Contents

Introduction .................................................................................................................. 2
Technical Overview ....................................................................................................... 2
  Target System Requirements ....................................................................................... 2
  Tools ............................................................................................................................ 3
Engines & Middleware .................................................................................................... 3
File Formats ................................................................................................................... 4
Technical Features ......................................................................................................... 5
  Custom Game Type ..................................................................................................... 5
  Rune System ................................................................................................................. 5
    Implementation A – Base the rune class on Pickup class ........................................ 5
    Implementation B – Base the rune class off CTF Flag .............................................. 6
    Implementation C – Base the rune Class on the Bombing Run Ball ....................... 6
    Implementation D – Base the rune class on Actor ................................................... 6
    Final Decision – Implementation A .......................................................................... 7
Altar System .................................................................................................................. 7
  Implementation .......................................................................................................... 7
Troll & Dwarf Pawns ....................................................................................................... 7
  Implementation .......................................................................................................... 7
Draught of Life ................................................................................................................ 8
  Implementation A ....................................................................................................... 8
  Implementation B ....................................................................................................... 8
Gods ............................................................................................................................... 8
  Implementation A – Custom Pawn Class for the Gods (preferred) ....................... 9
  Implementation B – Stationary Object as Gods (backup) ......................................... 9
Weapons ....................................................................................................................... 9
  Mjolnir’s Echo ......................................................................................................... 9
  Tears of Freya ......................................................................................................... 10
  Tyr's Cannon ........................................................................................................... 10
Custom Levels ............................................................................................................. 10
  Implementation ......................................................................................................... 11
HUD .............................................................................................................................. 11
Custom Menu and GUI ................................................................................................. 11
Development Plan ......................................................................................................... 11
Milestones .................................................................................................................... 11
  GDD and TDD complete ........................................................................................... 11
  Asset List and Development Plan Complete .......................................................... 11
Prototype .................................................................................................................... 12
  Alpha .......................................................................................................................... 12
Beta .............................................................................................................................. 12
  Gold ............................................................................................................................ 12
Software Architecture ................................................................................................ 12
  Directory Structure .................................................................................................. 12
  Build Process .......................................................................................................... 13
Coding Standards ....................................................................................................... 13
Introduction

Great Winter is a multiplayer, team-based game modification (mod) for Epic's Unreal Tournament 2004. Two teams representing rival factions in Norse mythology struggle to gain control over power-ups and destroy each other's god. The game style of Great Winter resembles Onslaught more than any other existing game style packaged with Unreal Tournament 2004. In Great Winter, the team power cores are replaced by their respective gods, and the power nodes are replaced by altars upon which team-based runes may be placed. When a player carries a rune to an altar, the altar powers up in the same way as an Onslaught power node; however, unlike a power node it grants the controlling team a special ability such as increased damage or speed. The concept of links between the nodes and cores is not relevant in Great Winter. Each team starts the game with their own fives runes, and the levels will have a minimum of three altars.

Technical Overview

Target System Requirements

Great Winter, like most Unreal Tournament 2004 game mods, will be developed for PC distributions of the game. Epic lists the following system requirements on the retail box:

- **Operating System:** Windows 98 / Me / 2000 / XP
- **Processor:** Pentium III 1.0 GHz or AMD Athlon 1.0 GHz or faster (1.2 GHz or faster recommended)
- **Memory:**
  - 128 MB RAM minimum
  - (256 MB RAM recommended)
- **Hard Disk Space:** 5.5 GB free space
- **CD-ROM Drive:** 8X Speed
- **Video:**
  - 32 MB Windows 98/Me/2000/XP-compatible video card (64 MB NVIDIA GeForce 2 or ATI Radeon Hardware T&L card recommended)
- **Sound:**
  - Windows 98/Me/2000/XP-compatible sound card (Dolby Digital Interactive Content Encoder required for Dolby Digital audio. Sound Blaster Audigy 2 ZS card recommended.)
- **DirectX:**
  - DirectX version 9.0b (included) or higher
- **Internet Connection:**
  - 33.6 Kbps modem for LAN/Internet play (Broadband Internet connection recommended)
**Tools**

All members of the development staff will have access to development computer with standard office programs such as Microsoft Word and Excel. In addition, all members will have access to the basic tools that accompany *Unreal Tournament 2004*.

<table>
<thead>
<tr>
<th>Users</th>
<th>Tool</th>
<th>Use</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art / LD/ SD</td>
<td>Workstation and base software</td>
<td>Dell XPS Gen 2 laptop with Microsoft office</td>
<td>$4000</td>
</tr>
<tr>
<td>Art / LD/ SD</td>
<td>Epic’s <em>Unreal Tournament 2004</em></td>
<td>Retail version of the game</td>
<td>$40</td>
</tr>
<tr>
<td>Art / LD/ SD</td>
<td><em>Unreal Ed</em></td>
<td>Level Editor, Package Editor</td>
<td>Included with game</td>
</tr>
<tr>
<td>Art / LD/ SD</td>
<td>Bugzilla</td>
<td>Server-based bug tracking</td>
<td>Free</td>
</tr>
<tr>
<td>Art / LD/ SD</td>
<td>Source Safe</td>
<td>Source Control</td>
<td>$60</td>
</tr>
<tr>
<td>Art/ LD</td>
<td>Autodesk’s 3DS Max 7</td>
<td>Character meshes and animation and static meshes</td>
<td>$3500</td>
</tr>
<tr>
<td>Art</td>
<td>Actor X Exporter</td>
<td>Export to <em>Unreal</em> formats</td>
<td>Free</td>
</tr>
<tr>
<td>Art/ LDs</td>
<td>Adobe’s Photoshop CS2</td>
<td>Textures and other 2D art</td>
<td>$650</td>
</tr>
<tr>
<td>Art</td>
<td>Wacom Intuos3 Tablet (6 x 8)</td>
<td>Input device for drawing</td>
<td>$330</td>
</tr>
<tr>
<td>SD</td>
<td><em>Unreal Development Environment 3.0</em></td>
<td>Script editing</td>
<td>Free</td>
</tr>
</tbody>
</table>

**Engines & Middleware**

Since *Great Winter* is a mod for an existing game, the actual game engine source code will not be needed nor will a third-party middleware solution be needed. The code will be written using the *Unreal Script* language and compiled with the tools that accompany the retail version of the game.
# File Formats

The following table lists the file formats used in the development of *Great Winter*:

<table>
<thead>
<tr>
<th>File Format</th>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animation</td>
<td>.psk</td>
<td>3ds skeletal animation</td>
</tr>
<tr>
<td></td>
<td>.psa</td>
<td>3ds mesh animation</td>
</tr>
<tr>
<td>Audio</td>
<td>.uax</td>
<td>Unreal audio</td>
</tr>
<tr>
<td>Maps</td>
<td>.ut2</td>
<td>Unreal map</td>
</tr>
<tr>
<td>Music</td>
<td>.ogg</td>
<td>Ogg Vorbis compressed audio</td>
</tr>
<tr>
<td>Static Meshes</td>
<td>.usx</td>
<td>Unreal static mesh</td>
</tr>
<tr>
<td></td>
<td>.3ds</td>
<td>3ds mesh</td>
</tr>
<tr>
<td>Scripts</td>
<td>.uc</td>
<td>Unreal Script class</td>
</tr>
<tr>
<td>System</td>
<td>.u</td>
<td>Unreal class package</td>
</tr>
<tr>
<td></td>
<td>.int</td>
<td>Unreal package resource and localization</td>
</tr>
<tr>
<td></td>
<td>.ucl</td>
<td>Unreal package interface</td>
</tr>
<tr>
<td></td>
<td>.ini</td>
<td>Unreal configuration file</td>
</tr>
<tr>
<td></td>
<td>.bat</td>
<td>System batch file</td>
</tr>
<tr>
<td>Textures</td>
<td>.utx</td>
<td>Unreal texture package</td>
</tr>
<tr>
<td></td>
<td>.tga</td>
<td>Targa texture</td>
</tr>
<tr>
<td></td>
<td>.bmp</td>
<td>Bitmap texture</td>
</tr>
</tbody>
</table>
Technical Features

Note: reading the Game Design Document before reading this document is essential.

Custom Game Type

The first thing needed is the new game type. Since the game works almost like Unreal's Onslaught game except the Nodes don't link together, and the 2 cores are replaced by gods. The most logical implementation is to base our new game code off Onslaught game class. Additionally, there are also the rune system, which need to be implemented and integrate into the rest of the Onslaught game code.

Rune System

The rune system is essentially a different kind of pickup. The rune system may be implemented by basing off one of the following Unreal's systems: Pickup, CTF Flag, Bombing Run Ball, or straight from Actor.

Implementation A – Base the rune class on Pickup class

The main benefit of modifying Unreal's pickup into our rune is the presence of Unreal's inventory system. When a pawn picks up an item, the item can either be placed in the pawn's Inventory or destroyed. This is very close to desired behavior, except that the runes are never destroyed, but set to different states. The runes can be held, at base, dropped or placed on an altar, and only one rune can be picked up at any time. Considering the abilities of Unreal script, creating and setting the states of the runes should not be difficult at all.

Another mechanism needed for the runes to be implemented is the runes’ effects on the team when a rune is placed on the Altar (discussed later). The effects include: more speed, more damage, more armor, regeneration and god's shield. This mechanism is something that needs to be implemented from scratch.

Benefit: Picking up and dropping pickups is already available. runes are logically pickups.
Implementation B – Base the rune class off CTF Flag

The similarity between the CTF flag and our rune system is that the flag can be picked up and placed at a specific location just like a rune; however, there is a major difference: there is only one flag for each team, while in our case there are five runes for each team. Moreover, there’s only one place for each team to place the flag, while in our case we need three places that do not belong to any team to place the runes.

Like in implementation A, the rune effects are still need to be implemented from scratch.

*Benefit:* The CTF flag is a lot like a rune: it can be held, dropped or placed at a base.

*Risk:* Although there are a lot of similarity between a CTF flag and a rune, there are some major differences as well. The CTF flag is designed specifically for the CTF game, which will probably require a lot of modification to make it work with another game type. In contrast, a pickup is more general and can be modified to suit any game type.

Implementation C – Base the rune Class on the Bombing Run Ball

Basing the rune on the Bombing Run ball is similar to basing it on the CTF flag, except there is only one ball in the game. We will probably run into similar problems, as the Bombing Run ball is designed specifically for the Bombing Run game, and would require major modification to make it conform to our specification.

Implementation D – Base the rune class on Actor

Actor is one of the lower level classes in unreal. If we cannot base our custom class on anything else, basing it on Actor would be our last hope.

*Benefit:* We can make an Actor into pretty much whatever we want.

*Risk:* Basing our rune on Actor gives us flexibility, but at the same time introduces a lot of risks and probably requires more time, because we would have to create almost everything from scratch.
**Final Decision – Implementation A**

First of all, it seemed to be most logical to base our rune class on *Unreal*'s Pickup class. Additionally, the work load involved in the implementing the rune’s states is less than that required to modify a CTF flag or a Bombing Run ball. *Unreal*'s pickup class also gives us a lot of flexibility where we need it, while Actor gives us too much flexibility, which could result in additional code.

*Resources:* 2-3 programmers, 10 different rune models, rune base.

**Altar System**

The altar system is already very similar to *Unreal*'s Onslaught node system. Basing the altar on Onslaught's node class is the most logical and most likely the easiest implementation.

**Implementation**

We will try to keep as much as possible of the original Onslaught node code and try to build other classes around it. The Node already has all of the functionality that we need, including the relation to the Link Gun, which we will later modify to be one of our weapons. All we need is to do is take away the linking mechanism of the node, and the altar is pretty much ready.

*Resources:* 1 programmer, 1 altar model.

**Troll & Dwarf Pawns**

Besides their appearance, Trolls and Dwarves are the same. Moreover, they function in a very similar manner to an unreal pawn. Hence, we only need to implement their appearance and animation.

**Implementation**

Creating a new character is very straight forward using *Unreal*. The most difficult part would be custom animation since the character animation and weapon animation has to match, which may be difficult. Moreover, both Trolls and Dwarves look very different, thus they will have different animation sets as well.
Aside from the animation, the character selection screen may need some modification as well.

Resources: 2 programmers, 3 artists, 2 character models.

**Draught of Life**
The Draught of Life (DOL) is essentially health pickup that is dropped by a pawn, where the amount of health is varied depending on the number of kills the killer has. The most logical implementation would be to base the DOL from Unreal’s HealthPickup class, or base it straight off Pickup like the Runes.

**Implementation A – Base off HealthPickup (preferred)**
Since Unreal’s pawn does not drop health pickup by default, the first thing that needs to be done is to try to spawn health pickup whenever a pawn dies. The next step would be to make the amount of health varies depending on the person who killed the pawn.

*Risk:* Risk here is quite minimal, the system seemed easy enough to implement with Unreal Script. The only tricky part is probably to get the number of kills to determine the amount of health.

**Implementation B – Base off Pickup (backup)**
This implementation is a backup plan for if by any reason, Unreal’s HealthPickup class does not allow us to do what we wanted. This way we can make our own pickup that match our specification, but will probably take more time.

Resources: 1 programmer, 1 artist, 1 pickup model.

**Gods**
The two gods, Loki and Odin, function the same, but there is nothing like them in the Unreal’s universe, which could make the gods the most problematic part to implement. The most logical implementation would be to base the gods’ class on Unreal’s pawn class with custom AI and a custom way point. Another plan, which should be easier, is to make the god a stationary object that attacks at random intervals, but this way we could not move the god as in original specification.
Implementation A – Custom Pawn Class for the Gods (preferred)

This is the preferred way. We need to implement the following: make the god bigger, add custom AI, a custom weapon, and a custom way point. The main game class also needs to know about the gods because when one of the gods is destroyed, the game will end. Moreover, the spawn points of both gods need to be place-able from Unreal Editor.

Additionally, the gods also need their own models and animations, which may be more problematic than those for the Trolls, and Dwarves because of their massive size.

*Risks:* Although the gods are essentially customized Bots, they behave very differently. There are a lot of things that can go wrong for the gods, especially with their AI. Unreal pawns can have very complex AI, most of which is not suitable for the gods. Finally, the size of the gods could introduce some problems as well.

Implementation B – Stationary Object as Gods (backup)

This is an easier way to implement the gods, but the gods will not move. All we need here is to base the god class on any stationary, destructible object; a turret would be a logical choice.

*Resources:* 2 programmers, 2 artists, 2 god models.

**Weapons**

There are three custom weapons: Mjolnir’s Echo, a hammer which functions similar to Unreal’s link gun, Tears of Freya, a crossbow which function similar to Unreal’s sniper rifle, and Tyr’s Cannon, a steam-thrower which functions similarly to Unreal’s flak cannon.

**Mjolnir’s Echo**

This is the most important weapon in the game because it’s the only weapon that can damage the gods, and heal Altars. Fortunately, the functionality of the Link Gun matches our specification (heals nodes, and make nodes charge faster). With all that works cut out, we just need to change the appearance of the link beam into something more like lightning, ideally something similar to the lightning gun.
The secondary area of effect attack is something that *Unreal* does not have, but that can be easily implemented. In *Unreal*, the Projectile class has hurt radius functionality already implemented for use with weapons like the rocket launcher. We could use the same functionality for our area of effect attack.

**Tears of Freya**

This is a sniping type weapon, which make it very simple to create. There is really nothing special about this weapon regarding implementation. All we need are custom meshes for projectiles and the crossbow itself.

The secondary attack is very simple as well; the only major different is the explosive projectile, which is really simple to implement.

The tricky part is how the weapon gets its ammo; the ammo can only be collected at a special charging station. Fortunately, the charging station can just be an ammo pickup and does not disappear.

**Tyr's Cannon**

This is an extremely close range weapon. It is very much like a flame thrower, but produces steam instead. Creating the visual for the steam could be tricky, but hopefully, not very difficult. *Unreal*’s flak cannon fires a number of projectiles at once. We can probably use the same attack and change the mesh into smoke particles. The only concern is that each steam particles has to get larger as it travels further then slowly disappear.

*Resources:* 2 programmers, 1 artist, 3 gun meshes and projectiles

**Custom Levels**

The levels need to be customized to suite the new gameplay. Fortunately, with *Unreal* Editor, it is very simple to enable the level designers to place the runes, altars and gods in levels.

*Resources:* 4 level designers, 3 custom levels
**HUD**

*Great Winter* has its own unique HUD with a mini map similar to the one in Onslaught. The only concern here is modifying the Onslaught map to meet our requirements.

**Implementation**

The HUD is nothing special, since the hardest part, the mini-map, is already done for us by the Onslaught HUD. The only things left for us to implement are custom graphics and the states of each rune. The player needs to know what state each rune is in; whether it is: held, dropped, on altar, or at a base.

*Resources:* 1 programmer, 1 artist.

**Custom Menu and GUI**

There are a lot of GUI elements in *Unreal Tournament 2004*, replacing them all would take too much time. Thus, only the main menu and loading screen will be customized. If time allows, more GUI elements may be customized.

*Resources:* 1 programmer, 1 artist

**Development Plan**

**Milestones**

**GDD and TDD complete**

Both documents are complete. The TDD contains coding practices and at least a general outline of how to implement the gameplay elements described in the GDD in addition to coding practices, tools, and risk analysis. Source control and bug tracking methods (Visual SourceSafe and Bugzilla) have been proposed.

**Asset List and Development Plan Complete**

An asset list with all assets and completion dates has been created. Source control and bug tracking solutions have been implemented.
Prototype

The code for Mjolnir's Echo is complete and the weapon's static mesh and animation have been added. The rune system and god bots have been implemented, but may have place-holder art or animation (most likely from UT2004). All runes (five) are implemented. The gods spawn correctly and killing the opposing team's god results in a victory. The gods use their area of effect attack on the opposing team. A preliminary HUD has been implemented which also may contain place-holder art. An installer / un-installer has been created.

Alpha

Mjolnir's Echo is bug free. The behavior of Tyr's Cannon and Tears of Freya has been coded and their static meshes and animations have been implemented and added to the game. The steam charger has been implemented and its static mesh has been added to the game. The final GUI and HUD have been implemented, including all art assets. The gods behave correctly under all circumstances. The mod is installable / un-installable on all test platforms.

Beta

The mod works correctly at internet speeds. Any bugs since alpha are squashed.

Gold

Final gameplay balancing has taken place. The game is totally compliant with the TDD and GDD. The TDD and GDD have been updated if necessary. All bugs have been fixed. Five CD's with the final game have been burned.

Software Architecture

Directory Structure

All art, level, and code assets will be placed under one subdirectory called 'GWMod' that lies under the 'UT2004' base directory. Assets will not be located across the default directories. This method keeps data organized and simplifies installation and un-installation. The list below shows the directory structure:

- UT2004
  - GWMod
• Animations – mesh and skeletal animations
• Classes – *Unreal Script* code
• Help – custom splash logo
• Maps – level files
• Static Meshes – exported mesh packages
• System – Game package files, custom initialization files and batch build executable
• Textures – custom texture packages
• ‘UT2K4Mod.ini’ – custom mod initialization file

**Build Process**

The mod will built using the GWCompile.bat script in the system directory. This script will automate the process of compiling the mod and generating the .int and .ucl files.

**Coding Standards**

1. All of the *Unreal Script* classes developed by the software team will start with ‘GW’. For example, GWrune.uc will be used.
2. Proper indentation will be used for braces. Beginning and ending braces will be placed on their own lines. Indents will be standard tab size within the editor, UDE.
3. Variable names will have their first word letters capitalized like ‘runePickup’.
4. A general effort will be made to keep the code as readable as possible using helpful comments.