone should be correctly outputting the appropriate sounds generated in the game environment.

- **Monitor**

  The monitor must be set to the display high quality image for the user, so the player's view when wearing the Oculus Rift must be similar to what is being displayed on the monitor excluding the 3D immersive depth.

8.4 Integration Testing

After rigorous testing of each individual module, a comprehensive test will be performed to ensure adequate compliance and compatibility of each subsystem to create a complete functional system. Complete inputs or actions will be supplied by team VR-X to ensure proper data flow from between subsystems.

8.5 Requirement Testing

After Integration testing have been perform, an additional requirement testing will be conducted to make sure every subsystem, and the system as a whole fulfills the requirements specified in the SRS.

9. Requirements Traceability Matrix

Input Layer
Processing Layer
Storage Layer

		Requirements									
		3.5	3.6	3.7	3.8	3.9	5.1	5.2	5.3	5.4	9.5
		Academic Report Generation	Entertaining	Interactive	Semantically Realistic	Configurable Controls	Frame Rate	Graphics Quality	Installation	Responsiveness	Safety
Modules	Player Input			X						X	
	Player Controller					X					
	Pause Menu			X							
	Pawn								X		
	Actors								X		
	HUD				X		X	X			X
	Game Events		X	X			X	X			
	Academic Report	X									
	Save Game State										
	Game Info								X		
	Main Menu			X							
	Load Screen						X				
	Static Meshes		X		X		X	X			
	Material		X		X		X	X			
	Textures		X		X		X	X			
	Audio		X		X						

Table 9.1 Requirements Traceability Matrix

9.1 Analysis:

The requirement traceability matrix shows that saving and loading do not play a large role in the creation of our product. While the feature is more important than the "Other Requirements" section listed in the SRS, this should not be a feature with high priority. Saving and loading will most likely be implemented in an "Autosave" fashion. Autosave refers to developers forcing the game to save when a certain event is triggered.

10. Acceptance Plan

10.1 Overview

This section will outline the requirements needed for the Virtual Reality Xplorer to be considered an acceptable product.

10.2 Installation and Packaging

The Virtual Reality Xplorer will be an accumulation of products composed of the Oculus Rift, an Xbox 360 Controller, a wireless headset, and a CD.

10.2.1 Installation CD

This CD will contain all data and executables that will allow our software to run. It will also include a user manual.

10.2.2 Oculus Rift

The Oculus Rift will come bundled with an HDMI/DVI cable for displaying visuals, a USB cable to interface with the CPU, and a cord for power supply.

10.2.3 Xbox 360 Controller

A standard Xbox 360 Wireless Controller with a wireless receiver will be included.

10.2.4 Wireless Headset

Quality sound headphones will be included.

10.3 Acceptance Testing

Testing for the Virtual Reality Xplorer will be executed in several ways. Testing of various components such as modules, layers, and subsystems will be conducted. Further details are included in the System Test Plan.

10.4 Acceptance Criteria

For the Virtual Reality Xplorer to be considered functional and acceptable, these criteria must be met:

Req. # (SRS)	Requirement Name	Description
3.1-3.4	Learning various scientific subjects.	The user will learn different scientific subjects such as force, environments, matter, refraction, etc.
3.5	Academic Report Generation	Teachers will track statistics from a previously attempted course such as time spent, number of retries, etc.
3.6	Entertaining	The game must be able to capture and hold the student's attention with elements such as story and environments.
3.7	Interactive	The environment allows the user to interface with the Menu/HUD and control components within the environment to an extent.
3.8	Semantically Realistic	The game must be able to provide a sense of realism.
3.9	Configurable Controls	The game must allow the user to customize their controls as the user sees fit.
5.1	Frame Rate	Frame rate of the Oculus Rift must be at minimum 60.
5.2	Graphics Quality	Resolution must be at minimum 1280 * 800 (for the Oculus Rift).
5.3	Installation	The package must allow for a user friendly installation of executables.
5.4	Responsiveness	The game must be responsive to the user's actions through the controller at minimum 1/60ths of a second.
9.5	Safety	The game must be created in such a way that the user's risk of nausea is reduced/eliminated.

Table 10.1 Acceptance Criteria