ROOM GENERATION, REMOVAL, AND RECOVERY IN UDK 3.5

Scott Johnson
Clemson University, smj2@g.clemson.edu

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

Recommended Citation
Johnson, Scott, "ROOM GENERATION, REMOVAL, AND RECOVERY IN UDK 3.5" (2013). All Theses. 1774.
https://tigerprints.clemson.edu/all_theses/1774

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.
ROOM GENERATION, REMOVAL, AND RECOVERY IN UDK 3.5

A Thesis
Presented to
the Graduate School of
Clemson University

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

by
Scott M. Johnson
December 2013

Accepted by:
Dr. Brian A. Malloy, Committee Chair
Dr. Sophie Jörg
Dr. Joshua A. Levine
Abstract

This paper explores algorithms and techniques for creating an extended game world that represents a radical departure from the widely used static content in current use with most games featuring procedurally generated game worlds. Using the Unreal Development Kit 3.5 game engine and focusing on a small set of rooms and doors the research sets to create a game world that is in constant flux and addresses issues of game development limitations on the size of game worlds.
# Table of Contents

<table>
<thead>
<tr>
<th>Abstract</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Games and Grids</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Zelda (1986)</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Unity and Houdini</td>
<td>6</td>
</tr>
<tr>
<td>2.3 The UDK 3.5</td>
<td>7</td>
</tr>
<tr>
<td>2.3.1 Rationale for Using the UDK</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Flash</td>
<td>8</td>
</tr>
<tr>
<td>2.4.1 Why was Flash needed for this research?</td>
<td>8</td>
</tr>
<tr>
<td>3 Related Work</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Rogue (1980)</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Dynamic Dungeon Prototype (2009)</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Fallen Enchantress (2012)</td>
<td>11</td>
</tr>
<tr>
<td>3.4 Faster Than Light (2012)</td>
<td>12</td>
</tr>
<tr>
<td>4 Setup and Use of the UDK 3.5</td>
<td>15</td>
</tr>
<tr>
<td>4.1 New Project</td>
<td>15</td>
</tr>
<tr>
<td>4.2 Preparing 3d Assets for UDK from Maya</td>
<td>16</td>
</tr>
<tr>
<td>5 Methodology and Implementation</td>
<td>19</td>
</tr>
<tr>
<td>5.1 GameMode.uc</td>
<td>20</td>
</tr>
<tr>
<td>5.1.1 Class Setup</td>
<td>20</td>
</tr>
<tr>
<td>5.2 PlayerController.uc</td>
<td>20</td>
</tr>
<tr>
<td>5.2.1 Class Setup</td>
<td>20</td>
</tr>
<tr>
<td>5.3 Room Generation</td>
<td>21</td>
</tr>
<tr>
<td>5.3.1 translateDoor()</td>
<td>21</td>
</tr>
<tr>
<td>5.3.2 setupTestRoom()</td>
<td>21</td>
</tr>
<tr>
<td>5.3.3 placeDoor()</td>
<td>22</td>
</tr>
<tr>
<td>5.3.4 setupDoorGrid()</td>
<td>22</td>
</tr>
<tr>
<td>5.3.5 buildRooms()</td>
<td>22</td>
</tr>
<tr>
<td>5.4 Room Removal</td>
<td>23</td>
</tr>
<tr>
<td>5.4.1 Constant Deletion</td>
<td>23</td>
</tr>
<tr>
<td>5.4.2 Connected Deletion</td>
<td>24</td>
</tr>
<tr>
<td>5.4.3 Random Deletion</td>
<td>24</td>
</tr>
<tr>
<td>5.5 Room Recovery</td>
<td>25</td>
</tr>
<tr>
<td>5.5.1 Pseudo Recovery</td>
<td>25</td>
</tr>
<tr>
<td>5.5.2 Whole Map Recovery</td>
<td>25</td>
</tr>
</tbody>
</table>
# Table of Contents (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.3 Single Room Recovery</td>
<td>26</td>
</tr>
<tr>
<td>5.6 Wall Generation</td>
<td>26</td>
</tr>
<tr>
<td>5.6.1 Getting a unique reference number</td>
<td>26</td>
</tr>
<tr>
<td>5.6.2 setupWalls()</td>
<td>28</td>
</tr>
<tr>
<td><strong>6 Case Study</strong></td>
<td>29</td>
</tr>
<tr>
<td>6.1 Walk-through</td>
<td>29</td>
</tr>
<tr>
<td><strong>7 Future Work and Conclusions</strong></td>
<td>34</td>
</tr>
<tr>
<td>7.1 Different room types into UDK</td>
<td>34</td>
</tr>
<tr>
<td>7.1.1 Basic Walls in UDK</td>
<td>34</td>
</tr>
<tr>
<td>7.1.2 Wall and Room Styles</td>
<td>34</td>
</tr>
<tr>
<td>7.2 Filling the Rooms</td>
<td>34</td>
</tr>
<tr>
<td>7.3 After Thoughts on UDK 3.5</td>
<td>35</td>
</tr>
<tr>
<td>7.4 After Thoughts on UDK 3.5</td>
<td>36</td>
</tr>
<tr>
<td>7.5 Conclusion</td>
<td>37</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>38</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>19</td>
</tr>
<tr>
<td>5.2</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>29</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6.2</td>
<td>Base room is created and player is spawned into world.</td>
</tr>
<tr>
<td>6.3</td>
<td>Player looks at East door of current room.</td>
</tr>
<tr>
<td>6.4</td>
<td>Player clicks on door causing a new room to be created.</td>
</tr>
<tr>
<td>6.5</td>
<td>Player moves into newly created room and looks back at old room.</td>
</tr>
<tr>
<td>6.6</td>
<td>Player looks at West door of current room.</td>
</tr>
<tr>
<td>6.7</td>
<td>Player clicks on West door causing the starting room to be deleted and</td>
</tr>
<tr>
<td></td>
<td>creating a new room off of the West door.</td>
</tr>
<tr>
<td>6.8</td>
<td>Player looks back at Starter room and see’s it has been deleted.</td>
</tr>
<tr>
<td>7.1</td>
<td>A mockup of a corner room piece ready for UDK with testing texture.</td>
</tr>
<tr>
<td>7.2</td>
<td>Concept art for possible advanced room look set in a decayed unkept state.</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction and Motivation

In video game development, the size of a game world can be bounded by real world restrictions such as budget, time, or hardware limitations [1][17]. These limitations can induce situations where a player has the possibility of entering areas of the game that lack detail and functionality. Even in games with massively sized worlds, such as the Elder Scrolls Oblivion and Skyrim, players may arrive at a point in the game where the invisible walls of game limitations can literally freeze them in their tracks as they reach the borders of the game world [14].

Since these game limitations are caused by the difficulty in predicting the manner in which players will explore a game world, some developers use procedurally generated media to create worlds that are crafted with algorithms, rather than artists, as the player continues to explore the game world. Developers spend less time on hand crafting specific game areas and instead maintain general media that can be used to create new and unique types of locations with as much variety as development and execution time permits.

These procedurally generated worlds escape the problem of elongated development times resulting from the intense artistic efforts required to develop a meaningfully large game world; however, these development efforts continue to suffer from hardware restrictions and limitations. As a player explores an ever expanding, procedurally generated game world, the size of the data required to describe that world grows as the size of the world expands. In a game such as Minecraft, a player can create and travel in any cardinal direction in what might appear to be an infinite space [8]. However the data that describes the world must be stored and retrieved, resulting in a game world that is limited by the existing hardware. Also, since a game world is typically described by a set of coordinates, players that venture too far from the point of origin may incur difficulties resulting from these hardware restrictions. The creator of Minecraft, Markus Persson, has documented situations where a
player has wandered too far from the original starting point in Minecraft, resulting in game anomalies, inconsistent functionality, and failure of algorithms that are central to Minecraft gameplay [11].

The research that we describe in this paper attempts to address some of these issues by removing the static nature of the procedurally generated world, and making most aspects of the world dynamic. This results in a game space that is constantly changing as the game is played in real time. By reducing the elements of a \textit{first person shooter} into a world consisting of rooms and doors, our approach attempts to create a starting point for further research and development of dynamically created game worlds.

To support the research an example game design has been conceptualized that takes full advantage of the dynamic game world structure. The concept game centers around the notion of a player as someone who enters the mind of mentally sick individuals in an attempt to eliminate the source. To illustrate the dynamic world and provide a proof of concept, a prototype game world was designed and implemented. The game world centers around the notion of a player as someone who enters the mind of mentally sick individuals in an attempt to eliminate the source of their mental strain. The mental space the player explores resembles a house that is remembered by the victim being explored. Since the game deals with memories and not physical places the house can be of any size or configuration and, like the hazy nature of memories, the rooms and other artifacts of the house need not always be in the same locations. The genre in which this game is placed must be one that compliments this dynamic approach to developing game worlds. For our prototype concept game, a \textit{survival horror genre} was selected. In the survival horror genre players are engaged in situations where they have limited resources and skills for defending themselves and the world is typically populated with adversaries that are difficult to defeat. Also, a key idea in the survival horror genre is that usually the environments themselves are
designed in a way to make the player uneasy and unable to continually predict what will occur next or where it will occur.

The notion of a game world that is constantly shifting has historically been used to create the uneasy feeling that is present in the horror genre. The goal of the game is to explore the shifting rooms of the house and find key items that relate to the events the sick patient has suffered through. Cryptic notes might be given to the player to encourage them to find these key items, as illustrated in figure [1.1]. While searching for these items players typically need to avoid traps, solve puzzles, and negate special nightmare guardians whose only goal is to keep these horrible memories from escaping, as illustrated in Figure [1.2]. The nature of the non-static level design ensures that the player running from these nightmares should have an endless supply of rooms to navigate, being forced to quickly react in locating doors and to avoid locked doors and traps. The game design structure that this example game might employ would be that of what Penny De Byl described as a *hub and spoke*, which is based on the idea of a central hub location that the player returns to after each challenge [2]. In our example game this would be a reoccurring safe room where the player can remain, confident that none of the nightmare creatures haunting him can enter.

After completing the required tasks, the player is cleared of bad memories and nightmares, and the game can end in a successful conclusion.

In the next chapter, we provide background to facilitate comprehension of our work. In Chapter [3] we describe the research and games that illustrate the horror genre. In Chapter [4] we describe our set up and use of the UDK, and in Chapter [5] we describe our methodology in building the prototype game implementation. We then describe the salient features of our prototype game and, finally, we draw some conclusions and describe future work.
Figure 1.1: The player might be given cryptic notes telling them what items they are supposed to find around the game world. As can be seen the game would be in a first person perspective to keep visibility low and tension high.

Figure 1.2: A player is shown peeking around a corner to see a nightmare guardian in the room. These guardians serve as an unstoppable force that the player must avoid during exploration of the game world.
Chapter 2

Background

The research described in this paper addresses the problem of algorithmically creating worlds that incorporate game design practices that have been around since the beginning of gaming history. This section looks at one of these practices, grid-based games, that represents a key focus of this research, motivates the tools that we chose to use, and discusses some of those we chose not to use.

2.1 Games and Grids

In video game design, grid’s provide designers with the means to efficiently handle game world information in a way that is easy to store, retrieve, and navigate. The basic constructs used to build grids can take a variety of shapes, but the most common shapes are squares, hexagons, and triangles [10]. Because of the nature of the content incorporated into the games that form the target of this research, we chose a square shaped grid for generating game content. The game space for this thesis is composed of rooms and doors. Since practical architecture usually relies on simple rectangular designs, the cube based nature of a uniformed square grid fits perfectly for this type of layout.

2.1.1 Zelda (1986)

The Zelda [9] game genre exemplifies a world based on a grid structure. Each part of this grid is either walkable or not, permitting the player and the enemies to either pass through the grid area or not. For example, Figure 2.1 illustrates a grid that is represented visually using a limited number of art assets; these art assets are illustrated in Figure 2.2, which lists the 144 tiles available as art assets for the environment in Zelda’s surface world. Zelda uses it’s grid structure to limit the total data size of the game while giving the player a massive hand-built world to explore.
2.2 Unity and Houdini

One of the original attempts at creating procedural worlds attempted to blend Houdini with Unity to generate playable levels in real time. Houdini has a robust system for procedural generation of architecture and there are quite a few examples of individuals using Houdini for generating building interiors\[5\]. Unity is an open-ended 3d gaming system with a strong user base and some cross platform support. The basic idea was to exploit Houdini’s facility to be accessed through a python shell \[12\] and a Unity plug-in that used python scripts in an attempt to combine the two programs in real time. The advantage to this approach was having a very flexible generation tool for non-standard architecture.

The main problem with this approach was the scope of the Zelda world, and the time required to build such a world. The process of getting this technique to work would involve learning both Houdini and Unity from the ground up, evaluating how much CPU time each program requires to generate items, and determining if it is even possible to create, export,
and load game data from Houdini into Unity. Due to these constraints, we decided to focus on a single game framework with a wider range of techniques available in the development environment.

### 2.3 The UDK 3.5

The Unreal Game Engine is a popular 3d framework currently being used in over 100 games ranging from AAA titles, such as Gears of War, Bioshock, and Borderlands, to smaller game titles such as Antichamber [21]. The Unreal Engine features three levels of programming. In the first level, there is the core engine that is written in C++. This part of the game engine is responsible for all computationally intense tasks such as 3d rendering and lighting, physics, and other computationally intense elements [20]. The second level contains the scripting language, UnrealScript, that allows programmers to manipulate base game functionality as well as programing NPC AI, weapons, and other elements of gameplay that isn’t incorporated into the core Unity engine. The third level is a node based coding language known as Kismet. Kismet is similar to UnrealScript in its range of uses and also has the ability to work in sync with UnrealScript.
2.3.1 Rationale for Using the UDK

The history of the Unreal engine began in 1998 with the release of the game *Unreal* and *Unreal Tournament*. As the series progressed the engines performed well, giving rise to Unreal Engine 2 in 2002, and then Unreal Engine 3 [20]. The success of the Unreal engine has helped to build a strong community of developers willing to share information. Also with such a positive history, the unreal engine has been widely used in venues other than gaming, including as TV shows, software applications, and training programs [20]. Most importantly, the coding structure has a small learning curve. The class structures of the scripting system, UnrealScript, is clearly explained on the UDK developer homepage, which serves as a launch point for application development.

2.4 Flash

Flash is a 2d application developed by Adobe for use in creating interactive web content. It features the scripting language Action Script that allows users to produce movement, interactivity between program and user, and other essential functions. Most importantly for this research it allowed the creation of rapid prototypes that could then be translated into UDK.

2.4.1 Why was Flash needed for this research?

The UDK coding system takes time to be compiled for each change that is made and also requires that the leveling editing software be reloaded each time as well. Because of this, coders who prefer to write small bits of code at a time are at a huge disadvantage. Not only is Flash’s work environment friendly to these types of coders, but it’s scripting structure also shares aspect with UDK’s UnrealScript that were important to this research:

- Both programs have a user created Library of assets they pull from.
- Both only support single dimension arrays.
• Both have a scripting language based over an inaccessible hard coded language.

• Both scripting languages are object orientated.

• Both share similar variable types.

• Both are heavily used for games.
Chapter 3

Related Work

The use of procedurally generation to create environments in games is becoming more prevalent in main stream industry and some of the best selling video games in the past few years have featured some use of procedural generation[18]. In this section, we consider three games and one prototype that feature procedurally generated game worlds in ways that inspired and motivated the research presented in this paper.

3.1 Rogue (1980)

*Rogue* is an exploration game where the player is tasked with finding exits in a game space that is different each time they play[16]. The game is famous for its harsh punishment of permanent death (where the game must be started over each time the player dies), its randomized items, and of course its procedurally created levels. Using a graphic system comprised of computer ASCII character’s, *Rogue*’s game world is a series of rooms and hallways as can be seen in figure 3.1. *Rogue*’s room and hallway generation is almost node based, with each room that is generated being flagged as connected or not connected. The program keeps connecting rooms to unconnected rooms until there are no more rooms left unconnected[6]. Many copies of this game style are being made even today and has given the game industry a completely new genre known as rogue-like[19].

3.2 Dynamic Dungeon Prototype (2009)

In 2009 the concepts of this research paper were already being tested by this research team on the ideas on dynamically generated game worlds. This early attempt at a prototype, seen in figure 3.2 is a rogue-like game where the pieces of the world always change when they are no longer in the player’s sight[7]. The basic premise was to reach a special location on
Rogue is a game that eventually would lead to a new genre in gaming called "rouge-likes". Focused on exploration, players explore procedurally generated dungeons visualized by ASCII characters.

As the world blocks would come into the player’s field of vision there was a 50/50 chance that the block would either be a passable floor or an impassable wall. If a player found themselves stuck they could go back and forth until a path seemed to open. This prototype served as an example of a constant dynamic game world and how the player can be a part of the world generation in real time.

3.3 Fallen Enchantress (2012)

Fallen Enchantress is a turned based strategy game that focuses its random generation on open world maps very similar to how Rogue works; however, the unique functionality of the technique in Fallen Enchantress is relevant to our research. Fallen Enchantress has a 3d world whose map system is based on a 2d grid layout as shown in figure 3.3. Like Rogue, Fallen Enchantress has a simple distinction between walkable areas and non-walkable areas: land and water. What makes Fallen Enchantress different from Rogue can be seen in
Figure 3.2: Screenshot of a dynamic game world prototype. Each time a piece of the world goes out of the player’s view it has a 50/50 chance of being a wall or floor piece.

Figure 3.4 in that *Fallen Enchantress* uses a set of prefabricated pieces to give a smoother transition from land to water versus *Rogue*’s dramatic cutoff between walls and open areas. These prefabricated pieces in *Fallen Enchantress* are designed to connect together seamlessly no matter how they are put together. Inspired by this technique, our research uses a similar system to handle the generation of walls for rooms.

### 3.4 Faster Than Light (2012)

*Faster Than Light* (FTL) [4] is a game where the player takes the roll of a star-ship captain guiding his/her crew on a suicide mission from one side of the galaxy to the other. FTL is a game filled with random events that give the player opportunities to make tough calls, avoid or engage in combat, and interact with the universe other various ways. However, the aspect of the game that is relevant to our work is that the game also gives the player the ability to choose the difficulty of the game while playing. Once reaching the end (or jumpgate) of the current planetary system the player is actively in they are given a choice of a few out-branching paths to take. These paths are randomly generated each time a
Figure 3.3: *Fallen Enchantress* is a turn-based strategy game that uses procedural generation to create new maps each time a game is played. The map is a system of terrain that is either passable or not passable.

game is started, consisting of Hostile systems (red), Friendly systems (green), and more unpredictable Nebula systems (purple). Although the player is stuck to certain choices they are still given the option to choose their own path. Traveling on a path that is almost all red will give the user a much harder experience, but one that may be more rewarding. Choosing a path that is mostly green the player will still have conflict but on a more manageable scale, but might leave some players bored. This mixture of limited choice paths was inspirational in how we approached the room and door design of our final prototype.
Figure 3.4: A visual demonstration of how *Fallen Enchantress*’ maps are based on a 2d grid with connecting 3d prefabricate land and water pieces making up the visual aspects of the world.
Chapter 4

Setup and Use of the UDK 3.5

Utilizing a 3d application for this research was critical from the beginning. The reason behind this was two fold. One, we felt that creating a constant dynamic game world was something that could easily be accomplished in a 2d based game. Adding in the extra dimension of space that 3d games present creates a lot of consistency problems that 2d games can avoid. The extra work required to make a 3d procedurally generated world look cohesive was a desirable challenge in this research. Two, procedurally generated worlds are fairly common in 2d games and it has only been recently that these concepts has found their way into the realm of 3d gaming [18] and the opportunity to explore a less charted territory was a great incentive for this research. The following section demonstrates what was needed to begin working with the 3d software chosen for the research, the Unreal Development Kit 3.5.

4.1 New Project

Getting a new basic project up and running for UDK 3.5 takes a little bit of navigating so it was thought to be valuable to share the process. Epic Games is kind enough to have a brief tutorial on the subject [3] and the following instructions will be a mix of their references and experience from this research.

1. Locate the directory UDK/Development/Src/. This is where all the UnrealScript files will be stored for the new project.

2. Decide on a name. From this point on the name of the new project will be very important to keep consistent as most of the new UnrealScript files will need to start with this name. For the sake of simplicity we will call our project "NewProject".

3. In UDK/Development/Src create a new folder named "NewProject".
4. Create another folder in the NewProject folder called "Classes". This is where the UDK compiler will look for our UnrealScripts.

5. The UDK compiler will not automatically see our new project folder so we will need to manually tell it that our project is there. Use your favored text editor and open up the file UDK/UDKGame/Config/UDKEngine.ini.

6. In UDKEngine.ini look for the section marked "[UnrealEd.EditorEngine]". Here we should see a list of packages Unreal uses when it compiles. At the end of this list use the command "ModEditPackages" and equal it to our project name. The result should look like this: "ModEditPackages=NewProject".

7. Save the changes made to UDKEngine.ini. The UDK compiler now knows to compile our project along with everything else.

8. Lastly we need to setup a GameInfo class in our NewProject/Classes folder. The GameInfo class is where most scripts in UDK start. In a text editor create a new file called "NewProjectGame.uc" and save it in our Classes folder. Notice how the name of our project is present in the class name. This is where naming convention starts to become very important.

9. The code for a new basic GameInfo class is very simple:

```plaintext
class NewProjectGame extends GameInfo;
```

Basically, we are taking the preexisting GameInfo class and making a new one specifically for our project. Here we can start changing aspects of the game and setting up other key files for our project.

### 4.2 Preparing 3d Assets for UDK from Maya

For our research Maya was used to create the 3d assets needed. The following instructions give detail on how to setup Maya to be compatible with UDK as well as how to prepare the
3d models to be imported into UDK’s asset library.

1. In Maya, go to Preferences->Settings and make sure your “up axis” is set to Z. If you don’t do this all your models will be sideways when you import them into the UDK.

![Figure 4.1: Maya’s preferences window.](image)

2. Unreal’s unit of measurement is the same as Maya’s default measurement so there is no need to change that.

3. Create or load your desired model.

4. Name your model. (see Figure 4.2)

![Figure 4.2: mayaBaseModel](image)

5. Create your bounding box. A bounding box is a set of simple geometry that represents where you wish a player or object to collide with the current model. UDK runs at its best with bounding boxes that are simple in shape. It is much better to have a number
of simple shaped boxes then it is to have a single mesh with tons of complicated angles.

6. Name bounding box the same name as your base model but with UCX infront of it. This lets Unreal know that the model is a bounding box and what model it is bound to (see Figure 4.3).

![Figure 4.3: mayaBaseCollision](image)

7. Select both your base model and bounding box model and export as a FBX file. Be sure to check the "include media" option in the FBX exporter if you want the textures on the model be there in UDK.

8. Import the FBX file into the UDK asset library and UDK should automatically recognize the different parts of your model.
Chapter 5

Methodology and Implementation

The goal of our research is the development of a prototype that provides the player with a set of game play options that can be followed or ignored at the player’s discretion. Our intention is that the player will have a limited set of procedural paths that are defined by the placement of doors, but the player is virtually never required to follow a specific direction. Figure 5.1 displays how player choice effects the flow of the program. As the player travels from room to room, opening and closing doors, previous rooms they have encountered are deleted and new ones created as needed. In this section, we describe our approach toward realizing this goal, including a description of the algorithms that we developed and incorporated into the UDK.

Figure 5.1: Flow diagram of the core algorithm for the research incorporating player choices.
5.1 GameMode.uc

GameMode.uc is the first scripting class called at the start of any UDK game. The class is set as an extension of GameInfo.uc which sets up a default first-person camera and control scheme and basic physical influences on the player such as gravity. In the defaultProperties function the PlayerController.uc is set, which houses the core of the code used in this research.

5.1.1 Class Setup

```uc
class CubeGameMode extends GameInfo;

defaultProperties(){
    SetPlayerControllerClass to our custom PlayerController.uc.
}
```

5.2 PlayerController.uc

PlayerController.uc gives coders access to functions that are directly involved with the player’s activities such as movement, jumping, clicking, etc. It is in this class that rooms are built and destroyed.

5.2.1 Class Setup

```uc
// Class Structs

struct pos{ // This makes it easy to assign and access locations in the imaginary grid.
    var int x;
    var int y;
};

struct gridPiece{ // Holds all key data a room piece can have.
    var pos rPos; // Position of room in imaginary grid.
    var int rType; // Room Type. 0 = room, 1 = door.
    var int wType; // Wall type.
    var int rNum; // Unique Room ID Number.
    var int rNumDoor; // Only used if piece is a door. Holds connecting room rNum.
    var bool doorDir; // Which direction a door is facing. True = up/down, false = left/right.
    var Actor aHold; // This Unreal based variable holds all the model data that is shown in the 3d world.
```
var bool deleteMe; // Bool to signal if this gridPiece needs to be reset during room deletion.

};

// Class Variables
var private array<gridPiece> gridArray; // Holds all rooms and is the base of the game grid.
var private int roomCount; // Used to give all rooms created a unique id number.
var private int gridSize; // How many Unreal units each grid piece is in size.

preBeginPlay(){ // Function called before game begins.
  Set the size of the gridWidth and gridHeight. (currently 25x25)
  Set gridSize to 256. (each piece of the grid is 256 Unreal units in size)
  For the gridHeight and gridWidth add a gridPiece to gridArray.
  Set roomCount to 1.
  Setup the first room in the middle of the grid. Give it four doors.
}

5.3 Room Generation

The situation players are presented in this research are a series of rooms connected by doors. Players have option of choosing which door they wish to enter or to ignore.

5.3.1 translateDoor()

When a player opens a door the translateDoor() function is called to decode which gridPiece has been clicked on and, if the piece is a door, and what the appropriate next action should be.

private function translateDoor(int tX, int tY){
  var int r = Decode tX and tY from a 2d array to a 1d array.
  Check to see if gridArray[r].rNumDoor equals 0.
  If no: exit function. Else Continue.
  Check to see if door goes up/down or left/right.
  Based on results above check both directions for an open space.
  Set gridArray[r].rNumDoor to roomCount.
  call SetupTestRoom(x,y) with appropriate coordinates.
}

5.3.2 setupTestRoom()

private function setupTestRoom(int tX, int tY, int tD = 1){
var int r = Decode tx and ty from a 2d array to a 1d array.
Set gridArray[r].rType to 0 for a room.
Set gridArray[r].rNum to roomCount.
For tD call placeDoor();
Call buildRooms();
++roomCount;
}

5.3.3 placeDoor()

private function placeDoor(){
Create var array<int>tempStore.
Go through gridArray[] and capture to tempStore[] any grid piece whose
rNum ID number is the same as roomCount and also is not a door.
Randomly go through tempStore and test each grid piece in all
directions around it if a valid door can be placed.
Let r = this valid door location.
Once a suitable location for the door has been found call
setupDoorGrid(r,direction); // direction is bool dependent
On if the door goes up/down or left/right.
If not suitable door location is found exit function.
}

5.3.4 setupDoorGrid()

private function setupDoorGrid(int r, bool dir){
Set gridArray[r].rType equal to 1, marking it as a door.
Set gridArray[r].rNum = roomCount.
Set gridArray[r].doorDir = dir.
Set gridArray[r].rNumDoor to 0 since its currently not connected to
another room.
}

5.3.5 buildRooms()

private function buildRoom(){
Create var array<int>tempStore.
Go through gridArray[] and capture to tempStore[] any grid piece whos
rNum equals roomCount.
Go through tempStore[] and see if a grid pieces rType is 0 or 1.
If rType is 0: generate a room block in UDK.
if rType is 1: generate a door block in UDK.
5.4 Room Removal

5.4.1 Constant Deletion

Constant Deletion is the current implemented method of removing rooms from the game. This method is called each time a new room is made. A safelist is created which then holds the rNum of the door that was activated and the new room that will be created. Effectively, what this process does is save the room the player is currently in and the new room being created while deleting all over rooms in the grid.

**setDelete()**

```plaintext
private function setDelete(int tX, int tY){

var int r = Decode tX and tY from a 2d array to a 1d array.
Create array of ints called safeNum.
Add the rNum from gridArray[r] to safeNum[].
If gridArray[r].rNumDoor does not equal 0 then add that value to safeNum[].
Go through gridArray[] to look at all the grid pieces.
If the current pieces rNum in not 0 and is a room:
Set pieces delete flag to True.
Compare the pieces rNum to that of the ones in safeNum[].
If any are the same set pieces delete flag back to False.
Else if the current pieces rNum is not 0 and is a door:
Set pieces delete flag to True.
Compare the pieces rNum to that of the ones in safeNum[].
If any are the same, set pieces delete flag back to False.
Compare the doors rNumDoor to safeNum[].
If none of the safe numbers are equal to rNumDoor:
Set rNumDoor to 0.
Call deleteRoom(r) if the pieces deleteFlag is still True.
}
```

**deleteRoom()**

```plaintext
private function deleteRoom(int r){
Set gridArray[r].aHold to be erased using the Destroy{} command.
Reset all other values back to 0.
}
```
5.4.2 Connected Deletion

Connected Deletion is very similar to constant deletion except that we add more numbers to the safe list. We add these new numbers by stepping back through doors and checking which rooms they are connected to. What this gives us is a web of rooms that are saved instead of just the current room and the new room, like in Constant Deletion. Below is a simple adaptation of this theory.

```plaintext
private function connectDelete(int tX, int tY) {  
  var int r = Decode tX and tY from a 2d array to a 1d array.  
  Create array of ints called safeNum.  
  Create array of ints called tempDoors.  
  Go through gridArray and add any doors to tempDoors[] who's rNum or rNumDoor are equal to gridArray[r].rNum.  
  Add the rNum from gridArray[r] to safeNum[].  
  Go through tempDoors[] and add the rNum and rNumDoor to safeNum[].  
  Go through safeNum[] and delete any duplicate numbers.  
  Go through tempDoors[] to look at all the grid pieces.  
  If the current pieces rNum in not 0 and is a room:  
    Set pieces delete flag to True.  
  Compare the pieces rNum to that of the ones in safeNum[].  
  If any are the same set pieces delete flag back to False.  
  Else if the current pieces rNum is not 0 and is a door:  
    Set pieces delete flag to True.  
  Compare the pieces rNum to that of the ones in safeNum[].  
  If any are the same, set pieces delete flag back to False.  
  Compare the doors rNumDoor to safeNum[].  
  If none of the safe numbers are equal to rNumDoor:  
    Set rNumDoor to 0.  
  Call deleteRoom(r) if the pieces deleteFlag is still True.  
}
```

5.4.3 Random Deletion

Random Deletion is also similar to Constant Deletion but the process is essentially delayed. Each time a new room is generated by opening a door, a class static variable has a value set by the programmer added to it. This static variable is compared to a random number and if a requirement is meet then the delete method is called. What this grants is a world where rooms stay constant for a while, but can suddenly shift without warning.

```plaintext
// Since Unrealscript doesn't support static variables we will need to make a new class variable in PlayerController.uc.  
var private int classVarRandDel;
```
5.5 Room Recovery

Down time is always apart of a well balanced game and plays an important role in giving the player a rest from intense action or the stress of wondering what is around the next corner [17]. Some sort of recurring room, or rooms, that the player knows is safe can fill this role of down time in world of every changing room layouts. It would also give some cohesion to the game world if players felt they were revisiting areas they had already been to. The time constraints of this research have left recovery on the back burning and although the algorithms haven’t been worked out for these functions, some basic concepts behind them have been.

5.5.1 Pseudo Recovery

Pseudo Recovery involves using a per-determined room created by the game developer that would occasionally be generated into the world instead of a randomized room. It would depend highly on where the room could fit and the frequency in which the developer wants the room to appear.

5.5.2 Whole Map Recovery

The idea behind whole map recovery is having a second array that represents the game world grid, but instead of holding all the information the normal world grid holds, this one would just hold room numbers and prefab information to be recalled when it would fit.
These phantom rooms would be called in occasionally when they fit to gives the player some sense that they are crossing paths they have ran before.

5.5.3 Single Room Recovery

Single room recovery would be similar to the whole map version but the grid used would be much smaller. Instead of capturing the whole map, this grid array would capture the information for just one room. This information would then be recalled at a random or predetermined time to once again give the player a feeling of crossing their own path.

5.6 Wall Generation

With the creation of a 3d world based on a grid there are two major ways to handle the 3d game assets of the procedurally generated rooms. One is the follow Minecraft’s example where the rooms are generated by empty space and the walls are the remaining solid blocks built within the game grid. This method results in very blocky looking rooms that tend to lack in personality. The second method, and the way the research chose, is the opposite of this where room pieces are created that fit into place like a puzzle. The result is that the majority of the game grid remains empty and the 3d game assets are just where the rooms are located. This method gives the option of creating much more detailed room architecture that still fits into an easy to read grid structure. This is the method seen in Fallen Enchantress in figure 3.4 where the game map is comprised of interlocking land/sea pieces. This method does require to have an algorithm that can take data from a cell and its surroundings to see which prefab type would be needed and convert this to a single easy to reference number.

5.6.1 Getting a unique reference number

Getting a unique reference number for each wall situation involves taking each grid piece before a 3d asset is chosen and checking its surroundings. Each side of the grid piece is given a value. North = 1, East = 10, South = 100, West = 1000. This can be seen more
clearly in figure 5.2. Whenever a grid side is facing an empty space or another room the corresponding side value is added to the grid pieces reference number. The results gives a unique number for different wall configurations. For example if a grid piece is in the middle of a room it’s reference number would be "0" because no values would have been added. In contrast, if a grid piece is completely surrounded by empty spaces it’s reference number would be "1111". More combinations of these values and results can be seen in figure 5.2.

Once this reference number is established the 3d wall assets would need to be sorted.
through a database and selected based on the type of room being constructed. The results of this process can be seen in figure [5.3] where in our Flash example we see two rooms defined without walls and then both rooms with walls after being run through the wall generation algorithm.

5.6.2 setupWalls()

```plaintext
private function setupWalls(){
  Create local var int refNum and set it to 0;
  For the length of gridArray[]
    Check the grid piece north of current grid piece.
    If space is blank or is a room with an rNum different then current grid pieces rNum:
      refNum += 1;
    Check the grid piece east of current grid piece.
    If space is blank or is a room with an rNum different then current grid pieces rNum:
      refNum += 10;
    Check the grid piece south of current grid piece.
    If space is blank or is a room with an rNum different then current grid pieces rNum:
      refNum += 100;
    Check the grid piece west of current grid piece.
    If space is blank or is a room with an rNum different then current grid pieces rNum:
      refNum += 1000;
  Use this newly calculated refNum and choose an appropriate corresponding 3d asset.
}
```
Chapter 6

Case Study

In the previous chapter we have described our methodology for basic room creation and deletion in the UDK. Although the rooms are represented by a single grid piece, the abstracted rooms and doors give the foundation for expansion into more realistic and visually exciting game spaces. In this chapter, we demonstrate the results of our prototype and provide a step by step walk through of a typical player experience.

6.1 Walk-through

The following section demonstrates a walk though of the working prototype and a typical situation a player would experience that demonstrates the basic aspects of the dynamic realtime level creation process. Each step of the process is represented by a top down view (on the left) of the game world grid and a first person view (on the right) of what the player is seeing in the actual game. Figure 6.1 shows the key for the top down view where a white square represents a blank space, yellow square a room block, and red square a door block. The player is represented by the black point with a line extending out to showing their facing direction.

![Figure 6.1: Key for top down view walk-through example.]

1. In figure 6.2 the player is facing north east standing on a room block and surrounded
by 4 doors.

Figure 6.2: Base room is created and player is spawned into world.

2. In figure 6.3, the player turns to the east. Their cross-hair aligned with the door east of them.

Figure 6.3: Player looks at East door of current room.

3. In figure 6.4, the player clicks on the east door. A new room block is generated to the east of this door and a new door is placed facing south.

4. In figure 6.5, the player walks east into the newly created room block and turns facing south-west. In their view they can still see that the previous room consisting of one room block and 4 doors is still in the game world.

5. In figure 6.6, the player turns to the south. Their cross-hair aligned with the door south
6. In figure 6.7, the player clicks on the south door and a new room block is generated to the south of this door. A new door block is generated to the east of this newly created room block. In the top down view on the left it can be seen that the old room block to the west the player had started in has been removed, as well as all the doors that were not attached to the room block the player is currently in.

7. In figure 6.8, the player turns to the west. They now see that the room and doors to the west are now gone from the world.
This chapter has given a glimpse into the final workings of the research presented in this paper. Although the basic process is quick in nature, by the creation of room blocks and door blocks a player is able to navigate a 3d game world that is constantly changing in an endless fashion. The prototype created is ready to be expanded and such expansion ideas are explored in the next section, Chapter 7.
Figure 6.8: Player looks back at Starter room and see’s it has been deleted.
Chapter 7

Future Work and Conclusions

7.1 Different room types into UDK

As of right now the rooms in this research all of our rooms and doors are represented by abstract blocks. Although these blocks go to prove the basic algorithms of this paper work it would be nice to expand into different types of room structures. Even expanding into basic shapes such as long narrow hallways, rectangular rooms, and closets would go along way to improve the visual connection to the world.

7.1.1 Basic Walls in UDK

Once different types of rooms were up and running it would be ideal to take the algorithm developed for the flash room prototypes for walls and apply it to UDK. Converting the code is simple enough, but what will take a bit more effort is getting the 3d assets developed and tested, such as the corner wall in figure 7.1.

7.1.2 Wall and Room Styles

Further down the road when walls and basic rooms are implemented it would be time to see what kind of variety of looks could be accomplished to set whatever mood was needed for the game space. Since the different pieces of rooms are brought into the world as prefabs their complexity can be quite high and will still be able to blend together seamlessly.

7.2 Filling the Rooms

Once basic room structures were implemented it would be idea to then come up with a realistic and procedural way to fill the rooms with content. A certain amount of imperfection
would be needed to create a room setting that didn’t feel to computer assembled and perhaps translating the research of Joshua Taylor and Ian Parberry on clutter [15] in a 2d space to a 3d space would be a great place to start.

7.3 After Thoughts on UDK 3.5

Through out this research we utilized the Unreal Development Kit 3.5 as the 3d engine for our generation of rooms. The UDK proved more then capable of handling the prototype created, responding well to the task of creating and delete 3d assets in realtime. It is unknown how UDK would handle itself once more complicated geometry was used. Another concern is the required heavy use of dymanic lighting. Currently the prototyophe uses a single dynamic light from a sky like position. Because the UDK is built primary to use baked lighting to render its more complicated scenes having a world comprised of multiple
dynamic lights may cause issue. The UDK performed well with our grid based method of room generation, but for the sake of procedural generation it would be ideal to have an engine that would allow us to build walls and rooms vector by vector. Coding with UnrealScript was effective, versatile, and easy to read/write.

7.4 After Thoughts on UDK 3.5

Through out this research we utilized the Unreal Development Kit 3.5 as the 3d engine for our generation of rooms. The UDK proved more than capable of handling the prototype created, responding well to the task of creating and delete 3d assets in real-time. It was not tested how UDK would handle itself once more complicated geometry was used, though.
Another concern is the required heavy use of dynamic lighting. Currently the prototype uses a single dynamic light from a sun-like position. Because the UDK is built primary to use baked lighting to render its more complicated scenes having a world comprised of numerous dynamic lights may cause issue. The UDK performed well with our grid based method of room generation, but for the sake of procedural generation it would be ideal to have an engine that would allow us to build walls and rooms vector by vector. Coding with UnrealScript was effective, versatile, and easy to read/write.

»»»> .r37

7.5 Conclusion

In the end this research has shown that it is possible to use UDK in creating grid based procedural worlds with simple geometry and that UDK has the capability to handle constant creation. This paper lays some ground work on a possible method for its creation and a solid direction for its future development.
Bibliography


