5-2012

Bringing Action-Adventure Gameplay to the iPad Using the Unreal Development Kit

Jonathan Lequire
Clemson University, jonathan.a.lequire@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses
Part of the Computer Sciences Commons

Recommended Citation
Lequire, Jonathan, "Bringing Action-Adventure Gameplay to the iPad Using the Unreal Development Kit" (2012). All Theses. 1305.
https://tigerprints.clemson.edu/all_theses/1305

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
BRINGING ACTION-ADVENTURE GAMEPLAY TO THE IPAD
USING THE UNREAL DEVELOPMENT KIT

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters
Digital Production Arts

by
Jonathan A. Lequire
May 2012

Accepted by:
Dr. Brian A. Malloy, Committee Chair
Dr. Jerry Tessendorf
Dr. Donald H. House
Abstract

Mobile game development is a fast growing industry that is primarily focused on casual games with short play sessions. In this paper, we describe an approach to the development of a game with a focus on lengthy, more traditional gameplay. We describe some techniques to exploit the UDK for this purpose, even though the UDK has been used, traditionally, for the development of shooter games. We have implemented these techniques, and we describe our port of the game to the Apple iPad. The result of our work is a playable game that runs on the iPad 2, and features interesting uses of both the UDK and mobile technology. These techniques can be used by other developers interested in exploiting the UDK for development of adventure games for mobile platforms.
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>List of Source Listings</td>
<td>viii</td>
</tr>
<tr>
<td>1 Introduction and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 What is a Game Engine?</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Unreal Engine History</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 What About Other Engines?</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Why Unreal Engine?</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Game Genres</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Anatomy of a Game</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Art Topics for Games</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1 8-bit and 16-bit</td>
<td>9</td>
</tr>
<tr>
<td>2.4.2 Sprites and Meshes</td>
<td>9</td>
</tr>
<tr>
<td>2.4.3 Particle Systems</td>
<td>10</td>
</tr>
<tr>
<td>2.4.4 Collision</td>
<td>10</td>
</tr>
<tr>
<td>2.4.5 Art Styles</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Programming Terms</td>
<td>11</td>
</tr>
<tr>
<td>3 Related Work</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Translating Popular 2D Game Play Into 3D</td>
<td>12</td>
</tr>
<tr>
<td>3.1.1 New Super Mario Bros.</td>
<td>12</td>
</tr>
<tr>
<td>3.1.2 Legend of Zelda Games for Nintendo DS</td>
<td>13</td>
</tr>
<tr>
<td>3.2 iOS Games Made With UDK</td>
<td>13</td>
</tr>
<tr>
<td>3.2.1 Dungeon Defenders</td>
<td>14</td>
</tr>
<tr>
<td>3.2.2 Sminis</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Action-Adventure RPGs on iOS</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1 Pocket RPG</td>
<td>15</td>
</tr>
<tr>
<td>3.3.2 Mage Gauntlet</td>
<td>15</td>
</tr>
<tr>
<td>3.4 What Makes Mistweaver Different?</td>
<td>15</td>
</tr>
<tr>
<td>4 Evolution of the Game Concept</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Learning the Unreal Development Kit</td>
<td>18</td>
</tr>
<tr>
<td>4.1.1 Building a First Person Shooter Map</td>
<td>19</td>
</tr>
</tbody>
</table>
Table of Contents (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.2 Using UnrealScript to Develop a New Weapon</td>
<td>20</td>
</tr>
<tr>
<td>4.1.3 Build a Simple iOS Game Using Kismet</td>
<td>21</td>
</tr>
<tr>
<td><strong>4.2 Game Design</strong></td>
<td></td>
</tr>
<tr>
<td>4.2.1 Identify Key Elements of Zelda Gameplay</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2 Making a New Experience from Zelda Gameplay Elements</td>
<td>25</td>
</tr>
<tr>
<td>4.2.3 The Story Outline</td>
<td>25</td>
</tr>
<tr>
<td>4.2.4 Designing the Main Character</td>
<td>27</td>
</tr>
<tr>
<td>4.2.5 The Magic of Mists</td>
<td>28</td>
</tr>
<tr>
<td>4.2.6 Picking a Subset for the Demo</td>
<td>29</td>
</tr>
<tr>
<td>4.2.7 Planning the Dungeon</td>
<td>29</td>
</tr>
<tr>
<td>4.2.8 Visual Style &amp; Mobile Considerations</td>
<td>30</td>
</tr>
<tr>
<td><strong>5 Setup and Use of the UDK</strong></td>
<td>31</td>
</tr>
<tr>
<td>5.1 Autodesk Maya to UDK Workflow</td>
<td>31</td>
</tr>
<tr>
<td>5.1.1 Setting the Correct Up Axis</td>
<td>31</td>
</tr>
<tr>
<td>5.1.2 Changing the Maya Grid</td>
<td>32</td>
</tr>
<tr>
<td>5.1.3 Collision Meshes</td>
<td>33</td>
</tr>
<tr>
<td>5.1.4 UV Layout Requirements &amp; Tips</td>
<td>34</td>
</tr>
<tr>
<td>5.2 UnrealScript Development Environments</td>
<td>36</td>
</tr>
<tr>
<td>5.3 Unreal Frontend Settings</td>
<td>36</td>
</tr>
<tr>
<td><strong>6 Game Construction and Port to iPad</strong></td>
<td>37</td>
</tr>
<tr>
<td>6.1 Creating a New Game Type in UDK</td>
<td>37</td>
</tr>
<tr>
<td>6.1.1 Creating A Custom Game Type</td>
<td>37</td>
</tr>
<tr>
<td>6.1.2 Config File Settings</td>
<td>38</td>
</tr>
<tr>
<td>6.1.3 Mobile UI</td>
<td>39</td>
</tr>
<tr>
<td>6.2 Building the Prologue Dungeon</td>
<td>39</td>
</tr>
<tr>
<td>6.2.1 Modular Design &amp; Construction</td>
<td>39</td>
</tr>
<tr>
<td>6.2.2 Defining a Camera Volume</td>
<td>40</td>
</tr>
<tr>
<td>6.2.3 Extending the Unreal Teleporter</td>
<td>42</td>
</tr>
<tr>
<td>6.2.4 Kismet Scripting For Gameplay Elements</td>
<td>42</td>
</tr>
<tr>
<td>6.2.5 Displaying Health in the HUD</td>
<td>46</td>
</tr>
<tr>
<td>6.3 Constructing the Main Character</td>
<td>47</td>
</tr>
<tr>
<td>6.3.1 Archetypes in Unreal</td>
<td>48</td>
</tr>
<tr>
<td>6.3.2 Skeletal Meshes</td>
<td>48</td>
</tr>
<tr>
<td>6.3.3 New Weapons</td>
<td>49</td>
</tr>
<tr>
<td>6.3.4 Custom Projectiles</td>
<td>50</td>
</tr>
<tr>
<td>6.3.5 Animations and the AnimTree</td>
<td>52</td>
</tr>
<tr>
<td>6.3.6 Casting a Spell</td>
<td>52</td>
</tr>
<tr>
<td>6.4 Developing Some Simple Enemies</td>
<td>53</td>
</tr>
<tr>
<td>6.4.1 Magical Turrets</td>
<td>53</td>
</tr>
<tr>
<td>6.4.2 Enemy Characters</td>
<td>54</td>
</tr>
<tr>
<td>6.4.3 Custom Kismet node for setup</td>
<td>55</td>
</tr>
</tbody>
</table>
Table of Contents (Continued)

6.5 Finishing Touches .............................................. 56  
   6.5.1 Custom Icon Set .......................................... 56  
   6.5.2 Changing the Startup movie .......................... 56  
   6.5.3 Creating the Main Menu ............................... 56  
   6.5.4 Installing on the iPad ................................. 57  

7 Realization of the Game Design ................................. 58  
   7.1 Entering Mistweaver: The Menu Screen ............... 58  
   7.2 The World of Mistweaver ................................. 59  
      7.2.1 PC Testing ........................................... 59  
      7.2.2 iPad ................................................ 60  
   7.3 Characters .................................................. 60  
   7.4 Items ....................................................... 61  
   7.5 Effects ..................................................... 62  
   7.6 Gameplay ................................................... 63  

8 Conclusion ................................................................ 65  

Appendices ............................................................... 66  
   A Mistweaver Source Code .................................... 67  
   B Additional Images ............................................. 95  

Bibliography ............................................................. 103
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 ZZT Menu</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Unreal Engine Rendering Comparison</td>
<td>5</td>
</tr>
<tr>
<td>2.3 UV Coordinate Example</td>
<td>10</td>
</tr>
<tr>
<td>3.1 New Super Mario Bros.</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Zelda on the Nintendo DS</td>
<td>13</td>
</tr>
<tr>
<td>3.3 iOS games made with UDK</td>
<td>14</td>
</tr>
<tr>
<td>3.4 Pocket RPG art style</td>
<td>15</td>
</tr>
<tr>
<td>3.5 Combat in Mage Gauntlet</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Results of Eat3D beginner tutorial</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Construction of the custom map</td>
<td>20</td>
</tr>
<tr>
<td>4.3 Eat3D iOS development tutorial, Cliff Attack</td>
<td>21</td>
</tr>
<tr>
<td>4.4 The ‘classic’ Zelda games</td>
<td>22</td>
</tr>
<tr>
<td>4.5 A boy and his sword</td>
<td>23</td>
</tr>
<tr>
<td>4.6 Examples of enemies</td>
<td>23</td>
</tr>
<tr>
<td>4.7 A small selection of dungeons</td>
<td>24</td>
</tr>
<tr>
<td>4.8 The Overworld of Hyrule</td>
<td>24</td>
</tr>
<tr>
<td>4.9 Ganon and Princess Zelda</td>
<td>25</td>
</tr>
<tr>
<td>4.10 The Temple of Mists</td>
<td>26</td>
</tr>
<tr>
<td>4.11 The Lightweaver and Darkweaver</td>
<td>27</td>
</tr>
<tr>
<td>4.12 Link through the years</td>
<td>28</td>
</tr>
<tr>
<td>4.13 The young mistweaver</td>
<td>28</td>
</tr>
<tr>
<td>4.14 Dungeon Planning Sketches</td>
<td>30</td>
</tr>
<tr>
<td>4.15 Infinity Blade II by Epic Games</td>
<td>30</td>
</tr>
<tr>
<td>5.1 Maya Axis Orientation Settings</td>
<td>32</td>
</tr>
<tr>
<td>5.2 Maya Grid Settings</td>
<td>32</td>
</tr>
<tr>
<td>5.3 A double door frame and the corresponding collision mesh</td>
<td>33</td>
</tr>
<tr>
<td>5.4 UDK Collision Examples</td>
<td>34</td>
</tr>
<tr>
<td>5.5 The proper naming setup in Maya for FBX export</td>
<td>34</td>
</tr>
<tr>
<td>5.6 UVs in the Static Mesh Editor</td>
<td>35</td>
</tr>
<tr>
<td>6.1 Mistweaver Wall Package</td>
<td>40</td>
</tr>
<tr>
<td>6.2 TDRPGCamVolumes in the editor</td>
<td>41</td>
</tr>
<tr>
<td>6.3 Kismet structure for handling keys</td>
<td>43</td>
</tr>
<tr>
<td>6.4 Draw Text properties</td>
<td>44</td>
</tr>
</tbody>
</table>
## List of Figures (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Custom font problem between previewer and iPad.</td>
<td>44</td>
</tr>
<tr>
<td>6.6</td>
<td>Kismet sequence for an instance of the locked door prefab.</td>
<td>45</td>
</tr>
<tr>
<td>6.7</td>
<td>Kismet sequence for an instance of the chest prefab.</td>
<td>45</td>
</tr>
<tr>
<td>6.8</td>
<td>Original Kismet sequence for health display.</td>
<td>47</td>
</tr>
<tr>
<td>6.9</td>
<td>Final Kismet sequence for health display and the new properties.</td>
<td>48</td>
</tr>
<tr>
<td>6.10</td>
<td>Skeletal mesh and underlying bone structure for the heroine.</td>
<td>49</td>
</tr>
<tr>
<td>6.11</td>
<td>Class diagram for Mistweaver weapon setup.</td>
<td>50</td>
</tr>
<tr>
<td>6.12</td>
<td>Magic Missile particle effect.</td>
<td>51</td>
</tr>
<tr>
<td>6.13</td>
<td>The AnimSet editor.</td>
<td>52</td>
</tr>
<tr>
<td>6.14</td>
<td>The AnimTree editor.</td>
<td>53</td>
</tr>
<tr>
<td>6.15</td>
<td>A simple Kismet node to configure MWBots</td>
<td>55</td>
</tr>
<tr>
<td>6.16</td>
<td>iOS Icons</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>The Mistweaver main menu.</td>
<td>58</td>
</tr>
<tr>
<td>7.2</td>
<td>Complete layout of the fourth floor.</td>
<td>59</td>
</tr>
<tr>
<td>7.3</td>
<td>Testing Mistweaver on the PC.</td>
<td>59</td>
</tr>
<tr>
<td>7.4</td>
<td>Mistweaver on the iPad.</td>
<td>60</td>
</tr>
<tr>
<td>7.5</td>
<td>Main character in action</td>
<td>61</td>
</tr>
<tr>
<td>7.6</td>
<td>Collecting keys.</td>
<td>61</td>
</tr>
<tr>
<td>7.7</td>
<td>Effects in Mistweaver</td>
<td>62</td>
</tr>
<tr>
<td>7.8</td>
<td>The trap in action</td>
<td>62</td>
</tr>
<tr>
<td>7.9</td>
<td>Floor traps.</td>
<td>63</td>
</tr>
<tr>
<td>7.10</td>
<td>Gameplay on the iPad.</td>
<td>64</td>
</tr>
<tr>
<td>B.1</td>
<td>Dungeon – Overview.</td>
<td>95</td>
</tr>
<tr>
<td>B.2</td>
<td>Dungeon – Floor 2.</td>
<td>96</td>
</tr>
<tr>
<td>B.3</td>
<td>Dungeon – Floor 3.</td>
<td>97</td>
</tr>
<tr>
<td>B.4</td>
<td>Dungeon – Floor 4.</td>
<td>98</td>
</tr>
<tr>
<td>B.5</td>
<td>Complete Dungeon – Floor 2.</td>
<td>99</td>
</tr>
<tr>
<td>B.6</td>
<td>Complete Dungeon – Floor 3.</td>
<td>100</td>
</tr>
<tr>
<td>B.7</td>
<td>Complete Dungeon – Floor 4.</td>
<td>101</td>
</tr>
<tr>
<td>B.8</td>
<td>Complete Dungeon – Other Rooms.</td>
<td>102</td>
</tr>
</tbody>
</table>
# List of Source Listings

<table>
<thead>
<tr>
<th>Source Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Custom Game Type Example</td>
<td>38</td>
</tr>
<tr>
<td>6.2 Mobile Input Configuration</td>
<td>39</td>
</tr>
<tr>
<td>6.3 TDRPGCamVolume Properties</td>
<td>41</td>
</tr>
<tr>
<td>6.4 State Programming Example</td>
<td>54</td>
</tr>
<tr>
<td>6.5 SeqAct_MWConfigureBot.uc</td>
<td>55</td>
</tr>
<tr>
<td>A.1 TDRPGame.uc</td>
<td>67</td>
</tr>
<tr>
<td>A.2 TDRPGPawn.uc</td>
<td>68</td>
</tr>
<tr>
<td>A.3 TDRPGPlayerController.uc</td>
<td>70</td>
</tr>
<tr>
<td>A.4 TDRPGPlayerCamera.uc</td>
<td>72</td>
</tr>
<tr>
<td>A.5 TDRPGCamVolume.uc</td>
<td>74</td>
</tr>
<tr>
<td>A.6 TDRPGTeleporter.uc</td>
<td>75</td>
</tr>
<tr>
<td>A.7 TDRPGInventoryManager.uc</td>
<td>78</td>
</tr>
<tr>
<td>A.8 MistweaverSpell.uc</td>
<td>79</td>
</tr>
<tr>
<td>A.9 MistweaverSpellMode.uc</td>
<td>82</td>
</tr>
<tr>
<td>A.10 MistweaverSpellMode_Projectile.uc</td>
<td>84</td>
</tr>
<tr>
<td>A.11 MistweaverProjectile.uc</td>
<td>85</td>
</tr>
<tr>
<td>A.12 MistweaverProjectile_Bouncer.uc</td>
<td>87</td>
</tr>
<tr>
<td>A.13 MWPawn_NPC.uc</td>
<td>89</td>
</tr>
<tr>
<td>A.14 MWBot.uc</td>
<td>90</td>
</tr>
<tr>
<td>A.15 SeqEvent_DrawPlayerHealth.uc</td>
<td>93</td>
</tr>
<tr>
<td>A.16 DefaultEngineUDK.ini Additions</td>
<td>94</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Motivation

In recent years, video games have seen tremendous increase in popularity and revenue. A recent report by the Entertainment Software Association (ESA) shows that the video game industry grew by over ten percent in a five year span, even though the American economy grew by only two percent in the same period [4]. In 2010 the video game industry generated more than twenty-five billion dollars in revenue [6]. In addition to their increases in revenue, video games are also widely enjoyed on a variety of platforms by a diverse audience [5]. A 2011 study by the ESA reports that seventy-two percent of American households play video games, forty-two percent of the players are women, and adult women represent a significantly large portion of the game-playing population [5].

In addition to their increase in revenue, mobile and hand-held devices have become increasingly popular for game play. For example, a 2011 report by the ESA states that fifty-five percent of gamers play games on a mobile device, such as an iPad or Android, or on hand-held devices, such as the Dual Screen (DS) device [5]. However, a game genre for mobile or hand-held devices that appears to be under represented is the action-adventure role-playing game in the style of The Legend of Zelda. This game features a top down view of a lone hero that must explore the world to find the power to defeat an evil sorcerer.

One reason for the under representation of the action-adventure genre on mobile and hand-held devices is the lack of information available to guide video game developers in the creation of these games using the popular game engines and, in particular, using the UDK. This information gap requires that many developers interested in building these types of games must suffer the start up overhead of developing a new engine for the action-adventure game that they wish to develop.

In this thesis, we investigate the development of an action-adventure game for a mobile device, the Apple iPad. We explore the use of the UDK for this purpose, even though tra-
ditionally, the UDK has been used for shooters developed for the PC and home consoles. We describe our design of an action-adventure game, Mistweaver, in the Zelda style. We provide details of our construction of a level for Mistweaver, including some helpful suggestions of UDK developers who wish to build a similar game for the iPad.

The remainder of this thesis is organized as follows. In the next chapter we review the basics of game development, followed by a brief history of the Unreal Engine platform, and an overview of game types and terms that we use in this thesis. Chapter 3 outlines some game projects that are similar to the one that we have developed, and Chapter 4 describes the process that we used to learn the UDK, and a review of the design of Mistweaver. Chapter 5 explains the steps required to set up the various software packages used in my research so that they can work with the UDK. Chapter 6 details the process used to build a level of our game, including examples of a broad range of UDK features that facilitated efficient development of Mistweaver. Chapter 7 reviews the result of this project, including images of the Mistweaver game running on an iPad 2. Finally, in Chapter 8, we draw some conclusions. The complete code for the Mistweaver project is included in Appendix A. Additional images, in higher resolution, can be found in Appendix B.
Chapter 2
Background

This chapter presents a brief overview of the concepts and terms that might more fully explain the approach that I have taken in this thesis. In the next section, I describe what I mean by a game engine, and then I apply this definition to some commonly used game engines including the Unreal Engine, Panda3D, Unity and the CryENGINE3. In Section 2.2, I describe the important game genres and place Mistweaver among these genre. In Section 2.3, I review the important features that are included in game engines and in Section 2.4 I review the art concepts that apply to my work. Finally, in Section 2.5 I review the programming terms and concepts that are used with game engines.

2.1 What is a Game Engine?

A game engine is a framework designed to aid in the creation and development of video games. The features offered by a game engine varies, but most include rendering support for either 2D or 3D graphics, collision detection and response, animation, and functionality to control gameplay.

In the early days of video games, each game was developed from first principles as a stand-alone entity to operate on the console hardware used at the time of development. Eventually, a few companies began to market their engines as third party software. It became cheaper for many studios to pay a company for the use of their software rather than develop a new proprietary solution.

2.1.1 Unreal Engine History

The concepts that make the Unreal Engine the powerful toolset that it is today began with one programmer: Epic Games founder and owner Tim Sweeney, during his time as a student at the University of Maryland. His first game was titled ZZT, which he completed in 1991.
ZZT featured an object-oriented language contained within the game to control gameplay without rewriting the base code which was a revolutionary idea. A menu screen from ZZT can be seen in Figure 2.1. [21]

Figure 2.1: ZZT Menu

“ZZT served as a conceptual blueprint for Unreal. A game engine with a high-productivity, what-you-see-is-what-you-get tools pipeline, bundled with a programming language aimed at simplifying gameplay logic.”

–Tim Sweeney [21]

In 1998, the first version of the Unreal Engine made its debut with a game simply titled: Unreal. The game was built in response to the increasing popularity of FPS games such as Doom[31], Quake[35], and Wolfenstein 3D[38]. Mike Thomsen, a writer for IGN.com states, “The game was arguably the best-looking shooter of its time, handling large, highly detailed indoor and outdoor areas with uncommon ease.” [21]

Unreal Engine 2 introduced the idea of using a single tool to develop games for multiple platforms to the Unreal methodology. Epic’s strategic focus had shifted to include the PS2, XBox, and GameCube. At the time, the PC and console markets were very separate entities. The difference in hardware power between PCs and consoles was the primary point of division. This broadening of focus came late in the life cycle of console hardware. Thus, Epic was forced to adapt the engine to outdated technology. [21]
Epic was able to begin development of Unreal Engine 3 before the next generation of console hardware was finalized. Mark Rein, vice president of Epic Games, boasted that he and Sweeney were able to convince Microsoft to double the RAM in the Xbox 360 based on images from *Gears of War* [33], illustrating the difference in quality that could be achieved. *Gears* was being developed as a graphical showpiece for the then upcoming Xbox 360. [21]

The Unreal Engine has always been on the cutting edge of graphics capabilities, a comparison of the graphical leaps between versions can be found in Figure 2.2. The character Malcolm has been recreated in each version of *Unreal Tournament*, providing a wonderful way to measure the power differences between versions of the Unreal Engine.

![Figure 2.2: Unreal Engine rendering comparison using the character Malcolm from *Unreal Tournament*.](image)

Unreal Engine 3, once an expensive software package only available to licensees, was released as part of the Unreal Development Kit in November 2009 [25]. This toolset allows anyone to learn, develop, and release games made with Unreal Engine 3. The toolset is still growing, and now includes support for both Apple iOS and Mac OS X applications, support for rendering over the web using Adobe Flash, as well as cutting edge features used in Epic’s latest games.
2.1.2 What About Other Engines?

Many game engines exist, and each has both strengths and weaknesses. Most engines are still proprietary software and are not made available for any form of external development. Three popular engines that are available to the general public for development are Panda3D, Unity, and CryENGINE.

Panda3D is designed as a framework for 3D rendering and game development for Python and C++ programs [19]. One of the most attractive and unique features of Panda3D is that it’s Open Source and free for any purpose, including commercial ventures. The Panda3D web site includes a plethora of documentation and examples to foster use for either commercial, educational, or recreational use.

Unity is one of the largest competitors for the UDK. It has many of the same features as the Unreal Engine and is distributed in both a free and pro version. One of the biggest strengths of Unity is that it, like UDK, provides a wide range of cross-platform options in the same package. [24]

CryEngine is developed by Crytek and has been used to make many visually impressive games, such as Crysis and Far Cry. A non-commercial version of CryENGINE 3 was recently released. Like the Unreal Engine this is a complete creation package that offers development options for PC, Xbox 360 and PS3. [2]

2.1.3 Why Unreal Engine?

The primary reason that I chose the Unreal Engine for this research was the plethora of features that it provides, and the potential for rapidly prototyping of a complete 3D video game. In the past I have experimented with level editors that were released with each versions of the Unreal Tournament games, but I was never able to see any projects through to a finished state, partly due to my inexperience and unfamiliarity with topics such 3D modeling and game programming logic. My education in computer science, graphics and modeling have filled these gaps and I now feel qualified to create something special with the Unreal Engine.
In addition, the UDK is an excellent choice because it is a well known and widely used toolset. Game studios, both large and small, are using the Unreal Engine to develop the games that people are currently playing on every major gaming platform, such as *Batman: Arkham City*, *Mass Effect 3*, and *Dungeon Defenders*. [27] The cross platform development is an essential feature to achieve my goal of developing a game for the iOS.

By contrast, Panda3D and CryENGINE are both excellent engines with broad feature sets. Neither of these engines provide facility for porting a game to a mobile device.

Unity is the only competitor reviewed in this paper that offers the cross platform features required for my research. The aspect of Unity that prohibited its use for my research is that basic mobile development, for iOS or Android, is a bonus feature in Unity and it must be purchased before it can be used in the editor. Support for iOS development is included for free in the UDK.

### 2.2 Game Genres

A game genre describes a style of gameplay, rather than the content of the story. Over the years many game subgenres have developed. The book *Level Up* [20], by Scott Rogers, features a comprehensive listing of game genres. The following list outlines only the genres relevant to the games discussed in this thesis.

- **Action** games rely on hand-eye coordination and quick reflexes to accomplish the goal of the game.

- **Adventure** games focus on item collection and inventory management, puzzle solving, and exploration.

- **Action-Adventure** is a combination of genres that incorporates aspects of the adventure genre into the fast paced nature of an action game.

- **First Person Shooter (FPS)** games primary gameplay element is firing projectiles at enemies from the character’s point of view.
• **Role-playing Games (RPG)** are based on pen and paper games like **Dungeons and Dragons**. Players usually choose a class and increase their powers through combat, exploration, and treasure collecting.

• **Tower Defense (TD)** games primary objective is to place towers strategically within a level to prevent waves of enemies from reaching something you want to protect, usually helpless villagers.

### 2.3 Anatomy of a Game

Regardless of the genre, most games are made of the same basic pieces.

The *Heads Up Display*, or simply *HUD*, is the term used to describe the on screen display of information to the player. Depending on the style and objective of the game this may include information like health, ammo, number of lives, timers, or available items.

The game *Camera* was introduced during the evolution of 3D games. The camera is an invisible entity that is located at the point from which the scene is rendered. Many early games struggled with providing a suitable control scheme or intelligent algorithm for moving the camera in relation to the player.

*Characters* in a game can refer to many different things. There are two main classifications: *Player Characters (PCs)*, which are characters that the player controls during the game, and *Non-Player Characters (NPCs)* – characters that are controlled by the game. NPCs include enemy characters, which are sometimes referred to as *bots* or *mobs* and any other character that a player can interact with, like a villager that sends the player on a quest.

Games are broken down into *levels*. The name for these levels can vary depending on the game. They are sometimes called *missions, dungeons, stages, or chapters*. Levels serve a couple purposes for a game; they provide a measure of progress for players and allow developers to break the game apart into easily manageable pieces for creation.

Most games include some *items*. These can be anything that a player might use to progress further in the game. Some common examples of items are coins, potions or medi-
cal supplies, ammunition, or temporary powerups.

The *cutscene* is a storytelling construct for games where gameplay stops to allow part of the story to be told in a more dramatic fashion.

### 2.4 Art Topics for Games

#### 2.4.1 8-bit and 16-bit

The terms 8-bit and 16-bit for game art are derived from the underlying hardware for which the game was created. When those systems were created the hardware could only store information in 8 or 16 bit blocks, resulting in a color palette of only 256 colors, or $2^8$, for 8-bit games and around 64,000 colors, $2^{16}$, for 16-bit games. Today’s hardware uses 24 or 32 bits to represent each color, resulting in over 16 million available colors.

#### 2.4.2 Sprites and Meshes

A *sprite* is a two dimensional image used to represent an element of the game. Sprites usually consist of multiple frames of animation that can be cycled through in a sequence to give the illusion of a character performing some action, running for instance. Sprites are still used in 3D engines, primarily on simple rectangular planes emitted by *particle systems*, which are discussed in Section 2.4.3.

A *mesh* is the three dimensional equivalent of a sprite. A mesh consists of a set of triangulated *faces*, usually referred to as *polygons*, that form the shape of an object or character in the game. Each triangular face is defined by a set of three *vertices*, which represent a single point in 3D space.

A 2D *texture* image is applied to these faces using *UV coordinates*. UV coordinates are a 2D translation of 3D coordinates such that each UV coordinate represents the location of the pixel in the texture for the associated 3D vertex. The colors between the vertices can then be applied across the face using various mathematical processes. A simple example of UV coordinates can be seen in Figure 2.3.
Modern game engines use *materials* or *shaders* instead of simple 2D textures. These materials make use of multiple textures, often referred to as *maps*, to represent more complex properties of the material. Some common maps are *specular*, *bump or normal*, and *emissive*. Specular maps define how shiny parts of the mesh should be. Bump or Normal maps provide a computationally cheap method to add small details to a surface without increasing the polygon count. Emissive maps define parts of a mesh that should be rendered as if those parts are emitting light.

### 2.4.3 Particle Systems

*Particle systems* are used to represent physical phenomena that cannot be represented using polygons. Common uses for particle systems in games include fire, smoke, explosions, falling leaves, and rain.

### 2.4.4 Collision

*Collision* is a cross-discipline problem in video games. Programmers must write code that detects and responds to collisions between components of the game. Artists must ensure that the produced assets are optimized for collision detection. Collision is included under art in this thesis because the detection and response code is supplied in the UDK, while new art assets must define the shape of the mesh for collision checking.


2.4.5 Art Styles

Like traditional art, many different styles are used in game development. In addition to the 16-bit style discussed above, which is currently popular on mobile devices, a few other styles are referenced in this thesis.

*Photo-realism* is the most common art style for big budget projects. This style strives to produce images that appear as real as possible. Many of the games produced with the Unreal Engine use this style.

*Hand painted* art styles use textures that have been painted by texture artists. This style usually appears slightly cartoon-like. However, many different results can be produced, depending on the amount of detail added to the textures and the effort put into hiding brush strokes.

2.5 Programming Terms

A powerful feature of the Unreal Engine is the use of *Binary Space Partitioning (BSP)*\cite{29} and *Constructive Solid Geometry (CSG)*\cite{30} to quickly build basic level geometry. BSP is a method used to recursively divide a space into convex shapes that can be represented using a tree. CSG is a method of modeling used in the Unreal editor that allows the creation of complex surfaces using simple geometric objects and common boolean operations such as union and difference.
Chapter 3

Related Work

This chapter explores current and past games that are related to Mistweaver. The following section describes various games and approaches that developers have taken to translating popular 2D games into 3D. Section 3.2 reviews games that have been developed for the iPad using the UDK, and Section 3.3 describes action-adventure games, that are similar to Mistweaver, that have been developed for the iPad using non-UDK frameworks. Section 3.4 concludes the chapter by comparing and contrasting Mistweaver with the games that I have reviewed in this Chapter.

3.1 Translating Popular 2D Game Play Into 3D

In this section, I review two Nintendo games that have made the transition from their 2D beginnings into a 3D rendered version of the original gameplay. I describe these games because Mistweaver is patterned after the popular 2D Zelda games and I, similar to some other developers, have developed Mistweaver using 3D techniques.

3.1.1 New Super Mario Bros.

The New Super Mario Bros. series currently has two games: the original game that was developed for the Nintendo Dual Screen (DS) [16], and a second game that was developed for the Nintendo Wii [17]. These games take Mario back to the original side-scrolling paradigm that helped make the Mario series so popular. The games follow the layout of the game Super Mario Bros. 3, which was developed for the Nintendo Entertainment System (NES). Super Mario Bros. 3 features an overhead map with nodes representing levels and other special stages. The art style of the new games closely matches the style used in Super Mario World for the Super Nintendo (SNES), but translated into three dimensional elements rather than two dimensional sprites, as illustrated in Figure 3.1.
3.1.2 Legend of Zelda Games for Nintendo DS

The Legend of Zelda series also features two games with a 2D to 3D translation, The Phantom Hourglass\[14\] and Spirit Tracks\[15\]; both games were released for the Nintendo DS. The games extend the original gameplay to exploit the touchscreen on the Nintendo DS and to add new twists to original game artifacts such as the boomerang. The art style is a blend of the cartoon-like characters from the Nintendo GameCube Zelda game, Wind Waker, and a more detailed texturing for the environments shown in Figure 3.2.

3.2 iOS Games Made With UDK

Support for iOS development using the UDK was announced in September 2010, and this feature became available to the public in the December 2010 build of the UDK. Subsequently, a number of games have been developed for the iOS using the UDK. Since I am also using the UDK for iOS development, I review two of these games in this section.
3.2.1 Dungeon Defenders

Dungeon Defenders was originally developed as a demo by Trendy Entertainment as a show piece for Epic Games to use in promoting the UDK. However, the demo was so popular that it has been transformed into a complete game that is available on most platforms supported by the Unreal Engine. Dungeon Defenders blends tower defense gameplay with RPG elements and features a hand painted art style, illustrated in Figure 3.3.

3.2.2 Sminis

Sminis is a recent addition to the iOS App Store. The goal of Sminis is to help small robots escape a factory where they have been enslaved. The player is given a single button for playing, but the game is nevertheless very challenging because the trigger must be carefully timed to start or stop the sminis in an effort to avoid obstacles in each level. A sample level from Sminis is shown in Figure 3.3.

Figure 3.3: Dungeon Defenders gameplay, left. A level from Sminis, right.

3.3 Action-Adventure RPGs on iOS

A number of games on the App Store fall into the action-adventure/RPG genre. Some of these games follow the Zelda pattern and a few incorporate an art style that is similar to Mistweaver.
3.3.1 Pocket RPG

Pocket RPG is a game that features an art style close to the style used for Mistweaver; Pocket RPG is shown in Figure 3.4. The gameplay focuses on battling waves of enemies as the player progresses through the levels, gaining both temporary and long term power increases and three different classes that change the game dramatically for each play through.

Figure 3.4: Pocket RPG art style.

3.3.2 Mage Gauntlet

Mage Gauntlet is the game that most closely resembles the 16bit art style of The Legend of Zelda: A Link to the Past, the Zelda game released on the SNES. Mage Gauntlet features a young girl who wants to understand why everything magical that she touches explodes. Eventually, a powerful wizard mentor arms her with an artifact that allows her to collect spells from within the world for one time use. Mage Gauntlet combat is illustrated in Figure 3.5

3.4 What Makes Mistweaver Different?

There are several features of Mistweaver that differentiate it from the other games that I have reviewed in this Chapter. One key difference that provided the inspiration for the
The Mistweaver project is my goal of adapting to, and making the game available on, the mobile market and, in particular, the iPad. As of November 2011, Nintendo executives remain firm in their resolve to only develop games for their own hardware, despite reports and claims by investors urging Nintendo to adapt to the changing market and work toward releasing games for mobile devices [8].

Reggie Fils-Aime, Nintendo of America president and COO, responded to the investor complaints in an interview, saying:

“First, we’re an entertainment company. We don’t make devices for the sake of making devices. We make our hardware in order to bring great entertainment experiences to life. Whether that’s the DS, the Wii or the 3DS—or even back to the NES and the SNES—that’s our philosophy. Therefore, the concept of having our core franchises on other systems really flies in the face of what we believe in[...]” [18]

Nintendo does offer digital versions of a select collection of their past titles on the Virtual Console for the Wii and 3DS, but this solution requires the consumer to first purchase the Nintendo hardware.

Mistweaver is not a Nintendo game, but if the concepts of Zelda gameplay are adapted properly to the touchscreen and successfully used to tell a story as deep and engaging as
that of the Zelda universe, then perhaps Nintendo will recognize the value of adapting their games for the mobile market.

The UDK games described above cover many genres and highlight the power of the Unreal Engine on mobile platforms. Mistweaver uses this same power as a platform to build a compelling art style and gameplay experience.

The current mobile game development market is very competitive, so very few resources exist beyond small samples from Epic Games and a few tutorials for prototyping a mobile game using Kismet, the node-based scripting language built into the UDK. Angry Rock Studios, the creators of Sminis, released a few developer journals during the course of development that give a nice overview of parts of their process [1]. The most detailed developer blog is from Trendy Entertainment’s Jeremy Stieglitz, in which he details the process of creating the Dungeon Defense demo in a month, this blog includes the code for the demo and a concise explanation for the features included in the demo without becoming tutorialized [3].

The Mistweaver project uses the approach of Dungeon Defense, providing code and explanations for how things are accomplished in the final demo so that others can make better games without having to figure out the basics for each new game type they choose to explore.

The graphical style of Mistweaver is the main difference from similar iOS games. A hand painted art style, applied to 3D models, is a very effective solution on mobile platforms. Pocket RPG is a great example of this.
Chapter 4

Evolution of the Game Concept

The Unreal Engine is a popular video game engine aimed primarily at the construction of first person shooters (FPS) [26]. One of the primary goals of this thesis is an investigation into the feasibility of using the Unreal Engine for construction of other video game genres.

This chapter begins with an overview of the process that we followed in learning the basics of the UDK editor, UnrealScript, and using the UDK for iOS development. Next, we provide an in depth deconstruction of The Legend of Zelda games that are the inspiration for the final product. Finally, an outline for a game, Mistweaver, is presented that provides a large number of the desired action adventure features to implement using the UDK.

4.1 Learning the Unreal Development Kit

The Unreal Engine became widely popular as an FPS engine, but its reputation for ease of game construction, extensive feature set, and its readily-available documentation, made it attractive for adaptation to other game genres. The ultimate realization of my endeavors is Mistweaver, but my first steps in creating Mistweaver began before the game design had been conceived. The process began with learning the basic feature set of the Unreal Development Kit (UDK) and exercising these features through tutorial based examples to determine the game types, other than FPS, that might be feasible.

The UDK is packaged with a handful of FPS demo maps and enough assets that the first steps most introductory tutorials follow is learning editor navigation, the content browser, and lighting setup through building a simple FPS map. Even though we had an alternative genre in mind, these tutorials seemed like a fitting beginning for my explorations.
4.1.1 Building a First Person Shooter Map

There are two tutorials that approach map construction in different ways, and we decided to work through these tutorials as an initial step. The first tutorial, from 3dBuzz, follows a more traditional approach in laying out a level using BSP (Binary Space Partitioning) and adding static meshes on top of those basic walls as decorations.[22] The second, from Eat3D takes a more modern approach and only uses BSP for a small portion of the level and builds the rest from the existing static meshes.[9]

We classify the 3dBuzz approach as traditional, based primarily on past experience with the Unreal Engine. The maps for the original Unreal Engine games were almost entirely composed of BSP surfaces, but advances in Unreal Engine 2 computer hardware utilization allowed instanced static meshes to play a much larger role in level design and construction. Unreal Engine 3, which is the base for UDK, exploits greater hardware utilization and most levels these days use very few, if any, BSP brushes for level geometry.

The final step of my introductory process, was to design and build a FPS map of my own. The purpose of this step was to become more familiar with the interface and included assets without the step by step help of a tutorial. We created a simple layout using BSP and then began building detail on top of the BSP structure. As we began the decoration process, we realized that the BSP geometry could have been built from the meshes used to add details in the modern style. Images from the construction of this level can be seen in

Figure 4.1: Results of Eat3D beginner tutorial.
4.1.2 Using UnrealScript to Develop a New Weapon

The next step was one of the most daunting—exploring and studying hundreds of thousands of lines of unfamiliar code and learning how to build new gameplay on top of the existing code base. Fortunately, Eat3D has a good tutorial for developing a simple first extension in the form of a new weapon type. For an experienced programmer, most of the tutorial is remedial, but basic programming concepts are covered in a clear manner for everyone to gain a basic knowledge of programming syntax. The final chapters of the tutorial offer some explanation of the class hierarchy in UnrealScript, which is a proprietary language inside of the Unreal Engine, while stepping through the process of creating the new weapon.

UnrealScript is an object-oriented programming language that is based on a unified type system, similar to C# and Java. Much of the syntax is similar to most modern programming languages, so that anyone familiar with any flavor of the C or Java languages should be able
to grasp the majority of UnrealScript. UnrealScript introduces additional syntactic features that are used to tie parts of the script to the UDK editor, described in Chapter 6.

4.1.3 Build a Simple iOS Game Using Kismet

The final pieces in learning the UDK center around developing content for mobile platforms. Shane Caudle, a Principal Artist at Epic Games, released a brief video tutorial that walks through the set up of a dual stick shooter game called Jazz Jackrabbit[7]. This game is created entirely in the UDK editor using Kismet, the Unreal Engine’s node-based scripting language, to set up and control all game logic.

Also worth mentioning here to keep things organized, though it was not available until much later in the process, is another video tutorial from Eat3d that goes through the process of building an airplane based game using the gyroscope on the iPad to control movement.[10] This video also includes a couple chapters on the real world example of porting an existing Unreal Engine 3 game to iOS, mostly centered around optimizing high end content to run on the less powerful hardware. This tutorial also focused on an entirely Kismet based setup.

4.2 Game Design

Many of the ideas that are incorporated into Mistweaver grew from my explorations into the UDK and the games that were developed using the UDK. Ideas for various game concepts
were maintained in a sketchbook for later review. Initially, these ideas described different ways to use pieces of the UDK to achieve various gameplay elements from existing games. Later, as the direction of the game that we wanted to build became more clear, the ideas began to coalesce around a central plot that would fit nicely into the Zelda gameplay archetype.

4.2.1 Identify Key Elements of Zelda Gameplay

An important first step in the construction of a Zelda type game is to explore the concepts that make a classic Zelda game memorable and engaging. To provide scope for the statements that follow, we consider the ‘classic’ Zelda games to be *The Legend of Zelda*[11], for the Nintendo Entertainment System, and *The Legend of Zelda: A Link to the Past*[13], for the Super Nintendo Entertainment System, illustrated in Figure 4.4. *Zelda II: The Adventure of Link*[12] deviates drastically from the formula of the other two games to the point of being mostly irrelevant for my design.

![Figure 4.4: The ‘classic’ Zelda games.](image)

At its heart, Zelda is an action/adventure game with the flavor and appeal of a role playing game (RPG). The player begins the game as a weak boy named Link, who quickly receives a sword and a warning that the world is a dangerous place, Figure 4.5. Subsequently, the direction the game takes is up to the player for the most part. Most modern Zelda games have deviated from this freedom to weave somewhat complex stories with a few more RPG elements included.

Attacking is accomplished primarily using the sword, which grows in power as the player progresses through the game, and defense is provided by a shield and nimble move-
ment. As the game progresses, the player gains access to various tools, some of which offer ranged attacks, such as the boomerang or bow, and some have more utilitarian uses, such as providing light in dark rooms.

Enemies are usually difficult to overcome or appear in packs large enough to make even the weakest enemy a nontrivial encounter. Colors are used to distinguish the difficulty of enemies of the same type as the game progresses, allowing for instant recognition of basic abilities while warning that the new version will be tougher to kill in some way. Some examples of large enemy packs and coloration can be found in Figure 4.6.

The majority of progressive gameplay takes place in dungeons, filled with enemies and puzzles. The design of each dungeon varies quite a bit from each dungeon before it which reduces the repetitive nature of the gameplay, Figure 4.7. The dungeons often feature a midpoint boss and always contain a very powerful final boss.

The dungeons are tied together with a large exterior map, called the Overworld, where most of the exploration gameplay takes place. The Overworlds of the classic Zelda games

Figure 4.6: Examples of enemies.
Figure 4.7: A small selection of dungeons.

are illustrated in Figure 4.8, the top image is the Overworld from The Legend of Zelda and the bottom two images show the Light and Dark Overworlds from A Link to the Past.

Figure 4.8: The Overworld of Hyrule.

The final two pieces that unify this formula are the primary antagonist and a princess.

The name of the primary antagonist has evolved over the years but is usually based on the name he wore in the classic Zelda games: Ganon. This character is the epitome of evil and wants to remake the world as he sees fit. And, of course, no Zelda game is complete without a princess in peril that the player must rescue, Princess Zelda. Figure 4.9 illustrates the A Link to the Past incarnations of these two characters.
4.2.2 Making a New Experience from Zelda Gameplay Elements

Most of the classic gameplay elements outlined in the previous section are incorporated into Mistweaver in some fashion. To deviate from the Zelda archetype Mistweaver features a female protagonist that is primarily a spellcaster. If she needs to swing her staff for defense things are going terribly wrong. The shield is a spell learned early on that can block a small amount of incoming damage before needing to be recast. Thus, the Mistweaver shield plays much differently from the Zelda shield in that the Mistweaver shield can work from any angle and should be able to absorb damage while the player is attacking. The staff is her primary weapon and throughout the game it can be replaced or visually upgraded using gem like additions.

4.2.3 The Story Outline

During the idea generation phase of this project the scope of the story quickly outpaced what one person might accomplish in a reasonable time period; thus, we present the full outline of these ideas in this chapter and later discuss which parts are to be included in the final project.

The prologue begins with a cinematic explaining the history of the Mistweavers leading to a setup for the initial location. The player is then presented with a choice to play as a Lightweaver or Darkweaver, each has a small selection of spells to give the flavor of each side of the Mist without overwhelming the player. These weavers are about to enter the sacred Temple of Mists, Figure 4.10 that has been lost for a thousand years to uncover the ancient secrets of their order.

The Lightweaver is an older man, mostly a stereotypical wizard without the pointy hat,
illustrated in Figure 4.11. The Darkweaver is an attractive woman of a slightly inhuman species, she is clearly an adult but has no identifiable age characteristics. They are both powerful Weavers, making the prologue dungeon quite trivial and often having spells that allow the player to skip portions of the dungeon, which is designed as a test for new initiates into the order.

Once the player finishes the prologue level with their chosen character, they encounter the other character for a confrontation. At this point our protagonist awakens from a dream of her father, the Lightweaver, battling for his life against the Darkweaver. Despite his earlier warnings of the dangers, she rushes into the Temple’s entryway to help him. This second playthrough of the prologue should highlight the vast difference in power that the novice daughter has in relation to the elder Weavers. The dungeon should now prove more challenging and also require a bit more exploration to get through all the obstacles.

The daughter arrives just as the mysterious Darkweaver, Figure 4.11, deals a fatal blow to her father and continues deeper into the dungeon in search of power. Her father charges her to take up the fight against the Darkweaver because the magic she seeks could destroy the world.

The rest of the story provides a build up of the conflict between Light and Dark. In the first act, the daughter tries to figure out what magic her father hoped to obtain in the temple and must find the keys to unlock the first wing of the main Temple that contains that magic. At the end of this wing she discovers that the Darkweaver has beaten her there and they

Figure 4.10: The Temple of Mists.
fight to a standstill, at which point the Darkweaver reveals that she is really our heroines mother and escapes in the stunned confusion that follows.

Act two begins with the daughter even more determined to stop her mother and find answers; however, another set of keys must be collected to unlock the next wing of the Temple and the daughter begins to discover that she might be a true Mistweaver, able to weave both aspects of the Mist. The climax of this act occurs in the second wing when the daughter finally defeats her mother, only to find out that her mother was being driven by a truly ancient evil. Before her mother dies, she reveals her search for the Ultimate Weaves, which can only be used by a true Mistweaver and might be the key to saving the world.

Act three follows the same dungeon pattern, with the final wing of the Temple requiring the Ultimate Weaves to open, which leads the daughter to the Staff of the Mistweaver, the final piece needed to assure victory over the Evil One. The story concludes with the daughter assaulting the Evil One’s stronghold and finally defeating him.

4.2.4 Designing the Main Character

One of the features that makes the Zelda protagonist, Link, work so well as a main character is his simple design. Figure 4.12 shows how little has changed in the past twenty five years for the young hero. A simple, unified color scheme with enough accessories to look decorated without distracting from the overall character. Most Zelda games feature at least
one costume change, which is usually a recolor from the classic green to another primary color. The only visual change for the hero comes as sword and shield upgrades throughout the game. This generic hero is important to the Zelda franchise because it makes it easy for players to become immersed in the story and identify themselves as the hero.

With the Zelda template in mind, we began a design of the daughter character in a fashion that embodied the features of Link. We kept her fairly young and generic, with some simple accessories such as a cloak and pouch to store her magical tools, Figure 4.13. The final touch included the addition of a snug vest to help define her feminine form without resorting to gratuitous skin exposure. This design allows for three clearly defined upgrade paths: cloak, dress, and staff, as well as the option to add some extraneous pieces of armor like the shoulder pieces depicted in the father and mother concepts.

4.2.5 The Magic of Mists

The magic of a Mistweaver is divided into two parts: the Dark and the Light. This system will, over time, act as an alignment system for the daughter character. Each time a strong
Dark or Light ability is used it can push the balance of magic within her towards one extreme or another, eventually weakening spells of the opposing alignment, something that Weavers with only one aspect of the magic do not have to worry about since they cannot control the other aspect.

The idea of weaving spells is borrowed from the The Wheel of Time series by Robert Jordan, who calls the process channeling. Channeling consists of five different powers: earth, air, fire, water, and spirit. Though some of the elemental feel of various powers might come into play for the animation of the weaves, in Mistweaver there are only two powers. Later in the game these powers might be combined to form the ultimate spells and different, more powerful, versions of basic spells.

4.2.6 Picking a Subset for the Demo

We chose to create the prologue of the story for the demo portion of my project and, as a result, the prologue has more details than the rest of the story. From the prologue, we chose to focus on the portion of the daughters trip through the dungeon since that should cover all the pieces of the gameplay without adding the complication of a wide variety of spells as well as making the dungeon more challenging. This reduces the character count to one, which is also a major time saver. The UDK is built with animation reuse in mind as long as the characters can use the same bone structure, however the modeling and setup for each character to be able to reuse those animations takes considerable time and effort.

The balance of magic system was not important enough this early in the game design to include a complete implementation. Only two spells were selected for implementation due to their similarity and usefulness in the final production, a fireball and a magic missile. This will force the player to rely on movement for defense.

4.2.7 Planning the Dungeon

Dungeon planning began with a few different thumbnail sketches of possible flows, Figure 4.14 shows the thumbnail that was produced. These were then refined at a higher resolution
and constantly reevaluated as construction took place. Most of the design of this dungeon is to include as much variety as possible while maintaining a manageable set of assets so construction can occur quickly and new ideas implemented primarily using existing assets.

4.2.8 Visual Style & Mobile Considerations

The Unreal Engine has the capability of bringing very impressive visuals to the iPad hardware, as demonstrated effectively by Epic Games in Infinity Blade and most recently Infinity Blade II, as seen in Figure 4.15. Most of the essential graphical features available for PC and console gaming now have counterparts in the mobile version of the Unreal Engine.

The style of Mistweaver needed to be simple to pay homage to its classic roots, while still being visually engaging. For this purpose I chose to use completely hand painted textures, an art style inspired by World of Warcraft[39] that many studios are currently moving toward.
Chapter 5

Setup and Use of the UDK

This chapter outlines the steps, and some possible missteps, that were used in developing Mistweaver using the UDK. The UDK can be daunting for a novice user and many of the concepts presented here can facilitate novice usage of the UDK, and provide a smoother transition into UDK game development. We first describe our setup and use of Autodesk Maya with the UDK, and then describe an approach to using meshes in the UDK. We then describe UV layout, and finish this chapter by detailing our use of UnrealScripting, the scripting facility in the UDK.

5.1 Autodesk Maya to UDK Workflow

The Autodesk Maya toolset provides a powerful platform for developing art assets for the Unreal Engine. This section covers the basics of setting up Maya for this purpose, this guide is intended for someone at an intermediate Maya knowledge level and only covers the settings specific to Unreal Engine production.

5.1.1 Setting the Correct Up Axis

The Unreal Engine coordinate system orientation is different from the one used by Maya. The most important difference is that the Z axis must be set as the up axis. An approach to adjusting this setting in Maya is illustrated in Figure 5.1. In the figure, the settings adjustment can be accomplished in Maya using the Preferences menu, through the sequence: Window Menu -> Settings/Preferences -> Preferences; select Settings on the left, and then select Z as the up axis under World Coordinate System.

The most recent versions of the UDK do not appear to need this setting to properly import models; axis orientation information is now stored as part of the export. However, there is the possibility that more advanced exports, such as rigs and animations, may not translate
properly; thus, we recommend setting Maya to match the Unreal Engine orientation when using all version of the UDK.

### 5.1.2 Changing the Maya Grid

The grid in the Unreal Editor is based on powers of two. Therefore, to take full advantage of the layout tools in the editor and not have tiny gaps that ruin the look of the level being created, we recommend setting the Maya grid to match this and to use snapping for any vertices, edges, or faces that need to align exactly with another mesh. The settings shown in Figure 5.2 produce a grid that will suffice for most modular mesh construction, these settings produce grid lines every 16 units. If a finer resolution is desired, the user may either decrease the *Grid Lines Every* setting to the next lower power of two, so the next setting would be 16, or increase the subdivision levels to a higher power of two. For *Mistweaver*, the modularity was simple enough that we were able to work at the settings shown in Figure 5.2.

Grid snapping in Maya can be accomplished two ways, the first is to enable it constantly by toggling the tool bar icon on and off. The second way is to hold the ‘x’ key before beginning to translate an object or component. Another useful hotkey is the ‘v’ key to snap to vertices if two points need to align exactly.
5.1.3 Collision Meshes

Collision checking can have a significant impact on the speed and playability of a video game. Thus, in order to guarantee smooth playability, every face of every object cannot be tested in real time. For most collision checks, a primitive based check is performed first, and then a per face check can be used if the primitive check found a collision. The UDK can create simple collision primitives, after a mesh is imported, that work well for simple shapes, but if the mesh has any concave areas or holes that characters need to pass through, a custom collision mesh must be defined. This collision mesh can consist of any number of primitive objects, but they must all be convex objects.

![Collision Mesh Example](image)

**Figure 5.3:** A double door frame and the corresponding collision mesh.

Collision meshes are detected upon import if and only if a set of naming conventions are followed. The FBX pipeline is slightly more robust in this area, but at the cost of being slightly more difficult to properly adhere to the naming conventions. For instance, the mesh and collision mesh shown in Figure 5.3 which has been combined in Maya and given the appropriate prefix and proper mesh name `UCX_DoorHole`, will export perfectly using the ActorX plugin. The same setup as exported using FBX is shown imported into the UDK in Figure 5.4. The figure also illustrates that the collision primitives have been combined into one convex collision mesh, which is optimal because more meshes take longer to check, but incorrect in this case because it is blocking the doorway.

To fix this problem, the collision meshes in Maya must be grouped together using the original name for the group, `UCX_DoorHole`, and as the beginning of the name for each
child piece of geometry, as shown in Figure 5.5. The collision now imports exactly like the ActorX exported version, allowing the player to pass through the opening of the door, Figure 5.4.

Figure 5.5: The proper naming setup in Maya for FBX export.

For most of this research, the ActorX plugin was used because it provided an easier setup for each mesh as discussed earlier. Once animation began, the FBX pipeline became more appealing. The fbx pipeline is covered in Section 6.3.

5.1.4 UV Layout Requirements & Tips

For StaticMeshes, the Unreal Engine typically requires two UV maps to properly render a mesh. The first UV map is for texture coordinates, by default, and the second map, which must have non-overlapping faces and be contained entirely in the zero to one range, is used for lighting storage. It is possible to adjust the default map indices for a completely non-overlapping layout for the texture, in this case only one map is required, but the engine
must be set to read map zero as the lighting UVs. This is not usually practical for most meshes because higher quality renders can be achieved by reusing parts of the texture sheet, even mirroring entire sections if another detail map or the geometry will hide the texture symmetry. Figure 5.6 shows the default Light Map Coordinate Index setting of one.

![Figure 5.6: UVs in the Static Mesh Editor.](image)

There are a few things to consider when laying out the lighting UVs. An initial approach may be to apply an automatic mapping, as a fast fix, which places every face of the mesh in the zero to one range and will ensure that nothing overlaps. However, this is the completely wrong approach and will result in many artifacts in how the final lighting appears in the game. Each UV island has the potential to contain a seam that will pick up a bad lighting value and render improperly. To avoid this it is best to minimize the number of UV islands and arrange them with at least a few pixels of space between islands that will be well lit, such as the exterior faces of a facade, and islands that will usually be dark, like the bottom of a crate. The other concern is the UV edge falling between pixels, this is another cause of lighting artifacts where an edge of the mesh has a small, black gradient even when fully lit. To help fix this, the snap UVs tool can be used specify a grid: right click on the icon and set the grid to the size of the Light Map Resolution for the mesh that is set in the Static Mesh Editor, as shown in Figure 5.6.
5.2 UnrealScript Development Environments

The UnrealScript language, like most programming languages, may be contained in plain
text files and does not require a special editor. There are many options for a game devel-
oper seeking assistance in navigating the code base. UnCodeX provides a class heirarchy
browser and can export the class trees to an html reference view. WOTgreal is a free IDE
that includes support for the UDK in its latest versions.

Perhaps the most powerful and well known IDE is Visual Studio, either 2008 or 2010
can be used, and an addon called nFringe from Pixel Mine Games that provides Intellisense
code completion as well as class browsing functionality. Properly installed, this addon can
also launch the UnrealScript compiler and then launch a copy of the newly compiled game
to test changes. nFringe is free for non-commercial projects so this was the development
environment used for most of this research. User Grash on the Epic Games community
forums provides en excellent guide for setting up Visual Studio 2010 with nFringe. [28]

One aspect of UnrealScript development that proved disconcerting was the inability
to compile scripts with the UDK editor open. Thus, when an error condition arises, the
compiler is unable to save the scripts and the developer should make sure the editor is
closed.

5.3 Unreal Frontend Settings

The Unreal Frontend is a fairly simple application used to compile scripts, cook assets, and
package the game. Cooking is a term used for reformatting the assets as used in the editor
into static copies in a format best read by the engine at run time. This application is also
used to deploy test versions of the game to the iPad.
Chapter 6

Game Construction and Port to iPad

In this chapter, the construction of the Mistweaver demo is broken down into several main topics which are explained in detail over many smaller sections.

6.1 Creating a New Game Type in UDK

The Unreal Engine game logic is controlled through the strategic use of game types. This concept is a powerful feature for games like Unreal Tournament, allowing the same map design to be reused for different game types. Object technology and inheritance, as used in the Unreal Tournament system in UnrealScript, facilitates this reuse where a base class, UTGame, can define all the settings and actions that a FPS game needs, and then a developer need only make changes to the UDeathmatch and UTCTFGame to produce very different game experiences.

6.1.1 Creating A Custom Game Type

At the core of a custom game type is a class that is a descendant of the GameInfo class. For Mistweaver, the TDRPGame type was created inheriting from SimpleGame. This inheritance gives a couple useful routines from FrameworkGame, which defines mobile input types, and the SimpleGame function to handle Unreal Tournament naming conventions for testing various existing maps with the new game type. For a completely custom game this functionality should not be needed.

The DefaultProperties block of an UnrealScript class provides a place to declare the default settings for an actor. For Mistweaver, it was necessary to override the DefaultPawnClass and PlayerControllerClass attributes with new extensions designed to load the new character model and handle the mobile input from a top-down point of view.
The DefaultProperties block is the proper location to change these classes for a new game type, as seen in Source Listing 6.1. This Source Listing is the entire extension needed for a simple game type; some additional functionality may be added for inventory management, which can be seen in Source Listing A.1.

Source Listing 6.1: Custom Game Type Example

```plaintext
class TDRPGame extends SimpleGame;
DefaultProperties
{
PlayerControllerClass=class’TDRPGame.TDRPGPlayerController’
DefaultPawnClass=class’TDRPGame.TDRPGPawn’
bRestartLevel=false
bWaitingToStartMatch=true
bDelayedStart=false
}
```

6.1.2 Config File Settings

The configuration files are one of the most difficult things to manage for a UDK game. Making completely different games often requires a separate installation of the UDK to keep the configurations from conflicting. However, simpler games and most scripting tests can coexist within the same installation.

The first, and only required, change any game or project using custom UnrealScript classes must make is to modify the configuration files to include the new source package in the compilation routines. This is accomplished in the DefaultEngine.ini file by adding the line `+EditPackages=<Folder_Name>` under the `[UnrealEd.EditorEngine]` heading. The order in the configuration file defines the compilation order, so it is always best to add new packages at the end of the list if they may reference anything from other packages. The `<Folder_Name>` is the subfolder of the \Development\src directory that is used for custom code.
6.1.3 Mobile UI

The most robust method for defining mobile input for a new game type is through another config file, `DefaultGameUDK.ini`. This is only used for the joystick in Mistweaver at this time, but as features are finalized it would make sense to expand this definition to include the main action buttons and any menu screens that would be needed.

To define a new mobile input, go to the end of the `DefaultGameUDK.ini` file and add the following section declaration: `[Zone_Name MobileInputZone]`. Then add the lines to define the custom properties of the zone such as input keys, placement, and color. These properties can be any variable defined in the MobileInputZone class. For the full example of the Mistweaver joystick, see Source Listing A.16. The final piece to make this work is to define a mobile input configuration for the TDRPGame type as seen in Source Listing 6.2, this should be included after all the required mobile input definitions.

Source Listing 6.2: Mobile Input Configuration

```plaintext
[TDRPGame.TDRPGame]
+RequiredMobileInputConfigs=(GroupName="GamePlayInputs",
  RequireZoneNames="("TDRPG_MoveZone")
```

6.2 Building the Prologue Dungeon

The world of Mistweaver currently consists of a single dungeon, as outlined in Section 4.2.7. This dungeon provides many opportunities for exploring different methods of construction in the UDK. Some examples for each method are included in this section.

6.2.1 Modular Design & Construction

Asset reusability is a key element in the Unreal Engine design, if every object on the screen in a typical Unreal game was unique, memory performance would suffer. If an asset is reusable, the model data only needs to be loaded once and then simple operations like translation and rotation can be quickly applied to each instance of the mesh.
With this in mind, the Zelda dungeon tile set translates very well into a set of three

dimensional assets. An asset size of 256 units was chosen to be the base for one cell from
the dungeon plan. It is important for this number to be a power of two for grid alignment in
the UDK and 256 gave a nice scale for the desired plan. The complete set of wall meshes
used for Mistweaver can be seen in Figure 6.1. The other useful attribute gained by using
a power of two for the base mesh size is the ability to scale the meshes predictably to still
align on the UDK grid.

Figure 6.1: Mistweaver Wall Package.

6.2.2 Defining a Camera Volume

Volumes are used for many different purposes in the Unreal Engine. A volume is a section
of space in the game world marked with a BSP brush that is invisible in the final game. The
most common type of volume is the Blocking Volume which prevents players from entering
the area contained in the volume. Other functions of volumes include changing the physics
of an area, instantly or slowly killing a player who enters, and defining a section of space for the lighting system to process in high detail.

For Mistweaver it was necessary to extend the BlockingVolume class to prevent the new camera system from going behind the walls of the dungeon while allowing the player to move in the camera’s dead zone. The new TDRPGCamVolume adds four new properties to the basic BlockingVolume class, shown in Source Listing 6.3.

Source Listing 6.3: TDRPGCamVolume Properties

```java
class TDRPGCamVolume extends BlockingVolume;
enum LockDirection { LOCK_NEGATIVE, LOCK_NONE, LOCK_POSITIVE };
var (TDRPG) LockDirection LockXDirection;
var (TDRPG) LockDirection LockYDirection;
var (TDRPG) bool LockZ;
var (TDRPG) bool LockToOrigin;
```

These values of these properties are read by the camera class to determine the final location of the camera in relation to the player. See Figure 6.2 for an example of a room in the editor with the new volumes in place. Notice the ability to define multiple levels as the character ascends stairs, or to force the camera closer to the character to give a better sense of moving in the depth of the scene.

Figure 6.2: The TDRPGCamVolume (pink wireframe) being used to define proper camera placement in the dungeon.

The camera then calls the built in trace routine for each required update, if it strikes a TDRPGCamVolume it locks the camera in the indicated axis, both axes for inside corners,
while still allowing movement in the other axis so a player can move down the length of the wall without moving out of sight. The complete code for the `TDRPPlayerCamera` can be found in Source Listing A.4.

### 6.2.3 Extending the Unreal Teleporter

The teleporter system in the UDK provided a quick method for moving between the rooms of the dungeon. A problem was quickly identified with this basic implementation, once the game reached a playable state and it was sufficiently playtested. After most teleports, the player became disoriented because the character was hidden inside of the door frame. A naive solution might entail using twice the number of teleporters: the original, which might serve as the exit teleporter so the character looks like she had walked out of the room, and a new teleporter, serving as the entrance point in the new room. This setup was tested and did not affect performance overall, but required twice the setup work for the same result of getting the player from point A to point B. A more flexible solution is to simply extend the teleporter class to include an offset value that places the transported actor at a configurable distance in front of the teleporter.

The second issue that teleportation might raise is that the camera can slide behind the character. This occurs because the position of the camera is interpolated slightly to give a smooth start and stop effect. To address this issue, the new teleporter contains additional attributes that permit the teleporter to delay teleportation long enough to play a “fade to black” sequence, then fade the image back in once teleportation is complete. This provides a very clean transition between rooms to complement the new offset feature. Source code for the `TDRPTeleporter` can be found in Source Listing A.6.

### 6.2.4 Kismet Scripting For Gameplay Elements

Kismet is a powerful node-based scripting environment inside of the UDK. Using the nodes to control gameplay is very straightforward, and after following a few examples figuring out new operations is usually quite intuitive. Many of the following examples would benefit
from being coded in UnrealScript for reusability in a complete game, but for demonstration purposes they were done using only Kismet and tools found in the editor.

**Keys and Money**

Keys were the first thing that must be implemented, once the base for the level is in place. Locked doors control the flow through the dungeon to some degree. To implement the acquisition and use of keys, two *remote events* are needed: one for collecting a key and another for using a key, as shown in Figure 6.3. A remote event is a subsection of the Kismet script that can be called from other sections of the Kismet script. The new events add or subtract one respectively from the variable named: *Keys*. Naming variables provides a way to make Kismet code less confusing. Variables in Kismet could be considered in global scope, one variable can be connected to any number of action nodes. Notice how three instances of the *Keys* variable are present in the figure. Without variable naming, each of those five connections made to the instances of *Keys* would be drawn to a single node, resulting in convoluted node structure with connections crossing everywhere.

![Figure 6.3: Kismet structure for handling keys.](image)

The third section of Figure 6.3 shows the simple node structure for drawing the portion of the HUD dedicated to keys. The settings for location, font, and icon are stored inside
of the Draw Text and Draw Image events, Figure 6.4. This figure also illustrates a nice feature of UDK property windows, the attributes that have been changed from the default value are shown in bold. The Authored Global Scale is used for scaling the HUD elements to work on different resolution devices and a Max Trigger Count of zero means that this event can be used any number of times.

![Draw Text properties](image)

Figure 6.4: Draw Text properties.

A problem that might confront a developer when porting a game to the iPad is that custom fonts may not display on the device. To address this issue, the custom font must be added to the EngineFonts package rather than a new package. Once these fonts are imported to the EngineFonts, the font will render correctly on the device. Figure 6.5 shows an example of a font that is not installed properly. In the mobile previewer (top image) the font displays correctly, however once installed on an iPad the font reverts to the default.

![Custom font problem between previewer and iPad](image)

Figure 6.5: Custom font problem between previewer and iPad.

Door setup

The doors use a simple proximity trigger, with locked doors checking the value of the Keys variable described in Section 6.2.4. The sequence begins with the conditional key check; if the player has one or more keys, the unlock and door opening sound is triggered immediately, 0.5 seconds later the UseKey remote event is called to decrement the Keys
variable, and finally the Matinee sequence of the door opens one second after the sequence is triggered to match up with the sound track. This sequence is shown in Figure 6.6.

![Figure 6.6: Kismet sequence for an instance of the locked door prefab.](image)

**Chest setup**

A chest in *Mistweaver* consists of five pieces in the editor and seven kismet nodes, as shown in Figure 6.7. This group of chest items can be handled together using an UDK feature called *Prefabs*, which permits a collection of objects and any related Kismet scripts to be grouped together as an asset. This prefab can then be placed any number of times in a level without the need to set up every object and node for each instance.

![Figure 6.7: Kismet sequence for an instance of the chest prefab.](image)

An important part of mobile game design is an intuitive and engaging use of the touch-
screen interface. For example, the chest in Mistweaver began as an object that needed to be
touched with the player’s pawn to open. However, the UDK provides a facility for handling
touch input on objects using the Mobile Object Picker node, shown in Figure 6.7. The
Mobile Object Picker node allows any object to become a target for touch input. Using
the object picker, it is possible to touch the object as long as it can be seen on the screen,
whether the character is within reach or not. Then, the proximity trigger can be used to
toggle the enabled state of the Mobile Object Picker node, once the pawn is within range
of the chest.

**It’s a trap!**

No self-respecting dungeon is complete without some traps, just ask Indiana Jones. The trap
in the second wing of the Mistweaver dungeon makes use of the Fracture Tool found in the
UDK’s Static Mesh Viewer. With this tool, it is possible to slice a mesh into chunks that
can be broken apart dynamically. For the trap to break a Radial Impulse Actor is placed
in the room with the meshes with a radius adjusted to cover the four pieces of fractured
floor. This actor applies a force throughout that radius when active and has an option to
cause fractured meshes to explode. The impulse actor is toggled on as part of the Kismet
sequence. In addition to breaking the floor, the trap system plays ten seconds of fire particles
on top of the two trapped chests and damages the player. Another useful node is featured in
this setup, and also used for the crossfade transitions, the Toggle Input node allows Kismet
to prevent the player from moving or turning.

### 6.2.5 Displaying Health in the HUD

The health system represents an interesting example of when Kismet scripting is more cum-
bersonse than actually writing Unrealscript code. For example, one approach to designing
the health system might be to use a Kismet script to ‘hard-code’ heart containers, check the
current health against multiple integers and set the active inputs of two Draw Image events
for each heart. A setup of this type is illustrated in Figure 6.8, which shows three hearts
worth of health. However, the classic Zelda games allow the character to gather additional hearts, and the player may accrue up to twenty hearts. Using a Kismet script in this manner would require almost seven times the current setup, and because this is purely a kismet solution this setup would need to be duplicated into each level.

A better solution entails writing code to replace Kismet for a cleaner setup, and overall simplification of the solution. It is important to note here that everything in the Unreal Engine derives from the base Object class, including all the Kismet nodes used in the editor. A quick search of the class trees reveals the code for a Draw Image node contained in the SeqEvent_HudRenderImage class. A better Health system can be implemented by introducing a few additional variables to control the new specialized node SeqEvent_DrawPlayerHealth, which extends from the SeqEvent_HudRenderImage class. This solution was incorporated into Mistweaver, and is shown in Figure 6.9; the complete code for the new node can be found in Code Listing A.15.

### 6.3 Constructing the Main Character

Character construction is a very broad and complex topic. Most of the subtopics will be touched upon in this section, but the information discovered by the Mistweaver project is a fairly shallow overview of these topics.

![Figure 6.8: Original Kismet sequence for health display.](image)
6.3.1 Archetypes in Unreal

Archetypes are a powerful tool for rapid game development. An archetype stores a complete collection of settings for the actor that is used to declare the archetype. This information allows a properly coded class to be adjusted directly in the editor. Removing the need to close the editor and recompile the source to see the difference between changing attributes, like a GroundSpeed setting of 220.0 instead of 240.0. It is possible to set up an entire game this way, from the player pawn to every pickup and weapon. For Mistweaver, archetypes are used for weapons and ammunition to demonstrate the technique, while leaving room for the demonstration of other development methods using other classes.

6.3.2 Skeletal Meshes

A skeletal mesh is any mesh that is bound to a bone structure in the 3D modeling package and then imported attached to that bone structure. This allows animations to be imported using that bone structure to affect all meshes that use an identical naming convention, but not necessarily all of the bones of the base skeleton. Characters and weapons are the primary uses of skeletal meshes. The skeletal mesh and bone structure used for testing the heroine character can be seen in Figure 6.10.
6.3.3 New Weapons

The bones of the weapon mesh, in addition to allowing for firing animations, provide a reference point to attach weapon effects through the use of sockets in UDK. A socket is simply a point with a set distance and orientation relative to a bone. Most weapons require one socket, typically named the MuzzleFlashSocket due to the gun based nature of most games produced with the Unreal Engine, that indicate the location to play a particle effect every time the weapon fires. An additional socket is supported by the Mistweaver class that indicates the location to spawn the spell projectile.

SpellModes define the type of spell that is cast from each firing mode available on the weapon. These spell modes consist of two classes. The MistweaverSpellMode base class that handles common functionality like playing the firing effect particle system and defining the common attributes of all spell modes. This class is extended for each actual fire mode, currently only MistweaverSpellMode_Projectile is used because the two demo spells are projectile based: fire balls and magic missiles. This projectile spell mode handles the logic for picking the start and aim point of the projectile flight path. Source Listings A.9 and A.10 contain the code for the MistweaverSpellMode and
MistweaverSpellMode_Projectile classes.

To make the weapons function correctly, an extension of InventoryManager is needed to handle the weapon archetypes. This will later be used to keep track of the weapons available to a pawn as well as managing the fire sequence for the weapon, both functionalities are inherited and will not be covered here.

### 6.3.4 Custom Projectiles

Projectiles are required for any spell mode that has a visible effect once used. For a gun-based game, weapons like a pistol do not use projectiles because the bullets are too small and travel too quickly to be seen, a rocket launcher requires that the missile be drawn on its somewhat slower path to its destination. The MistweaverProjectile class derives directly from the Projectile base class rather than farther down the tree because most of the Unreal Tournament routines are unnecessary bloat for a game like Mistweaver. This class contains attributes for the particle and sound effects associated with the projectile and
a couple routines for creation and destruction of the projectile. An archetype of this class is used to define the Magic Missile projectile that is used by the character in the demo version of Mistweaver.

An extension of the projectile class was made to allow projectiles to bounce off walls and floors. MistweaverProjectile_Bouncer adds a number of new attributes to control the life of the projectile, if the projectile should bounce up and down as well as from walls, and the amount of energy lost per bounce. If enough energy is lost before the fuse timer depletes, then the projectile will also explode. A couple new functions were added to support this functionality, including an override of the HitWall function that processes the current velocity and reflects it appropriately about the normal of the hit wall. Two archetypes were created using this class: one for fire balls that use gravity to bounce similar to the fire balls in the Mario games and one that forces the fire balls to fly straight from the source and only ricochet on walls. Code for the MistweaverProjectile class can be found in Source Listing A.11 and for the bouncing extension in Source Listing A.12.

For these projectiles to be visible in game, a particle system must be set up to visibly represent the effect as seen in Figure 6.12. This system consists of three emitters: one spawns the rotating arrow, one handles the streaking trails which are an animated texture on a simple twisted mesh, and the final produces the sparkle clouds.

Figure 6.12: Magic Missile particle effect.
6.3.5 Animations and the AnimTree

Animations are easily exported using the fbx pipeline. Each fbx file may contain one animation that can be imported in the AnimSet Editor. That editor is opened by double-clicking the AnimSet that is created automatically when the original skeletal mesh is imported, by default it will use the name of the root bone. The skeletal mesh file can also contain an animation, but this is not recommended in case any modifications need to be made to the rig or mesh later. The heroine can be seen previewing the walking animation in Figure 6.13.

![Figure 6.13: The AnimSet editor.](image)

The final piece of character setup needed is called the AnimTree. This node-based tree provides the Unreal Engine with the logic required to blend between the animations of a given AnimSet. The nodes of the AnimTree usually feature controls that allow the developer to quickly test various conditions and their output, as shown in Figure 6.14. The orange paths are the currently active settings, notice the changes in the BlendBySpeed and BlendByPhysics nodes near the left of the tree between the three images.

6.3.6 Casting a Spell

In order to incorporate action into Mistweaver, characters must be able to fire the weapons. This can be accomplished using another group of config settings, like the joystick example. For a complete game, it should be done that way, but for the sake of providing a different
example the spell casting button is created using Kismet. This is also a very simple solution, but Kismet is contained in the level not in the game, so it would need to be copied into every level in the game. The previously mentioned tutorials led me to believe that this would be a nice complicated example, but Mistweaver uses actual weapons with a functioning inventory manager, instead of a pure Kismet solution for the game. The entire solution in Kismet is contained in one node, which should easily translate into the config file settings later, the Add Input Zone node contains all the settings needed to draw the button and trigger a fire event. The Input Key set to MOBILE_Fire allows the button to trigger the firing code in the inventory manager.

6.4 Developing Some Simple Enemies

Enemy construction includes all of the steps outlined for characters, with the addition of needing some form of artificial intelligence (AI). The AI can be a set of instructions to follow in the most basic case, or an entire decision making process that uses weighted values for currently available actions.

6.4.1 Magical Turrets

There is an excellent example for a turret class available on the Unreal Developer Network[23]. The purpose of this tutorial is to learn state based programming in UnrealScript. Every class can define one or more states that allow a programmer to define different behav-
ior for functions depending on which state is currently active, functions outside of states are available to all states. A basic example of state declarations can be seen in Source Listing 6.4

Source Listing 6.4: State Programming Example

```cpp
function BasicFunc() { Log("Basic Function called"); }

state StateOne {
  function Tick() {
    BasicFunc();
    Log("In State One");
    GotoState(StateTwo);
  }
}

state StateTwo {
  function Tick() {
    BasicFunc();
    Log("In State Two");
  }
}
```

Using states, the turrets in Mistweaver slowly scan in a circle until the player comes into view, then another state locks the turret onto the player and follows until the player moves out of view, while firing a projectile at fixed intervals.

### 6.4.2 Enemy Characters

Enemy characters require two classes: a Pawn based class, set up to give control the second class, a controller class derived from AIController. The MWPawn_NPC is declared as placeable, which allows a level designer to place the pawn in the level directly. This works well for a Zelda style game that is based on a set number of enemies per room. The NPC pawn should be spawnable at run time as well using Kismet if multiple waves are needed for the game design, but that feature is untested at this time. The code for the MWPawn_NPC class is available in Source Listing A.13.

The MWBot class uses a few parameters to define the pawns’ behavior. Two path nodes are given as patrol points and a maximum search radius to determine how close a player must be before the pawn will attack. Given this basic framework, it should be a trivial
extension to allow for an arbitrary number of path points that might be used for a complex guard pattern that a player must sneak through.

### 6.4.3 Custom Kismet node for setup

The other new Kismet node used by *Mistweaver* is a simple interface to configure *MWBot* patrol paths and ranges as shown in Figure 6.15. Code for the node can be seen in Source Listing 6.5. These connections are used by the MWConfigureBot function in the MWBot class that can be seen in Source Listing A.14.

![Figure 6.15: A simple Kismet node to configure MWBots.](image_url)

**Source Listing 6.5: SeqAct_MWConfigureBot.uc**

```plaintext
class SeqAct_MWConfigureBot extends SequenceAction;

var() float MaxFireRange;
var() Object PatrolStart;
var() Object PatrolEnd;

defaultproperties {
  ObjName="Configure a Mistweaver Bot"
  ObjCategory="TDRPGame"
  HandlerName="MWConfigureBot"
  Target= None
  MaxFireRange=256.f
  PatrolStart= None
  PatrolEnd= None
  VariableLinks(1)=(ExpectedType=class'SeqVar_Float', LinkDesc="Max. Range",
                   bWriteable=true, PropertyName=MaxFireRange)
  VariableLinks(2)=(ExpectedType=class'SeqVar_Object', LinkDesc="Patrol Start",
                   bWriteable=true, PropertyName=PatrolStart)
```

55
6.5 Finishing Touches

6.5.1 Custom Icon Set

When packaging the final game with Unreal Frontend the icon set is included automatically. The icons are located in the \UDKGame\Build\iPhone\Resources\Graphics directory of the UDK installation. The complete icon set has six resolutions for use on various Apple devices. Including a new icon set is as simple as making a backup of the originals for reference and overwriting the images in this directory with the new icons. The original icon and the final Mistweaver icon can be found in Figure 6.16. The corners are rounded off automatically by the iOS device.

Figure 6.16: iOS Icons

6.5.2 Changing the Startup movie

The startup movie is also contained in the Resources directory, inside the Movies folder. Overwriting the Startup.m4v file will play a different movie while the game loads. This movie will quit automatically once the game is finished loading, so this is a bad place to put story elements.

6.5.3 Creating the Main Menu

The main menu setup is a simple combination of previously discussed topics. The menu is contained in a separate level file and should use the MobileMenuGame game type.
The level must contain a player start to function properly and should have a button that loads the first game level. This button is created using Kismet, like the fire button in Section 6.3.6. For the Mistweaver menu, the button does not use a texture, and fills the entire screen, so that wherever the player taps the game begins loading. Loading another level uses the Kismet Console Command node, which executes the command: ‘open insert-filename-here’.

Other embellishments can be added to the menu, this is a complete level as far as the Unreal Engine is concerned so anything can be used for decoration, either displaying a static image using the HUD Draw Image over a black screen or building an entire Matinee cutscene behind the logo.

### 6.5.4 Installing on the iPad

Installing Mistweaver on the iPad requires a properly set up Unreal Frontend profile. For an iPad game, a clone of the EpicCitadel profile is needed, then remove the EpicCitadel.udk map from the ‘Maps to Cook’ list. Add the main menu level and the dungeon level, at the bottom of the map list check the Override Default box and select the main menu map. If this is the first attempt to install a game on the iPad, a developer must click the Application Settings button and follow the directions on the dialog to register as an Apple developer before the Frontend will allow the installation. Finally, the Start button must be pressed, which will initiate a lengthy process of compiling, cooking, packaging, and deploying the game, which is now ready to test on the connected iPad.
Chapter 7

Realization of the Game Design

To demonstrate the feasibility of our approach, we have implemented the techniques that we have described in previous chapters. The result of this implementation is a playable game, Mistweaver, consisting of one level. We have ported Mistweaver to the Apple iPad and we present some results of this implementation in this chapter.

7.1 Entering Mistweaver: The Menu Screen

Figure 7.1 shows the main menu screen, which features a first person view of the entrance to the dungeon. The logo and instructions complete the basic menu. When the player touches the menu, it dims and displays the text ‘Level Loading.’ The music featured during the menu display is a recording of an Orff ensemble piece performed by an after-school group from Kensington Elementary School.

Figure 7.1: The Mistweaver main menu.
7.2 The World of Mistweaver

Figure 7.2 shows the layout of the level design. There is a small deviation from the original layout plan where the top of the tower is a completely separate floor with a connecting room that allows some forced perspective stretching of the main vestibule; this arrangement provides a greater sense of vastness and magnitude for the final boss battle.

Figure 7.2: Complete layout of the fourth floor. The rest of the final dungeon layouts can be seen in better detail in Appendix B, Figures B.5, B.6, B.7, and B.8

7.2.1 PC Testing

The majority of testing for a UDK game is done on a PC, even for games intended for consoles and iOS. It is more efficient to initialize the previewer on a PC rather than recompiling and packaging the entire app for test on a game console or an iPad. The screenshot in Figure 7.3 shows the Mobile Previewer version of a UDK testing window, which correctly emulates the final iOS product. The Mobile Previewer is more true to the look and play on the iPad than the preview provided by a standard Play In Editor window.

Figure 7.3: Testing Mistweaver on the PC.
7.2.2 iPad

Figure 7.4 shows the basic look of Mistweaver on the iPad. The walls use bump offset mapping to create an illusion of depth to the blocks. The top image shows the basic color palette, based upon the primary colors of the two factions of Mistweavers, with a blend of purple to represent the mix of powers in a true Mistweaver.

The bottom left image shows the portal room where every door leads to another, unexpected door, in the same room and the lighting in this initial room is darker and mysterious. The bottom right image shows the forced perspective stretching of the entryway, shown in the top image, to make the final room of the tower appear much higher than it would have been if built exactly on top of the vestibule.

Figure 7.4: Mistweaver on the iPad.

7.3 Characters

The main character can be seen in Figure 7.5 along with a mobile enemy in the second image. The enemy uses basic, rule-based AI to recognize that the player is in range and to harass the player.

The player uses a very basic AnimSet of walking, idling, and falling to accomplish all

\footnote{An AnimSet was described in Chapter 5}
movements in a smooth manner. The final version of a complete game might include many other animations to accomplish tasks, such as opening the treasure chests, casting a spell, and taking damage.

7.4 Items

The two items of interest currently incorporated into Mistweaver are keys and gold, examples of these items are illustrated in Figures 7.6. The keys are used to unlock new areas of the dungeon, forcing players to explore the areas available to them before progressing to a final confrontation in each wing of the dungeon; this confrontation rewards the player with a color-coded key this is used to unlock the next wing of the dungeon.
Currently, there is not a way to spend the gold you collect, but in a complete game there would be shops in villages to purchase additional items.

7.5 Effects

Various effects are used in Mistweaver to create magic spells for the player and enemies, shown in Figure 7.7. The magic missile effect, described in Chapter 6, is shown in action in the first screen shot. The fireballs shot by the turrets, shown in the second image, are created as follows: a rotating mesh for the ball, an emitter for the fire trail, and a second emitter for the smoke trail.

Other dynamic effects in the game center around breaking various ground traps: two of these are shown in Figures 7.8 and 7.9. Figure 7.8 shows the camera move and break sequence for the trap described in Chapter 6.
The goal for the shattering floor, shown in Figure 7.9 is to add drama to a final boss fight, which will be incorporated into future iterations of Mistweaver. Each section of flooring, between the colored beams, will break at a set time when instigated by a boss attack; the effect of the shattering floor is to shrink the playable area of the room as the fight progresses.

Figure 7.9: Floor traps.

7.6 Gameplay

Mistweaver is a playable game with “light” combat and ample exploration challenges. The first image in Figure 7.10 illustrates basic light combat in the vestibule.

The maze in the second image illustrates hidden chests that provide opportunities for discovery as the player travels through the dungeon, and the image highlights several nice features of the UDK Lightmass lighting system. First, the bookcases are using a highly saturated placeholder texture in the image, which creates a large amount of bounce light. This bounce light is precomputed using photon mapping and is stored in texture maps, described in Chapter 5. Additionally, the lighting information is stored in a three dimensional grid system that is incorporated into the character’s lighting environment to accurately color moving actors based on the computed light information. In Figure 7.10 the character assumes a red hue while inside the maze.
Figure 7.10: Gameplay on the iPad.
Chapter 8

Conclusion

We have described our approach to using the UDK for the construction of an action-adventure video game for use on mobile gaming platforms. We have described our working prototype of this game, Mistweaver, and provided ample details of our construction and port to the Apple iPad. Mistweaver is a functional game prototype that utilizes many aspects of The Legend of Zelda gameplay. The Mistweaver demo is fully tested on the Apple iPad 2. We are in the process of extending some HUD samples into a package that will provide Kismet node extensions, similar to the health display featured in Section 6.2.5, to support other HUD features of classic games for use in UDK development. We intend to submit a description of our technique to the Computer Games Conference (CGAMES).
Appendices
Appendix A

Mistweaver Source Code

Source Listing A.1: TDRPGame.uc

class TDRPGame extends SimpleGame;

event AddDefaultInventory(Pawn Pawn) {
  local TDRPGInventoryManager TDRPGInventoryManager;
  local TDRPGPawn TDRPGPawn;
  local int i;

  if (Pawn == None) {
    return;
  }

  TDRPGInventoryManager = TDRPGInventoryManager(Pawn.InvManager);
  if (TDRPGInventoryManager == None) {
    return;
  }

  TDRPGPawn = TDRPGPawn(Pawn);
  if (TDRPGPawn == None) {
    return;
  }

  if (TDRPGPawn.WeaponArchetypes.Length > 0) {
    for (i = 0; i < TDRPGPawn.WeaponArchetypes.Length; ++i) {
      if (TDRPGPawn.WeaponArchetypes[i] != None) {
        TDRPGInventoryManager.CreateInventoryFromArchetype(TDRPGPawn.
          WeaponArchetypes[i]);
      }
    }
  }

  defaultproperties {
    PlayerControllerClass='TDRPGame.TDRPGPlayerController'
    DefaultPawnClass='TDRPGame.TDRPGPawn'
    bRestartLevel=false
    bWaitingToStartMatch=true
    bDelayedStart=false
  }

}
class TDRPGPawn extends UDKPawn;

var bool bSnapMove;
// Light environment component used by the pawn mesh
var (Pawn) const LightEnvironmentComponent LightEnvironment;
// Weapon socket name to attach weapons
var (Pawn) const Name WeaponSocketName;
// Current weapon archetype used for attachment purposes
var (Weapons) const array<MistweaverSpell> WeaponArchetypes;
var MistweaverSpell WeaponAttachmentArchetype;
var int MaxHealth;

DefaultProperties {
  InventoryManagerClass=class’TDRPGame.TDRPGInventoryManager’
  Begin Object Class=DynamicLightEnvironmentComponent Name=MyLightEnvironment
    bSynthesizeSHLight=true
    bIsCharacterLightEnvironment=true
    bUseBooleanEnvironmentShading=false
    InvisibleUpdateTime=1.f
    MinTimeBetweenFullUpdates=0.2f
  End Object
  LightEnvironment=MyLightEnvironment
  Components.Add(MyLightEnvironment)

  Begin Object Class=SkeletalMeshComponent Name=SkeletalMeshComponent0
    SkeletalMesh=SkeletalMesh‘TestCharacters.Daughter.Daughter_Base2’
    AnimTreeTemplate=AnimTree‘TestCharacters.Test’
    AnimSets(0)=AnimSet‘TestCharacters.Daughter.b_X_root’
    Translation=(Z=-46.0)
    Scale=0.85

    //General Mesh Properties
    bCacheAnimSequenceNodes=FALSE
    AlwaysLoadOnClient=true
    AlwaysLoadOnServer=true
    bOwnerNoSee=false
    CastShadow=true
    BlockRigidBody=TRUE
    bUpdateSkelWhenNotRendered=false
    bIgnoreControllersWhenNotRendered=TRUE
    bUpdateKinematicBonesFromAnimation=true
    RBChannel=RBCC_Untitled3
    RBCollideWithChannels=(Untitled3=TRUE)
    bOverrideAttachmentOwnerVisibility=true
    bAcceptsDynamicDecals=FALSE
    bHasPhysicsAssetInstance=true
    TickGroup=TG_PreAsyncWork
    MinDistFactorForKinematicUpdate=0.2
    bChartDistanceFactor=true
    RBDominanceGroup=20
    MotionBlurInstanceScale=0.0
}
bUseOnePassLightingOnTranslucency=True
bPerBoneMotionBlur=true
LightEnvironment=MyLightEnvironment;
End Object
Mesh=SkeletalMeshComponent0
Components.Add(SkeletalMeshComponent0)

Begin Object Name=CollisionCylinder
  CollisionRadius=+0021.000000
  CollisionHeight=+0044.000000
End Object
CylinderComponent=CollisionCylinder

bSnapMove=true
GroundSpeed=220.0
AirSpeed=440.0
WaterSpeed=220.0
AccelRate=2048.0
WalkingPct=+1.0
MaxStepHeight=16

Health=6
MaxHealth=6

WeaponSocketName=RightHandSocket
WeaponArchetypes(0)=MistweaverSpell’MW-Spells.Arch.MagicMissile_A’
class TDRPGPlayerController extends SimplePC;

// Scales input zones per aspect ratio
var float InputZoneScale;
var bool bCanMove;

// change setup for ipad vs. iphone
function SetupZones() {
    MPI = MobilePlayerInput(PlayerInput);
    StickMoveZone = MPI.FindZone("TDRPG_MoveZone");
}

function UpdateRotation( float DeltaTime ) {
    local Rotator DeltaRotation, NewRotation, ViewRotation;
    local float JoyX, JoyY, eps;
    eps = 0.45;
    ViewRotation = Rotation;
    JoyX = PlayerInput.RawJoyRight;
    JoyY = PlayerInput.RawJoyUp;

    //determine which axis is dominant
    if (Abs(JoyX) > Abs(JoyY)) {
        eps *= Abs(JoyX);
        // if the vertical is almost the same as the horizontal, angle
        if ((Abs(JoyY) > Abs(JoyX) - eps) && (Abs(JoyY) < Abs(JoyX) + eps)) {
            if (JoyY > 0.0) {
                ViewRotation.Yaw = 57344;
            } else if (JoyY < 0.0) {
                ViewRotation.Yaw = 40960;
            } else {
                ViewRotation.Yaw = 49152;
            }
        } else if (JoyX > 0.0) {
            if ((Abs(JoyY) > Abs(JoyX) - eps) && (Abs(JoyY) < Abs(JoyX) + eps)) {
                if (JoyY > 0.0) {
                    ViewRotation.Yaw = 8192;
                } else if (JoyY < 0.0) {
                    ViewRotation.Yaw = 24576;
                } else {
                    ViewRotation.Yaw = 16384;
                }
            } else {
                eps *= Abs(JoyY);
                if (JoyY > 0.0) {
                    if ((Abs(JoyY) > Abs(JoyX) - eps) && (Abs(JoyY) < Abs(JoyX) + eps)) {
                        if (JoyX > 0.0) {
                            ViewRotation.Yaw = 8192;
                        } else if (JoyX < 0.0) {
                            ViewRotation.Yaw = 57344;
                        } else {
                            //...
ViewRotation.Yaw = 0;
} else if (JoyY < 0.0) {
    if ((Abs(JoyX) > Abs(JoyY) - eps) && (Abs(JoyX) < Abs(JoyY) + eps)) {
        if (JoyX > 0.0)
            ViewRotation.Yaw = 24576;
        else if (JoyX < 0.0)
            ViewRotation.Yaw = 40960;
    } else
        ViewRotation.Yaw = 32768;
}
}

ProcessViewRotation( DeltaTime, ViewRotation, DeltaRotation );
ViewShake( deltaTime );
NewRotation = RInterpTo( Rotation, ViewRotation, DeltaTime, 200000, true );
SetRotation( NewRotation );
if ( Pawn != None ) Pawn.FaceRotation( NewRotation, DeltaTime );
}

state PlayerWalking {
    function PlayerMove( float DeltaTime ) {
        if (PlayerInput.aForward < 0)
            PlayerInput.aForward *= -1.0;
        if (PlayerInput.aStrafe < 0)
            PlayerInput.aStrafe *= -1.0;

        PlayerInput.aForward += PlayerInput.aStrafe;
        PlayerInput.aStrafe = 0.0;
        PlayerInput.aForward /= 2.0;
        if (PlayerInput.aForward > 1000.0)
            PlayerInput.aForward = 1000.0;

        Super.PlayerMove( DeltaTime );
    }
}

DefaultProperties { 
    CameraClass=class 'TDRPGame.TDRPGPlayerCamera'
    AutoRotationAccelRate=10000.0
    AutoRotationBrakeDecelRate=10000.0
    MaxAutoRotationVelocity=300000
    BreathAutoRotationAccelRate=250.0
    BreathAutoRotationBrakeDecelRate=1.0
    MaxBreathAutoRotationVelocity=75
    TimeBetweenCameraBreathChanges = 2.0
    RangeBasedYawAccelStrength=8.0
    RangeBasedAccelMaxDistance=512.0
    bCanMove=true
}
class TDRPGRPlayerCamera extends Camera;

var Vector CamOffset;

// How far above the floor are the collision volumes supposed to be
const CamVolumeHeight = 800;

/* Calculate the new camera position given collision with a CamVolume
 * that requires locking in the given direction.
 */
function float ModifyDirection(LockDirection LockDir, float Origin, float Extent, float Offset) {
    switch (LockDir) {
        case LOCK_POSITIVE:
            Origin -= Extent;
            break;
        case LOCK_NEGATIVE:
            Origin += Extent;
            break;
    }
    return Origin + Offset;
}

function UpdateViewTarget(out TViewTarget OutVT, float DeltaTime) {
    local Vector HitLocation, HitNormal, NewLocation, TraceHeight;
    local Rotator NewRotation;
    local Actor HitActor;
    local TDRPGCamVolume HitCV;

    // Early exit if:
    // - We have a pending view target
    // - OutVT currently equals ViewTarget
    // - Blending parameter is lock outgoing
    if (PendingViewTarget.Target != None && OutVT == ViewTarget && BlendParams.bLockOutgoing) {
        return;
    }

    // setup trace to begin above the default height for cam volumes
    TraceHeight.Z = CamVolumeHeight + 32;

    // base case camera calculation
    NewLocation = OutVT.Target.Location + CamOffset;

    // check for camera volumes with a trace
    HitActor = Trace(HitLocation, HitNormal, OutVT.Target.Location, OutVT.Target.Location+TraceHeight, FALSE, vect(12,12,12));
    if (HitActor != none) {
        if (HitActor.IsA('TDRPGCamVolume')) {
            HitCV = TDRPGCamVolume(HitActor);
            if (HitCV.LockToOrigin) {
            } else {
                // calculate new position
            }
        } else {
            // calculate new position
        }
    }
}

//
// figure out camera position given the cam volume information
if (HitCV.LockXDirection != LOCK_NONE)
    NewLocation.X = ModifyDirection(HitCV.LockXDirection, HitCV.
    CollisionComponent.Bounds.BoxExtent.X, CamOffset.X);
if (HitCV.LockYDirection != LOCK_NONE)
    NewLocation.Y = ModifyDirection(HitCV.LockYDirection, HitCV.
    CollisionComponent.Bounds.BoxExtent.Y, CamOffset.Y);
}
// Set camera height based on the collision with the cam volume
if (HitCV.LockZ) NewLocation.Z = HitLocation.Z;
}

if (TDRPGPawn(OutVT.Target).bSnapMove) {
    OutVT.POV.Location = NewLocation;
    TDRPGPawn(OutVT.Target).bSnapMove = false;
} else
    OutVT.POV.Location = VInterpTo(OutVT.POV.Location, NewLocation,
    DeltaTime, 3.9);
// Make the camera point towards the target’s location
NewRotation.Pitch = -75.0f * DegToUnrRot;
OutVT.POV.Rotation = NewRotation;
}
defaultproperties {
    CamOffset=(X=-200.0,Y=0.0,Z=800.0)
    DefaultFOV=90.f
}
class TDRPGCamVolume extends BlockingVolume;

// extend camera blocking functionality for defining room boundaries
enum LockDirection { LOCK_NEGATIVE, LOCK_NONE, LOCK_POSITIVE };

var (TDRPG) LockDirection LockXDirection;
var (TDRPG) LockDirection LockYDirection;
var (TDRPG) bool LockZ;
var (TDRPG) bool LockToOrigin;

DefaultProperties {
  Begin Object Name=BrushComponent0
  CollideActors=true
  BlockActors=false
  BlockZeroExtent=false
  BlockNonZeroExtent=true
  BlockRigidBody=false
  bDisableAllRigidBody=false
  RBChannel=RBCC_BlockingVolume
  bBlockCamera=true
  bCollideActors=true
  bSkipActorPropertyReplication=true
  LockXDirection = LOCK_NONE;
  LockYDirection = LOCK_NONE;
  LockZ = true
  LockToOrigin = false
}

Source Listing A.5: TDRPGCamVolume.uc
Source Listing A.6: TDRPGTeleporter.uc

```plaintext
/*===============================================================
* Teleports actors either between different teleporters within a
* level or to matching teleporters on other levels.
* Based on code in Teleporter.uc from Epic Games
*=============================================================*/
class TDRPGTeleporter extends Teleporter
placeable;

// How far to place the pawn in front of the teleporter
var (TDRPG) float OutOffset;

// How long to wait for the camera fade
var (TDRPG) float TimeDelay;

// Replication of URL for easier setup
var (TDRPG) string TargetName;

// Replication of Tag for easier setup
var (TDRPG) name MyName;

// Storage so the time delay will work
var Actor Inc;
var Actor Src;

event PostBeginPlay() {
    URL = TargetName;
    Tag = MyName;
    Super.PostBeginPlay();
}

//forums.epicgames.com/threads/776140-How-To-Unrealscript-run-function-in-Kismet
function TriggerRemoteKismetEvent( name EventName ) {
    local array<SequenceObject> AllSeqEvents;
    local Sequence GameSeq;
    local int i;

    GameSeq = WorldInfo.GetGameSequence();
    if (GameSeq != None) {
        // find any Remote Events events that exist
        GameSeq.FindSeqObjectsOfClass(class'SeqEvent_RemoteEvent', true, AllSeqEvents);
        // activate them
        for (i = 0; i < AllSeqEvents.Length; i++) {
            if(SeqEvent_RemoteEvent(AllSeqEvents[i]).EventName == EventName)
                SeqEvent_RemoteEvent(AllSeqEvents[i]).CheckActivate(WorldInfo, None);
        }
    }
}

// Accept an actor that has teleported in.
simulated event bool Accept( Actor Incoming, Actor Source) {
    Inc = Incoming;
    Src = Source;
    TriggerRemoteKismetEvent('DoorTransition');
    SetTimer(TimeDelay, false, 'AcceptTimer');
```
return true;

function AcceptTimer() {
    AcceptTimed(Inc, Src);
    ClearTimer("AcceptTimer");
}

simulated event bool AcceptTimed( actor Incoming, Actor Source ) {
    local rotator NewRot, oldRot;
    local float mag;
    local vector oldDir, newLoc;
    local Controller C;
    if ( Incoming == None )
        return false;

    TDRPGPawn(Incoming).bSnapMove = true;
    // Move the actor here.
    Disable('Touch');
    NewRot = Incoming.Rotation;
    if (bChangesYaw) {
        oldRot = Incoming.Rotation;
        NewRot.Yaw = Rotation.Yaw;
        if ( Source != None ) {
            NewRot.Yaw += (32768 + Incoming.Rotation.Yaw - Source.Rotation.Yaw);
        }
    }
    if ( Pawn(Incoming) != None ) {
        //tell enemies about teleport
        if ( Role == ROLE_Authority ) {
            foreach WorldInfo.AllControllers(class 'Controller', C) {
                if ( C.Enemy == Incoming ) {
                    C.EnemyJustTeleported();
                }
            }
        }
        //Added to make characters appear in front of the door
        newLoc = Location + Vector(Rotation) * OutOffset;
        if ( !Pawn(Incoming).SetLocation(newLoc) ) {
            'log(self" Teleport failed for ":$Incoming);
            return false;
        }
        if ( (Role == ROLE_Authority)
            || (WorldInfo.TimeSeconds - LastFired > 0.5) ) {
            NewRot.Roll = 0;
            Pawn(Incoming).SetRotation(NewRot);
            Pawn(Incoming).SetViewRotation(NewRot);
            Pawn(Incoming).ClientSetRotation(NewRot);
            LastFired = WorldInfo.TimeSeconds;
        }
        if ( Pawn(Incoming).Controller != None ) {
            Pawn(Incoming).Controller.MoveTimer = -1.0;
Pawn(Incoming).SetAnchor(self);
Pawn(Incoming).SetMoveTarget(self);
}
Incoming.PlayTeleportEffect(false, true);
} else {
  if (!Incoming.SetLocation(Location)) {
    Enable("Touch");
    return false;
  }
  if (bChangesYaw)
    Incoming.SetRotation(NewRot);
  Enable("Touch");
  if (bChangesVelocity)
    Incoming.Velocity = TargetVelocity;
  else {
    if (bChangesYaw) {
      if (Incoming.Physics == PHYS_Walking)
        OldRot.Pitch = 0;
      oldDir = vector(OldRot);
      mag = Incoming.Velocity.Dot oldDir;
      Incoming.Velocity = Incoming.Velocity - mag * oldDir +
                          vector(Incoming.Rotation);
    }
    if (bReversesX)
      Incoming.Velocity.X *= -1.0;
    if (bReversesY)
      Incoming.Velocity.Y *= -1.0;
    if (bReversesZ)
      Incoming.Velocity.Z *= -1.0;
  }
  Incoming.PostTeleport(self);
  return true;
}
defaultproperties {
  Begin Object NAME=CollisionCylinder
    CollisionRadius=+00040.000000
    CollisionHeight=+00080.000000
    CollideActors=true
  End Object

  Begin Object NAME=Sprite
    Sprite=Texture2D'EditorResources.tdrpg_tele'
  End Object

  OutOffset = 100;
  TimeDelay = 0.6;
  RemoteRole=ROLE_SimulatedProxy
  bChangesYaw=true
  bEnabled=true
  bCollideActors=true
  bStatic = false;
}
Source Listing A.7: TDRPGLnventoryManager.uc

class TDRPGLnventoryManager extends InventoryManager;

simulated function Inventory CreateInventoryFromArchetype(Inventory
    InventoryArchetype, optional bool bDoNotActivate) {
    local Inventory Inv;

    // ensure that the archetype is valid
    if (InventoryArchetype == None)
        return None;

    // spawn the inventory
    Inv = Spawn(InventoryArchetype.Class, Owner,\,, InventoryArchetype);
    if (Inv != None) {
        // destroy the inventory if the add fails
        if (!AddInventory(Inv, bDoNotActivate)) {
            Inv.Destroy();
            return None;
        } else {
            return Inv;
        }
    }
    return None;
}

DefaultProperties {
    PendingFire(0)=0
    PendingFire(1)=0
}
/**
 * MistweaverSpell
 * The base class to handle spell specific weapon mechanics.
 */

class MistweaverSpell extends UDKWeapon
{
    HideCategories(Movement, Display, Attachment, Collision, Physics, Advanced, Debug, Object);

    var (Weapon) const editinline instanced array<MistweaverSpellMode> FireModes;

    simulated event PostBeginPlay() {
        local int lcv;
        Super.PostBeginPlay();
        // set up mode owners
        if(FireModes.Length > 0) {
            for (lcv = 0; lcv < FireModes.Length; ++lcv) {
                if (FireModes[lcv] != None)
                    FireModes[lcv].SetOwner(Self);
            }
        }
    }

    simulated function SendToFiringState(byte FireModeNum) {
        // make sure spell mode is valid
        if (FireModeNum >= FiringStatesArray.Length)
            return;
        // set the spell mode
        SetCurrentFireMode(FireModeNum);
        // transition to the weaving state
        GoToState(FiringStatesArray[FireModeNum]);
    }

    simulated function FireAmmunition() {
        // use ammunition
        ConsumeAmmo(CurrentFireMode);
        // forward the fire call to the FireMode object
        if (CurrentFireMode < FireModes.Length && FireModes[CurrentFireMode] != None)
            FireModes[CurrentFireMode].Fire();
        NotifyWeaponFired(CurrentFireMode);
    }

    simulated function PlayFireEffects(byte FireModeNum, optional vector HitLocation) {
        // forward to FireMode object
        if (FireModeNum < FireModes.Length && FireModes[FireModeNum] != None)
            FireModes[FireModeNum].PlayFiringEffects(HitLocation);
    }
simulated function StopFireEffects(byte FireModeNum) {
  // Forward the stop fire effects to the FireMode object
  if (FireModeNum < FireModes.Length && FireModes[FireModeNum] != None)
    FireModes[FireModeNum].StopFiringEffects();
}

simulated function AttachToPawn(Pawn NewPawn) {
  local TDRPGPawn TPawn;
  // check the mesh and pawn’s mesh
  if (Mesh != None && NewPawn != None && NewPawn.Mesh != None)
    TPawn = TDRPGPawn(NewPawn);
  if (TPawn != None && TPawn.Mesh.GetSocketByName(TPawn.WeaponSocketName) != None) {
    // Attach to instigator’s skeletal mesh
    TPawn.Mesh.AttachComponentToSocket(Mesh, TPawn.WeaponSocketName);
    // Set the lighting environment
    Mesh.SetLightEnvironment(TPawn.LightEnvironment);
    // Set the weapon’s shadow parent to the instigator’s skeletal mesh
    Mesh.SetShadowParent(TPawn.Mesh);
    // Set the attachment archetype so the spell is visible
    TPawn.WeaponAttachmentArchetype = MistweaverSpell(ObjectArchetype);
  }
}

// This state is the transition from inactive to active states
simulated state WeaponEquipping {
  simulated function WeaponEquipped() {
    if (bWeaponPutDown) {
      PutDownWeapon();
      return;
    }
    AttachToPawn(Instigator);
    GoToState('Active');
  }
}

simulated function TimeWeaponFiring(byte FireModeNum) {
  if (CurrentFireMode < FireModes.Length && FireModes[CurrentFireMode] != None && FireModes[CurrentFireMode].RequiredTickDuringFire)
    return;
  Super.TimeWeaponFiring(FireModeNum);
}

simulated state WeaponFiring {
  simulated function Tick(float DeltaTime) {
    Global.Tick(DeltaTime);
    // if the fire mode needs a tick, forward to player if still
firing

if (CurrentFireMode < FireModes.Length && FireModes[CurrentFireMode] != None && FireModes[CurrentFireMode].RequiredTickDuringFire) {
    if (ShouldRefire()) {
        FireModes[CurrentFireMode].Tick(DeltaTime);
    } else {
        HandleFinishedFiring();
    }
}

simulated function EndState(Name NextStateName) {
    local int i;

    // stop firing effects
    if (FireModes.Length > 0) {
        for (i = 0; i < FireModes.Length; i++) {
            if (FireModes[i] != None)
                FireModes[i].StopFiringEffects();
        }
    }

    DefaultProperties {
        Begin Object Class=SkeletalMeshComponent Name=
            MySkeletalMeshComponent
        bUpdateSkelWhenNotRendered=false
        bIgnoreControllersWhenNotRendered=true
        bAcceptsDynamicDecals=false
        CastShadow=true
        TickGroup=TG_DuringAsyncWork
        End Object
        Mesh=MySkeletalMeshComponent
        Components.Add(MySkeletalMeshComponent)
        FiringStatesArray(0)="WeaponFiring"
    }
}
// Handles weapon firing mechanics

class MistweaverSpellMode extends Object
  HideCategories(Object)
  EditInLineNew
  Abstract;

  // sockets
  var(SpellMode) const Name MuzzleSocketEffectName;
  var(SpellMode) const Name FireSocketName;

  // options
  var(SpellMode) const bool HasRecoil;
  var(SpellMode) const bool RequiredTickDuringFire;

  // sounds
  var(Sounds) const SoundCue FireSoundCue;
  var(Sounds) const SoundCue StopFireSoundCue;
  var(Sounds) const SoundCue ContinuousFireSoundCue<EditCondition=RequiredTickDuringFire>;

  // particles
  var(ParticleSystems) const ParticleSystem FireParticleSystem;

  protected write MistweaverSpell Owner;
  protected write ParticleSystemComponent MuzzleFlashParticleSystem;
  protected write AudioComponent ContinuousFireAudioComponent;
  protected write bool HasPlayedFiringSound;

  simulated function SetOwner(MistweaverSpell NewOwner) {
    if (NewOwner != None)
      Owner = NewOwner;
  }

  final simulated function Fire() {
    if (Owner != None)
      BeginFire();
  }

  simulated function PlayFiringEffects(optional Vector HitLocation) {
    local SkeletalMeshComponent SkeletalMeshComponent;
    local Vector SocketLocation;
    local Rotator SocketRotation;
    if (Owner == None || Owner.WorldInfo.NetMode == NM_DedicatedServer)
      return;
    if (!(HasPlayedFiringSound || !RequiredTickDuringFire) && FireSoundCue != None && Owner.Instigator != None) {
      Owner.Instigator.PlaySound(FireSoundCue, true);
      HasPlayedFiringSound = true;
    }
    if (ContinuousFireSoundCue != None && ContinuousFireAudioComponent == None && Owner.Instigator != None) {
      ContinuousFireAudioComponent = Owner.Instigator.
      CreateAudioComponent(ContinuousFireSoundCue, true, true);
    }
  }
ContinuousFireAudioComponent.FadeIn(0.1f, 1.0f);

// handle socketed particle effects
SkeletalMeshComponent = SkeletalMeshComponent(Owner.Mesh);
if (SkeletalMeshComponent != None && SkeletalMeshComponent.GetSocketByName(MuzzleSocketEffectName) != None) {
    if (FireParticleTemplate != None) {
        if (MuzzleFlashParticleSystem == None) {
            MuzzleFlashParticleSystem = new ParticleSystemComponent;
            MuzzleFlashParticleSystem.SetTemplate(FireParticleTemplate);
            SkeletalMeshComponent.AttachComponentToSocket(MuzzleFlashParticleSystem, MuzzleSocketEffectName);
        }
    }
    // trigger the muzzle flash
    if (MuzzleFlashParticleSystem != None) {
        SkeletalMeshComponent.GetSocketWorldLocationAndRotation(MuzzleSocketEffectName, SocketLocation, SocketRotation);
        MuzzleFlashParticleSystem.ActivateSystem();
    }
}

simulated function StopFiringEffects() {
    if (Owner == None || Owner.WorldInfo.NetMode == NM_DedicatedServer)
        return;

    HasPlayedFiringSound = false;
    if (StopFireSoundCue != None && Owner.Instigator != None)
        Owner.Instigator.PlaySound(StopFireSoundCue, true);
    if (ContinuousFireAudioComponent != None) {
        ContinuousFireAudioComponent.FadeOut(0.1f, 0.f);
        ContinuousFireAudioComponent = None;
    }
    if (MuzzleFlashParticleSystem != None)
        MuzzleFlashParticleSystem.DeactivateSystem();
}

protected function BeginFire();

function simulated Tick(float DeltaTime);

DefaultProperties {
    HasRecoil = false
class MistweaverSpellMode_Projectile extends MistweaverSpellMode;

var (SpellMode_Projectile) const MistweaverProjectile
ProjectileArchetype<AllowAbstract>;

protected function BeginFire() {
  local Vector StartTrace, EndTrace, RealStartLoc, AimDir,
SocketLocation;
  local Rotator SocketRotation;
  local ImpactInfo TestImpact;
  local Projectile SpawnedProjectile;
  local SkeletalMeshComponent SkeletalMeshComponent;

  Owner.IncrementFlashCount();

  if (Owner.Role == Role_Authority) {
    SkeletalMeshComponent = SkeletalMeshComponent(Owner.Mesh);
    if (SkeletalMeshComponent != None && SkeletalMeshComponent.
      GetSocketByName(FireSocketName) != None) {
      // get the socket’s world location and rotation
      SkeletalMeshComponent.GetSocketWorldLocationAndRotation( 
        FireSocketName, SocketLocation, SocketRotation);
      RealStartLoc = SocketLocation;
      // Use owner rotation instead of socket rotation for now, this
      // ensures the projectile will travel parallel to the ground
      // until proper spellcasting animations are added.
      AimDir = Vector(Owner.Owner.Rotation);
    } else {
      StartTrace = Owner.Instigator.GetWeaponStartTraceLocation();
      AimDir = Vector(Owner.GetAdjustedAim(StartTrace));
      // this is the location where the projectile is spawned.
      RealStartLoc = Owner.GetPhysicalFireStartLoc(AimDir);
      if (StartTrace != RealStartLoc) {
        // If projectile is spawned at different location of crosshair
        // then simulate an instant trace where crosshair is aiming at
        // Get hit info.
        EndTrace = StartTrace + AimDir * Owner.GetTraceRange();
        TestImpact = Owner.CalcWeaponFire(StartTrace, EndTrace);
        // Then realign projectile aim to match where the crosshair
        // hit.
        AimDir = Normal(TestImpact.HitLocation - RealStartLoc);
      }
      // Spawn Projectile
      SpawnedProjectile = Owner.Spawn(ProjectileArchetype.Class,,
        RealStartLoc,, ProjectileArchetype);
      if (SpawnedProjectile != None && !SpawnedProjectile.bDeleteMe) {
        SpawnedProjectile.Init(AimDir);
      }
    }
  } else {
  }

DefaultProperties {
class MistweaverProjectile extends Projectile

    HideCategories(Attachment, Physics, Advanced, Debug, Object);

var (Projectile) const class<DamageType> ProjectileDamageType<
    AllowAbstract>;

var (ParticleEffects) const ParticleSystem FlightParticleTemplate;
var (ParticleEffects) const ParticleSystem ExplosionParticleTemplate;

var (Sounds) const SoundCue AmbientFlightSoundCue;
var (Sounds) const SoundCue ExplosionSoundCue;

var ParticleSystemComponent FlightParticleSystemComponent;
var AudioComponent AmbientAudioComponent;
var bool HasExploded;

simulated event PostBeginPlay() {
    Super.PostBeginPlay();

    // spawn particle effect
    if (FlightParticleTemplate != None) {
        FlightParticleSystemComponent = new () class 'ParticleSystemComponent';
        if (FlightParticleSystemComponent != None) {
            FlightParticleSystemComponent.SetTemplate(FlightParticleTemplate);
            AttachComponent(FlightParticleSystemComponent);
        }
    }

    // create flight sound if it exists
    if (AmbientFlightSoundCue != None) {
        AmbientAudioComponent = CreateAudioComponent(AmbientFlightSoundCue,
            true, true);
    }

    MyDamageType = ProjectileDamageType;
}

simulated function Explode(vector HitLocation, vector HitNormal) {
    if (HasExploded) {
        return;
    }

    if (WorldInfo.NetMode != NM_DedicatedServer) {
        if (WorldInfo.MyEmitterPool != None) {
            if (FlightParticleSystemComponent != None) {
                FlightParticleSystemComponent.DeactivateSystem();
            }

            if (ExplosionParticleTemplate != None) {
                WorldInfo.MyEmitterPool.SpawnEmitter(ExplosionParticleTemplate,
                    Location, Rotator(HitNormal));
            }
        }
    }

    if (ExplosionSoundCue != None) {
        PlaySound(ExplosionSoundCue, true);
//apply damage
if (Damage > 0 && DamageRadius > 0) {
    if (Role == ROLE_Authority) {
        MakeNoise(1.0);
    }
    ProjectileHurtRadius(HitLocation, HitNormal);
}

// fade out ambient audio
if (AmbientAudioComponent != None) {
    AmbientAudioComponent.FadeOut(0.15f, 0.f);
}
HasExploded = true;
LifeSpan = 2.f;

DefaultProperties {
}
class MistweaverProjectile_Bouncer extends MistweaverProjectile;

// projectile life range
var (Bouncer) const float MinFuseTime;
var (Bouncer) const float MaxFuseTime;
var (Bouncer) const int MaxBounceCount;
// should the projectile bounce on the floor too
var (Bouncer) const bool UseGravity;
// additional Z velocity if affected by gravity
var (Bouncer) const float TossZ;
var (Bouncer) const float Damping;
// time in seconds before the bounce sound can play again
var (Bouncer) const float BounceSoundInterval;
var float NextBounceSoundTime;
var int BounceCount;

simulated event PostBeginPlay() {
    local float FuseTime;
    Super.PostBeginPlay();
    if (Role == Role_Authority) {
        FuseTime = RandRange(MinFuseTime, MaxFuseTime);
        SetTimer(FuseTime, false, NameOf(ExplodeTimer));
    }
}

function Init(vector Direction) {
    SetRotation(Rotator(Direction));
    Velocity = Normal(Direction) * Speed;
    BounceCount = 0;
    if (UseGravity) {
        Velocity.Z += TossZ + (FRand() * TossZ / 2.f) - (TossZ / 4.f);
    }
}

function ExplodeTimer() {
    Explode(Location, Vect(0.f, 0.f, 1.f));
}

function HitWall(vector HitNormal, Actor Wall, PrimitiveComponent WallComp) {
    bBlockedByInstigator = true;

    // play the bounce sound
    if (WorldInfo.NetMode != NM_DedicatedServer && WorldInfo.TimeSeconds >= NextBounceSoundTime) {
        PlaySound(ImpactSound, true);
        NextBounceSoundTime = WorldInfo.TimeSeconds + BounceSoundInterval;
    }

    // check to make sure we didn’t hit a pawn
    if (Pawn(Wall) == None) {
        BounceCount++;
    }
// Reflect off wall with damping
Velocity = Damping * ((Velocity dot HitNormal) * HitNormal * -2.f + Velocity);
Speed = VSize(Velocity);
if (Velocity.Z > 400.f) {
    Velocity.Z = 0.5f * (400.f + Velocity.Z);
}
if (!UseGravity) {
    Velocity.Z = 0.f;
    Velocity = Normal(Velocity) * Speed;
}
if (Speed < 20.f || Pawn(Wall) != None || BounceCount > MaxBounceCount) {
    ImpactedActor = Wall;
    SetPhysics(PHYS_None);
    Explode(Location, HitNormal);
}
else if (Wall != Instigator) {
    Explode(Location, HitNormal);
}

simulated function Tick(float DeltaTime) {
if (UseGravity) {
    Velocity.Z += WorldInfo.GetGravityZ() / 100.f;
}
SetRotation(Rotator(Normal(Velocity)));
}

DefaultProperties {
bBounce=true
LifeSpan=0.f
TossZ=128.f
Damping=0.95f
MaxBounceCount=5
UseGravity=false
}
class MWPawn_NPC extends UDKPawn
placeable;

var (NPC) SkeletalMeshComponent NPCMesh;
var (NPC) class <AIController> NPCController;

simulated event PostBeginPlay() {
  if (NPCController != None) {
    // set the existing controller class to our npc controller
    ControllerClass = NPCController;
  }
  Super.PostBeginPlay();
}

DefaultProperties {
  Begin Object Class=SkeletalMeshComponent Name=NPCMesh0
  SkeletalMesh=SkeletalMesh'MW-Elemental.Elemental_Base'
  AnimSets(0)=AnimSet'MW-Elemental.joint1'
  // Needs physics asset and animtree
  Translation=(Z=-46.0)
  End Object

  NPCMesh=NPCMesh0
  Mesh=NPCMesh0
  Components.Add(NPCMesh0)

  Begin Object Class=DynamicLightEnvironmentComponent Name=
    MyLightEnvironment
    bSynthesizeSHLight=true
    bIsCharacterLightEnvironment=true
    bUseBooleanEnvironmentShadowing=false
    InvisibleUpdateTime=1.f
    MinTimeBetweenFullUpdates=0.2f
  End Object

  LightEnvironment=MyLightEnvironment
  Components.Add(MyLightEnvironment)

  NPCController=class 'MWBot'
  Health=2
  GroundSpeed=150.0
  bCanTeleport=False
}
class MWBot extends UTBot;

var() Actor MWPatrolStart;
var() Actor MWPatrolEnd;
var() Actor MW_Target;
var() Actor MW_Target;
var() TDRPGPlayerController MWTarget;
var() float MWMaxFireRange;

function MWConfigureBot(SeqAct_MWConfigureBot MyAction) {
MWPatrolStart = Actor(MyAction.PatrolStart);
MWPatrolEnd = Actor(MyAction.PatrolEnd);
MWMaxFireRange = MyAction.MaxFireRange;
SetTimer(0.3, true, 'ScanForPlayers');
}

function bool ShouldStrafeTo(Actor WayPoint) {
return false;
}

function ScanForPlayers() {
foreach WorldInfo.AllControllers(class 'TDRPGPlayerController',
MWTarget) {
'Log(MWTarget);
if(MWTarget != None && VSize(MWTarget.Pawn.Location - Pawn.
Location) < MWMaxFireRange && FastTrace(MWTarget.Pawn.Location
, Pawn.Location,,true)) {
Focus = MWTarget;
Enemy = MWTarget.Pawn;
VisibleEnemy = Enemy;
EnemyVisibilityTime = WorldInfo.TimeSeconds;
bEnemyIsVisible = true;
if(!IsInState('ChargingNoStrafe'))
{
GotoState('ChargingNoStrafe');
}
}
}

state Defending {
ignores EnemyNotVisible;
}

function Restart( bool bVehicleTransition ) {} 

function bool IsDefending() {
return true;
}

function EnableBumps() {
enable('NotifyBump');
}

event MonitoredPawnAlert() {
WhatToDoNext();
}
function ClearPathFor(Controller C) {
}

function SetRouteGoal() {
    local Actor NextMoveTarget;
    local bool bCanDetour;
    bCanDetour = true;
    if(ActorReachable(RouteGoal))
        NextMoveTarget = RouteGoal;
    else
        NextMoveTarget = FindPathToward(RouteGoal, bCanDetour);
    if(NextMoveTarget == None)
        NextMoveTarget = FindPathToward(RouteGoal, bCanDetour);
    if(NextMoveTarget == None) {
        CampTime = 3;
        GotoState('Defending','Pausing');
    }
    Focus = NextMoveTarget;
    MoveTarget = NextMoveTarget;
}

function EndState(Name NextStateName) {
    MonitoredPawn = None;
    SetCombatTimer();
    bShortCamp = false;
}

function BeginState(Name PreviousStateName) {}
StopMovement();
SetFocalPoint(Pawn.Location + Vector(MoveTarget.Rotation) * 10.f);
Sleep(0.5f);
// paused at one destination, select the other
if(RouteGoal != none && RouteGoal == MWPatrolEnd)
    RouteGoal = MWPatrolStart;
else
    RouteGoal = MWPatrolEnd;
LatentWhatToDoNext();

state ChargingNoStrafe extends Charging {
    function bool StrafeFromDamage(float Damage, class<DamageType>
        DamageType, bool bFindDest) {
        return false;
    }
    function bool TryStrafe(vector sideDir) {
        return false;
    }
}
DefaultProperties {
    MWMaxFireRange=512
    MWPatrolEnd=None
    MWPatrolStart=None
    MWTarget=None
}
class SeqEvent_DrawPlayerHealth extends SeqEvent_HudRenderImage;

var (HUD) Texture2D HalfDisplayTexture;
var (HUD) Texture2D FullDisplayTexture;
var (HUD) int CurrentHealth;
var (HUD) int MaxHealth;
var (HUD) int UnitsPerLine;
var (HUD) float SpaceBetween;

function Render (Canvas TargetCanvas, Hud TargetHud) {
    local float UsedX, UsedY, UsedXL, UsedYL, UsedSpace;
    local float GlobalScaleX, GlobalScaleY;
    local int lcv, offsetX, offsetY;
    if (bIsActive) {
        PublishLinkedVariableValues();
        // Code from Epic Games to scale graphics for iOS devices
        // cache the global scales
        GlobalScaleX = class 'MobileMenuScene'.static.GetGlobalScaleX() / AuthoredGlobalScale;
        GlobalScaleY = class 'MobileMenuScene'.static.GetGlobalScaleY() / AuthoredGlobalScale;
        // if floating point values, multiply by the canvas size
        // else, apply GlobalScaleFactor, and undo the author scale
        UsedX = (DisplayLocation.X < 1.0f) ? DisplayLocation.X * TargetCanvas.SizeX : (DisplayLocation.X * GlobalScaleX);
        UsedY = (DisplayLocation.Y < 1.0f) ? DisplayLocation.Y * TargetCanvas.SizeY : (DisplayLocation.Y * GlobalScaleY);
        UsedXL = (XL <= 1.0f) ? XL * TargetCanvas.SizeX : (XL * GlobalScaleX);
        UsedYL = (YL <= 1.0f) ? YL * TargetCanvas.SizeX : (YL * GlobalScaleY);
        UsedSpace = (SpaceBetween <= 1.0f) ? SpaceBetween * TargetCanvas.SizeX : (SpaceBetween * GlobalScaleX);
        offsetX = 0;
        offsetY = 0;
        for (lcv = 1; lcv < MaxHealth; lcv+=2) {
            TargetCanvas.SetPos(UsedX + ((UsedXL + UsedSpace) * offsetX), UsedY + ((UsedYL + UsedSpace) * offsetY));
            TargetCanvas.DrawTile(DisplayTexture, UsedXL, UsedYL, U, V, UL, VL, DisplayColor);
            if (CurrentHealth >= lcv) {
                TargetCanvas.SetPos(UsedX + ((UsedXL + UsedSpace) * offsetX), UsedY + ((UsedYL + UsedSpace) * offsetY));
                TargetCanvas.DrawTile(HalfDisplayTexture, UsedXL, UsedYL, U, V, UL, VL, DisplayColor);
            }
            if (CurrentHealth >= lcv + 1) {
                TargetCanvas.SetPos(UsedX + ((UsedXL + UsedSpace) * offsetX), UsedY + ((UsedYL + UsedSpace) * offsetY));
                TargetCanvas.DrawTile(FullDisplayTexture, UsedXL, UsedYL, U, V, UL, VL, DisplayColor);
            }
        }
    }
}
Source Listing A.16: DefaultEngineUDK.ini Additions

```plaintext
[TDRPG_MoveZone MobileInputZone]
InputKey=MOBILE_AForward
HorizontalInputKey=MOBILE_AStrafe
Type=ZoneType_Joystick
bRelativeX=true
bRelativeY=true
bRelativeSizeX=true
bRelativeSizeY=true
X=0.075
Y=-0.285
SizeX=0.15
SizeY=1.0
bSizeYFromSizeX=true
VertMultiplier=-1.0
HorizMultiplier=1.0
bScalePawnMovement=true
RenderColor=(R=255,G=255,B=255,A=255)
InactiveAlpha=0.3
bUseGentleTransitions=true
ResetCenterAfterInactivityTime=3.0
ActivateTime=0.6
DeactivateTime=0.2
TapDistanceConstraint=5

[TDRPGGame.TDRPGame]
+RequiredMobileInputConfigs=(GroupName="GamePlayInputs", RequireZoneNames=("TDRPG_MoveZone"))
```
Appendix B

Additional Images

Figure B.1: Dungeon – Overview.
Figure B.2: Dungeon – Floor 2.
Figure B.3: Dungeon – Floor 3.
Figure B.4: Dungeon – Floor 4.
Figure B.5: Complete Dungeon – Floor 2.
Figure B.6: Complete Dungeon – Floor 3.
Figure B.7: Complete Dungeon – Floor 4.
Figure B.8: Complete Dungeon – Other Rooms.
Bibliography


103


