

AN INTRODUCTION TO MODDING
FOR
AGE OF MYTHOLOGY
&
AGE OF MYTHOLOGY: THE TITANS

by
STEPHEN CAINES

Contents

0.	CONDITIONS OF USE AND DISCLAIMER	4
1.	INTRODUCTION	5
2.	GETTING STARTED	6
2.1	SETTING UP A WORK AREA.....	6
2.2	EXTRACTING THE SOURCE FILES	7
3.	CHARACTER PROTO DEFINITIONS	11
3.1	CONVERTING THE PROTO(X) FILE	11
3.2	UNDERSTANDING CHARACTER DEFINITIONS	12
3.3	DEFINING DIFFERENT CHARACTER ACTIONS	20
3.4	CREATING A NEW CHARACTER.....	23
4.	THE ROLE OF MODELS	26
5.	ANIMATIONS	28
5.1	ANIM FILE SYNTAX	31
5.2	DETAIL OF AN ANIM FILE	32
5.3	MODIFYING AN ANIM FILE	36
5.4	MORE ADVANCED ANIM FEATURES	38
6.	TEXTURES	41
6.1	CONVERTING TEXTURES TO BITMAPS.....	43
6.2	CREATING A NEW BODY TEXTURE	45
6.3	CREATING A NEW HEAD TEXTURE	48
6.4	CREATING A NEW SHIELD TEXTURE.....	49
6.5	ICONS.....	50
7.	IN-GAME TEXT	52
7.1	ADDING NEW IN-GAME TEXT	52
8.	HISTORIES	56
9.	SOUNDS	57
10.	CHECKING OUR NEW UNIT	59
11.	BUILDINGS	62
11.1	ENABLING AN EXISTING BUILDING TO TRAIN A NEW CHARACTER.....	62
11.2	USING BUILDING TO TRAIN UNITS	65
11.3	MODIFYING A BUILDING'S PROTO DEFINITION	66
11.4	CREATING A NEW BUILDING PROTO DEFINITION.....	66
11.5	CREATING A NEW BUILDING ANIM.....	69
11.6	CUSTOMISING THE BUILDING.....	70
11.7	ADDING THE BUILDING'S IN-GAME TEXT	72
11.8	ADDING THE BUILDINGS HISTORY	73
11.9	CHECK THE NEW BUILDING	73
12.	ENABLING PROTO UNITS	76
12.1	CONVERTING THE TECHTREE(X) FILE.....	76
12.2	ENABLING NEW PROTOUNITS.....	77
12.3	TESTING THE COMPLETED MOD	79
13.	MAKING MORE NEW UNITS	81

13.1	IT'S A TOUGH WORLD OUT THERE.....	81
14.	TECHNOLOGIES	82
14.1	WHAT ARE TECHNOLOGIES?	82
14.2	GENERAL DETAILS.....	83
14.3	PREREQUISITES	85
14.4	EFFECTS.....	88
	14.4.1 <i>Type Effects</i>	89
	14.4.2 <i>Amount Effects</i>	90
	14.4.3 <i>Action Effects</i>	93
	14.4.4 <i>Status Effects</i>	94
	14.4.5 <i>Generator Effects</i>	94
	14.4.6 <i>Culture Effects</i>	95
14.5	EXCLUSIONS	95
14.6	CREATING A NEW TECHNOLOGY	96
	14.6.1 <i>Making the Technology Accessible</i>	99
	14.6.2 <i>Adding the In-Game Text</i>	99
	14.6.3 <i>Technology Icon</i>	99
14.7	TESTING THE TECHNOLOGY	100
14.8	CREATING MORE TECHNOLOGIES.....	100
15.	RELICS	102
16.	CIVILISATIONS	103
17.	MINOR GODS	108
18.	GOD POWERS.....	111
18.1	PLACEMENT GOD POWERS.....	113
18.2	UNIT SWAP GOD POWERS.....	114
18.3	EPIC GOD POWERS.....	115
18.4	CREATING YOUR OWN GOD POWERS	117
19.	RANDOM MAPS	118
19.1	ENABLING A RANDOM MAP FOR NEW CIVILIZATIONS.....	118
20.	ARTIFICIAL INTELLIGENCE	123
21.	WHERE TO NEXT?.....	124

0. Conditions of Use and Disclaimer

In using this Guide to Age of Mythology Modding, the user acknowledges:

- a) That they are a licensed user of Age of Mythology or Age of Mythology The Titans;
- b) That the Terms of Conditions of their License Agreement with Ensemble Studios are in no way changed as a result of the use of the techniques described in this guide;
- c) That they are an Individual and that any modifications made using the techniques described in this Guide will be used solely for their own personal use, or the use of similarly licensed individuals;
- d) That the information contained in this guide is the author's interpretation of how to perform the modifications described in this Guide and the author has not relied on any information from Ensemble Studios in deriving these interpretations;
- e) That the author's interpretations are just that and the user shall make their own judgment on the validity of these interpretations and take all responsibility for any consequence of these interpretations being false. The author in no way warrants the accuracy of the information provided;

This guide may not be used for training purpose (either paid or free of charge) by any organization or group (either for profit, not for profit, or charitable) without the written consent of the author. Such consent will not be unreasonable withheld but will subject to written assurance from such a group that any third party involved in such training will be bound to similar conditions and acknowledge similar disclaimers as outlined above.

For further information or clarification, please contact outsidethedots@optusnet.com.au.

1. Introduction

Undertaking a modding project in Age of Mythology can be a frustrating affair. There is so much that can be done and lots of advice on specific areas of modding available through online forms. However, for a newbie modder, or an experienced modder trying to remember how they got something to work last time, I thought it may be useful to write a guide to modding that makes no assumptions on past experience, starts with the basics and tries to give a feel for how the various parts of the game hang together.

This guide is in no way complete, as I am yet to understand all the possibilities available for modifying AoM, and every week someone seems to find a new way to extend the Age of Mythology playing world, but it is hoped that this guide will at least get you started on your modding career.

This Modding Guide uses the following tools:

- Ykkrosh's AOM Data File Converter – this application is essential for extracting the individual files provided with AOM and for converting files between source and compiled versions – a brilliant piece of code.
- Vachu's AOM Game Text Editor – this application lets you add to the list of text messages used in the game.

The guide also assumes you have access to a text editor (notepad will suffice) and a tool for editing bitmaps. In the early stages of modding, MSPaint will suffice (particularly if you just want to cut and paste existing artwork), however if you want to start creating artwork from scratch it is highly recommended that you get a more advanced tool such as GIMP.

Finally, this guide has been written using AoM The Titans file names, however the same principles apply to standard AoM modding. The only difference is in the names of the data files involved and these will be highlighted at the start of each chapter.

2. Getting Started

Serious modding in AoM will result in the creation of a large number of files and can get confusing, so the first thing to do is to create a separate area for modding activity. Once a mod is ready for testing, you can then copy them to the appropriate AoM folder. This will save a lot of frustration when a mod fails, as it will be easier to identify the problem file and you will not have to reinstall AoM (and lose any other modifications you have done or downloaded).

The following approach is recommended:

2.1 *Setting up a Work Area*

1. Create a **New Folder** and call it **AoM Mods**.
2. In this new **AoM Mods** folder, create two sub-folders called:
 - a) **Source Files**;
 - b) **Mod Files**;
3. In each of the two folders from step 2, create 10 sub-folders called:
 - a) **ai**
 - b) **anim**
 - c) **bit maps**
 - d) **data**
 - e) **god powers**
 - f) **history**
 - g) **models**
 - h) **rm**
 - i) **sound**
 - j) **textures**
4. In each of the two **textures** folders, create a new sub-folder called **icons**.

You may have noticed that with the exception of the **Bit Maps** folders, these folders have the same names as those used in the standard AoM installation. The idea is that we will install the relevant source files under their appropriate folders in the **Source Files** folder, and then edit and save our modifications into the appropriate folder in the **Mod Files** folder. This may sound like over-kill, but when hundreds of files are involved, it makes it much easier to manage the mod installation and fallback process.

If you plan a major modification (such as a new civilization) I would also make a third set of folders where you can store the latest "working release", in this way the worst that can happen if you really mess things up, is that you only risk losing the latest modification.

5. Once your folders are set up, run the installation programs for **Ykkrosh's AOM Data File Converter** and **Vachu's AOM Game Text Editor** included with this guide installing them into your **AoM Mods** folder.

The following files/file structure should be in place

📁	AoM Mods		
	- Age of Mythology Center	1KB	Internet Shortcut
	- data.vdf	357KB	OLD File
	- compile	757KB	VHT File
	- Text Editor	380KB	Application
	- Read Me	2KB	Text Document
	- aomed	1372KB	Application
📁	Mod Files		
📁	ai		
📁	anim		
📁	bit maps		
📁	data		
📁	god powers		
📁	history		
📁	models		
📁	rm		
📁	sound		
📁	textures		
📁	icons		
📁	Source Files		
📁	ai		
📁	anim		
📁	bit maps		
📁	data		
📁	god powers		
📁	history		
📁	models		
📁	rm		
📁	sound		
📁	textures		
📁	icons		

2.2 Extracting the Source Files

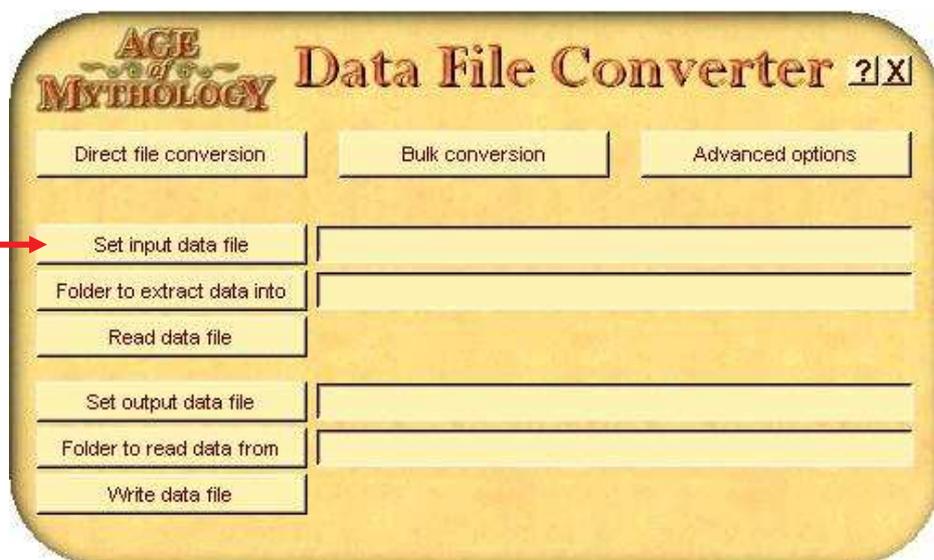
6. We now need to copy the relevant source files from the original AoM folders to our **Source Files** folders. For the time being, we are not going to worry about ai or rm files.

In AoM, the individual program files are stored in collections of like files called **.BAR** files. In the first instance, we want to copy these files into their correct **Source Files** folder where we can extract the individual program files. *Note: that if after reading through this chapter, you feel confident with what we are trying to achieve you can omit this step (6) and extract the files directly from the standard AoM folders. This recommended approach is to avoid extracting the files into the AoM folders by mistake (rather than the new folders where we want to do our mods). However to follow this approach you will need at least 1Gbyte of free disk space. To copy the required .BAR files:*

- a) Find the default AoM folder it should be C:\Program Files\Microsoft Games\Age of Mythology, expand the folder and you will see it contains a number of sub-folders, some with the same names we used in our own filing system.
- b) From the **anim** folder within the AoM default folder copy the **ANIM.BAR** and **ANIM2.BAR** files to your **Source Files\anim** folder. If you are not using the Titans Expansion, you will not have the second file (ditto for the other copies below).
- c) From the **data** folder within the AoM default folder copy the **DATA.BAR** and **DATA2.BAR** files to your **Source Files\data** folder.
- d) From the **god powers** folder within the AoM default folder copy the **GODPOWERS.BAR** and **GODPOWERS2.BAR** files to your **Source Files\god powers** folder.
- e) From the **data** folder within the AoM default folder copy the **MODELS.BAR** and **MODELS2.BAR** files to your **Source Files\models** folder.
- f) From the **sounds** folder within the AoM folder copy the **SOUNDS.BAR** and **SOUNDS2.BAR** files to your **Source Files\sounds** folder.
- g) From the **textures** folder within the AoM folder copy the **TEXTURES.BAR** and **TEXTURES2.BAR** files to your **Source Files\textures** folder.

7. The next step is to extract the individual program files for each file type. To do this:

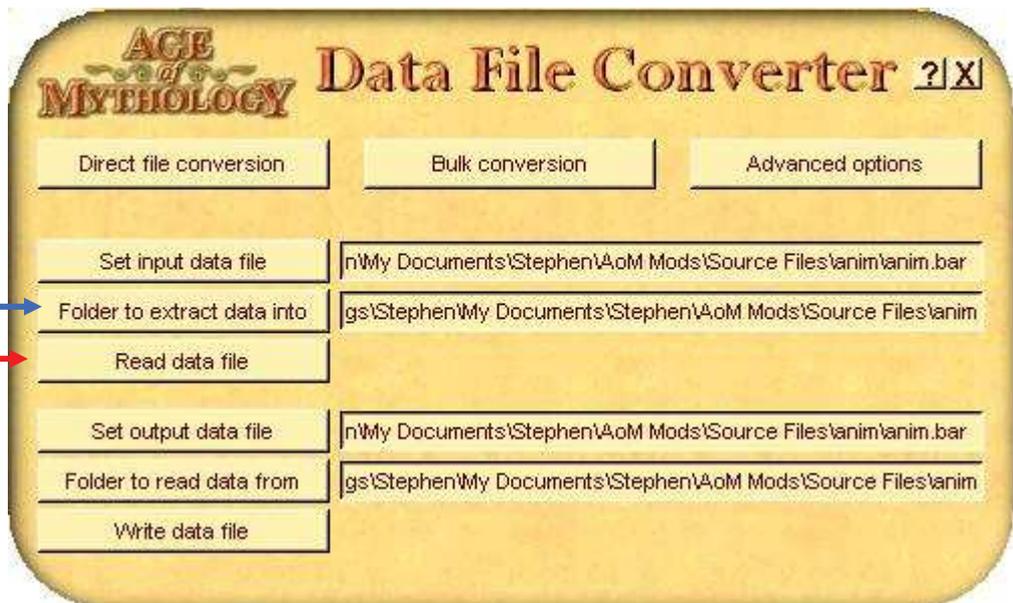
- a) Launch **Ykkrosh's AOM Data File Converter** by double clicking the relevant icon in your **AoM Mods** folder. It will bring up the following:



- b) In the first instance, we need to extract the individual files from the .BAR files we have just copied to our **Source Files** sub-folders. To do this click on the **Set input data file** button.

You will then be asked to **Select Data File for Input** and be presented with a screen for browsing your file system. Click through until you reach the **AoM Mods\Source Files\anim** folder; select the **ANIM.BAR** file and click open.

Follow the same process to select the **Folder to extract data into** field. We want this to be the same as the source folder.



Click the **Read data file** button and a Log window will open and a list of files will appear as they are extracted from the .BAR file, ending this the line **Finished!**. Close the Log Window (⊗) and click on the **anim** directory and you will see a long list of text files in the for **character name_anim**. These are the anim (or animation) files for Age of Mythology.

Repeat this process, this time selecting the **ANIM2.BAR** file. This will extract the AoM Titans Expansion anims.

- c) To extract the other files we just repeat the same process for the other .BAR files:
- **DATA.BAR** and **DATA2.BAR** files in your **Source Files\data** folder
 - **GODPOWERS.BAR** and **GODPOWERS2.BAR** files to your **Source Files\god powers** folder.
 - **MODELS.BAR** and **MODELS2.BAR** files to your **Source Files\models** folder.
 - **SOUNDS.BAR** and **SOUNDS2.BAR** files to your **Source Files\sounds** folder.
 - **TEXTURES.BAR** and **TEXTURES2.BAR** files to your **Source Files\textures** folder.

Remembering each time to click **Set Input Data File**; browse through to the appropriate folder; select the required file and **Open** it; select the **Folder to extract data into**; click **Read data file**, and check that the Log is OK.

- d) Once you have successfully extracted all the files, you can delete the .BAR files (copied in step 6) if space is a premium. **DO NOT DELETE THE .BAR FILES IN THE AOM FOLDERS.**

Note that the .XMB and .DDT files you have extracted are not in an editable format at this stage. This is what the **Direct file conversion** option is for. We will convert these as required, in later chapters.

If you followed the instructions in this chapter, you should now have a working environment in which you can start modding.

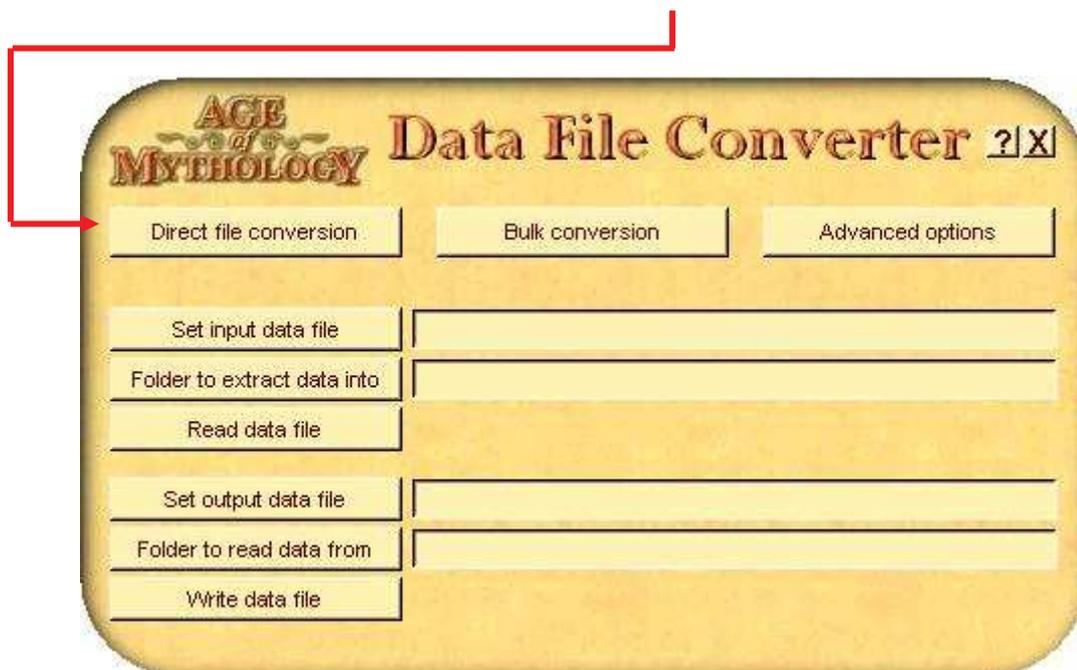
3. Character Proto Definitions

The first place we will start modding is in the creation of new characters for use in game play. This can be done from scratch by creating new models for completely new characters, however at this stage we will start by modifying an existing character.

3.1 Converting the PROTO(X) File

In AoM and AoMTT all characters available within the game are defined in the **PROTO.XMB** file (AoM) or **PROTOX.XMB** file (AoMTT). To add a new character we first need to convert the appropriate file into a format we can modify. To do this:

- 1) Launch **Ykkrosh's AOM Data File Converter** by double clicking the relevant icon in your **AoM Mods** folder, and click the **Direct file conversion** button.



- 2) You will then be asked to **Select data file for input**. Go to your **Source Files\data** folder, locate the PROTOX.XMB file (AoMTT) or PROTO.XMB file (AoM), and click **Open**.
- 3) A pop-up window will inform you that it is **Converting to XML – select an output file in the following window**. Press the **OK** button, and then press the **Save** button on the following window. This will save the file to the default file name – **PROTOX.XML**. The conversion will start and a pop up window will advise **Conversion Finished** when complete. Press **OK** and close (⌫) the **AOM Data File Converter**.
- 4) Copy this file from your **AoM Mods\Source Files\data** folder to your **AoM Mods\Mod Files\data** folder. You can retain the version in Source Files as an original.

You now have a file **PROTOX.XML** that can be edited using **Notepad** or any other .XML editor.

3.2 Understanding Character Definitions

For the time being, open this file using **Notepad** and you will see some header information, followed by a list of units representing the characters, buildings, embellishments and special effects used in the game. In the standard files, there are 800 units defined in the PROTOX file for AoMTT and 622 units defined in the PROTO file for AoM. In this chapter, we are solely concerned with defining new characters.

To begin with, we will look at a basic infantry unit Greek Hero Jason (PROTOX unit 542, PROTO unit 478).

The following is a copy of the PROTOX definition of Jason.

```
<unit id="542" name="Hero Greek Jason">
  <dbid>2296</dbid>
  <displaynameid>16898</displaynameid>
  <footprint>Footprint Military</footprint>
  <icon>Hero G jason</icon>
  <maxcontained>1</maxcontained>
  <containedattack>0.0500</containedattack>
  <initialhitpoints>250.0000</initialhitpoints>
  <maxhitpoints>250.0000</maxhitpoints>
  <los>16.0000</los>
  <portraiticon>Hero G Jason Icon 64</portraiticon>
  <obstructionradiusx>0.7000</obstructionradiusx>
  <obstructionradiusz>0.7000</obstructionradiusz>
  <soundvariant>Hack</soundvariant>
  <birthreplacement>Hero Birth</birthreplacement>
  <deadreplacement>Hero Death</deadreplacement>
  <ballisticsplashproto>Splash</ballisticsplashproto>
  <ballisticbounceproto>Dust Medium</ballisticbounceproto>
  <formationcategory>Mobile</formationcategory>
  <maxvelocity>4.3000</maxvelocity>
  <movementtype>land</movementtype>
  <turnrate>18.0000</turnrate>
  <unitaitype>HandCombative</unitaitype>
  <populationcount>2</populationcount>
  <trainpoints>9.0000</trainpoints>
  <buildlimit>1</buildlimit>
  <allowedage>1</allowedage>
  <cost resourcetype="Food">100.0000</cost>
  <cost resourcetype="Gold">50.0000</cost>
  <bounty resourcetype="Favor">2.1600</bounty>
  <bountyfactor resourcetype="Favor">1.0000</bountyfactor>
  <rollovertextid>16601</rollovertextid>
  <rolloverbonusdamageid>17277</rolloverbonusdamageid>
  <rolloveruseagainstid>17352</rolloveruseagainstid>
  <rollovercounterwithid>17571</rollovercounterwithid>
  <rolloverupgradeatid>17589</rolloverupgradeatid>
  <buttonpos column="0" row="0"></buttonpos>
  <armor damagetype="Hack" percentflag="1">0.25</armor>
  <armor damagetype="Pierce" percentflag="1">0.35</armor>
  <armor damagetype="Crush" percentflag="1">0.99</armor>
  <allowedculture>Greek</allowedculture>
```

```

<unittype>LogicalTypeVolcanoAttack</unittype>
<unittype>LogicalTypeSuperPredatorsAutoAttack</unittype>
<unittype>LogicalTypeCanBeHealed</unittype>
<unittype>LogicalTypeAffectedByRestoration</unittype>
<unittype>LogicalTypeAffectedByVortex</unittype>
<unittype>LogicalTypeMilitaryUnitsAndBuildings</unittype>
<unittype>LogicalTypeParticipatesInBattlecries</unittype>
<unittype>LogicalTypeTornadoAttack</unittype>
<unittype>LogicalTypeSuperPredatorsAttack</unittype>
<unittype>LogicalTypeValidFlamingWeaponsTarget</unittype>
<unittype>LogicalTypeConvertsHerds</unittype>
<unittype>LogicalTypeValidBoltTarget</unittype>
<unittype>LogicalTypeFimbulWinterTCEvalType</unittype>
<unittype>LogicalTypeEarthquakeAttack</unittype>
<unittype>LogicalTypeFavoriteUnit</unittype>
<unittype>LogicalTypeValidSpyTarget</unittype>
<unittype>LogicalTypeGreekHeroes</unittype>
<unittype>LogicalTypeValidShiftingSandsTarget</unittype>
<unittype>LogicalTypePredatorsAttack</unittype>
<unittype>LogicalTypeNeededForVictory</unittype>
<unittype>LogicalTypeHandUnitsAutoAttack</unittype>
<unittype>LogicalTypeValidFrostTarget</unittype>
<unittype>LogicalTypeLandMilitary</unittype>
<unittype>LogicalTypeImplodeAttack</unittype>
<unittype>LogicalTypeValidSPCUnitsDeadCondition</unittype>
<unittype>LogicalTypeAffectedByHealingSpring</unittype>
<unittype>LogicalTypeUnitsNotBuildings</unittype>
<unittype>LogicalTypeRangedUnitsAutoAttack</unittype>
<unittype>LogicalTypeVillagersAttack</unittype>
<unittype>LogicalTypeHandUnitsAttack</unittype>
<unittype>LogicalTypeRangedUnitsAttack</unittype>
<unittype>LogicalTypeTartarianGateValidOverlapPlacement</unittype>
<unittype>LogicalTypeGarrisonOnBoats</unittype>
<unittype>LogicalTypeValidForestFireTarget</unittype>
<unittype>LogicalTypeValidMeteorTarget</unittype>
<unittype>LogicalTypeMinimapFilterMilitary</unittype>
<unittype>LogicalTypeIdleHero</unittype>
<unittype>LogicalTypeIdleMilitary</unittype>
<unittype>Unit</unittype>
<unittype>UnitClass</unittype>
<unittype>Military</unittype>
<unittype>Hero</unittype>
<flag>ObscuredByUnits</flag>
<flag>CollidesWithProjectiles</flag>
<flag>DontRotateObstruction</flag>
<flag>ShowGarrisonButton</flag>
<flag>CorpseDecays</flag>
<flag>ApplyHandicapTraining</flag>
<flag>HideGarrisonFlag</flag>
<flag>Tracked</flag>
<contain external="1">Relic</contain>
<action name="HandAttack">
  <param name="MaximumRange" value1="0.1"></param>
  <param name="Damage" type="Hack" value1="9"></param>
  <param name="DamageBonus" type="MythUnit" value1="7"></param>
  <param name="Rate" type="All" value1="5.0"></param>
  <param name="DamageBonus" type="SetAnimal" value1="3"></param>
</action>
<action name="PickUp">
  <param name="TypedRange" type="Relic" value1="1"></param>
</action>
<action name="DropOff">
  <param name="TypedRange" type="AbstractTemple" value1="1"></param>
  <param name="Rate" type="AbstractTemple" value1="1"></param>

```

```
</action>
</unit>
```

While at first sight it may appear very complex, the structure of the character definition is simply made up of a number attributes and options that determine how the unit will behave. The following summarizes what each part of the definition is for:

```
<unit id="542" name="Hero Greek Jason">
```

Every unit requires a unique **unit id** and these unit ids must be sequential. For example, the first unit id you will add will be 801 for AoMTT and 623 for AoM. The format of the definition begins with `<unit id = ...>` and ends with `</unit>` (at the bottom of the definition) and you will notice that each attribute in the definition uses the same syntax (e.g. `<dbid>...</dbid>`).

Whenever adding or modifying character definitions, remember you must have a unique in sequence id and the syntax is mandatory. If not when you try and convert the file back to an .XMB file for use in the game, it will fail to compile. The **name** is used to identify the appropriate anim file, in this case **Hero Greek Jason_anim.txt**. Note that the name that appears in game is not always the same as the proto unit name.

```
<dbid>2296</dbid>
```

The **dbid** needs to be unique (though not sequential) the first available dbid is 2848 for AoMTT and 2653 for AoM.

```
<displaynameid>16898</displaynameid>
```

The **displaynameid** field is a reference to a text id contained in the `xpacklanguage.dll` file (AoMTT) or `language.ddl` file (AoM) and is discussed in more detail in Chapter 7. The number needs to be unique, but you can have gaps in the numbering. If you start your own text numbering at 60000 you will leave plenty of room for any future ES enhancements. The **displaynameid** field points to the text entry that will appear in game or in the editor when placing new units. There is also an **editornameid** field that can be used if you wish to use a different name in the scenario editor, if making a civilization you could use this to include a prefix, which would allow you to keep your new units together in the menus when using the Scenario Editor to place your new units.

```
<footprint>Footprint Military</footprint>
```

You may have noticed that when walking over snow or going across water units leave footprints (defined earlier in the PROTO/X definitions). This field simply tells the game which of the various footprint options will be used. There are a large number of footprint options available including Hoofprint (medium animals), Oar Footprint (for boats), Footprint Villager (for villagers), Footprint Animal Small (goats, pigs), Footprint Cavalry (cavalry), Footprint Animal Large (giraffe, elephant), and a number of specialized footprints such as Flaming Footprint, Footprint Scarab, Footprint Wheel, Footprint Siege. Generally the footprint defined for your base unit will be the one to use.

```
<icon>Hero G jason</icon>
```

In game, you will notice that a picture of a character, technology and so forth is displayed in the information box or in help menus. These are called **icons** and come in varying sizes from 32x32 bits to 256x256 bits. The **icon** field refers to the name of the file in the `\textures\icons` folder that will be used in specific situations (depending on how big the icon is on a specific screen). We will learn how to create icons in Chapter 6. Unfortunately, Jason

does not adopt the standard naming format, in most cases icons use a naming convention of **unitname Icon size**, e.g. this icon reference should have been Hero G Jason Icon 32, with the larger portrait icon defined below being Hero G Jason Icon 64. If you keep to this naming standard you can set up the proto definition and create the icons later.

```
<maxcontained>1</maxcontained>
```

The **maxcontained** field is more typically used for buildings or boats that can garrison people, but it is also required for units that may be required to contain other units (for example carrying a relic). You can use this field for other special features such as the ability for one character to carry another (e.g. I have used it to allow a mounted version of a Legolas character to garrison Gimli). Just set the amount to the number of units you want to be able to garrison.

```
<containedattack>0.0500</containedattack>
```

The **containedattack** field defines what percentage of the units attack capability will be added to the attack strength of a unit it is garrisoned in (if the GarrisonBonus flag is set). Generally logic would dictate that ranged units such as archers would have a higher containedattack (they can shoot out windows) while hand attack units would do little to increase the strength of the building they are in.

```
<initialhitpoints>250.0000</initialhitpoints>  
<maxhitpoints>250.0000</maxhitpoints>
```

These fields set the initial and maximum hitpoints for a character in its original format (e.g. they do not include any additional hitpoints the character may receive because of technology upgrades). Usually they are set to the same amount, but you could set the maxhitpoints to a higher number and use a regenerate action (see below) to improve their hitpoints from a lower starting level. The hitpoints can be any amount you choose, but to avoid making an unreasonably strong character you should look at other similar units and not go overboard.

```
<los>16.0000</los>
```

The **los** field determines the unit's line of site in its original format. The higher the number the further it sees. There is no limit to los, but again a unit that has line of site over the entire game field is of little interest in a game (just use the cheat if you want to see everything).

```
<portraiticon>Hero G Jason Icon 64</portraiticon>
```

This is a larger version of the characters icon and refers to a file of the given name in the \textures\icons folder.

```
<obstructionradiusx>0.7000</obstructionradiusx>  
<obstructionradiusz>0.7000</obstructionradiusz>
```

The obstructionradiusx/z fields determine how much space a character "blocks" and are used by the game to stop characters walking through each other (or buildings being built through other buildings or characters). The larger the numbers, the larger the obstruction area. It is usually best to stick the values contained in the base unit you use for modding. If you start increasing these values you will end up with a situation where units have difficulty navigating around the game board and getting trapped between units.

```
<soundvariant>Hack</soundvariant>
```

```
<birthreplacement>Hero Birth</birthreplacement>  
<deadreplacement>Hero Death</deadreplacement>  
<ballisticsplashproto>Splash</ballisticsplashproto>  
<ballisticbounceproto>Dust Medium</ballisticbounceproto>
```

The above fields define the basic soundsets the game will use in association with this character, in order the default sound file used when the unit attacks, is born/dies and is attacked. Generally, the ones that existed from the base protounits can be retained. They can also be added to in the anim definitions and _snd file for the unit.

```
<formationcategory>Mobile</formationcategory>
```

The formation category determines how the unit will behave in group situations, and can be defined as Protected, **Mobile**, **Ranged** and **Body**. As far as I can tell these definitions mean – Protected, the unit will stay at the center of the group so that it can be protected by the units surrounding it; Mobile – the unit can move independently of the group, Ranged – in attack situations the unit can hang back from the group and fire from a distance; Body – the unit will stay in a solid formation within the group. In general, weak units are protected; heroes, cavalry and other fast units are Mobile, ranged units such as archers, slingers, and war ships are Ranged and standard infantry are Body.

```
<maxvelocity>4.3000</maxvelocity>
```

The maxvelocity field defines how quickly the unit will move in its initial form (pre any tech upgrades). In general cavalry move the fastest, followed by light infantry and villagers, heavy infantry, and siege weapons. You can set the amount as high as you like – a maxvelocity of 100 would allow a unit to cross a standard game map in 3-4 seconds.

```
<movementtype>land</movementtype>
```

The movementtype field tells the game how the unit will move; the options are **air**, **land** and **water**. Basically "air" can go anywhere whereas "land" and "water" are restricted from movement over the other (e.g. land can't cross water).

```
<turnrate>18.0000</turnrate>
```

As far as I can tell turnrate is used to set the priority of movement within the game – a higher turnrate equates to a higher priority and hence should be used for priority units (though I stand ready to be corrected on this). In general, stick to the rate defined for the base protounit.

```
<unitaitype>HandCombative</unitaitype>
```

This field defines what type of unit this prototype is for AI purposes. This is discussed further in Chapter 14, but in general the AI needs to know the unit type so it can create the right balance of units. The main unitaitypes are **Civilian** – the unit will be selected to gather resources; **Hand Combative**, **Ranged Combative**, **RangedSiege**, **Ram**, **Scout** – the unit will be selected based on the military AI goals in place. If the unitaitype field is not consistent with what the unit actual does it will just make the AI act inappropriately.

```
<populationcount>2</populationcount>
```

This field defines how many units towards the population limit the protounit will use. In general, stronger units have a higher population count. This amount can be zero, or a fraction if you so wish.

```
<trainpoints>9.0000</trainpoints>
```

```
<buildlimit>1</buildlimit>
<allowedage>1</allowedage>
```

The above fields provide some basic info about unit creation. The trainpoints defines how long it will take to build (higher the number, longer the training time). The buildlimit defines the maximum number of the unit that can be used in a game (pre any tech upgrades which may affect this value) and the allowedage defines the earliest age in which the unit can be built (e.g. age 1-4). Typically heroes or special characters will be limited to one, more exotic characters will be prohibited until later ages, and the better the unit the longer it will take to create, however there are no hard and fast rules.

```
<cost resourcetype="Food">100.0000</cost>
<cost resourcetype="Gold">50.0000</cost>
```

These fields define how much a unit will cost (pre any tech upgrades which may change these values). As well as **Food** and **Gold**, the cost resourcetype can also be "**Wood**" or "**Favor**". The main challenge in defining the cost of the unit is to ensure that you retain a balance in resource costs – e.g. if everything costs Food - Wood and Gold become useless, too much emphasis on Favor may limit your ability to quickly build a defending army and so forth.

```
<bounty resourcetype="Favor">2.1600</bounty>
```

The Norse culture earns favor by destroying enemies, so each unit has a favor value that is received by the opponent on its death. Generally the harder the unit is to kill, the higher its bounty should be. While the standard game uses this feature only for favor, it can be used to define other resource types for example you could create a civilization that survives on plunder and various buildings and characters could result in bounties of food, wood and gold.

```
<bountyfactor resourcetype="Favor">1.0000</bountyfactor>
```

This field establishes a factor that is applied to a units favor gathering ability (praying or combat). It is usually set to one, however it could be varied so that weaker units gain a favor premium.

```
<rollovertextid>16601</rollovertextid>
<rolloverbonusdamageid>17277</rolloverbonusdamageid>
<rolloveruseagainstid>17352</rolloveruseagainstid>
<rollovercounterwithid>17571</rollovercounterwithid>
<rolloverupgradeatid>17589</rolloverupgradeatid>
```

These fields define a number of other texts messages that are used with the protounit and as with the **displaynameid** are simply references to an index number in the xpacklanguage.dll or language.dll files. The texts have the following purposes. The **rollovertextid** is an expanded explanation of the unit and is displayed (usually in the bottom left-hand side of the screen when the unit is selected) it is a brief explanation of what the unit is for. The **rolloverdamagebonusid** is used in the main information screen for the unit (when you press the information icon) and should be based on the bonusdamage settings you use in the units action parameters (see below). The **rolloveruseagainstid** and **rollovercounterwithid** similarly tell game players what units the protounit is best used to attack and what to use to defend against it. The **rolloverupgradeid** tells game players which building or buildings they need to upgrade the unit (e.g. where they can initiate the technology upgrades associated with this protounit). The text ids need to be unique. As advised above, just start your own text ids at 60000 and keep adding one for each new text. Including the actual text is discussed in Chapter 7.

```
<buttonpos column="0" row="0"></buttonpos>
```

This field is used in conjunction with the `unittransform` and `commandpanel` functions and need not be worried about at this time. It just tells the game where this button will be and would only be changed if you wanted to use this position for something else.

```
<armor damagetype="Hack" percentflag="1">0.25</armor>
<armor damagetype="Pierce" percentflag="1">0.35</armor>
<armor damagetype="Crush" percentflag="1">0.99</armor>
```

These fields define the initial level of armor protection for the unit (e.g. prior to any armor upgrades). The amounts are the percent by which attacks are reduced. Hack are usually hand attacks, pierce - ranged attacks and crush - siege attacks, though some attacks use combinations of these. The higher the amount used the more resistant a protounit is to a given form of attack.

```
<allowedculture>Greek</allowedculture>
```

This field tells the game what protounits are available to a given culture (Greek, Egyptian, Norse, and Atlantean) and is mainly used to limit the build options available in some multi-purpose buildings (e.g. town centers). Use this field if you want to limit the availability of the unit.

```
<unittype>LogicalTypeVolcanoAttack</unittype>
<unittype>LogicalTypeSuperPredatorsAutoAttack</unittype>
<unittype>LogicalTypeCanBeHealed</unittype>
<unittype>LogicalTypeAffectedByRestoration</unittype>
<unittype>LogicalTypeAffectedByVortex</unittype>
<unittype>LogicalTypeMilitaryUnitsAndBuildings</unittype>
<unittype>LogicalTypeParticipatesInBattlecries</unittype>
<unittype>LogicalTypeTornadoAttack</unittype>
<unittype>LogicalTypeSuperPredatorsAttack</unittype>
<unittype>LogicalTypeValidFlamingWeaponsTarget</unittype>
<unittype>LogicalTypeConvertsHerds</unittype>
<unittype>LogicalTypeValidBoltTarget</unittype>
<unittype>LogicalTypeFimbulWinterTCEvalType</unittype>
<unittype>LogicalTypeEarthquakeAttack</unittype>
<unittype>LogicalTypeFavoriteUnit</unittype>
<unittype>LogicalTypeValidSpyTarget</unittype>
<unittype>LogicalTypeGreekHeroes</unittype>
<unittype>LogicalTypeValidShiftingSandsTarget</unittype>
<unittype>LogicalTypePredatorsAttack</unittype>
<unittype>LogicalTypeNeededForVictory</unittype>
<unittype>LogicalTypeHandUnitsAutoAttack</unittype>
<unittype>LogicalTypeValidFrostTarget</unittype>
<unittype>LogicalTypeLandMilitary</unittype>
<unittype>LogicalTypeImplodeAttack</unittype>
<unittype>LogicalTypeValidSPCUnitsDeadCondition</unittype>
<unittype>LogicalTypeAffectedByHealingSpring</unittype>
<unittype>LogicalTypeUnitsNotBuildings</unittype>
<unittype>LogicalTypeRangedUnitsAutoAttack</unittype>
<unittype>LogicalTypeVillagersAttack</unittype>
<unittype>LogicalTypeHandUnitsAttack</unittype>
<unittype>LogicalTypeRangedUnitsAttack</unittype>
<unittype>LogicalTypeTartarianGateValidOverlapPlacement</unittype>
<unittype>LogicalTypeGarrisonOnBoats</unittype>
<unittype>LogicalTypeValidForestFireTarget</unittype>
<unittype>LogicalTypeValidMeteorTarget</unittype>
<unittype>LogicalTypeMinimapFilterMilitary</unittype>
<unittype>LogicalTypeIdleHero</unittype>
<unittype>LogicalTypeIdleMilitary</unittype>
```

The above list of **unittype** definitions tells the game how it is to treat a unit in various circumstances. In the first instance it is easier to keep these the same as the base unit you have used for your modified protounit but when you get more experience you can add or delete unittype definitions where you want a unit to be immune from certain attacks or god powers.

```
<unittype>Unit</unittype>
<unittype>UnitClass</unittype>
<unittype>Military</unittype>
<unittype>Hero</unittype>
```

The next set of unittype definitions defines a number of unittype classes, which are used in the game to cluster units into like groups. These definitions are useful when defining new technologies as the effects of these technologies can be made either to a specific protounit or to a class of units (e.g. armor and weapons upgrades to all military units). Other common unit types that enable such global upgrades are AbstractArcher, AbstractInfantry and AbstractCavalry. The first of the definitions is also used in the Editor when placing protounits, whereby they can be separated into Units, Buildings and Embellishments. You can use as many classes as you like, just make sure that when you use them you are not inadvertently granting a unit multiple upgrades because they are in multiple classes.

```
<flag>ObscuredByUnits</flag>
<flag>CollidesWithProjectiles</flag>
<flag>DontRotateObstruction</flag>
<flag>ShowGarrisonButton</flag>
<flag>CorpseDecays</flag>
<flag>ApplyHandicapTraining</flag>
<flag>HideGarrisonFlag</flag>
<flag>Tracked</flag>
```

The flags listed above switch on or off some of the options associated with protounits and should kept as standard from the base protounit. If making a hero that you do not wish to die you should include line:

```
<flag>HeroDeath</flag>
```

This will allow the hero unit to resurrected when allied troops move into the area.

```
<contain external="1">Relic</contain>
```

This allows a unit to carry a relic.

The last section of the unit definition is the actions that unit can perform. The number of actions can vary for a unit and can include various attack actions, economic actions. In the example of Jason only three actions are defined:

```
<action name="HandAttack">
  <param name="MaximumRange" value1="0.1"></param>
  <param name="Damage" type="Hack" value1="9"></param>
  <param name="DamageBonus" type="MythUnit" value1="7"></param>
  <param name="Rate" type="All" value1="5.0"></param>
  <param name="DamageBonus" type="SetAnimal" value1="3"></param>
</action>
```

This first action is a basic hand attack definition and has five parameters:

- The maximum range parameter, tells the game how far away an enemy must be for the attack to take effect. For a hand attack this is typically 0.1 though you can

increase here or via a technology to give the unit an advantage of reach over its opponents.

- The **Damage** and damage **type** caused. You could define additional damage for Pierce and Crush if required (for example a swordsman may inflict Hack and Pierce, a Maceman could inflict Hack and Crush Damage). The total of these damages would be reduced by the opponent's armor settings and applied against its hitpoints.
- The **DamageBonus** allows you set an additional damage amount for either a specific protounit or in this case a unit class. Jason has two Damagebonuses against unit classes MythUnit and SetAnimal.
- The Rate parameter allows you define the frequency of attack, a higher number being more "swings per second".

```
<action name="PickUp">
    <param name="TypedRange" type="Relic" value1="1"></param>
</action>
<action name="DropOff">
    <param name="TypedRange" type="AbstractTemple" value1="1"></param>
    <param name="Rate" type="AbstractTemple" value1="1"></param>
</action>
```

The above actions allow the protounit to pick up and drop off relics and should be used as is for any protounit that you require to collect relics.

```
</unit>
```

This signifies the end of the definition for this protounit. If it was the last unit in the PROTOX file it would be followed by </proto> signifying the end of the definition file.

3.3 Defining Different Character Actions

Aside from the definition of actions, most protounits have the same features in terms of a name, cost and basic attributes. Making your characters interesting is all about them doing interesting things so it is worth looking at the other types of actions units can perform:

Regenerate

We hate it when our heroes die, and so the best way to prolong them is to allow them to regenerate or recover. The following standard action enables regeneration.

```
<action name="Regenerate">
    <param name="Rate" type="All" value1="1.0"></param>
    <param name="Persistent"></param>
</action>
```

To increase regeneration ability just increase the amount in value1. The value is roughly equal to one hitpoint regeneration per second.

Heal

You may want units such as priests or buildings such as temples to be able to heal other units. The action required to do this is:

```
<action name="Heal">
    <param name="MaximumRange" value1="3.0"></param>
```

```

        <param name="Rate" type="LogicalTypeCanBeHealed" value1="1.5"></param>
    </action>

```

The action has two parameters the **MaximumRange** (e.g. how far away the unit needs to be to be affected) and the **Rate** at which healing takes place (roughly hit points per second)

Convert

A convert or chaos attack is where a unit can convert those around them to Gaia, whereby they attack all players around them. The convert attack uses the following format:

```

<action name="ConvertAttack">
    <param name="MaximumRange" value1="7"></param>
    <param name="MuteDamage"></param>
    <param name="NoWorkOnFrozenUnits"></param>
    <param name="NoWorkOnStoneUnits"></param>
    <param name="Rate" type="LogicalTypeAffectedByChaos" value1="1.0"></param>
    <param name="NoWorkOnChaosUnits"></param>
    <param name="AttackAction"></param>
    <param name="ChargeAction"></param>
    <param name="ConvertToGaia"></param>
</action>

```

The main parameters here are the **MaximumRange** and **Rate** of attack (in this case one at a time). The parameter "**ChargeAction**" means that this attack can only be used once and then it must be recharged. To define the recharge time you would insert the following (just above the **maxvelocity** definition earlier on in the unit's proto definition):

```

<rechargeTime>10.0000</rechargeTime>

```

Where the amount is the number of seconds required to recharge the attack.

Ranged Attack

The ranged attack is the standard attack type for units such as archers, ranged siege weapons or buildings that shoot. It takes the following basic format:

```

<action name="RangedAttack">
    <param name="MinimumRange" value1="6.0"></param>
    <param name="MaximumRange" value1="28"></param>
    <param name="Damage" type="Pierce" value1="19"></param>
    <param name="Accuracy" value1="1.0"></param>
    <param name="AttackAction"></param>
    <param name="AccuracyReductionFactor" value1="0.1"></param>
    <param name="AimBonus" value1="25"></param>
    <param name="SpreadFactor" value1="0.10"></param>
    <param name="MaxSpread" value1="3.0"></param>
    <param name="TrackRating" value1="5.0"></param>
    <param name="UnintentionalDamageMultiplier" value1="0.3"></param>
    <param name="HeightBonusMultiplier" value1="1.25"></param>
    <param name="DamageBonus" type="Warg" value1="7"></param>
    <param name="DamageBonus" type="Warg Rider" value1="7"></param>
    <param name="DamageBonus" type="Orc Pikeman" value1="7"></param>
    <param name="Rate" type="All" value1="1.5"></param>
</action>

```

In this example the **MinimumRange** is used as the character in question (Legolas) uses a sword for close combat. If the unit only had a ranged attack you would not use this

parameter. The other parameters are similar to those used in a basic hand attack and define the level and type of damage caused and any damage bonuses. Ranged Attacks also allow control over the accuracy of firing include the **SpreadFactor** and **MaxSpread** (how far the projectiles will spray from their intended target) based on how far away it is; how well it will target moving objects via its **TrackRating** (in this case the **AccuracyReductionFactor** is applied when speed exceeds 5.0; and the extent to which accuracy and damage will increase where the ranged unit is higher in altitude than the target (its **HeightBonusMultiplier**). This basic format is used across all ranged attacks. The variation that creates different projectiles that are displayed during game-play are defined earlier in the protounit definition, typically just after the turnrate, using one the following definitions:

```
<projectileprotounit>Arrow Flaming</projectileprotounit>  
<projectileprotounit>Ballista Shot</projectileprotounit>  
<projectileprotounit>Lampades Bolt</projectileprotounit>
```

Look at the definition of the ranged unit most like the one you are creating if you are not sure.

Autogather

It is also handy sometimes to define units or buildings that autogather resources – A Plenty Vault is one example of this type of unit, but it is also useful for fattening up livestock and so forth. The format is straightforward – just define the type and rate of resource gathering.

```
<action name="AutoGather">  
  <param name="Persistent"></param>  
  <param name="Rate" type="Favor" value1="0.50"></param>  
</action>
```

A number of other standard attack variations are available such as LightingAttack, JumpAttack, ChargedRangeAttack and ChargedHandAttack. If you do a search through the PROTOX files you can see you they work.

The **MOST IMPORTANT** thing to remember when defining actions, is that each action will require a separate animation definition in the units anim file (see Chapter 5) and these animations will in turn call different Models. While it would be nice to create a unit with all of these actions, if suitable models do not exist (unless you can create additional models) the actions will not look very realistic in game play. More of this later.

The other point to remember when defining the initial proto unit is that you are defining how the character will behave with the Age 1 technology set. As we will discuss in Chapter 10, the units speed, los, armor strength, recharge time, training time, attack strength, attack speed, and so forth can be upgraded during the game through technologies.

In general, it is more interesting to start with a weaker unit and force players to achieve certain technologies, than to start with an Age 1 monster that can simply wander around the game destroying enemies at random.

3.4 Creating a New Character

It will take a bit of practice for all of this to sink in, but to start with we will just keep it simple. We are going to create a new proto unit definition – Aragorn – based on the Jason Character.

- 1) Open the PROTO(X).XML file scroll or search your way down to Greek Hero Jason (PROTOX unit 542, PROTO unit 478). Copy the full proto definition for Jason – i.e. everything between

```
<unit id="542" name="Hero Greek Jason">
.....
.....
</unit>
```

- 2) Go to the end of the PROTO(X).XML file and before the line containing `</proto>` paste the copied protounit definition.

- 3) Now make the following changes to the pasted definition:

- a) Change the first line to `<unit id="801" name="Aragorn"> "623"` if using AoM.
- b) Change the second line to `<dbid>2848</dbid> 2653` for vanilla AoM.
- c) Change the `<icon>` field from **hero g jason** to **aragorn icon 32**.
- d) Change the `<portraiticon>` field from **hero g jason icon 64** to **aragorn icon 64**.
- e) Change the six (6) references to in-game texts to the numbers we will be using in Chapter 7.

```
<displaynameid>60000</displaynameid>
.....
.....
<rollovertextid>60001</rollovertextid>
<rolloverbonusdamageid>60002</rolloverbonusdamageid>
<rolloveruseagainstdid>60003</rolloveruseagainstdid>
<rollovercounterwithid>60004</rollovercounterwithid>
<rolloverupgradeatid>60005</rolloverupgradeatid>
```

- f) Change the `<allowedculture>` field from **Greek** to **Norse**.

```
<allowedculture>Norse </allowedculture>
```

- 4) After the last action:

```
<action name="DropOff">
  <param name="TypedRange" type="AbstractTemple" value1="1"></param>
  <param name="Rate" type="AbstractTemple" value1="1"></param>
</action>
```

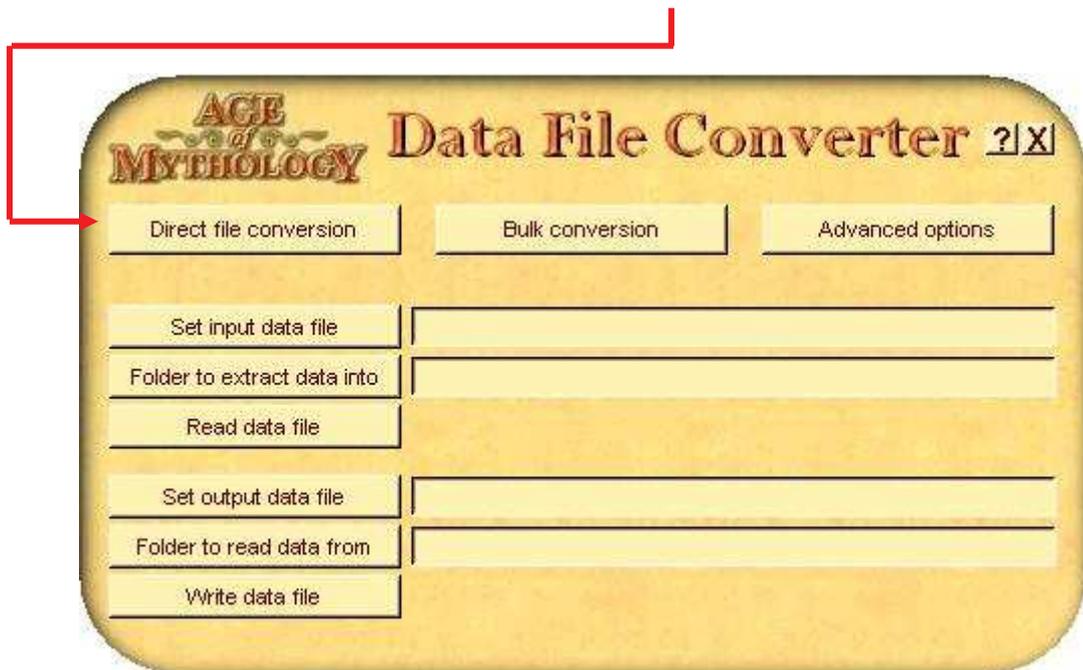
but before the end of the unit definition:

</unit>

insert a new action to allow our hero to regenerate.

```
<action name="Regenerate">  
  <param name="Rate" type="All" value1="1.0"></param>  
  <param name="Persistent"></param>  
</action>
```

- 5) Save the file (takes a little while causes it is quite large).
- 6) Launch **Ykkrosh's AOM Data File Converter** by double clicking the relevant icon in your **AoM Mods** folder, and click the **Direct file conversion** button.



- 7) You will then be asked to **Select data file for input**. Go to your **Source Files\data** folder and locate the **PROTOX.XML** file (AoMTT) or **PROTO.XML** file (AoM) and click **Open**.
- 8) A pop-up window will inform you that it is **Converting to XMB – select an output file in the following window**. Press the **OK** button, and then press the **Save** button on the following window. This will save the file to the default file name – **PROTO(X).XMB**. The conversion will start and a pop up window will advise **Conversion Finished** when complete. Press **OK** and close (ⓧ) the **AOM Data File Converter**.

You now have a new file **PROTO(X).XMB** ready for use by the game. If for some reason the conversion fails this will almost certainly be due to a syntax error. So check.

- a) That the unit definition preceding the new Aragorn units ends with </unit>
- b) That the Aragorn unit definition ends with </unit>
- c) That immediately after the Aragorn unit definition is the end of protounit definitions </proto> and that there is nothing after it.

- d) That the other lines changed have the correct syntax in terms of the definition start and finish.

If the conversion was successful copy the new PROTO(X).XMB file to the \data folder in your standard AoM folder

While we have defined a new character it is not of much use to us yet, first we must animate it.

4. The Role of Models

Before moving onto animations, we need to discuss Models. I have yet to master the art of modeling, however in order to understand how animations and textures work it is important to understand Models.

Everything you see in game play is based on a set of pre-built models provided with the game, however it is not as simple as a one-one ratio of characters and models. Each protounit will have a number of models that are used to animate different actions. For example there will be different models for when the unit is idle, walking, attacking, bored, dying, dead or at birth. Economic units such as villagers will also have different models that animate them when mining, chopping wood, sowing seeds, gathering food and so forth.

To add to this complexity, some animations require multiple models. For example, in a number of infantry units the head is a separate model and there will be multiple versions of the head, usually associated with different technologies. The head of a champion infantry unit may be a head wearing a helmet and so forth.

Other models may also be attached to a unit, such as weapons, or items the unit may be carrying – a basket for example.

So before we move on to animating and applying new textures to our new character, we need to discuss the implications of models.

The first issue is that when deciding which character (and in turn which set of models) you want to use as the basis of a new character you need to see what action models are available for that character. In the case of Aragorn, who we are going to model on Greek God Jason, we have nine models to work with:

```
hero g jason_attacka  
hero g jason_boreda  
hero g jason_carry  
hero g jason_carrybored  
hero g jason_carryidle  
hero g jason_deatha  
hero g jason_flaila  
hero g jason_idlea  
hero g jason_walka
```

While in some cases, models of different characters are similar enough (when the same texture is applied) to extend the options available, generally in basic animation we are limited to the actions available with that model.

In the case of Jason (and hence Aragorn), there is little point in defining a ranged attack for example – you can use the idle model and have it magically fire arrows, or extract food by telepathy, but as stated in the last chapter the effect is not very realistic. Similarly you cannot have a jump attack as there is no model that jumps.

So before selecting a base unit it is always wise to do a search on the Models folder to find which models are available for the character. This will save later disappointments when you realize that a certain character simply cannot perform the actions you require.

Unless you know how to create new models, the available AoM models will be your starting point for modding. If you want to build a new unit that uses a jump attack as its main attack for example, you would do a search of your Source Files\models folder for "jump". The naming convention for models is fairly standardized – unitname_action – so searching for a specific action – jump, pray, etc., will give you a list of all protounits with that ability and once you have found a likely candidate you can then search for all the models associated with that protounit to see what else they can do. In this way you can target in on the preferred base unit for your intended mod.

In some instances, characters will look similar enough (particularly when using the same textures), so that you can mix and match them. For example, in a Legolas character I developed, I used **archer x hero** as the basic model, but used the **hero x kastor adult_attack** model to let Legolas have a hand attack animation for close combat. The units are of a similar size and build and the loss of the archer's quiver when in hand attack mode is not too obvious.

While consistency of appearance is important for some units, another approach is to accept that the models may look different but use the available models to create transformational characters – for example you could base a character on **villager g greek female**, but use **TitanXGaia_attack** for the hand attack. When attacked the harmless female would transform into a 60 foot monster, only to shrink back to normal size when her attacker was vanquished.

As you get more experienced with modding these possibilities will become more apparent, however in the first instance we will just stick the basic models available for a single protounit.

5. Animations

A characters anim file is a definition of the models and visual effects that the game will use for the various actions a character will perform during game play. The format of the file is broken into two sections – a definitions section, where various attachments and visual effects are defined, and an anim section, where the each of the actions the character will perform are defined.

Every unit defined in the PROTO(X) file will have a corresponding anim file, called **protounitname_anim.txt** where protounitname is identical to the name defined in the PROTO(X) file.

The following is a copy of the **hero greek jason_anim** file that we will use as the basis of our Aragorn anim.

```
//=====
// ajax_anim.txt
//=====

//=====
define heroglow
{
    set hotspot
    {
        version
        {
            Visualparticle SFX A Hero Glow Small
        }
    }
}

//=====
anim Idle
{
    SetSelector
    {
        ContainLogic
        SetSelector
        {
            set hotspot
            {
                version
                {
                    Visual Hero G Jason_idleA
                    Connect FRONTABDOMEN heroglow hotspot
                }
            }
        }
        SetSelector
        {
            set hotspot
            {
                version
                {
                    Visual Hero G Jason_CarryIdle
                    Connect FRONTABDOMEN heroglow hotspot
                }
            }
        }
    }
}
```

```

}
}

//=====
anim Bored
{
  SetSelector
  {
    ContainLogic
    SetSelector
    {
      set hotspot
      {
        version
        {
          Visual Hero G Jason_boredA
          Connect FRONTABDOMEN heroglow hotspot
        }
      }
    }
  }
  SetSelector
  {
    set hotspot
    {
      version
      {
        Visual Hero G Jason_CarryBored
        Connect FRONTABDOMEN heroglow hotspot
      }
    }
  }
}
}

//=====
anim attack
{
  SetSelector
  {
    set hotspot
    {
      version
      {
        Visual Hero G Jason_attackA
        //-- auto generated by the bang animation tool
        //-- do not hand edit these values
        tag Attack 0.32 true
        tag GenericSound 0.40 true
        //-- end auto generated section
        Connect FRONTABDOMEN heroglow hotspot
      }
    }
  }
}

//=====
anim death
{
  SetSelector
  {
    set hotspot
    {
      version
      {

```


5.1 Anim File Syntax

So what does all this mean?

The syntax of anims uses a number of functions – SetSelector, set hotspot, version and so forth to define attributes for a particular situation. In more advanced anims these functions allow you to control various aspect of the character look based on a range of external factors. The role of these functions is as follows:

SetSelector tells the game that the definition that follows is dependent on the result of some form of logical test. In the case of the first SetSelector in the anim section the test is – is the unit idle?, is the unit bored? is the unit attacking? and so forth. However in many situations we will want a number of nested conditions, so in the idle anim for example, we have a second SetSelector after the function ContainLogic. So in this instance, we have two tests – Is the idle? If so, is the unit carrying anything? The outcome of these two tests determines which animation to use. In the definition area you will note that there is no SetSelector as in this case there is no test required, the game will always define this visual effect.

Set hotspot tells the game that you are defining an animation, which will be a collection of visual properties. The hotspot is the sequence of movements that you will see in game every time the particular action occurs.

Version can be best thought of as a frame in a sequence. While most of the sequences in AoM are achieved with a single model, some more complex sequences use several. The hippikon attack sequence for example uses three. In this case the **hotspot** (or sequence) is based on three **versions** (or frames). When the action is triggered the game will cycle through these three versions for as long as the action is in progress.

Within each **Version** we have a number of functions that allow us to define and if required modify the models we use.

Visual denotes that the following filename is the model we require for this particular version (or frame). For example `Visual hero greek jason_idleA` means use the file **hero greek jason idleA.BRG** from the AoM Models folder. The functions **Visualparticle** and **VisualGranny** (the latter not used in this anim) do the same thing but for .PRT files and .GRN files. Just look up the file in the \models directory to find the files extension type for the model you want if you are not sure which to use.

The next function we see is **Connect**. Connect allows us to attach other models (such as weapons) or visual effect (such as hero glows) to our base model. As you look through the various anim files you will find a large number of potential Attachpoints. In the Jason anim we use it to connect the heroglow visual particle to Jason's **FrontAbdomen**, but at other times we may use it to attach a sword to a models **RightHand** or **LeftHand**, a shield to a models **LeftForearm** or **RightForearm** or a predefined head to a models **TopOfHead**.

These attachpoints are defined when creating a model, and will vary from model to model (although the names are standard where they exist).

Finally in the Jason example, we see the use of the function **tag**. Tags are used in models that involve complex movements and are used as parameters to determine things like how big steps should be when walking or the timing of attack sequences. As the comments preceding these functions suggest these values are generated by the AoM animation tools and should not be changed.

One other function that we do not find in the Jason anim, but that will be used frequently in our modding activities is the **ReplaceTexture** function. This is the function we use to replace a models default texture with our modified texture and how we change the appearance of the models.

5.2 Detail of an Anim File

We will go into this in more detail later in this Chapter, but for now we will have a more detailed look at the Jason anim and discuss what each part of the anim is doing.

```
//=====
// ajax_anim.txt
//=====
```

In anims (and a number of other AoM files), the `///` denotes a comment line and normally you would put in the name of the file for reference (in this case they obviously edited from an ajax anim and forgot to change it).

```
//=====
define heroglow
{
    set hotspot
    {
        version
        {
            Visualparticle SFX A Hero Glow Small
        }
    }
}
}
```

The only entry in the definitions section for this anim is defining a visual particle called a heroglow. You can call define these by any name you like – usually you just call them what they are: head, sword, basket, etc. These definitions are then available to connect to the main model in the individual anim definitions. In this definition, there is no logic involved so you will notice that there is no SetSelector declaration. This will not always be the case.

You will also notice that in some anims, they will use the **Import** function. This allows you to import other anim files into your anim and is often used for items such as weapons, which are common across a number of units. These imports save you redefining these objects every time you want to use them.

```
//=====
anim Idle
{
    SetSelector
```

This tells the game to use this anim definition when the unit is idle.

```
{
```

ContainLogic

Jason has two models for use when he is idle. One for when he is carrying something (a relic) and one when he is not. ContainLogic is the function that the game uses to test if the unit is carrying something. The function will deliver the result No or Yes.

SetSelector

This next SetSelector is nested inside the first one and deals with a No response from the ContainLogic function.

```
{
  set hotspot
```

We then define the model sequence for this occurrence.

```
{
  version
```

And the frames used in the sequence (in this case there is only one frame).

```
{
  Visual Hero G Jason_idleA
```

We define the model we wish to use (in this case a reference to \models\hero g jason_idlea.brg).

```
Connect FRONTABDOMEN heroglow hotspot
```

And connect an embellishment in the form the heroglow visual particle define earlier.

```
    }
  }
}
SetSelector
```

We then use SetSelector again to manage the Yes response from the ContainLogic function (note that it is nested at the same level as the No response).

```
{
  set hotspot
  {
    version
    {
      Visual Hero G Jason_CarryIdle
      Connect FRONTABDOMEN heroglow hotspot
    }
  }
}
```

We then repeat the anim definition for this occurrence, using the appropriate model hero g jason_CarryIdle.brg.

```
  }
}

//=====
anim Bored
{
```

The anim Bored definition follows the same logic as the idle anim, but in this instance it will point to two different models hero g jason_boredA.brg or hero g jason_CarryBored.brg.

```

SetSelector
{
    ContainLogic
    SetSelector
    {
        set hotspot
        {
            version
            {
                Visual Hero G Jason_boredA
                Connect FRONTABDOMEN heroglow hotspot
            }
        }
    }
}
SetSelector
{
    set hotspot
    {
        version
        {
            Visual Hero G Jason_CarryBored
            Connect FRONTABDOMEN heroglow hotspot
        }
    }
}
}
}

```

```

//=====
anim attack
{

```

The anim attack definition only has one set of possible models. You will notice that in game play, units put down relics when fighting – the reason being that there are no carryattack models! The attack model requires the parameters passed by the two **tag** statements.

```

SetSelector
{
    set hotspot
    {
        version
        {
            Visual Hero G Jason_attackA
//-- auto generated by the bang animation tool
//-- do not hand edit these values
            tag Attack 0.32 true
            tag GenericSound 0.40 true
//-- end auto generated section
            Connect FRONTABDOMEN heroglow hotspot
        }
    }
}
}
}

```

```

//=====
anim death
{

```

The anim death definition is as simple as they get and simply tells the game to use the model file hero g jason_deathA.brg.

```

SetSelector

```



```

SetSelector
{
  set hotspot
  {
    version
    {
      Visual Hero G Jason_flailA
      Connect FRONTABDOMEN heroglow hotspot
    }
  }
}
}
}

```

While there are certainly more complex version of anims, they all conform to this basic structure.

In the case of Jason, we have an anim that creates a single look unit. That is, while different "action" versions of the model will be used the basic features of that model – textures and weapons will be constant. There are a number of ways of adding variety to the animations, which we discuss later in this Chapter. But for starters let us look at what is involved in using the Jason anim for our new Aragorn character.

5.3 *Modifying an Anim File*

For our first anim edit all we are going to do is make the changes necessary to change the appearance of Jason to that of Aragorn. We will do this by telling the anim to use a different texture when displaying the model by way of the **ReplaceTexture** function.

The ReplaceTexture function has two parameters old_texture and new_texture with the following syntax:

ReplaceTexture old_texture/new_texture.

In Chapter 6 we will create some new texture files for Aragorn, but to enable us to make our changes to the anim we first need to find out the name of the default texture used by the Jason anim. To do this:

Click on the Source Files\textures folder and do a search of the folder for all files containing "jason". You will get a list of seven (7) .DDT files:

```

hero g jason corpse bodya
hero g jason corpse skeletona
hero g jason head standard
hero g jason shield
hero g jason standard
hero g jason icon 64
hero g jason

```

If you remember back to the proto definition you will recall that the latter two files are the names of the icon files used in the proto definition. The remaining files are the texture files for this character. The two corpse textures are similar for all models but the other three

(head standard, shield and standard) are the textures we will be replacing with new Aragorn textures.

We will keep to the same naming conventions when we get to creating the new textures in Chapter 6, so the new commands we will be requiring when we create our new Aragorn_anim file will be three ReplaceTexture entries:

```
ReplaceTexture hero g jason standard/aragorn standard  
ReplaceTexture hero g jason head standard/aragorn head standard  
ReplaceTexture hero g jason shield/aragorn shield
```

So let us create the new anim:

- 1) In your **Source Files\anim** folder locate the **hero greek jason_anim.txt** file and **Copy** it, then **Paste** it into your **Mod Files\anim** folder.
- 2) **Rename** the version of the **hero greek jason_anim.txt** file you just placed in your **Mod Files\anim** folder to **aragorn_anim.txt**.
- 3) Open the **aragorn_anim.txt** file using Notepad.
- 4) Change the second line of the anim from **ajax_anim.txt** to **aragorn_anim.txt**.
- 5) Now scroll through the anim file and after every occurrence of the command

```
visual hero g jason_action
```

insert the three ReplaceTexture commands listed above.

For example, for the idle anim with ContainLogic of No you will end up with the following

```
version  
{  
    Visual Hero G Jason_idleA  
    ReplaceTexture hero g jason standard/aragorn standard  
    ReplaceTexture hero g jason head standard/aragorn head standard  
    ReplaceTexture hero g jason shield/aragorn shield  
    Connect FRONTABDOMEN heroglow hotspot  
}
```

You need to make sure you do this for every anim definition or you will find your character reverting back to the original Jason textures for some actions. There are a total of nine (9) insertions you will need to do.

- 6) Once you have made all the changes, save the **Aragorn_anim.txt** file and copy it to the **\anim** folder in the main **Age of Mythology** folder.

We now have an anim file for our new character.

5.4 More Advanced Anim Features

The Jason/Aragorn anim is very simple and often you will want to add variations into your anim files. A number of functions are available for use in anims to assist with this.

The first of these is **VariationLogic**. VariationLogic tells the game to cycle through a list of defined hot spots so that the look of the character varies. An example of this could be where you want some variety in your villager – let us say we want red heads, blondes and brunettes. We could achieve this variation by creating three new texture files called villager g female head (red/blonde/brown) and creating the following anim definition (I will use the anim idle for this example).

```
anim idle
{
  SetSelector
  {
    VariationLogic
    set hotspot
    {
      version
      {
        visual villager g female_idleA
        ReplaceTexture villager g female head/villager g female head red
      }
    }
    set hotspot
    {
      version
      {
        visual villager g female_idleA
        ReplaceTexture villager g female head/villager g female head blonde
      }
    }
    set hotspot
    {
      version
      {
        visual villager g female_idleA
        ReplaceTexture villager g female head/villager g female head brown
      }
    }
  }
}
```

You would need to repeat these definitions for all of the various anim definitions.

The resulting anim when called in the game would cycle through the three head formats.

Another use of the VariationLogic Command is for things like weapons. For example you may wish to base a unit on the greek hippikon but to have a mix of spear and sword carrying cavalry. You would do this by defining a weapon at the start of the anim and then connecting this weapon. For example:

```
define weapon
{
  SetSelector
  {
```

```

VariationLogic
set hotspot
{
  version
  {
    visual attachments a spear copper
  }
}
set hotspot
{
  version
  {
    visual attachments n sword standard
  }
}
}

```

You would then to a global edit to replace the existing `connect RightHand greekSword hotspot` with

```
connect RightHand weapon hotspot
```

When these units are created during a game, half will have an altantean spear and half a Norse sword as their weapon.

The TechLogic command allows for similar variations to be included but links the logic for inclusion of these variations to the availability of certain technologies.

The various *weapons_anim* files employ this technique, for example the `Greek Sword_anim`:

```

define GreekSword
{
  SetSelector
  {
    TechLogic none/Copper Weapons/Bronze Weapons/Iron Weapons
    set hotspot
    {
      version
      {
        Visual Attachments G Sword Standard
      }
    }
    set hotspot
    {
      version
      {
        Visual Attachments G Sword Copper
      }
    }
    set hotspot
    {
      version
      {
        Visual Attachments G Sword Bronze
      }
    }
    set hotspot
    {
      version
      {
        Visual Attachments G Sword Iron
      }
    }
  }
}

```

```
}  
}  
}
```

Similar approaches are taken with Shield defines and in the anims for most standard infantry units usual use TechLogic with the none/medium/heavy/champion technologies for the relevant military type to change the appearance of the units texture and head during game play.

Another specific variation technique is the use of the **ConstructionLogic** function. This is typically used with buildings to link anims to the different models to be used to represent the stages of building construction and destruction as required. The command uses the syntax:

```
ConstructionLogic %1 %2 %3 ... where %1,%2, %3 etc and integers between 0 and 99
```

for example:

```
ConstructionLogic 0 25 50 75
```

would use five hotspots for the variations stages of construction or destruction.

If you look at the anim for any building you will how this is employed in the anim idle and anim death definitions.

With these functions there is no end to the level of variation you can create in your animations and the increased visual interest you can build into your game play. For example, the characters standing behind Aragorn (ROTK) and Faramir in the picture below are 8 of the 15 possible variations of a Men of Gondor mod I made using VariationLogic.



Similarly by linking a unit's armor, helmet, shield and sword separately to the relevant technologies you can have 256 different versions of a unit during the course of a game.

It is just a matter of creating the textures.

6. Textures

The quality of the textures you use in your mods will make or break them, but you do not need to be an artist to make reasonable textures, it just takes a bit of patience. If you can create a new texture from scratch all the better, but there are plenty of photos available on the Internet or in books that you can use to cut and paste a new texture for you characters, as is show by the example below:

Original Artwork



Finished Unit



The trick is understanding how the texture files work and how they map to the models used in the game. Once you get the hang of this you can build up a wide range of characters using pictures of various examples of chain mail, plate mail, various armors, shields and helmets from history or from the movies.

The starting point is the texture files for the unit you plan to base your new character on, as these will have the layouts required to map correctly over the models your new character will use.

Most characters use a number of different texture files that are mapped to the various models used in building the character. Typically separate texture files are used for the body and the head (to give the head a sharper definition), with additional textures used for attachments such as shields, weapons and capes. Cavalry units will also have a separate texture file for the horse. Units also have icon textures that contain the picture of the character to be displayed in game play. These usually come in 32x32 bit and 64x64 bit formats.

Characters that change in appearance through the course of the game (i.e. as technologies are acquired) may use multiple versions of each texture type. To check on the relevant textures for the character you wish to base your mod on, do a search for files that include the characters name in your **AoM Mods\Source Files\Textures** folder. While ES has not always stuck to a standard naming convention usually the following names are used:

Body

character standard map
character copper map
character bronze map
character iron map

Head

character head standard
character head copper
character head bronze
character head iron

Cape

character cape standard
character cape copper
character cape bronze
character cape iron

Horse

character horse standard
character horse copper
character horse bronze
character horse iron

Shield

attachment *c* shield standard
attachment *c* shield copper
attachment *c* shield bronze
attachment *c* shield iron

where *c* = g (greek), n (norse), e (egyptian), a (atlantean)

Weapon

attachment *c* *weapon* standard
attachment *c* *weapon* copper
attachment *c* *weapon* bronze
attachment *c* *weapon* iron

where *c* = g (greek), n (norse), e (egyptian), a (atlantean); and
weapon = sword, spear, axe, lance, etc.

Icons

character icon 32
character icon 64

Each character unit also has textures for its corpse body and corpse skeleton. These are fairly standard and I find they do not warrant replacement.

6.1 Converting Textures to Bitmaps

To create the textures for our Aragorn Character:

- 1) Do a search on "Jason" in the AoM Mods\Source Files\textures folder. You will get a list of seven (7) .DDT files:

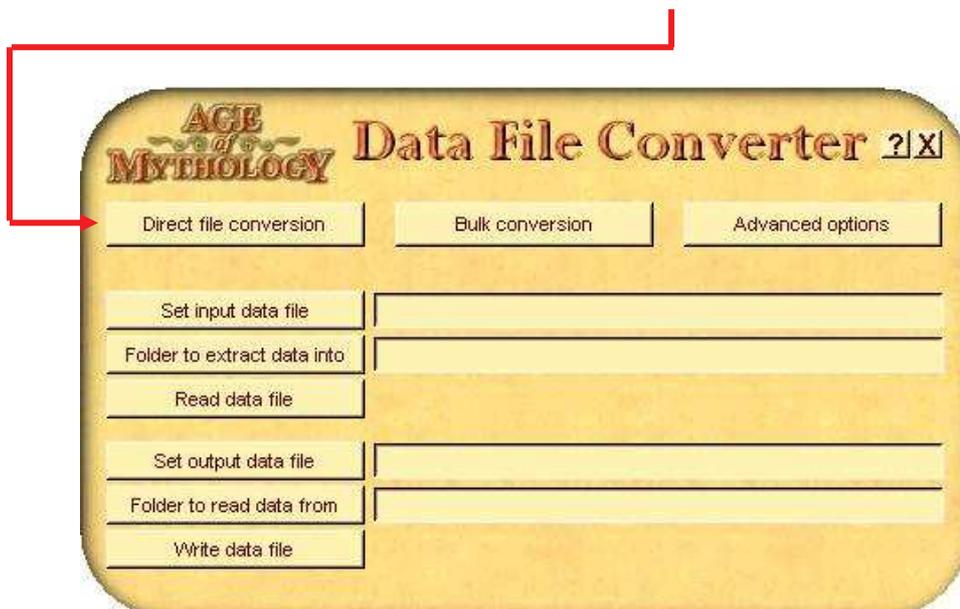
```
hero g jason corpse bodya
hero g jason corpse skeletona
hero g jason head standard
hero g jason shield
hero g jason standard
hero g jason icon 64
hero g jason
```

The first two "corpse" files we will not worry about but we want to **Copy** the next three (3) files and **Paste** then to our **AoM Mods\Mod Files\textures** folder and the last two files we want to **Copy** and **Paste** to our **AoM Mods\Mod Files\textures\icons** folder.

The first three files are the textures for Jason's head, shield and body. The latter two files are Jason's 64x64 bit and 32x32 bit icon files.

Now rename each of the copied files, replacing **hero g jason** with **aragorn** (in the case of the 32x32 bitmap call this aragorn icon 32).

- 2) The files we have copied are in .DDT format and need to be converted to bitmaps so that we can modify them.
- 3) Launch **Ykkrosh's AOM Data File Converter** by double clicking the relevant icon in your **AoM Mods** folder, and click the **Direct file conversion** button.



- 4) You will then be asked to **Select data file for input**. Go to your **Mod Files\textures** folder and locate the **aragorn head standard** file and click **Open**.
- 5) A pop-up window will inform you that it is **Converting to BMP – select an output file in the following window**. Press the **OK** button, and then press the **Save** button on the following window.

This will save the file to the default file name – **aragorn head standard.BMP** and you will be prompted with the message **DDT image format "16-bit, 0 alpha [0], 5 mip-map levels (remember this number)**. This information is required when you convert bitmaps to .DDT format.

I keep track of these numbers by renaming the .bmp file to include this information.

For example rename **aragorn head standard.bmp** to **aragorn head standard 16 0 [0] 5**.

- 6) Press **OK**, and repeat steps 3 – 5 for the aragorn shield and aragorn standard files in the **Mod Files\textures** folder; and for the two icon files in the **Mod Files\textures\ icons** folder.

Note that a number of different image formats will be used in the conversion so make sure you use the right one when renaming your files. You should end up with the following five (renamed) bitmap files:

AoM Mods\Mod Files\textures

aragorn head standard 16 0 [0] 5.bmp
aragorn shield 15 1 [1] 5.bmp
aragorn standard 15 1 [1] 5.bmp

AoM Mods\Mod Files\textures

aragorn icon 32 p16 0 [2] 1.bmp
aragorn icon 64 p16 0 [2] 1.bmp

Do not worry about what the formats mean at this stage, as normally you will just stick to the format of the unit you base your mod on. Basically the different formats just tell the game (p)alette whether to use an expanded palette set or not, whether the image uses transparent areas, player colors (or not) and how many facets are included in the bitmap. The only real thing to remember is that if you get them wrong the texture will not work.

Image Size

AoM can handle texture sizes up to 256x256bits, which equates to a 256x512 bit image if a format that supports transparency or player colors is used. The larger the image size the greater the detail you will be able to obtain from your in-game characters.

Most of the standard textures used in AoM are of a very low resolution, however these can be scaled up when modding. This is what we will be doing with our Aragorn textures.

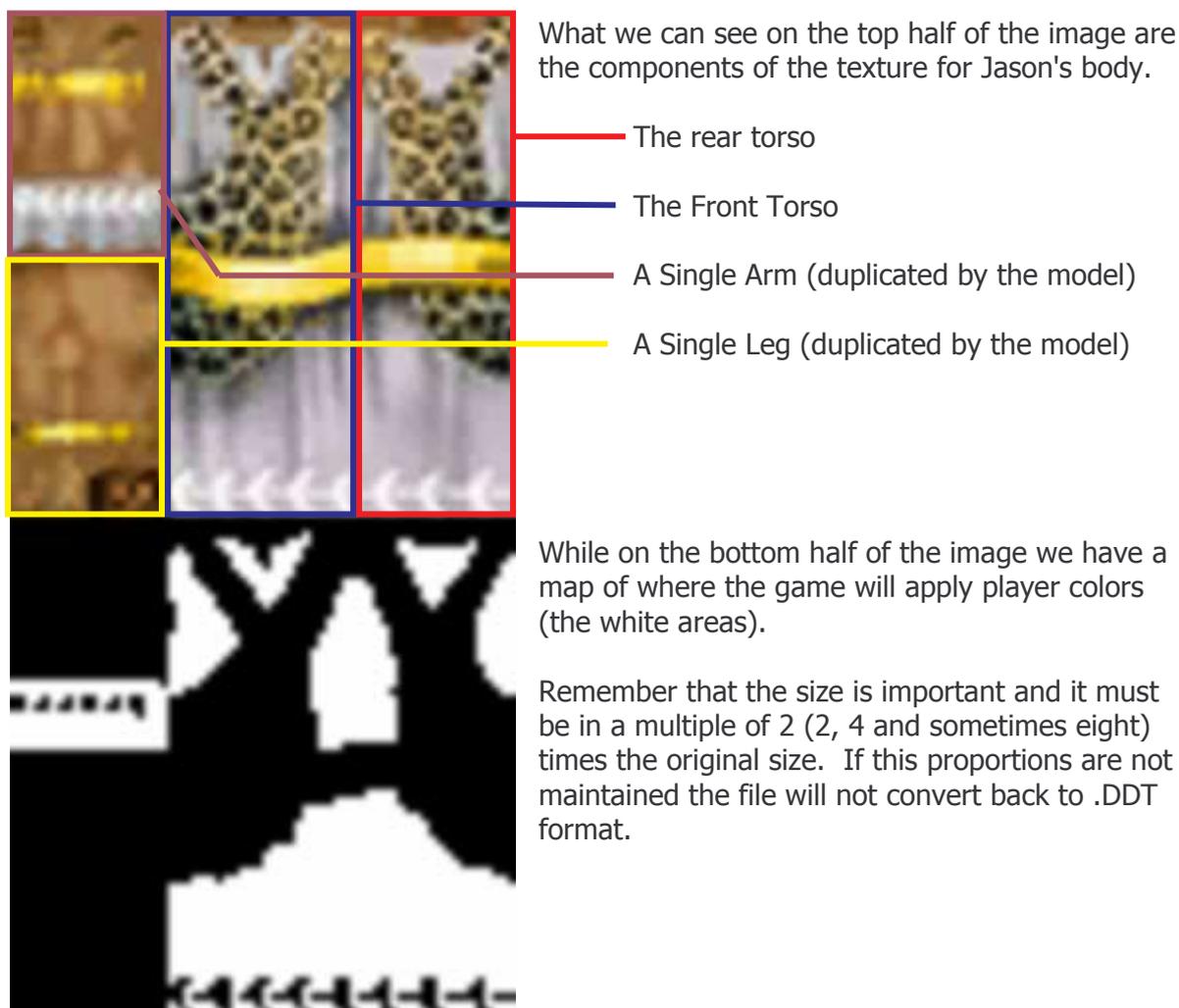
6.2 Creating a New Body Texture

To create Aragorn's body texture we are going to enlarge the jason body bit map and using some pre-developed artwork make a new texture.

- 1) In the **AoM Mods\Mod Files\textures** folder right-click on the **aragorn standard.bmp** file and Open With **Paint**. You will see the following (small) bitmap.



- 2) From the **Paint** menu select [**Image**], **Stretch/Skew** and change the **Vertical** and **Horizontal Stretch** to 400 (e.g. we want the image to be 4 times larger).



- 3) To create our aragorn body texture we are simply going to cut and past the segments for a new aragorn texture from the images provided with this guide. The following shows the four segments contained in the **aragorn body segments.bmp** file (note: the following is only a jpeg copy of this file).



Find this file and open it using **Paint**.

- 4) In the copy of **Paint** that has the **aragorn standard** bitmap, select [**Image**] from the menu and turn off **Draw Opaque** (there should be no tick next to it).
- 5) Now go into the **aragorn body segments.bmp** and one section at a time, lay it over the original segment (arm – top left), leg (bottom left), front torso (middle) rear torso (right).

- 6) Now fill the bottom half of the bitmap with black (we do not want player colors). Just draw a solid black square to do this.

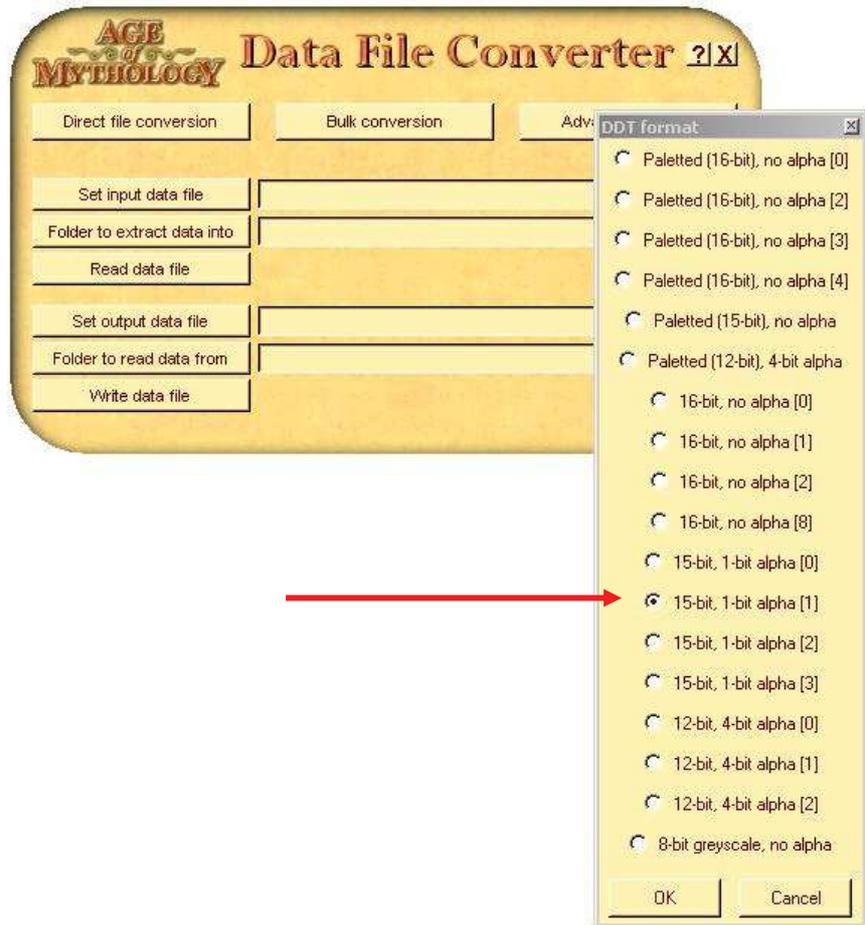
The complete bit map should look like this.



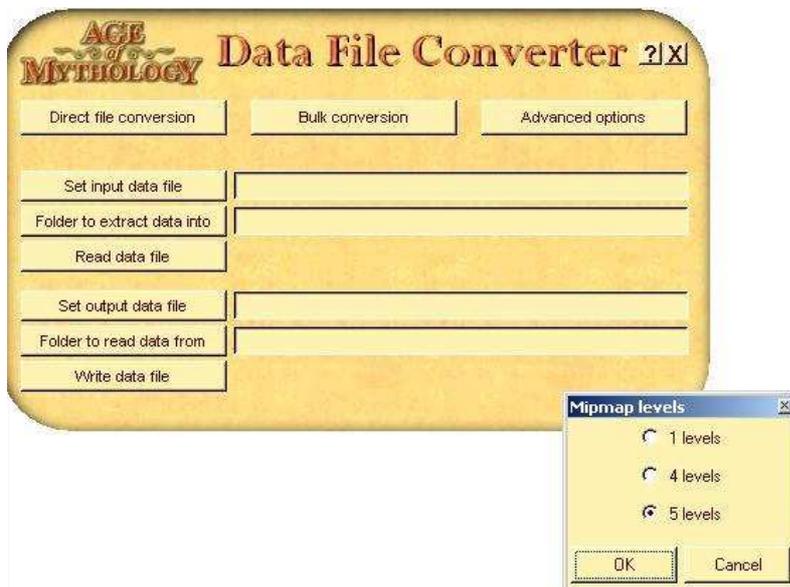
- 7) Save the file.
- 8) Launch **Ykkrosh's AOM Data File Converter** if it is not already running, and click the **Direct file conversion** button. You will be prompted with **Select data file for input**. Find the file we just created (**aragorn standard 15 1 [1] 5.bmp**) in the **AoM Mods\Mod Files\textures** folder and click **open**.
- 9) You will then be prompted with the message **Converting to DDT – select an output file in the following window**. Click **OK**.
- 10) The next screen will prompt **Select data file to create**. Type in **aragorn standard.ddt**.

Remember to type the ".ddt" or it will save it to the wrong format file.

- 11) A pop-up box will then prompt you for the DDT format. Click the button next to the correct format (in this case **15-bit, 1-bit alpha [1]**) and then **OK**.



- 12) A second pop-up box will prompt for the number of Mipmap levels (in this case **5**) and press OK.



- 13) You will then be prompted **Conversion finished**. Press **OK**, and close/minimize the Editor. You will now have a file in your **AOM Mods\Mod Files\textures** folder called **aragorn standard.ddt**. Copy this file to the **\textures** folder in your standard Age of Mythology folder.

Unless you mistakenly messed up the proportions of the bitmap you should have no problems with the conversion to .DDT format. If it fails to convert:

- a) Check the attributes of the bitmap. Its size in pixels should be 256 wide and 512 high.
- b) Make sure you used the correct DDT Format and Mipmap levels.

OK, that is the general process for making new textures.

- 1) Convert the original .DDT file to a bitmap using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**.
- 2) If required, increase the size of the file using the Stretch/Skew function so that the file is either 32, 64, 128 or 256 pixels wide and 32, 64, 128 or 256 pixels high for 16-bit textures (the square ones) and 64, 128, 256 or 512 pixels high for the 15-bit textures (the rectangular ones). The larger the size, the sharper the definition of your new character.
- 3) Make the appropriate changes to the texture, using the initial texture areas as your guide.
- 4) Convert the new .bmp file back into .DDT format using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**.

We now need to repeat this process for aragorn's head and shield.

6.3 *Creating a New Head Texture*

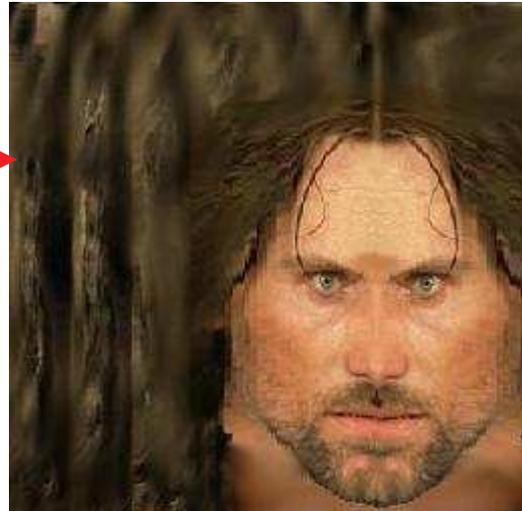
- 1) Open the aragorn head standard 16 0 [0] 5 bitmap and stretch it to 800% (8 times its initial size). The stretch function only allows stretching up to 500% so you will need to stretch it by 400% and then again by 200%. Check its attributes, and it should now be 256 pixels wide and 256 pixels high. It will look like this. 

The head texture essentially sits on a three-sided face model with the blue bordered section representing the face and the purple bordered area representing the sides and back of the head (e.g. it is



replicated on both sides). The main trick with a new face texture is to ensure both sides of the head match up with the face and that the level of the eyes and mouth are in proportion. If you allow a small gap for where the neck would be the eyes are roughly half way down the texture and the mouth three-quarters of the way down (much like a human face).

- 2) Open the **aragorn head segment.bmp** file provided with the guide, select all of it and **Copy** and then **Paste** it into the **aragorn head standard** bitmap. It should now look like this:



- 3) Save the file.
- 4) Using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**, convert the bitmap to a .DDT format using filename = **aragorn head standard.ddt**, DDT format **16-bit no alpha [0]** and **Mipmap levels = 5**.

- 5) Copy this file to the \textures directory in your default Age of Mythology folder.

6.4 Creating a New Shield Texture

- 1) Open the aragorn shield 15 1 [1] 5 bitmap and stretch it to 400% (4 times it initial size). Check its attributes, and it should now be 256 pixels wide and 256 pixels high. It will look like this.

The shield texture consists of four segments the front of the shield (top left), the back of the shield (top right), the player coloring for the front of the shield (bottom left) and the player coloring for the back of the shield (bottom right).



- 2) Open the **aragorn shield segment.bmp** file provided with the guide, select all of it and **Copy** and then **Paste** it into the **aragorn shield** bitmap. It should now look like this:



You will notice that we are only changing the front of the shield (top left) and removing its player colors (bottom left)

- 3) Save the file.
- 4) Using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**, convert the bitmap to a .DDT format using:

filename = **aragorn shield.ddt**
 DDT format = **15-bit 1 alpha [1]**
 Mipmap levels = **5**.

- 5) Copy this file to the \textures directory in your default Age of Mythology folder.

6.5 Icons

Unlike the other texture files the Icon files used in AoM are placed directly into the user interface screens and need to be of a set size. In the case of user icons these are 32x32 pixel and 64x64 pixel formats, however for major and minor god icons we also use 128x128 pixel and 256x256 pixel formats.

When making character icons, the quickest approach is create the 64x64 pixel icon; save it; shrink it to 50%; and save it again as the 32x32 pixel icon file.

- 1) Open the **aragorn icon 64 p16 0 [2] 1.bmp** file from your AoM Mods\Mod Files\textures\icons folder. It will look like this.  It is a portrait icon – that is what you see is what you get.
- 2) Open the **aragorn icon segment.bmp** file provided with this guide; select all on the bitmap; copy and paste it to the **aragorn icon 64** bitmap, which will then look like this: 
- 3) Make sure the files attributes show that width and height are both 64 pixels and **Save** the file.
- 4) Using the **stretch/skew** function reduce the image to 50% of its current size (width and height) and Save as **aragorn icon 32 16 0 [2] 1.bmp**. You will need to over-right the current file.
- 5) Using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**, convert the bitmap to a .DDT format using filename = **aragorn icon 64.ddt**, DDT format **Paletted 16-bit no alpha [2]** and **Mipmap levels = 1**; and repeat these steps for the smaller icon using filename = **aragorn icon 32.ddt**, DDT format **Paletted 16-bit no alpha [2]** and **Mipmap levels = 1**;
- 6) Copy these two new .DDT files **aragorn icon 32.ddt** and **aragorn icon 64.ddt** to the **\textures\icons** directory in your default Age of Mythology folder.

We have now created the new textures for our Aragorn character and you should have the following files in you Age of Mythology folder:

```
\texture\aragorn standard.ddt  
\texture\aragorn head standard.ddt  
\texture\aragorn shield.ddt  
\texture\icons\aragorn icon 32.ddt  
\texture\icons\aragorn icon 6432.ddt
```

We now just need to add the in-game text, and our modification is complete.

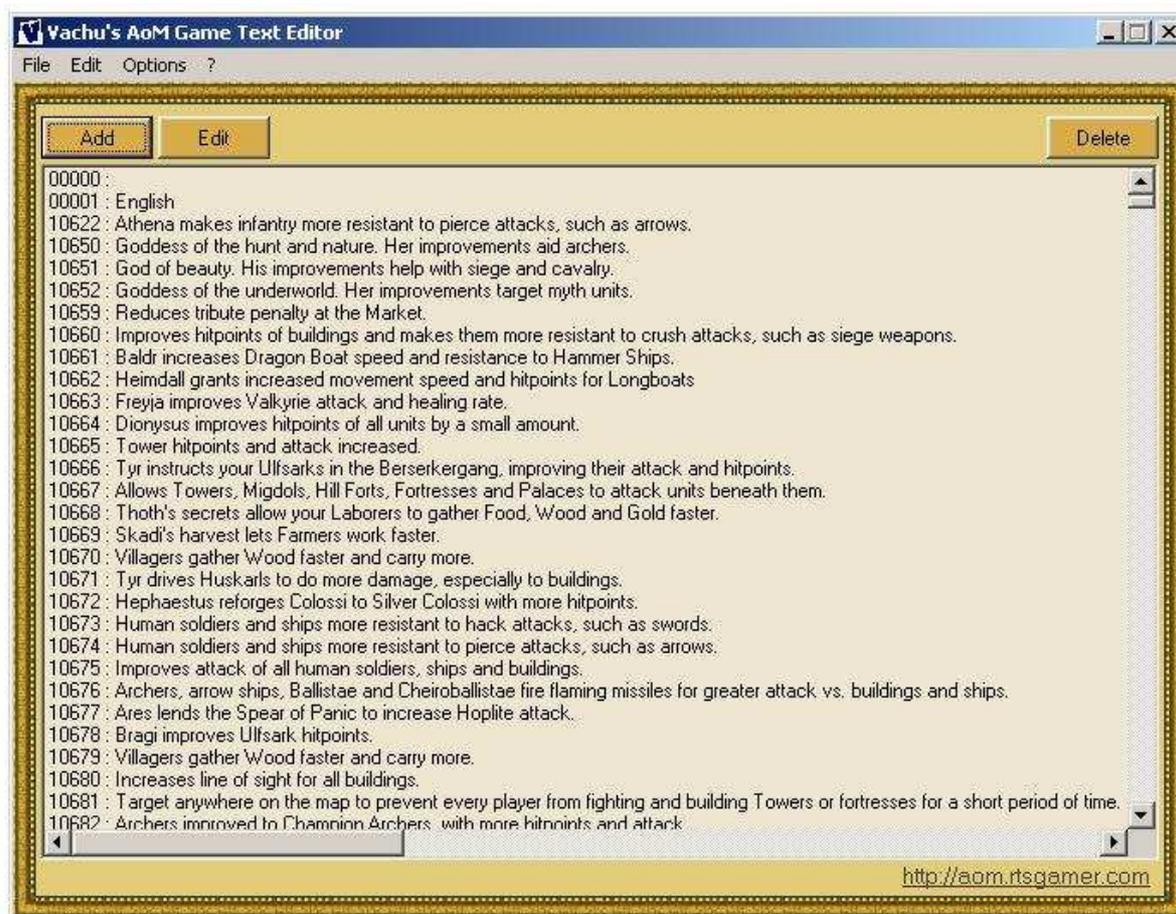
7. In-Game Text

IMPORTANT NOTE: The Game Text Editor will automatically create a new language.dll file when you save any changes and **it will save this file into your default Age of Mythology folder**. So before making any changes go into your Age of Mythology folder and locate the files language.dll (AoMTT and AoM) and xpacklanguage.dll (AoMTT only) and rename these files for back-up purposes, e.g. languageold.dll, xpacklanguageold.dll. If you want to revert to these files at a future date, just delete any new files you may have created and rename these files back to their original names.

The unit names, messages and other texts used throughout the game are held in a separate file called xpacklanguage.dll in AoMTT and language.dll in AoM. This file is simply an indexed list of messages in the format *text_id text*.

7.1 Adding New In-Game Text

To edit this file we use **Vachu's AOM Game Text Editor**. Double-clicking the Game Text Editor icon in you **AoM Mods** folder will bring up the following screen.



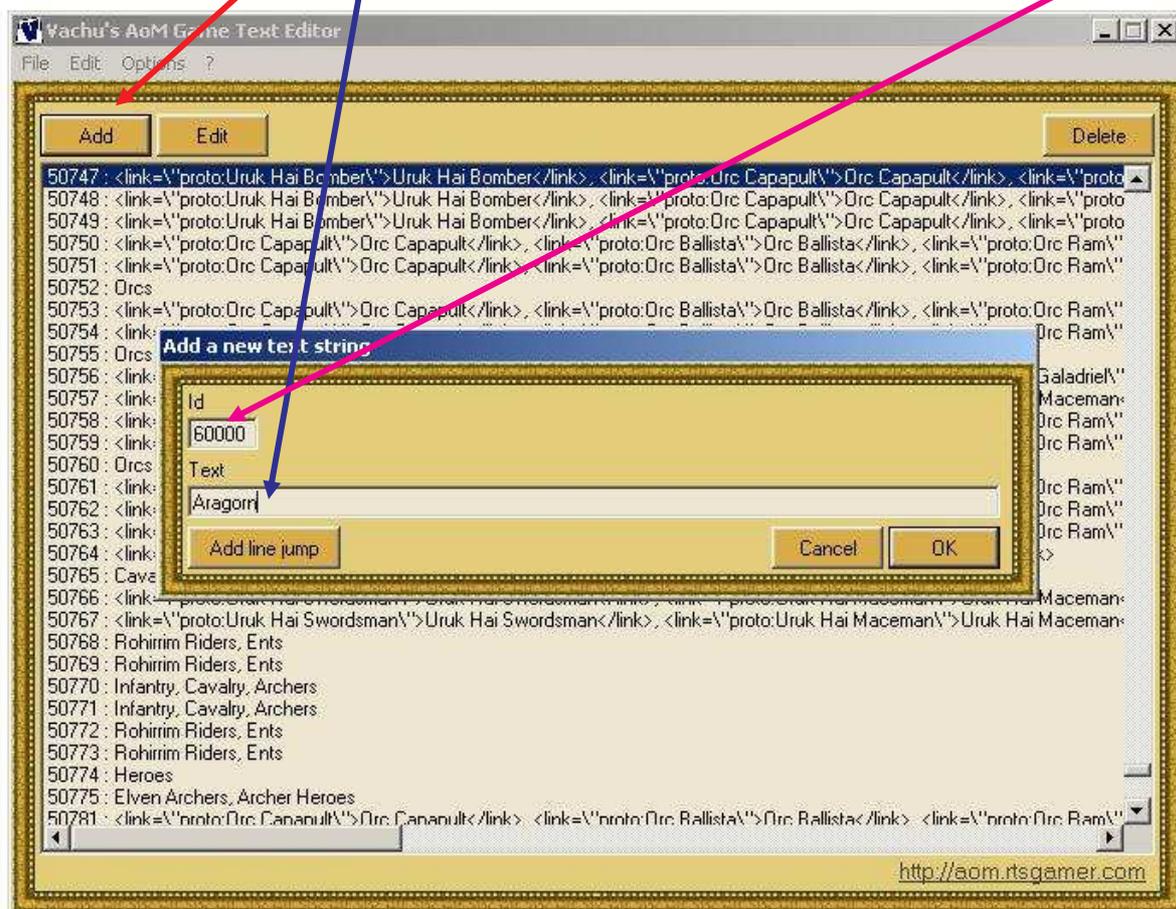
This screen allows us to **Add**, **Delete** or **Edit** text entries and on Saving the file a new **language.dll** file will be created.

Text messages can be simple text or use hot links to overview files held in your **\History** folder. This linking feature allows you to navigate around the unit information screens during game play – e.g. you can look at information on the character, click on the building where upgrades for the character are undertaken and click on specific technologies available from that particular building.

For our Aragorn character we need to create the six (6) text entries we made reference to in our proto unit definition:

```
<displaynameid>60000</displaynameid>  
<rollovertextid>60001</rollovertextid>  
<rolloverbonusdamageid>60002</rolloverbonusdamageid>  
<rolloveruseagainstid>60003</rolloveruseagainstid>  
<rollovercounterwithid>60004</rollovercounterwithid>  
<rolloverupgradeatid>60005</rolloverupgradeatid>
```

Our first text entry is the name we want displayed for our character (Aragorn). To enter this we press the **Add** button in **Vachu's AOM Game Text Editor**. Enter **60000** in the **Id** field and **Aragorn** in the **Text** field



Then press OK.

Now repeat this for our rollovertextid – this is the message that is displayed in the bottom left hand corner when the unit is selected.



For our next four text messages we want to create links to other information held in the game.

The first of these – the rolloverbonusdamageid – refers to the units where we defined a special damage bonus when defining the proto units attack. For Aragorn we gave a generic damage bonus for all units of type Myth Unit. For our text message we want to make a link the topic of Myth Units.

Within the text editor we can make links to files held in the history(2) folder by defining a link to either a topic – a file in the history2 folder or a proto – a file in the history2\units, history2\techs or history2\mods sub-folders.

The format of these links is

```
<link=\"topic:filename\">linktext</link> or  
<link=\"proto:filename\">linktext</link>
```

where filename is the name of the file from the History2 folder or its sub-folders and linktext is the text that will appear on the screen (highlighted and enabled as a hot link).

Hot links can be concatenated using a mixture of topic and proto references:

```
<link=\"topic:archers\">archers</link>, <link=\"proto:Axeman\">Axeman</link>,  
<link=\"proto:Hypaspist\">Hypaspist</link>, <link=\"proto:Throwing  
Axeman\">Throwing Axeman</link>
```

This will set links to the topic of archers and to the unit descriptions of the key ranged unit within the game.

A mixture of text and links can also be used, for example.

```
Wall, <link=\"proto:Settlement Level 1\">Town Center</link>
```

The in-game text will display Wall, Town Center with a hot link on the Town Center entry.

For Aragorn we will just do a simple link to the Myth Unit Topic, as follows:



For the RolloverUseAgainstId we can use a link to both Myth Units and Human Soldiers (after all Aragorn is great against anyone!)

So add another text string with Id=60003 and Text = <link=\\\"topic:myth units \\\">myth units </link>,<link=\\\"topic:human soldiers \\\">human soldiers </link>.

For the RolloverCounterWithId we will just create a link to Heroes. This will tell opponents the best unit to attack Aragorn with. So add our next text string Id=60004 and Text = <link=\\\"topic:Heroes\\\">Heroes</link>.

Our final text string for this unit is the RolloverUpgradeAtId. This tell players which buildings to use to upgrade the Aragorn unit. Note we have not created a building to create Aragorn or added any technologies to upgrade him yet, so for the time being we will just use text entries. Our last text string will be Id = 60005 and Text = House of Elrond, Elven Armory.

Once you have made the changes click **File** and **Save** in the Text Editor. This will create a new **language.dll** file in your default Age of Mythology folder. If you are using AoMTT you will need to rename this file to **xpacklanguage.dll**.

8. Histories

The last thing we need to do is to create a history file for Aragorn. We do this by creating a file in our **AoM Mods\Mod Files\history** folder call **Aragorn.txt**. The contents of this file follow the same format as we used in the text entries – they can be freeform text or links to topics or proto descriptions. For the time being, just Copy the following Text and Paste it into your **Aragorn.txt** file. Then save the file and copy it into the **history2\units** folder in your default Age of Mythology folder.

A descendant of the lost line of the ancient kings of Men, Aragorn is fated to one day claim the empty throne of Gondor. Aragorn is a mighty warrior, wielding his blade with great adeptness in defense of Helm's Deep. He fights with passion and bravery, but also with wisdom, which earns him the respect and admiration of Théoden, King of the Rohirrim. In The Return of the King, he will face several challenges that will determine the fate of Middle-earth.

9. Sounds

Each unit in AoM(TT) has a sound definition file that associates standard or your own sound files to specific actions or situations the unit encounters. Sound file names use the format ***unitname_snd.xml*** with the converted version called ***unitname_snd.xmb***. Both production files are stored in the default **Age of Mythology\sound** folder.

The following is the sound file for our Aragorn character.

```
<?xml version="1.0" encoding="UTF-8"?>
<protounitsounddef>
  <protounit name="Aragorn">
    <soundtype name="Select">
      <soundset name="MilitaryAtlanteanSelect"></soundset>
    </soundtype>
    <soundtype name="Grunt">
      <soundset name="MaleGrunt"></soundset>
    </soundtype>
    <soundtype name="Hit">
      <damagetypelogic>
        <choice name="Crush">
          <soundset name="CrushFlesh"></soundset>
        </choice>
        <choice name="Fire">
          <soundset name="HackFlesh"></soundset>
        </choice>
        <choice name="Hack">
          <soundset name="MetalSlice"></soundset>
        </choice>
        <choice name="Pierce">
          <soundset name="PierceFlesh"></soundset>
        </choice>
        <choice name="Slash">
          <soundset name="MetalSlice"></soundset>
        </choice>
      </damagetypelogic>
    </soundtype>
    <soundtype name="Death">
      <soundset name="MaleDeath"></soundset>
    </soundtype>
    <soundtype name="Creation">
      <soundset name="HeroBirth"></soundset>
    </soundtype>
    <soundtype name="Acknowledge">
      <targetlogic>
        <choice name="default">
          <soundset name="MilitaryAtlanteanAcknowledge"></soundset>
        </choice>
        <choice name="enemy">
          <soundset name="MilitaryAtlanteanAttack"></soundset>
        </choice>
      </targetlogic>
    </soundtype>
  </protounit>
</protounitsounddef>
```

It is simply a copy of the standard atlantean military unit sound file with the **protounit** name field set to our new proto unit name (Aragorn).

The various soundset names are help in a file called soundsets-xpack.xml (AoMTT) or soundset.xml (AoM) and these contain a list of all the soundsets and the sound files they relate to.

The following is an extract from the start of this file. You can get this by using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**, to convert the .XMB version of the file to .XML format.

```
<?xml version="1.0" encoding="UTF-8"?>
<soundsetdef>
  <soundset maxnum="3" name="GatherChop" volume="0.4000">
    <sound filename="Woodchop1.wav" volume="1.0000"></sound>
    <sound filename="Woodchop2.wav" volume="1.0000"></sound>
    <sound filename="Woodchop3.wav" volume="1.0000"></sound>
  </soundset>
  <soundset maxnum="3" name="GatherFlay" volume="0.1500">
    <sound filename="meatgather1.wav" volume="1.0000"></sound>
    <sound filename="meatgather2.wav" volume="1.0000"></sound>
  </soundset>
  <soundset maxnum="3" name="GatherMine" volume="0.5000">
    <sound filename="mine1.wav" volume="1.0000"></sound>
    <sound filename="mine2.wav" volume="1.0000"></sound>
    <sound filename="mine3.wav" volume="1.0000"></sound>
  </soundset>
  <soundset maxnum="2" name="Arrow" volume="0.3000">
    <sound filename="Arrow1.wav" volume="1.0000"></sound>
    <sound filename="Arrow2.wav" volume="1.0000"></sound>
    <sound filename="Arrow3.wav" volume="1.0000"></sound>
    <sound filename="Arrow4.wav" volume="1.0000"></sound>
    <sound filename="Arrow5.wav" volume="1.0000"></sound>
  </soundset>
```

If you want to add new sounds, you need to store them in the sounds folder and create a new unique `<soundset ..>` definition. Once defined this soundset can be referred to from any of the **__snd.xmb** files.

For now just create a file name aragorn_snd.txt in your Mod Files\sound folder, and copy and paste the earlier details. Save the file and using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter**, to convert the .XML version of the file to .XMB format.

Then place a copy of both files into the \sound folder of your default Age of Mythology folder.

10. Checking Our New Unit

Before we check out our new unit, let us just do a quick check:

- 1) Our new **PROTOX.XML** or **PROTO.XML** file has been successfully converted to .XMB format and the new **PROTO(X).XMB** file has been copied into the **\data** folder in the default Age of Mythology folder.
- 2) Our new anim file – **aragorn_anim.txt** – has been copied to the **\anim** folder in the default Age of Mythology folder.
- 3) The three character textures for aragorn – standard, head standard and shield have been successfully converted to .DDT format and the .DDT files have been copied to the **\textures** folder in the default Age of Mythology folder.
- 4) The two icon textures for aragorn – aragorn icon 32 and aragorn icon 64 have been successfully converted to .DDT format and the .DDT files have been copied to the **\textures\icons** folder in the default Age of Mythology folder.
- 5) Our new in-game text has been added and saved to create a new **language.dll** file in the default Age of Mythology folder, and if we are using AoMTT this file has been renamed **xpacklanguage.dll**.
- 6) Our new history file **aragorn.txt** has been created and copied into the **history\units** folder (AoM) or **history2\units** folder (AoMTT) in the default Age of Mythology folder.

Once each of these nine (9) files is correctly installed in the default Age of Mythology folder you are ready to test the new character.

- 1) Log into AoM(TT) and go into the Editor. Select Place Unit and the Aragorn character should be on the list and on selecting this character you should be able to place it.

If not

If Aragorn is not in the list of available units, the game is not picking up the new PROTO(X) file and you should recheck that you placed it in the correct folder. It should be in the AGE OF MYTHOLOGY\DATA folder.

If the characters are there, but their names are preceded by three asterisks (e.g. *** Aragorn), the game is not picking up the xpacklanguage.dll file. It should be in the default Age of Mythology folder (e.g. not in a sub-folder).

If the character's name is present but the unit is blank the game is either not picking up the anim file or the anim file cannot find the relevant model. In case of the former, the anim file is probably in the wrong place (check that they are in your Age of Mythology\anim directory). In the latter, it may be that the anim is calling a model that you do not have installed. If so, check that you did not accidentally change the name of the original hero g jason visual definitions).

If the unit is there, but looks like an existing unit (e.g. if Aragorn still looks like Jason) then the game is not picking up the new texture files. Check that they are in the AGE OF MYTHOLOGY\TEXTURES directory.

Unless you were just missing a texture you will need to exit AoM(TT) fix the problem and restart.

Otherwise you should have a character that looks like this:



- 2) Now click on the **Scenario Menu** and select **Scenario Data** and then change the Scenario settings to not use **Victory Conditions** (there should be no x next to it). Then Play Test the Scenario.
- 3) When you select Aragorn, his icon should be displayed in the bottom center of the screen. If it is not there or is garbled then either the icon is not in the **Age of Mythology\textures\icons** folder or when it was converted to .DDT format the incorrect format or mipmap levels were set.
 - a) Check that the file is in the right place (if not copy it into the correct folder).
 - b) If it is there, recheck your two icon bitmaps and make sure they are 32x32 pixels and 64x64 pixels.
 - c) Convert your bitmaps again using the instructions at the end of Chapter 6.

Note that Texture files are loaded dynamically so you **do not** need to restart the game.

- 4) When you place your mouse over Aragorn or his icon the rollovertext should be displayed in the bottom left hand corner. If not check that the changes you made for text id 60001 are OK. If you did not get "*** before Aragorn in the unit placement the language file is being picked up OK. You will need to restart the game to pick up any changes.
- 5) Double-click on the Aragorn Icon and this will bring up the in-game help. In the middle section of the help screen will be the other rollover texts you created, click on the hot links to check that they work. If they are not there, recheck the text entries (ids 60002-5) making sure you check that the text id and the syntax of the link statements is correct. You will need to restart the game to pick up any changes.
- 6) The bottom of the screen should display the text from Aragorn's history file. If not make sure that the **aragorn.txt** file is in the correct folder (**\history** for AoM and **\history2** for AoMTT). These files are dynamically loaded so you do not need to restart.

Congratulations you have just created a new character!

11. Buildings

In order to use a character in normal Single Player mode you need somewhere to train (create) the character. You can achieve this by modifying an existing building's proto definition (the easiest way), or by creating a new building.

When creating a new building we end up with a chicken and egg situation whereby in order to create a new building we need a builder – which again can be an existing or new character and so forth. The result of this is that to create a building the chain of events that leads to that building must start with a character that is created in a Town Center that is placed during game initialization. As we will discuss later, this can be our own customized Town Center as part of our own culture, but for the time being we are going to look firstly at adding our unit to an existing building, and secondly at creating a new building and enabling a standard AoM(TT) character to build it.

11.1 Enabling an Existing Building to Train a New Character

To begin with we are going to modify the **Longhouse** protounit to enable it to train Aragorn. The **Longhouse** is **unit id="485"** in AoMTT and **unit id="421"** in AoM.

The following is a copy of the proto unit description for the Longhouse. You will notice that the definition is similar to that used for normal characters. The main difference is the center section of the definition. This is the part we are interested in. It defines units that can be trained at this building and the technologies that can be researched there, and is discussed in more detail later. The differences between Building and Character definitions are discussed through the definition.

```
<unit id="485" name="Longhouse">
  <dbid>2210</dbid>
  <displaynameid>19701</displaynameid>
  <icon>icon building barracks</icon>
  <maxcontained>5</maxcontained>
```

The maxcontained field is used for buildings that can garrison other units and equals the total population of the units that can be contained in the building.

```
<initialhitpoints>1200.0000</initialhitpoints>
<maxhitpoints>1200.0000</maxhitpoints>
<los>9.0000</los>
<portraiticon>Building Barracks Icon 64</portraiticon>
<obstructionradiusx>4.0000</obstructionradiusx>
<obstructionradiusz>4.0000</obstructionradiusz>
<deadreplacement>Destruction 4x4</deadreplacement>
```

When buildings are destroyed they leave a rubble mound on the ground. The deadreplacement field tells the game which anim to use for this. There are 15 anims named **destruction xxxx.txt** that you can use. Usually the file is determined by the buildings size, e.g. Destruction 4x4, Destruction 8x8 etc.

```
<maxvelocity>0.0000</maxvelocity>
```

Buildings do not move so their maxvelocity is zero.

```
<movementtype>land</movementtype>
<buildpoints>30.0000</buildpoints>
<buildingworkrate>1.0000</buildingworkrate>
```

Each unit or technology has a number of points that determine how long it will take to train/research. The buildingworkrate determines how many buildpoints per second the building will achieve. This parameter is useful when developing technologies that improve the production speed for all units and technologies from a given building. Instead of reducing the trainpoints for each unit and technology you can get the same effect by increasing the buildings workrate. God powers such as pestilence can also manipulate this value to slow down production.

```
<allowedheightvariance>4.0000</allowedheightvariance>
```

This parameter is used to determine how steep a slope the building can be built on. In this case an allowedheightvariance of 4 roughly equates to a 45% angle.

```
<allowedage>2</allowedage>
<cost resourcetype="Wood">110.0000</cost>
<bounty resourcetype="Favor">4.3200</bounty>
<bountyfactor resourcetype="Favor">1.0000</bountyfactor>
<bounty resourcetype="Gold">5.0000</bounty>
<rollovertextid>16548</rollovertextid>
<rollovercounterwithid>17685</rollovercounterwithid>
<rolloverupgradeatid>17729</rolloverupgradeatid>
<buttonpos column="0" page="2" row="0"></buttonpos>
<decay delay="0.0000" duration="1.0000"></decay>
```

The decaydelay determines how long (in minutes) the deadreplacement model will remain in use after the destruction of the building.

```
<armor damagetype="Hack" percentflag="1">0.30</armor>
<armor damagetype="Pierce" percentflag="1">0.96</armor>
<armor damagetype="Crush" percentflag="1">0.05</armor>
<allowedculture>Norse</allowedculture>
<unittype>LogicalTypeVolcanoAttack</unittype>
<unittype>LogicalTypeSuperPredatorsAutoAttack</unittype>
<unittype>LogicalTypeAffectedByRestoration</unittype>
<unittype>LogicalTypeTornadoAttack</unittype>
<unittype>LogicalTypeSuperPredatorsAttack</unittype>
<unittype>LogicalTypeConvertsHerds</unittype>
<unittype>LogicalTypeFimbulWinterTCEvalType</unittype>
<unittype>LogicalTypeEarthquakeAttack</unittype>
<unittype>LogicalTypeSiegeAutoAttack</unittype>
<unittype>LogicalTypeNeededForVictory</unittype>
<unittype>LogicalTypeHandUnitsAutoAttack</unittype>
<unittype>LogicalTypeBuildingNotTitanGate</unittype>
<unittype>LogicalTypeValidDeconstructionTarget</unittype>
<unittype>LogicalTypeBuildingsThatTrainMilitary</unittype>
<unittype>LogicalTypeBuildingsNotWalls</unittype>
<unittype>LogicalTypeBuildingsNotHouses</unittype>
<unittype>LogicalTypeTimeshift</unittype>
<unittype>LogicalTypeRangedUnitsAutoAttack</unittype>
<unittype>LogicalTypeVillagersAttack</unittype>
<unittype>LogicalTypeHandUnitsAttack</unittype>
<unittype>LogicalTypeRamAttack</unittype>
<unittype>LogicalTypeRangedUnitsAttack</unittype>
<unittype>LogicalTypeTartarianGateValidOverlapPlacement</unittype>
<unittype>LogicalTypeValidForestFireTarget</unittype>
```

```

<unittype>LogicalTypeSeaSerpentAttack</unittype>
<unittype>LogicalTypeValidMeteorTarget</unittype>
<unittype>LogicalTypeMinimapFilterMilitary</unittype>
<unittype>Building</unittype>
<unittype>BuildingClass</unittype>
<unittype>MilitaryBuilding</unittype>
<unittype>AbstractBarracks</unittype>
<unittype>ConvertibleBuilding</unittype>
<unittype>Age2Building</unittype>

```

The above unittype definitions are used to determine which building technologies are available.

```

<train column="1" page="1" row="0">Throwing Axeman</train>
<train column="0" page="1" row="0">Ulfsark</train>
<train column="3" page="1" row="0">Raiding Cavalry</train>
<train column="2" page="1" row="0">Hero Norse</train>
<tech column="0" page="1" row="1">Medium Infantry</tech>
<tech column="0" page="1" row="1">Heavy Infantry</tech>
<tech column="0" page="1" row="1">Champion Infantry</tech>
<tech column="0" page="1" row="2">Lone Wanderer</tech>
<tech column="0" page="1" row="2">Eyes in the Forest</tech>
<tech column="4" page="1" row="1">Thundering Hooves</tech>
<tech column="1" page="1" row="2">Call Of Valhalla</tech>
<tech column="1" page="1" row="1">Huntress Axe</tech>
<tech column="3" page="1" row="1">Heavy Cavalry</tech>
<tech column="3" page="1" row="1">Medium Cavalry</tech>
<tech column="3" page="1" row="1">Champion Cavalry</tech>
<tech column="5" page="1" row="1">Sons of Sleipnir</tech>
<tech column="3" page="1" row="2">BerserkerGang</tech>
<tech column="2" page="1" row="2">Swine Array</tech>
<tech column="2" page="1" row="1">Hall of Thanes</tech>
<tech column="5" page="1" row="0">Levy Longhouse Soldiers</tech>
<tech column="5" page="1" row="0">Conscript Longhouse Soldiers</tech>
<tech column="4" page="1" row="2">Axe of Muspell</tech>

```

This is the train/tech section we will discuss in more detail below.

```

<flag>PaintTextureWhenPlacing</flag>
<flag>NoIdleActions</flag>
<flag>ObscuresUnits</flag>
<flag>HasGatherPoint</flag>

```

This flag lets you set a Gather Point for units created at the building.

```

<flag>Immoveable</flag>
<flag>NoBloodOnDeath</flag>
<flag>NonAutoFormedUnit</flag>
<flag>CollidesWithProjectiles</flag>
<flag>DontFadeInOnBuild</flag>
<flag>Doppeld</flag>
<flag>InitialGarrisonOnly</flag>

```

This flag tells the game that only units created at the building can be garrisoned there.

```

<flag>SelectWithObstruction</flag>
<flag>FlattenGround</flag>

```

This flag tells the game to flatten the terrain around the building before placing it.

```
<\unit> <flag>Tracked</flag>
```

11.2 Using Building to Train Units

In the standard proto definition the Longhouse can train 4 units – Throwing Axeman, Ulfsark, Raiding Cavalry and Hero Norse (Hersirs). On each line you will notice that it defines the column, page, and row. These definitions tell the game where to locate the train/tech icon for each particular unit and technology.

```
<train column="1" page="1" row="0">Throwing Axeman</train>
<train column="0" page="1" row="0">Ulfsark</train>
<train column="3" page="1" row="0">Raiding Cavalry</train>
<train column="2" page="1" row="0">Hero Norse</train>
<tech column="0" page="1" row="1">Medium Infantry</tech>
<tech column="0" page="1" row="1">Heavy Infantry</tech>
<tech column="0" page="1" row="1">Champion Infantry</tech>
<tech column="0" page="1" row="2">Lone Wanderer</tech>
<tech column="0" page="1" row="2">Eyes in the Forest</tech>
<tech column="4" page="1" row="1">Thundering Hooves</tech>
<tech column="1" page="1" row="2">Call Of Valhalla</tech>
<tech column="1" page="1" row="1">Huntress Axe</tech>
<tech column="3" page="1" row="1">Heavy Cavalry</tech>
<tech column="3" page="1" row="1">Medium Cavalry</tech>
<tech column="3" page="1" row="1">Champion Cavalry</tech>
<tech column="5" page="1" row="1">Sons of Sleipnir</tech>
<tech column="3" page="1" row="2">BerserkerGang</tech>
<tech column="2" page="1" row="2">Swine Array</tech>
<tech column="2" page="1" row="1">Hall of Thanes</tech>
<tech column="5" page="1" row="0">Levy Longhouse Soldiers</tech>
<tech column="5" page="1" row="0">Conscript Longhouse Soldiers</tech>
<tech column="4" page="1" row="2">Axe of Muspell</tech>
```

These icons are displayed on the bottom left hand side of the screen when the building is selected; and as you may remember, this area allows for up to 3 rows of 5 icons, with the ability for a second page.

The column therefore defines the icons position horizontally (1-5), the page defines the page number (1-2) and the row defines the vertical position (0-2).

When adding additional units of technologies to a building the only trick is to pick a location not already in use. If you pick a location already used the last reference to that position will take it, and the other character or technology will be unavailable. If you look at the last two technologies you will notice that they take the same position – in this case it is intentional as the "Levy" technology is the prerequisite for the "Conscript" technology. When this icon is activated by the **techtree(x)** the "Levy" technology will already be active and the "Levy" icon no longer required.

In the **Longhouse** definition there is a gap at **column="4" page="1" row="0"**. We will use this gap to insert a train statement for Aragorn.

11.3 Modifying a Building's Proto Definition

- 1) In your **AoM Mods\Mod Files** folder open the **PROTO(X).XML** file.
- 2) Scroll down to the train section for unit **485** (PROTOX) or unit **421** (PROTO)
- 3) Between the train definition for Hero Norse and the Tech definition for Medium Infantry insert the train statement for Aragorn as follows:

```
<train column="2" page="1" row="0">Hero Norse</train>  
<train column="4" page="1" row="0">Aragorn </train>  
<tech column="0" page="1" row="1">Medium Infantry</tech>
```

making sure that you use the correct column, page and row values.

- 4) Save the File and using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter** convert the saved PROTO(X).XML file into .XMB format.
- 5) Copy the new **PROTO(X).XMB** file from your **AoM Mods\Mod Files\data** folder to the default **Age of Mythology\data** folder.

This building now has the capability to build Aragorn – once the protounit has been enabled by the Techtree(X) as discussed in Chapter 11.

This approach works fine when you are creating a single unit, however to expand your character while retaining the standard AoM protounits eventually you will want to be creating your own buildings.

11.4 Creating a New Building Proto Definition

Creating a building involves the same process as making a unit – create a new proto definition, anim, textures, text, and history. For the sake of this example we are going to keep our new building simple creating a **House of Elrond** based on a **Hill Fort** (unit id **489** (AoMTT) or unit id **425** (AoM)). Note: AoMTT users who want a more elven looking building may prefer to base their **House of Elrond** on the **Palace** proto unit (unit id 706).

- 1) Open the **PROTO(X).XML** file and select the protounit you wish to use and **Copy** the full definition beginning with

```
<unit id="489" (or 425 or 706)  
.....  
.....  
to the corresponding <\unit>.
```

- 2) Go to the end of the **PROTO(X).XML** and between the last **<\unit>** (the end of the Aragorn definition) and the **<\proto>** statement, **Paste** the copied definition.
- 3) On the first line of the copied definition change the **unit id** to **802** and the **name** to **"House of Elrond"**.

- 4) Change the second line to `<dbid>2849</dbid> 2654` for vanilla AoM.
- 5) Change the four (4) references to in-game texts to the numbers we will be using.

```

<displaynameid>60006</displaynameid>
.....
.....
<rollovertextid>60007</rollovertextid>
<rollovercounterwithid>60008</rollovercounterwithid>
<rolloverupgradeatid>60009</rolloverupgradeatid>

```

- 6) Change the `<allowedage>` field to **1**.

```

<allowedage>1</allowedage>

```

- 7) Go to the train/tech section and delete all train/tech definitions other than the first train entry.

- 8) Change the train entry to

```

<train column="1" page="1" row="0">Aragorn </train>

```

- 9) You will notice that whereas the Longhouse had no actions associated with it there are three actions defined for both the Hill Fort and Palace (Hand Attack, Ranged Attack and Enter) just before the `</unit>` definition.

The first action – Hand Attack is similar to that used in the character definition but with a higher maximum range. This range needs to be higher as the game associates the center of the building with its location (for calculating range). As the building has a 5x5 footprint a maximum range of 1 would mean that units would not get close enough to the building to be within its hand attack range. Also note that the third parameter is "Inactive". This means that the building has no hand attack in its initial Age 1 format – it is made active using the **Boiling Oil** technology. This will often be the case, as you may want certain actions to be deferred to later in the game. The important thing to remember is that they still need to be defined up-front.

```

<action name="HandAttack">
  <param name="MaximumRange" value1="4.0"></param>
  <param name="Rate" type="All" value1="5.0"></param>
  <param name="Inactive"></param>
  <param name="Damage" type="Pierce" value1="10"></param>
  <param name="AttackAction"></param>
  <param name="DamageBonus" type="Ship" value1="3"></param>
</action>

```

The Ranged Attack is also similar to the action we discussed in Chapter 3, however it introduces two useful parameter options `VolleyMode` and `NumberProjectiles`. The `NumberProjectiles` determines how many projectiles will be fired from the building and `VolleyMode` signifies that all projectiles will be fired in synchronized volleys (this is preferred when rushed by a large number of units as at least one projectile in a volley has a good chance of hitting the target).

```

<action name="RangedAttack">
  <param name="MinimumRange" value1="4"></param>
  <param name="MaximumRange" value1="18"></param>
  <param name="Damage" type="Pierce" value1="10"></param>

```

```

<param name="Accuracy" value1="0.9"></param>
<param name="Rate" type="All" value1="1.0"></param>
<param name="AttackAction"></param>
<param name="MaxSpread" value1="5.0"></param>
<param name="SpreadFactor" value1="0.1"></param>
<param name="TrackRating" value1="5.0"></param>
<param name="UnintentionalDamageMultiplier" value1="0.5"></param>
<param name="AccuracyReductionFactor" value1="1.5"></param>
<param name="AimBonus" value1="15"></param>
<param name="VolleyMode"></param>
<param name="NumberProjectiles" value1="3.0"></param>
<param name="DamageBonus" type="Ship" value1="6"></param>
<param name="HeightBonusMultiplier" value1="1.25"></param>
</action>

```

The Enter action, is a standard requirement for any building used to garrison units. The range determines how far away the unit has to be (from the center of the building) to be garrisoned.

```

<action name="Enter">
  <param name="MaximumRange" value1="10"></param>
</action>

```

If you used the Longhouse as the basis of your building insert these 3 actions just before the House of Elrond's </unit> definition.

- 10) As the House of Elrond is also a House of Healing, after the last action , (<action name ="enter">) and before the </unit> definition insert the following new action

```

<action name="Heal">
  <param name="MaximumRange" value1="10.0"></param>
  <param name="Rate" type="LogicalTypeCanBeHealed" value1="2.0"></param>
</action>

```

This actions states that any injured unit of LogicalTypeCanBeHealed will be healed at the rate of 2 hitpoints per second in within 10 units of the center of the building.

That completes the definition of the House of Elrond. Now we need to let someone build it.

To do this we are going to change the proto unit definition for Villager Norse (AoMTT unit id = **484**, AoM unit id = **420**).

- 11) Find the relevant protounit in your PROTO(X).XML file and scroll down to the line that says:

```

<train column="0" page="1" row="0">Farm</train>

```

after this line insert the train command for the House of Elrond

```

<train column="1" page="1" row="0">House of Elrond</train>

```

Now scroll down a bit further to the action definition for Build and insert the Rate information for House of Elrond.

```

<action name="Build">
  <param name="Rate" type="Farm" value1="1.0"></param>

```

—————> `<param name="Rate" type="House of Elrond" value1="1.0"></param>`
`</action>`

All we have done here is added another type of building that the villager can build/train and defined how many of its build points per second the builder will accomplish. Now instead of only being able to build farms Villager Norse will be able to build a House of Elrond.

- 11) Save the PROTO(X).XML file.
- 12) Launch **Ykkrosh's AOM Data File Converter** and using the **Direct file conversion** option, convert the PROTO(X).XML file to .XMB format.

If for some reason the conversion fails this will almost certainly due to a syntax error. So check.

- a) That the unit definition preceding the new House of Elrond unit (Aragorn) ends with `</unit>`
- b) That the House of Elrond unit definition ends with `</unit>`
- c) That immediately after the Aragorn unit definition is the end of protounit definitions `</proto>` and that there is nothing after it.
- d) That the other lines changed have the correct syntax in terms of the definition start and finish.

If the conversion was successful copy the new PROTO(X).XMB file to the `\data` folder in your standard AoM folder.

This version of the PROTO(X).XMB file will enable Aragorn to be created at a standard Norse Longhouse or at the House of Elrond. Note: If you want to keep your standard AoM building "pure", you can delete the changes you made to the Longhouse definition.

11.5 Creating a New Building Anim

For the time being, we do not need to do anything fancy with our new building – we just want to a place to create Aragorn. As such all we need to do for our **House of Elrond**_anim is:

- 1) Find the anim for our base unit in the AoM Mods\Source Files\anim directory (**Longhouse_anim.txt** or **Palace_anim.txt**).
- 2) **Copy** the file and **Paste** it to your **AoM Mods\Mod Files\anim** folder.
- 3) Rename the file to **House of Elrond_anim.txt**.
- 4) Copy the new **House of Elrond_anim.txt** file to the default **Age of Mythology\anim** folder.

This anim will create a direct replica of the original building when a House of Elrond is placed, but will use the new proto unit definition allowing Aragorn to be trained.

11.6 Customizing the Building

To customize this building you would need to re-edit the **house of elrond_anim.txt** file and include the relevant **ReplaceTexture** statements to allow you to link the chosen model to a new set of textures.

If you based it on the **Hill Fort**, after every occurrence of:

Visual Building N Hillfort **AND** Visual Building N Hillfort Age4

you would insert the line

ReplaceTexture building n hillfort/house of elrond

You would then use the **building n hillfort** texture as the basis of your house of elrond texture.

Note: the other visuals are used as in-construction models and do not need their textures changed.

If you based it on the **Palace**, after every occurrence of:

Visual Building X Palace Age2 **AND** Visual Building X Palace Age3

you would insert the lines

ReplaceTexture building x palace age2/house of elrond
ReplaceTexture building x palace roof/house of elrond roof

OR

ReplaceTexture building x palace age3/house of elrond
ReplaceTexture building x palace roof age3/house of elrond roof

You would then use the **building x palace age2** and **building x palace roof** textures as the basis of your house of elrond texture.

To create new textures for the House of Elrond you would:

- 1) Copy the relevant texture files from the **AoM Mods\Source Files\textures** folder to the **AoM Mods\Mod Files\textures** folder:

building n hillfort.ddt

OR

building x palace age2.ddt

building x palace age roof.ddt

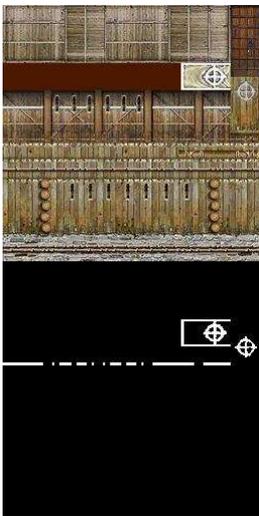
- 2) Rename the files **house of elrond.ddt** and **house of elrond roof.ddt** (if using the Palace).
- 3) Using the **Direct file conversion** option of **Ykkrosh's AOM Data File Converter** convert the files from .DDT to .BMP format, as discussed in Chapter 6 and remembering to record the **DDT format** and **Mipmap levels**.

15-bit, 1-bit alpha [1], 5 mip-map levels for the house of elrond.bmp

16-bit, 0-bit alpha [0], 5 mip-map levels for the house of elrond roof.bmp

Depending on your base unit you will have the following bitmaps:

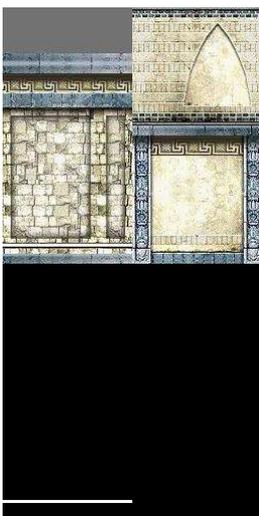
Hillfort (50% scale)



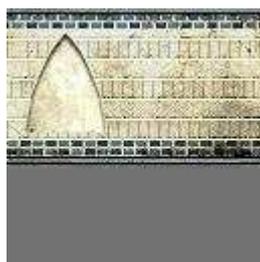
In-game View



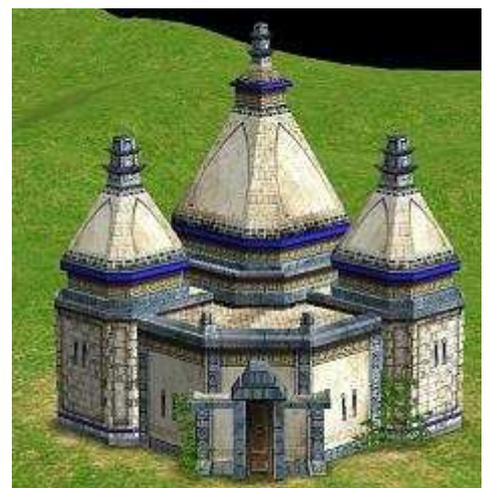
Palace (50% scale)



Palace Roof (100% scale)



In-game View



- 4) Look at the textures and the in-game view and you will see how the various frames in the textures are mapped to the building. You will also notice that not all of the features are shown. For example, the hill fort roof - this texture is probably derived

from a file call woodroof.bti, which is a general purpose texture used in the game which cannot be modified. Similarly with the palace the window slits are part of the model and are therefore fixed (as I assume is the portico – I cannot find where it gets these defined). Other features such as the vines on the Palace are defined in the anim file and use separate texture files.

With careful editing you can change the various frames to modify the look of your building. Artwork or samples of textures (marble for example) could be used to this end.

- 5) Once the texture(s) are modified you would convert the files to .DDT format using the correct **DDT Format** and **Mip-map** settings, and then copy them to the default **Age of Mythology\textures** folder.

11.7 Adding The Building's In-Game Text.

Earlier when we created the proto unit definition for the House of Elrond we made reference to four (4) in-game texts:

```
<displaynameid>60006</displaynameid>  
<rollovertextid>60007</rollovertextid>  
<rollovercounterwithid>60008</rollovercounterwithid>  
<rolloverupgradeatid>60009</rolloverupgradeatid>
```

To add these we again use **Vachu's AOM Game Text Editor** as described in Chapter 7.

We need to make the following new text entries

- 1) **Displaynameid** – this is just a plain text entry.

```
id = 60006  
text = House of Elrond
```

- 2) **Rollovertextid** –another plain text entry.

```
id = 60007  
text = The Last Homely House of Elrond at Rivendell
```

- 3) **Rollovercounterwithid** – against buildings you usually recommend siege weapons, so we need a link to this topic.

```
id = 60006  
text = <link="topic:Siege Weapons">siege weapons</link>
```

- 4) **Rolloverupgradeatid** – General building upgrades (increased hitpoints and crush armor) such as **Masons** apply improvement to buildings with a unit type `LogicalTypeBuildingNotTitanGate` (see the proto unit definition we made earlier). These upgrades are initiated at the **Town Center** in default AoM/AoMTT.

```
id = 60007
```

text = `<link="proto:Settlement Level 1">Town Center</link>`

- 5) Once you have made the changes click **File** and **Save** in the Text Editor. This will create a new **language.dll** file in your default **Age of Mythology** folder. If you are using AoMTT you will need to rename this file to **xpacklanguage.dll**.

11.8 Adding the Buildings History

The final step in creating our building mod is to create a history file for the unit. We do this in the same way we created our character history in Chapter 8, by creating a file in our **AoM Mods\Mod Files\history** folder call **House of Elrond.txt**. The contents of this file follow the same format as we used in the text entries – they can be freeform text or links to topics or proto descriptions. For the time being, just Copy the following Text and Paste it into your **House of Elrond.txt** file. Then save the file and copy it into the **history2\units** folder in your default Age of Mythology folder.

House of Elrond or the Last Homely House was home to Elrond, his sons Elladan and Elrohir, and daughter Arwen, located in the valley of Rivendell.

11.9 Check the New Building

Before we check out our new building, let us just do a quick check:

- 1) Our new **PROTOX.XML** or **PROTO.XML** file has been successfully converted to .XMB format and the new **PROTO(X).XMB** file has been copied into the **\data** folder in the default Age of Mythology folder.
- 2) Our new anim file - **house of elrond_anim.txt** – has been copied to the **\anim** folder in the default Age of Mythology folder.
- 3) If we created new textures for our building the textures for have been successfully converted to .DDT format and the .DDT files have been copied to the **\textures** folder in the default Age of Mythology folder.
- 4) If you created new icon textures (great initiative!), that these have been successfully converted to .DDT format and the .DDT files have been copied to the **\textures\icons** folder in the default Age of Mythology folder.
- 5) Our new in-game text has been added and saved to create a new **language.dll** file in the default Age of Mythology folder, and if we are using AoMTT this file has been renamed **xpacklanguage.dll**.
- 6) Our new history file **house of elrond.txt** has been created and copied into the **history\units** folder (AoM) or **history2\units** folder (AoMTT) in the default Age of Mythology folder.

Once each of these files is correctly installed in the default Age of Mythology folder you are ready to test the new building.

- 1) Log into AoM(TT) and go into the Editor. Select Place Unit and the House of Elrond character should be on the list and on selecting this character you should be able to place it.

If not

If House of Elrond is not in the list of available units, the game is not picking up the new PROTO(X) file and you should recheck that you placed it in the correct folder. It should be in the AGE OF MYTHOLOGY\DATA folder.

If the characters are there, but their names are preceded by three asterisks (e.g. *** House of Elrond), the game is not picking up the xpacklanguage.dll file. It should be in the default Age of Mythology folder (e.g. not in a sub-folder).

If the building's name is present but the unit is blank the game is either not picking up the anim file or the anim file cannot find the relevant model. In case of the former, the anim file is probably in the wrong place (check that they are in your Age of Mythology\anim directory). In the latter, it may be that the anim is calling a model that you do not have installed. If so, check that you did not accidentally change the name of the original visual definitions).

If you created new textures, but looks like the existing unit, then the game is not picking up the new texture files. Check that they are in the AGE OF MYTHOLOGY\TEXTURES directory.

Unless you were just missing a texture you will need to exit AoM(TT) fix the problem and restart.

- 2) Now click on the **Scenario Menu** and select **Scenario Data** and then change the Scenario settings to not use **Victory Conditions** (there should be no x next to it). Then Play Test the Scenario.
- 3) If you created a new icon, when you select the House of Elrond, this icon should be displayed in the bottom center of the screen. If it is not there or is garbled then either the icon is not in the **Age of Mythology\textures\icons** folder or when it was converted to .DDT format the incorrect format or mipmap levels were set.
 - a) Check that the file is in the right place (if not copy it into the correct folder).
 - b) If it is there, recheck your two icon bitmaps and make sure they are 32x32 pixels and 64x64 pixels.
 - c) Convert your bitmaps again using the instructions at the end of Chapter 6.

Note that Texture file are loaded dynamically so you do not need to restart the game.

- 4) When you place your mouse over the House of Elrond or its icon the rollovertext should be displayed in the bottom left hand corner. If not check that the changes you made for text id 60007 are OK. If you did not get "**** before House of Elrond in the unit placement the language file is being picked up OK. You will need to restart the game to pick up any changes.

- 5) Double-click on the House of Elrond Icon and this will bring up the in-game help. In the middle section of the help screen will be the other rollover texts you created, click on the hot links to check that they work. If they are not there, recheck the text entries (ids 60008-9) making sure you check that the text id and the syntax of the link statements is correct. You will need to restart the game to pick up any changes.
- 6) The bottom of the screen should display the text from the history file. If not make sure that the **house of elrond.txt** file is in the correct folder (**\history** for AoM and **\history2** for AoMTT). These files are dynamically loaded so you do not need to restart.

Congratulations you have just created a new building!

At this stage, in actual game play, the building will not be able to create Aragorn and the Villager Norse will not be able to build the House of Elrond, but we have checked that from a placement perspective that the mod has worked. To use the characters in our games, we need to enable these protounits. To do this, we must make changes to the TECHTREE(X) to enable these protounits.

12. Enabling Proto Units

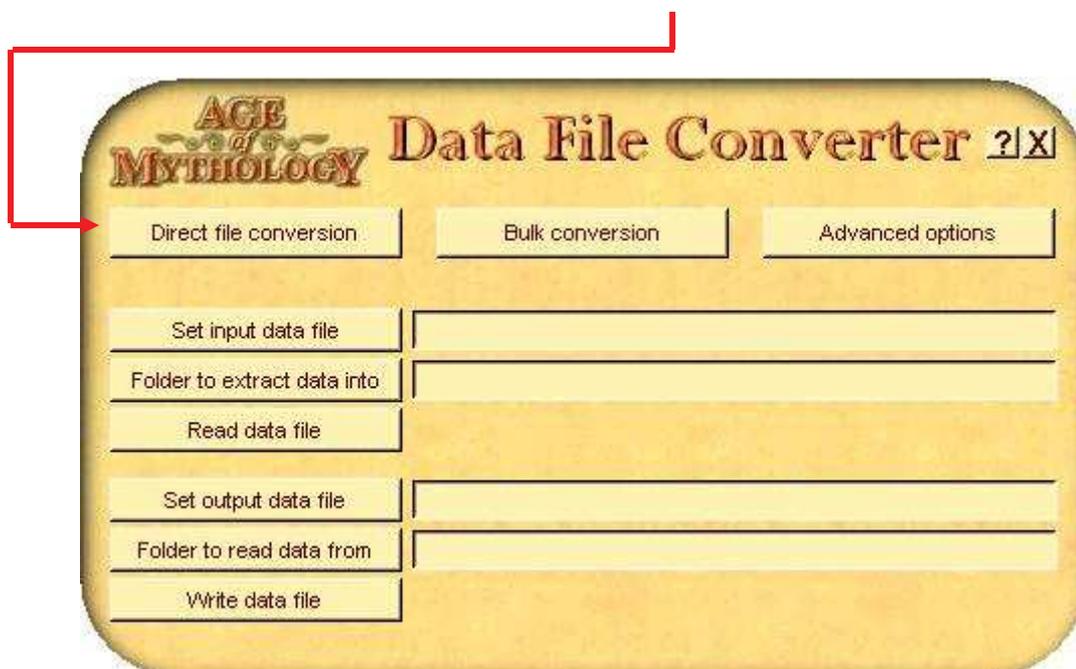
To use our proto units we need to modify the default TECHTREE(X) file to enable these units for game play.

In Chapter 12, we will be looking at Technologies in more detail, however for now we are simply interested in the changes we need to make to support our modifications.

12.1 Converting the TECHTREE(X) File

The first thing we need to do is to convert the TECHTREE(X).XMB file to a .XML format that we can edit.

- 1) Launch **Ykkrosh's AOM Data File Converter** by double clicking the relevant icon in your **AoM Mods** folder, and click the **Direct file conversion** button.



- 2) You will then be asked to **Select data file for input**. Go to your **Source Files\data** folder and locate the **TECHTREE.XMB** file (AoMTT) or **TECHTREE.XMB** file (AoM) and click **Open**.
- 3) A pop-up window will inform you that it is **Converting to XML – select an output file in the following window**. Press the **OK** button, and then press the **Save** button on the following window. This will save the file to the default file name – **TECHTREE(X).XML**. The conversion will start and a pop up window will advise **Conversion Finished** when complete. Press **OK** and close (⌘) the **AOM Data File Converter**.
- 4) Copy this file from your **AoM Mods\Source Files\data** folder to your **AoM Mods\Mod Files\data** folder. You can retain the version in Source Files as an original.

You now have a file **TECHTREE(X).XML** that can be edited using **Notepad** or any other .XML editor.

12.2 Enabling New ProtoUnits

To enable our new units we need to understand the technology structure that determines when our characters are to be enabled. This will be discussed in detail in Chapter 12, for now we will focus on two parameters we set in our proto unit definitions:

```
<allowedage>1</allowedage>
<allowedculture>Norse</allowedculture>
```

These indicate that we want these protounits to be available to the Norse Culture in Age 1.

Most unit enabling takes place in the Age Upgrade Technologies and for the Norse Culture, one of three Age 1 technologies:

```
Age 1 Odin
Age 1 Thor
Age 1 Loki
```

If we open out TECHTREE(X).XML file with notepad and scroll through to the first of these technologies we see the following:

```
<tech name="Age 1 Odin" type="Normal">
  <dbid>316</dbid>
  <displaynameid>11050</displaynameid>
  <researchpoints>0.0000</researchpoints>
  <status>AVAILABLE</status>
  <icon>God Major Odin Icon 64</icon>
  <rollovertextid>17906</rollovertextid>
  <flag>AgeUpgrade</flag>
  <prereqs>
    <civilization>
      <civname>Odin</civname>
    </civilization>
  </prereqs>
  <effects>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Dwarf</target>
    </effect>
    <effect status="unobtainable" type="TechStatus">Age 1 Thor</effect>
    <effect status="unobtainable" type="TechStatus">Age 1 Loki</effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Raven</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Ox Cart</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Villager Norse</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="GrantedTech" tech="Great
Hunt" type="Data">
      <target type="Player"></target>
    </effect>
```

```

    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Ulfark</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Fishing Ship Norse</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">House</target>
    </effect>
    <effect amount="1.10" relativity="BasePercent" subtype="Hitpoints" type="Data">
      <target type="ProtoUnit">Jarl</target>
    </effect>
    <effect amount="1.20" relativity="BasePercent" subtype="Hitpoints" type="Data">
      <target type="ProtoUnit">Huskarl</target>
    </effect>
    <effect amount="1.20" relativity="BasePercent" subtype="Hitpoints" type="Data">
      <target type="ProtoUnit">Ballista</target>
    </effect>
    <effect amount="1.20" relativity="BasePercent" subtype="Hitpoints" type="Data">
      <target type="ProtoUnit">Portable Ram</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Hero Norse</target>
    </effect>
    <effect action="Gather" amount="1.20" relativity="BasePercent"
subtype="WorkRate" type="Data" unittype="Hunttable">
      <target type="ProtoUnit">AbstractVillager</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Dryad</target>
    </effect>
    <effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
      <target type="ProtoUnit">Hesperides Tree</target>
    </effect>
  </effects>
</tech>

```

We will not explain all of this in detail yet, but the command we are interested has the format

```

<effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
  <target type="ProtoUnit">UnitName</target>
</effect>

```

It says that for the given protounit set the data for the enable parameter to one (in other words, switch it on).

In the **Age 1 Odin** example, you can see that the technology enables 9 ProtoUnits: **Raven, Ox Cart, Villager Norse, Ulfark, Fishing Ship Norse, House, Hero Norse, Dryad** and **Hesperides Tree** – all of the units that can be trained in Age 1 for the civilization **Odin**.

If you scroll down you will see the **Age 1 Odin** technology is followed by **Age 1 Thor** and **Age 1 Loki**. These technologies are similar but, for example you may notice that **Thor** does not enable ProtoUnit: **Raven**, but enables ProtoUnit: **Dwarf** and **Dwarf Foundry**, Similarly **Loki** does not enable ProtoUnit: **Raven**, but provides a number of economic and military improvements instead.

For the purpose of our modification we are going to limit the use of **aragorn** and the **house of elrond** to Age 1 Loki. To do this scroll to the end of the definition for Age 1 Loki

and after the effect definition for enabling the Hesperides Tree but before the <\effect> statement insert the following two enabling statements.

```
<effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
  <target type="ProtoUnit">Hesperides Tree</target>
</effect>
<effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
  <target type="ProtoUnit">Aragorn</target>
</effect>
<effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
  <target type="ProtoUnit">House of Elrond</target>
</effect>
</effects>
</tech>
```

Save the **TECHTREE(X).XML** file in your **AoM Mods\Mod Files\data** folder and using **Ykkrosh's AOM Data File Converter** convert this file into **.XMB** format (again in your **AoM Mods\Mod Files\data** folder), and copy the new **TECHTREE(X).XMB** file to the default **Age of Mythology\data** folder.

12.3 Testing The Completed Mod

With the new TECHTREE(X).XMB file in the default **Age of Mythology\data** folder, we are ready to test the completed mod.

It is assumed that you have successfully completed the testing outlined in Chapter 9 and Chapter 10.9. If you have not done these tests, do so now as if any problems occur in these tests the following will not work.

- 1) Launch AOM(TT) and Select the Single Player option. For Player 1 (you), select **Loki** as your Age 1 Major God and start a Random Map game.
- 2) Select a **Villager Norse** and the second icon in the bottom left corner should have the **House of Elrond** Icon. If the Icon is not available:
 - a) Recheck your **PROTO(X).XML** to make sure that the **train** statement in the **Villager Norse** proto unit definition is correct. If so,
 - b) Check the **TECHTREE(X).XML** file to make sure that the **enable** definition for the **House of Elrond** proto unit in the **Age 1 Loki** technology is correct.

As both of these files have been converted any error is most likely going to be spelling rather than syntax. **You will need to restart the game to pick up new versions of either of these files.**

- c) If both of these files are correct, confirm that you have the correct versions of the PROTO(X).XMB and TECHTREE(X).XMB files in your default **Age of Mythology\data** directory – check the creation dates.
- d) Finally, if it still does not work make sure you have used **Loki** as you major god. If the earlier tests worked (refer Chapters 9 & 10.9) items a-d are the only possible causes of failure. If you are still having problems redo the earlier tests and take any remedial actions required.

- 3) Select the House of Elrond Icon and get the **Villager** to build a **House of Elrond**. Once the building is complete, select the **House of Elrond** and in bottom left corner should have the **Aragorn** Icon. If the Icon is not available:
 - a) Recheck your **PROTO(X).XML** to make sure that the **train** statement in the **House of Elrond** proto unit definition is correct. If so,
 - b) Check the **TECHTREE(X).XML** file to make sure that the **enable** definition for the **Aragorn** proto unit in the **Age 1 Loki** technology is correct.

As both of these files have been converted any error is most likely going to be spelling rather than syntax. These are the only two possible causes of failure. If you are still having problems redo the earlier tests and take any remedial actions required.
- 4) Click on the **Aragorn** Icon and the unit should commence training, the Aragorn Icon should change to a used format (black and red version of the icon) indicating no further Aragorn's can be built..
- 5) Use one of your **Ulfarks** to build a **Longhouse**. The (used) **Aragorn** Icon should be on the top row in column 4. If not, the **train** statement in the **Longhouse** proto unit definition is incorrect.

Congratulations, you have now created your first addition to AoM(TT).

13. Making More New Units

The techniques we have covered in the first eleven Chapters of this Guide will enable you to create a wide selection of new characters for use in your single player games.

It is recommended that you consolidate these techniques with a few characters of your own before moving on to the remaining Chapters.

In particular before embarking on technologies it helps to be proficient in creating anims and there associated textures for multi-version characters (e.g. characters whose appearance and powers change as new technologies are researched). It also helps to gain experience with all the various actions (attack and economic) and a general knowledge of what works and what does not and how the game actually hangs together.

The main thing to remember when creating new characters is to do a bit of research first. Find a base character and have a look at what models it uses, any special effects associated with it, the types of actions it performs and the various textures it uses and how these are affected by technologies. You can waste a lot of time trying to get a character to something that just isn't possible – but you can save a lot of time by figuring it out in your head before you start changing files.

After a few attempts, the process of changing files will become routine. Just remember syntax, syntax, syntax, spelling, spelling, spelling. I don't know how many hours I've wasted trying to get something to work only to realize the logic was right I just left out a "}".

Once you get tired of the standard models, you can also expand into the area creating new models; a number of articles exist on this subject (and if I get around to learning modeling I will add a new chapter or two).

13.1 It's a Tough World Out There

Before you create that character you have always wanted to build and post it for peer group approval, just remember there are some tough critics out there.

The majority of the standard ES characters would probably score a "1" if posted as new mods – mainly due to low resolution textures. The screen prints included earlier in this guide are probably 10-times the size of the in-game character with the default camera pitch, so remember your work will be judged under a microscope.

Attention to detail in creating your textures is vital, being one pixel out in a texture can result in an unwanted line on an in-game character. But do not let this deter you, seeing your own characters in a game is a great buzz – of even better put your own face on a character and join the battle.

14. Technologies

In moving on to area of technologies, I have assumed that the reader has consolidated the techniques covered in the first 12 Chapters.

In particular, when we talk about which file to change, I will assume that everyone can distinguish between which files are **source** files (form our **AoM Mods\Source Files** folder), which are **mod** files (form our **AoM Mods\Mod Files** folder), and which are **production** files in our default **Age of Mythology** folder. I will also assume that the reader is familiar with the sub-folders and will simply speak of files and folders in terms of **source**, **mod** and **production**.

I will also assume that the reader has mastered the use of **Ykkrosh's AOM Data File Converter** and **Vachu's AOM Game Text Editor** and that within the context of the discussion will now what is involved to convert files between formats or to add text or links to the in-game texts.

If you do not think your ready for this, but still want to proceed, please reread the Disclaimer and proceed at your own risk.

14.1 What are Technologies?

Technologies in Age of Mythology are essentially a logical tree of potential player capabilities. The units available to a player; any upgrades to unit capabilities (including those achieved through age upgrades); God Power and Minor God options and availability; all these variables are controlled through the use of technologies.

When a game is initialized only one Technology is active – the **Age 1 Technology**, and all subsequent technologies are determined by prerequisites.

When we create a Technology Mod, we are changing the way the game behaves and this obviously has wider implications than simply adding a new character.

The TECHTREE(X) file is simply a list of all the technologies available to the game. Each technology has a standard definition format starting:

```
<tech name="Name" type="Type">  
    GENERAL DETAILS SECTION  
    PREREQUISITES SECTION  
    EFFECTS SECTION  
    EXCLUDE SECTION  
</tech>
```

The Technology Name is a label that will be used when referring to a technology and type can either be "Normal" – a technology that is not a god power, or "Power" – a God Power technology that needs to invoke the relevant god power script (see Chapter 14) when activated.

14.2 General Details

The general details section of the tech definition define the basic information about the technology. The following example uses the general details from the **Medium Archers** technology:

```
<dbid>73</dbid>
<displaynameid>11166</displaynameid>
<cost resourcetype="Wood">150.0000</cost>
<cost resourcetype="Gold">150.0000</cost>
<researchpoints>20.0000</researchpoints>
<status>OBTAINABLE</status>
<icon>improvement medium archer icon</icon>
<rollovertextid>10751</rollovertextid>
<buttonpos column="2" row="2"></buttonpos>
```

These various statements have the following meaning:

- 1) The **<dbid>** is simply a unique (not necessarily sequential) identification number.
- 2) The **<displaynameid>** has the same role as the field of the same name in the **PROTO(X)** file. It is just a reference to the **(xpack)language.dll** file and identifies the correct in-game text.

A second text messages can also be defined where prerequisites are used and this text is displayed when a Player attempts to activate a technology before the prerequisites are achieved. This uses the `<prereqnotmetrollovertextid>` statement, which again is simply a reference to a text id in the **(xpack)language.dll** file.

- 3) The **<cost resourcetype="">** field has the same effect as when used in the PROTO(X) file. It defines the cost in terms of **Food, Wood, Gold** and **Favor** of the technology. As with the PROTO(X) file combinations of resourcetype can be used.
- 4) The **<researchpoints>** field tells the game how long it takes to research the technology, with the actual time being **<researchpoints>** divided by the **<buildingworkrate>** (as defined in the initiating unit's proto unit definition); with the resultant value being the number of seconds the research will take.

In this example, the **<researchpoints>** is set to zero as it is a technology that is researched as a result of the civilization being selected when starting a game (hence we want it to occur immediately); with other technologies you would set this to a more appropriate amount (say between 15 and 30 seconds).

- 5) The **<status>** field defines the initial state (or availability) of the technology. The **<status>** of a technology can be **OBTAINABLE, AVAILABLE, ACTIVE** or **PERSISTENT**.

The different **<status>** options have the following meanings:

ACTIVE means that as soon as the **TECHTREE(X)** file is processed (e.g. during game start up the technology (and its **<effects>**) will immediately take effect. The reason that the **Age 1** technology is the only technology that has this **<status>** is that it is the only technology that has no prerequisites.

AVAILABLE means that the technology is available for research. During game start-up the only technologies that are **AVAILABLE** are those associated with the Age 1 minor gods (**Age 1 Zeus, Age 1 Hades, Age 1 Loki**, etc). These technologies need to be available at start-up as they are initiated by the choice of **civilization** selected during the game initiation process. All other technologies are made **AVAILABLE** as the result of successfully researching other technologies.

OBTAINABLE means that the technology can be made **AVAILABLE**, but only once specific prerequisites are achieved. The majority of technologies in the game are **OBTAINABLE**, based on a variety of **prerequisites** covering the player's culture, civilization, availability of specific proto units, of research of other technologies.

PERSISTENT has a similar meaning to **ACTIVE** except that is a technology that may need to be repeated throughout the game. The various "respawn" technologies are examples of this. An **ACTIVE** technology may, for example increase the hitpoints of a particular unit, and then cease to function. A **PERSISTENT** technology may need to repeat an action multiple times during a game, whenever the same situation arises. **PERSISTENT** technologies include an additional General Detail parameter:

<flag>Volatile</flag>

This tells the game that the technology is repeatable and the prerequisites must be rechecked, and technology reapplied each time the prerequisite occurs. For example with the respawn technologies, each time a unit is killed, it must be recreated at the appropriate building.

UNAVAILABLE means that the technology cannot be used. You would usually use this status when you want the technology to be triggered by another technology.

- 6) The **<icon>** field defines the name of the icon file for this technology, i.e. the image displayed in-game that initiates or represents this technology. Technology icons are 64x64 bitmaps converted into .DDT format; and are stored in the default **Age of Mythology\textures\icons** folder. They are created in the same way as character icons as described in Chapter 6.5.
- 7) The **<rollovertextid>** has the same function as in the earlier description of this field for characters and buildings. It is a reference to a text entry in the (xpack)language.dll file that gives a more detailed description of the technology (displayed in the bottom lefthand corner of the game field) when the cursor rests over the technology icon.
- 8) The **<buttonpos>** appears to define the location of the technology icon on the in-game screen – however it seems to be overridden by the row, page, column information in the proto unit definitions(?). I just tend to replicate whatever I used in the PROTO(X).
- 9) The General Details can also include one or more **<flags>**. Five different flags are used in the technology definitions:

Volatile – which as discussed tells the game to recheck for the stated prerequisite on a periodic basis. This period is defied with the **<delay>** function. For example,

including the statement `<delay>60.0000</delay>` in the General Details section will force the game to recheck a given prerequisite every 60 seconds.

AgeUpgrade – This tells the game that the technology is an Age Upgrade.

HideFromDetailHelp – This is used for behind the scenes technologies (tidying up messy bits that you do not want players to be aware of).

AlwaysShowButton – which I think is just a safeguard that avoids other icons being placed over the specified position for the relevant technology icon.

DynamicCost – this is only used with the Omniscience Technology and I assume calls a specific piece of code that calculates the cost based on the current populations.

- 10) The last parameter that is used in the General Details section is the delay function, which uses the format:

```
<delay>0.0000</delay>
```

As it suggests, this function delays the execution of the technology effects for a set time period of time after the prerequisites for that technology arise. The value is in minutes, e.g. 1.0000 = 60 seconds.

14.3 Prerequisites

Prerequisites, define the conditions that must occur for the technology to become **AVAILABLE**.

The prerequisite section is used in all technologies other than the initial Age 1 technology and is simply a list of prerequisite statements positioned between the start `<prereqs>` and end `</prereqs>` prerequisite commands.

```
<prereqs>
  PREREQUISITE 1
  PREREQUISITE 2
  PREREQUISITE 3
  ....
  ....
</prereqs>
```

Prerequisites statements come in a number of formats.

- 1) `<specificage>` tells the game that the player must reach a certain age, e.g. **Archaic Age**, **Classical Age**, **Heroic Age** or **Mythical Age** before the technology can become **AVAILABLE**. The statement uses the following syntax:

```
<specificage>Classical Age</specificage>
```

When using `<specificage>` as a prerequisite you define the earliest age in which the technology is to be made **AVAILABLE**. It will remain **AVAILABLE** in all future ages unless inhibited using another technology.

- 2) The **<culture>** prerequisite statement tells the game that the technology is limited to specific cultures, and then lists one or more **<culturename>** statements that tell the game which cultures the technology can be made **AVAILABLE** to, e.g. **Greek, Egyptian, Norse** or **Atlantean** (AoMTT only). The **<culture>** statement uses the following syntax:

```
<culture>
    <culturename>Norse</culturename>
</culture>
```

This would make the technology AVAILABLE to the Norse Culture only.

If a technology is to be made **AVAILABLE** to a number of cultures, a separate **<culturename>** prerequisite statement is required for each culture, e.g. to make a technology available to both the Atlantean and Norse cultures you would use the following prerequisite statement:

```
<culture>
    <culturename>Atlantean</culturename>
    <culturename>Norse</culturename>
</culture>
```

If the technology is to be made available to **all** cultures you do **not** need to include a **<culture>** prerequisite statement.

- 3) The **<civilization>** prerequisite statement works in the same way as the **<culture>** statement but where it is included it limited the availability of the technology to the list of **<civname>** entries. The civilizations used are those defined for the 12 Major Gods (9 in AoM) that are defined when selecting the Major God during game start-up, e.g. **Zeus, Poseidon, Hades, Ra, Isis, Set, Odin, Thor, Loki, Gaia, Kronos** and **Oranos**.

For example, to limit the availability of a technology to a single civilization (in this case Loki) you would include the following **<civilization>** prerequisite statement.

```
<civilization>
    <civname>Loki</civname>
</civilization>
```

- 4) A number of technologies within AoM(TT) are set up as a sequence of technologies, e.g. **Medium Archers, Heavy Archers, Champion Archers**. As well as having **<specificage>** prerequisites these technologies require the earlier technology to the **ACTIVE** before they can become **AVAILABLE**. This is achieved with the **techstatus** statement. For example the **Heavy Archers** technology will include the following prerequisite statement;

```
<techstatus status="Active">Medium Archers</techstatus>
```

This command has two parameters, **status**, which can be **UNAVAILABLE, OBTAINABLE, AVAILABLE, ACTIVE, PERSISTENT**, and the **Technology Name** you wish to check. It returns a true or false answer.

If required you can multiple techstatus prerequisite, all will need to return a true value if the technology is to become **ACTIVE**.

- 5) The next prerequisite type is used to check for the availability of a specific proto unit type. For example, the standard age upgrades require either a Temple, an Armory or a Market to be built first. To check for the presence of these units we use the <typecount> prerequisite statement. This statement has the following structure:

```
<typecount count="1.00" operator="gte" state="noneState aliveState " unit="Aragorn">
</typecount>
```

The statement has four parameters: count, operator, state and unit.

count is simply the number you want to use in your comparison with the actual typecount of a particular unit.

operator is the comparison you wish to make. This can be lt (less than), gt (greater than) or gte (greater than or equal to)

state refers to the unitstate of the proto unit you are counting. The value "noneState aliveState" is format used in all standard AoM(TT) typecount prerequisites and this just means units alive or in training/being built.

unit refers to the proto unit name you wish to count. This must be the same as defined as the unit name in the PROTO(X) file.

So in the statement example above it is simply asking the logical argument do I have one or more Aragorn units. The answer will be True or False.

- 6) The final concept we need to discuss in relation to prerequisites is **implied** prerequisites. Whenever we set up a prerequisite that is dependent on the whether the Player is a specific civilization; another technology is AVAILABLE; or a specific unit is present, the prerequisites that apply to that aspect of the game also become prerequisites for our new technology. It is therefore important to understand what these prerequisites are so that the new technologies prerequisites can logically return a True statement. If they cannot the technology will never be used.

For example, if we set our prerequisites to comprise <cname>Loki</cname> with a <typecount> for unit **Hero Greek Jason** of greater than or equal to 1. We could never get a **True** condition; as Hero Greek Jason is not available to the Loki civilization.

In summary we can define a range of prerequisites for each technology, and all defined prerequisites must be met (TRUE) before a technology becomes AVAILABLE. A number of specific prerequisite statements are used to check different aspects of the game, and a number of different prerequisites can be defined for a technology.

The following table gives a give reference to the correct one to use:

Required effect	Prerequisite Used
I want to defer the technology to a later age	<specificage>
I want to limit the technology to specific cultures	<culture>
I want to limit the technology to specific civilizations	<civilization>
I want the technology to be dependent on other technologies	<techstatus ...
I want the technology to be dependent on the availability of a specific unit	<typecount ...

14.4 Effects

Having defined a technologies basic details and the prerequisites that will determine when it becomes **AVAILABLE**, the next stage is to define what **effects** it will created when activated.

The **effects** section of the technology definition provides a list of effects relevant to a particular technology bound by the start and end effects command statements:

```
<effects>
    EFFECT 1
    EFFECT 2
    EFFECT 3
    EFFECT 4
    .....
    .....
</effects>
```

Effects are divided into five categories – **type**, **amount**, **action**, **status**, **generator**, and **culture**. Each of these categories is described in subsequent sections, however a brief description of the role of each category follows:

- 1) **<effect type ...>** statements are used to perform three in-game functions: displaying in-game text **<effect type "TextOutput">**, playing sounds **<effect type "Sound">**, and initiating Age Upgrade actions **<effect type "SetAge">**. These are discussed in Section 13.4.1.
- 2) **<effect amount ...>** statements are used to modify the values defined in the **body** of the various proto unit definitions in the **PROTO(X)** file. For example hitpoints, armor levels, carry capacity, speed, los, unit availability, etc. These are discussed in Section 13.4.2.
- 3) **<effect action ...>** statements are used to modify the values defined in the **actions section** of the various proto unit definitions in the **PROTO(X)** file. For example attack range, attack damage, gathering rates, turning on and off actions, etc. These are discussed in Section 13.4.3.
- 4) **<effect status ..>** statements are used to change the status of other technologies. These are discussed in Section 13.4.4.

- 5) **<effect generator ..>** statements are used when we want the technology to initiate an in game action – such as generating the start-up units for a Player. These are discussed in Section 13.4.5.
- 6) **<effect culture ..>** statements are used when we want to change the names of characters for a particular player as a result of an upgrade. For example changing the in-game name of a Hippikon to a Medium Hippikon. These is discussed in Section 13.4.6.

So in a typical <effects> section we would do some or all of the following:

- a) Trigger the display of a text message announcing the start of the technologies research.
- b) Alter the amounts from the protounit definition, making units available and improving the strength and speed of others.
- c) Alter the parameters used in a units actions to make them stronger or more efficient.
- d) Activate, make obtainable or turn off other technologies.
- e) Automatically create new units.
- f) Change the names of the units affected by the new technologies.
- g) Trigger the display of a text message announcing the end of the technology's research and play a sound file to attract the Players attention to the text message.

14.4.1 Type Effects

In defining our technologies there are three statement formats we can use to trigger events in the game – TextOutput, Sound and SetAge.

The "**TextOutput**" effect is used whenever we want to display a message during game play, and it uses the following format:

```
<effect type="TextOutput">textid</effect>
```

It is simply a reference to the id of a text entry defined in the (xpack)language.dll file. The text entries themselves, are set up as described in Chapter 7.

Normally, we use a "**TextOutput**" at the start and end of the **<effects>** section to inform the Player on the status of the research. If we want the message to be displayed to all players we add the **all="true"** parameter to the statement.

```
<effect all="true" type="TextOutput">textid</effect>
```

The "**Sound**" effect is used to initiate one of the sound files stored in the `\sounds` folder and is usually used to announce the completion of a technology's research or an age upgrade, and uses the following format:

```
<effect type="Sound">researchcomplete1.wav</effect>
```

Age upgrades use the sound file `AgeAdvance`. If you want to add your own sound files just change `researchcomplete1.wav` to the name of the new sound file.

The "**SetAge**" effect tells the game to initiate any other actions associate with an Age Upgrade – for age based technologies, textures, etc. It uses the format:

```
<effect type="SetAge">Age</effect>
```

Where `Age` will be either `Classical Age`, `Heroic Age`, or `Mythic Age`.

As the name suggests it is only used for technologies involving age upgrades as discussed in Chapter 16.

14.4.2 Amount Effects

Amount effects (along with action effects) are main type of effect used in technologies. Most of our technologies will improve certain aspects of a unit's performance and we set the new values for this increased or reduced performance using the `<effect amount ...>` definition.

The first amount we will want to effect is the **ENABLE** value of our protounits. In AoM(TT) no units are enabled until a technology enables them. So for example, in the **Age 1** technology we see the following statement that **ENABLEs** the proto unit **Settlement Level 1** (the Town Center).

```
<effect amount="1.00" relativity="Absolute" subtype="Enable" type="Data">
  <target type="ProtoUnit">Settlement Level 1</target>
</effect>
```

We used this statement in Chapter 12 to enable our Aragorn and House of Elrond units in the technology Age 1 Loki.

There are a wide range of formats for `<effect amount>` statements, however three standard parameters are used – amount, relativity and target type.

Relativity tells the game how the **amount** is to be applied and can be one of four options- Absolute, Assign, Percent, BasePercent:

- a) **Absolute** means that the value in the **amount** field is to be added to the existing value for the field in question– in case of enable this is zero;
- b) **Assign** means that the value in the **amount** field is to replace whatever value was currently stored in the nominated field;

- c) **Percent** means that the current value of the field is to be varied in percentage terms by the value in amount field; and
- d) **BasePercent** means that the current value of the field is to be varied in percentage terms of its original value (i.e. the one originally defined in the PROTO(X) file. For example, this option avoids the potential compounding effect that could occur using multiple armor upgrades.

The **amount** field will therefore be a positive or negative integer – where relativity is **ABSOLUTE** or **ASSIGN**; or a decimal percent (e.g. 1.0000 = 100%) where relativity is **PERCENT** or **BASEPERCENT**.

The **target type** is the name of the proto unit as defined in the PROTO(X) file.

Once you get used to tech modding you will find translating an effect becomes second nature, so we will not explore all of the available options, rather we will look at the main formats.

1) **Upgrading Basic Attributes**

The basic attributes of a unit (its hitpoints, speed, line of sight and trainpoints, etc.) are separate subtypes within the game and use similarly structured <effects>. The standard statement to increase a unit's hitpoints is:

```
<effect amount="1.10" relativity="BasePercent" subtype="Hitpoints" type="Data">
  <target type="ProtoUnit"> unitname</target>
</effect>
```

This tells the game to increase the nominated **ProtoUnits Hitpoints** by a value equal to 10% of its original hitpoints. If we set the original hitpoints to 300, this command would add 10% of this amount (e.g. 30 hitpoints) and our in-game character would now have 330 hitpoints available. Note, to cut down on effects, *unitname* can be a single proto unit or a **unittype**, e.g. if we used **HumanSoldier** as our *unitname* all protounits of this unittype would benefit from the improvement.

For **line of sight** ...

```
<effect amount="2.00" relativity="Absolute" subtype="LOS" type="Data">
  <target type="ProtoUnit"> unitname</target>
</effect>
```

(add 2 to the current value). For **speed** ...

```
<effect amount="1.10" relativity="BasePercent" subtype="MaximumVelocity" type="Data">
  <target type="ProtoUnit">Men of Rohan</target>
</effect>
```

(add 10% of the original speed to the current value). For **trainpoints**

```
<effect amount="0.80" relativity="Percent" subtype="TrainPoints" type="Data">
  <target type="ProtoUnit">Men of Gondor</target>
</effect>
```

(reduce the trainpoints to 80% of the current value).

In these basic attributes we have an amount, a relativity, a subtype and a protounit. Other basic attributes include **TributePenalty**, **AllowedAge**, **BuildLimit**, **MaxContained**, **RechargeTime**, **PopulationCapAddition**, **LifeSpan**, and **NumberProjectiles**.

The easiest way to remember when to use the basic attribute effect format is if the **PROTO(X)** definition is in the format `<attribute>value</attribute>` you use the basic attribute effect format where `subtype="attribute"`.

The difference that arises with other amount attributes discussed below, is that the subtype may have subtypes of its own, and therefore requires additional parameters.

2) **Increasing Armor Strength**

When increasing armor strength we need to remember two things: 1) that our units have three different types of resistance – hack, pierce and crush, and 2) AoM thinks in terms of armor vulnerability, so we are not **increasing** the armor's strength but **reducing** its vulnerability. The basic command format is:

```
<effect amount="-0.10" damagetype="Pierce" relativity="Percent" subtype="ArmorVulnerability"
    type="Data">
    <target type="ProtoUnit">unitname</target>
</effect>
```

In this example, we need to include a **damagetype** parameter to tell the game which component of **ArmorVulnerability** we are interested in. You will also notice that our amount is negative. If the proto unit had Pierce Armor of 35%, then the effect of this command would be to increase this by 6.5% - e.g. its original armor vulnerability is 65% and we are reducing this by 10%. The reason for doing it this way, is that if you stick to a number <1.0000 (100%) and reduce vulnerability rather than increase resistance you can never get >100% resistance (I assume the game would not handle this very well, as an attack would increase hitpoints).

To reduce a unit or unit types vulnerability to other attacks we would include similar commands for `damagetype="Hack"` and `damagetype="Crush"`. In the standard AoM technologies each damagetype is upgraded with a separate series of technologies, but you can mix and match as you choose.

3) **Increasing Carrying Capacity**

When increasing a unit's carrying capacity we need to specify which resource they will be able to carry more of, for example:

```
<effect amount="2.00" relativity="BasePercent" resource="Food" subtype="CarryCapacity"
    type="Data">
    <target type="ProtoUnit">AbstractVillager</target>
</effect>
```

This increases the unittype AbstractVillager's food carrying capacity to 200% of the initial value. If they could initially carry 20 units of food before needing to deposit it, they can now carry 40 units. We would use a similar format for `resource="Wood"` and `resource = "Gold"`.

4) Changing a Units Cost

To change the cost of a unit we need to use separate effect statements for each resource type cost we want to change, for example:

```
<effect amount="0.80" relativity="Percent" resource="Wood" subtype="Cost" type="Data">
  <target type="ProtoUnit"> unitname </target>
</effect>
```

This effect reduces the wood cost of the unit to 80% of its current value. We use the same format for `resource="Gold"`, `resource="Food"`, and `resource="Favor"`.

Similar amount effects can be used to manage `ResourceTrickleRate`, and `BuyFactor`, `SellFactor` values at markets. If you search your `TECHTREE(X)` file for these terms you can see how they work.

14.4.3 Action Effects

The second main area where we apply our new technologies is to the actions the unit performs. If you remember our earlier discussion on actions in Chapter 3, there are a range of actions a unit can perform and each of these action has a range of parameters that effect how they perform. The `<effect action ..>` commands allow us to control these variables through our technologies.

To start with let us look at a number of effects associated with a `RangedAttack`. These effects could be used with a ranged unit or a `BuildingThatShoots`. They all effect different attributes (or subtypes) of `action="RangedAttack"`.

```
<effect action="RangedAttack" amount="1.00" relativity="Absolute" subtype="ActionEnable"
type="Data">
  <target type="ProtoUnit"> unitname</target>
</effect>
```

```
<effect action="RangedAttack" amount="1.15" damagetype="Pierce" relativity="BasePercent"
subtype="Damage" type="Data">
  <target type="ProtoUnit"> unitname</target>
</effect>
```

```
<effect action="RangedAttack" amount="6.00" relativity="Absolute" subtype="DamageBonus"
type="Data" unittype="unittype">
  <target type="ProtoUnit"> unitname</target>
</effect>
```

```
<effect action="RangedAttack" amount="2.00" relativity="Absolute" subtype="MaximumRange"
type="Data">
  <target type="ProtoUnit"> unitname</target>
</effect>
```

```
<effect action="RangedAttack" amount="10.00" relativity="Assign" subtype="TrackRating"
type="Data">
  <target type="ProtoUnit"> unitname </target>
</effect>
```

The first effect **ENABLE**s the attack, this is common for buildings that cannot use their ranged attack until certain technologies are development, but it could be applied to other characters if required.

The next effect modifies the **DAMAGE** caused by the unit (by 15% of its initial value) and includes the *damagetype* (Pierce, Hack Crush), amount and basis for applying that amount.

The third effect modifies the **DamageBonus** (adding 6 to the current value). Because we can define multiple **DamageBonus** values in our **PROTO(X)** definition we need to specify which one we mean using the *unittype* parameter.

Similarly the fourth effect changes the **MaximumRange** value (plus 2.0) and the last the **TrackingRate** value (sets the current value to 10.0). If you look at a **RangedAttack** action definition in your **PROTO(X)** you will see that we are merely changing parameters already defined. So for example, we can also change the **MinimumRange**, **Accuracy**, **AccuracyReductionFactor**, **AimBonus**, **SpreadFactor** and so forth.

All of the different actions types used in the **PROTO(X)** file can be modified in this way by specifying *action="AreaAttack", "Autogather", "Boost", "BuckAttack", "Build", "Gather", "HandAttack", "Heal", "Trade"* and so forth; and defining the parameters that you want to vary, and which unit or unittype you want these changes to apply to.

14.4.4 Status Effects

The `<effect status ...>` command is used to change the status of technologies. We need this command as the logic of the game dictates that certain technologies preclude the future use of other technologies. If for example, we select Loki as our Major God, Thor and Odin should no longer be available so we use the effect commands:

```
<effect status="unobtainable" type="TechStatus">Age 1 Thor</effect>
<effect status="unobtainable" type="TechStatus">Age 1 Odin</effect>
```

Using these commands, these two technologies (originally defined as AVAILABLE) and any other technologies that use these technologies as prerequisites become unobtainable. The `<exclude>` command discussed below has a similar effect.

We can use this effect to change any given Technology any of the valid status values (UNOBTAINABLE, OBTAINABLE, AVAILABLE, ACTIVE or PERSISTENT) and in doing so micro-manage the range of technologies a Player has access to..

14.4.5 Generator Effects

The next effect we will look at is `<effect generator ...>`. This effect is used to force build a unit and used the following syntax.

```
<effect generator="buildingname" mute="true" type="CreateUnit" unit="unittype">
  <pattern minradius="0.00" quantity="4.00" radius="0.00" speed="0.00" type="Leaving">
    <offset x="-15.00" y="0.00" z="0.00"></offset>
  </pattern>
</effect>
```

The *buildingname* is the building protounit that the created units will appear to be created from. This is just for the "visual effect" and the building itself does not actually have to be able to create the *unittype* you request. The quantity parameter determines the number of the unittype that are created while the other parameters determine the *speed*, manner (*type*) and *offset* (direction from the building) that the units will disperse. If you want your units to

gather at a specific point just set the `type` parameter to "Leaving" and define the required `x` and `z offset` coordinates, `z= +North,-South`; `x=-East,+West`. If you want your units to just wander off set the `type` parameter to "Scatter" and the `speed` to a value greater than 0.00.

The generator effect is used in the **Starting Units Culture** technologies, various Respawn Technologies and to grant Myth Units at various stages during the game.

14.4.6 Culture Effects

The last area we will look at is changing unit names – not the protounit, just the in-game text. We do this by simply telling the game to use a new text id in our (xpack)language.dll file.

```
<effect culture="Greek" newname="textid" proto="unitname" type="SetName"></effect>
```

Once all of the effects are defined we close the effects section with the `</effects>` statement.

14.5 Exclusions

The final section of the technology definition lists the `<exclude>` statements. The `<exclude>` statement uses the following syntax:

```
<exclude>techname</exclude>
```

It has the same effect as using the status effect to make a technology unobtainable e.g. `<effect status="unobtainable" type="TechStatus">techname </effect>`. You can include multiple `<exclude>` statements in a technology.

Whether you use the exclude format (after the `<effects>` section) or the status format (in the `<effects>` section) is a matter of choice.

14.6 Creating a New Technology

The first technology we are going to create is an Age 4 Norse technology called Return of the King, which will make Aragorn super-powerful. This technology will cost 500 units of Food and 20 units of Favor. The following is the tech logic.

```
<tech name="Return of the King" type="Normal">
```

Our new technology's name, it is a normal technology not a god power.

```
<dbid>2500</dbid>
```

We need a unique database id. It does not need to be contiguous and starting with 2500 gives plenty of scope for future AoM(TT) expansions.

```
<displaynameid>60010</displaynameid>
```

Our reference to the relevant text entry (we will add all of these later).

```
<cost resourcetype="Food">500.0000</cost>
<cost resourcetype="Favor">20.0000</cost>
<researchpoints>30.0000</researchpoints>
```

These define the cost of the technology and how long it will take to research (30 seconds).

```
<status>OBTAINABLE</status>
```

This states that the technology is obtainable and it will become AVAILABLE once the prerequisites are met.

```
<icon>improvement aragorn rotk icon</icon>
```

This is a reference to the icon that will appear in the game for this technology.

```
<rollovertextid>60011</rollovertextid>
```

Our reference to the relevant text entry (we will add all of these later).

```
<buttonpos column="0" row="2"></buttonpos>
```

Where we want to locate the technology button on the activating building (our House of Elrond).

```
<prereqs>
  <specificage>Mythic Age</specificage>
  <culture>
    <culturename>Norse</culturename>
  </culture>
```

This technology will only have 2 prerequisites – it must be in Age 4, and the culture must be Norse.

```
</prereqs>
<effects>
  <effect type="TextOutput">60012</effect>
```

This text entry will announce that the research has commenced.

```
<effect amount="500.00" relativity="Absolute" subtype="Hitpoints" type="Data">
  <target type="ProtoUnit">Aragorn</target>
</effect>
```

Grant Aragorn an additional 500 hitpoints.

```
<effect action="HandAttack" amount="20.00" damagetype="Pierce" relativity="Absolute"
subtype="Damage" type="Data">
  <target type="ProtoUnit">Aragorn</target>
</effect>
```

Increase Aragorn's hand attack by 20 hitpoints.

```
<effect amount="-0.40" damagetype="Pierce" relativity="Percent"
subtype="ArmorVulnerability" type="Data">
  <target type="ProtoUnit">Aragorn</target>
</effect>
```

Decrease Aragorn's Pierce armor vulnerability by 40%.

```
<effect amount="-0.40" damagetype="Hack" relativity="Percent"
subtype="ArmorVulnerability" type="Data">
  <target type="ProtoUnit">Aragorn</target>
</effect>
```

Decrease Aragorn's Hack armor vulnerability by 40%.

```
<effect amount="10.00" relativity="Absolute" subtype="LOS" type="Data">
  <target type="ProtoUnit">Aragorn</target>
</effect>
```

Increase Aragorn's Line of Sight by 10 units.

```
<effect amount="2.00" relativity="BasePercent" subtype="MaximumVelocity" type="Data">
  <target type="ProtoUnit"> Aragorn</target>
</effect>
```

Double Aragorn's speed.

```
<effect action="Regenerate" amount="10.00" relativity="Absolute" subtype="WorkRate"
type="Data" unittype="All">
  <target type="ProtoUnit">Aragorn </target>
</effect>
```

Increase Aragorn's regeneration speed by 10 hitpoints per second.

```
<effect culture="Norse" newname="60013" proto="Aragorn" type="SetName"></effect>
```

Change Aragorn's in-game name to King Ellesar.

```
<effect type="TextOutput">60014</effect>
```

Send an in-game message signaling the end of the research.

```
<effect type="Sound">researchcomplete1.wav</effect>
```

Play the research complete sound file to announce its completion.

```
</effects>
</tech>
```

The following is a copy of the above technology, Copy it and Paste it into you mod version of the TECHTREE(X).XML between the last </tech> statement and the final </techtree>.

```

<tech name="Return of the King" type="Normal">
  <dbid>2500</dbid>
  <displaynameid>60010</displaynameid>
  <cost resourcetype="Food">500.0000</cost>
  <cost resourcetype="Favor">20.0000</cost>
  <researchpoints>30.0000</researchpoints>
  <status>OBTAINABLE</status>
  <icon>improvement aragorn rotk icon</icon>
  <rollovertextid>60011</rollovertextid>
  <buttonpos column="0" row="2"></buttonpos>
  <prereqs>
    <specificage>Mythic Age</specificage>
    <culture>
      <culturename>Norse</culturename>
    </culture>
  </prereqs>
  <effects>
    <effect type="TextOutput">60012</effect>
    <effect amount="500.00" relativity="Absolute " subtype="Hitpoints" type="Data">
      <target type="ProtoUnit">Aragorn</target>
    </effect>
    <effect action="HandAttack" amount="20.00" damagetype="Pierce" relativity="Absolute"
      subtype="Damage" type="Data">
      <target type="ProtoUnit">Aragorn</target>
    </effect>
    <effect amount="-0.40" damagetype="Pierce" relativity="Percent"
      subtype="ArmorVulnerability" type="Data">
      <target type="ProtoUnit">Aragorn</target>
    </effect>
    <effect amount="-0.40" damagetype="Hack" relativity="Percent"
      subtype="ArmorVulnerability" type="Data">
      <target type="ProtoUnit">Aragorn</target>
    </effect>
    <effect amount="10.00" relativity="Absolute" subtype="LOS" type="Data">
      <target type="ProtoUnit">Aragorn</target>
    </effect>
    <effect amount="2.00" relativity="BasePercent" subtype="MaximumVelocity" type="Data">
      <target type="ProtoUnit"> Aragorn</target>
    </effect>
    <effect action="Regenerate" amount="10.00" relativity="Absolute" subtype="WorkRate"
      type="Data" unittype="All">
      <target type="ProtoUnit">Aragorn </target>
    </effect>
    <effect culture="Norse " newname="60013" proto="Aragorn" type="SetName"></effect>
    <effect type="TextOutput">60014</effect>
    <effect type="Sound">researchcomplete1.wav</effect>
  </effects>
</tech>

```

Now convert the file into .XMB format and place a copy of the new TECHTREE(X). XMB file into your production \data folder.

While we have created the new technology, we need to integrate this into the game. This involves:

- 1) Making the technology available at the House of Elrond;
- 2) Adding the required in-game texts;
- 3) Creating a new icon for the Technology;

14.6.1 Making the Technology Accessible.

To make a technology available in game play it needs to be initiated either from a building, a unit, automatically activated by another technology (using the status effect) or in the case of relic technologies through the garrisoning of a relic unit (refer Chapter 15).

In this instance we want our technology to be made available at the House of Elrond. This requires the insertion of a <tech> statement into the proto unit definition for the House of Elrond. The syntax for this statement is:

```
<tech column="0" page="1" row="2">Return of the King</tech>
```

Open you mod version of the PROTO(X) file and insert this line after our <train> definition for Aragorn. Save the file, convert it into .XMB format and place a copy of this new PROTO(X).XMB file in you production \data folder.

14.6.2 Adding the In-Game Text

For our technology we need to add five new texts to our (xpack)language.dll file

```
id=60010    text=Return of the King
id=60011    text=Aragorn is unveiled as King Ellesar and with the reforged Anduril
             leads the people of Middle Earth to victory
id=60012    text=Researching the Return of the King
id=60013    text=King Ellesar
id=60014    text=Aragorn is crowned King Ellesar
```

Use the in-game text editor to add these new texts, save and create a new language.dll file and if using AoMTT rename this file to xpacklanguage.dll in your AoM production folder.

Create a file called **return of the king.txt** in your \history folder and use the text from text id 60011 as some sample text. Save this file and copy it into your production \history (AoM) or \history2 (AoMTT) folder.

14.6.3 Technology Icon

Using the instructions in Chapter 6.5, create a 64x64 pixel bitmap called **improvement aragorn rotk icon** in your mod \texture folder and use the following artwork to create the icon. Save the file and convert it to DDT format **Paletted 16-bit no alpha [2]** and **Mipmap levels = 1**. When



this is completed successfully place a copy of the new .DDT file in you production \textures\icons folder.

14.7 Testing the Technology

To test the technology you can either play a game through to Age 4, or go into the editor and using the trigger function create a trigger that uses a condition of CHAT CONTAINS setting whatever text trigger you want and an effect of SetTechnologyStatus, scrolling down to RETURN OF THE KING and setting its state to Active.

First thing, it should be in the editor list - if not the new technology is not being picked up. Go over the last few pages and make sure every new file is where it is meant to be.

If the technology was in the list, the game has picked it up, so place an Aragorn unit and a House of Elrond, and Play Test the Scenario (with no victory conditions).

Select the House of Elrond and click on its info icon. The **improvement aragorn ROTK icon** should be visible on the screen as a technology that can be developed at this building.

Select Aragorn and he should have his default settings. Then Enter the chat trigger that will activate the new technology – Aragorn's (sorry King Ellesar's) name and unit stats should change – the most obvious change will be his line of sight increasing. Click on his icon to check that all the changes took place. If not go back over you <effect> entries and make sure that they are correct.

If all has gone well you have a working new technology.

If you want to make the upgrade more visual, the following is a texture you can use (it is the same as used in the picture at the end of Chapter 5). Just paste it over your existing aragorn texture and save it under a new name. Then change the aragorn_anim file to include a **TechLogic None/Return of the King** statement in each anim definition, and use the new texture in the **ReplaceTexture** commands where the tech is active.



14.8 Creating More Technologies

So the process for creating a technology involves:

- 1) Decide what you want a technology to do, which units it will effect and check to make sure that the basic protounit is capable of receiving the enhancement either by updating or turning on an INACTIVE capability you have already defined in your proto unit definitions.

- 2) Decide whether you want to modify an existing technology, or create a new technology.

If modifying an existing technology, choose the appropriate technology and ensure that its prerequisites are suitable for your requirements. If modifying a technology, we do not want to change its prerequisites as this may make the technology unavailable for its intended use.

If creating a new technology, think about what prerequisites you require and what implied prerequisites this will entail, and determine what technologies (if any) you may need to exclude if the technology is used. Set up the new technologies Basic, Prerequisite and Exclusion sections.

- 3) Decide what effects you want the technology to perform, and decide of the appropriate effect statement format. I find it useful to stick to a standard layout:
 - a) Display the Initial text message (tech starting);
 - b) Define or the Unit Enabling effects;
 - c) Set the new values for unit basic attributes (refer 14.4.2)
 - d) Set the new value for unit action attributes (refer 14.4.3)
 - e) Generate any new units (14.4.5)
 - f) Change the names (if required) of any upgraded units;
 - g) Display the Complete text message and appropriate sound file.
- 4) Save your changes and create the new **TECHTREE(X).XMB** file and copy this into you production folder.
- 5) Make the necessary **PROTO(X)** changes to allocate the technology to the required unit using a `<tech>` statement.
- 6) Make the required changes to your **(xpack)language.dll** file and create a new history file and improvement icon.
- 7) Test the technology.

You can use technologies to upgrade your units, automatically gather resources for a player, or create mythical creatures (or other units) as required. While this can make you all powerful just remember that a unit with 250,000 hitpoints a 2,000 hitpoint attack, lightning speed and almost infinite regeneration power, is not going to lead to very interesting games.

Also once you start experimenting with technologies it helps to create your own buildings rather than adding technologies to existing buildings. Some existing buildings have a few slots spare, but it will quickly get messy (and may effect standard AoM(TT) campaigns). Start by creating a University (my favorite building from AoE) and think about the sort of things that would be researched in the time of your home-grown heroes.

15. Relics

Relics are simply technologies that are attached to a relic protounit and are made active when they are garrisoned in a suitable building. In terms of defining them as technologies they follow the same rules as other Normal technologies with the exception that they always have a name starting with "Relic " and require the basic settings to include:

```
<tech name="Relic name" type="Normal">
  <researchpoints>0.0000</researchpoints>
  <status>UNOBTAINABLE</status>
```

Once you have a relic technology you will need let the game know it is available.

To do this locate the **RELICS(2).XMB** in your source **\mods** folder and convert this to **.XML** format. Open this file and you will see it is just a list of relic technology names enclosed by `<relicdata>` and `</relicdata>` statements.

```
<relicdata>
  <relic tech="Relic Ankh of Ra"></relic>
  <relic tech="Relic Anvil of Hephaestus"></relic>
  <relic tech="Relic Armor of Achilles"></relic>
  <relic tech="Relic Arrows of Alfar"></relic>
  ....
  ....
  <relic tech="Relic of Ancestors SPC" reserved=""></relic>
  <relic tech="Relic of Earthquake SPC" reserved=""></relic>
</relicdata>
```

Insert your relic using the syntax:

```
<relic tech="Relic My Relic Name"></relic>
```

This will put the relic technology on the random selection list. The game allocates these to the relics placed in the game (seemingly at random)

If you don't like the hit and miss nature of the allocation of Relic technologies you can get rid of an existing relic entry, unless it includes the parameter `reserved=""` or you can assign specific relic powers using the Scenario Editor.

Personally, I prefer not to delete or alter too much of the standard code and find that the existing relics are fine – as long as you keep the standard unittypes, your new units will benefit from existing relic improvements.

16. Civilizations

Once you have mastered the art of creating new characters, buildings and technologies the obvious challenge is to combine these into you own civilization.

The main problem with civilizations is (as far as I know) the no one has figured our how the game defines the **<culture>** and **<civilization>** variables in the game, so until someone does we need to trick the game into using our minor god age stream rather than use the standard streams.

In understanding how this works, we need to think about what is actually happening during the start up phase of the game.

- 1) When starting a **Random Map** game or creating our own **Scenario** we define the **Major God** we want to use. Selecting this **Major God** defines the **<culture>** and **<civilization>** variables.
- 2) When the game is started it loads the **TECHTREE(X)** file. The **Age 1** technology is already **ACTIVE** and is executed and the only other technologies that are **AVAILABLE** are the **Age 1 God** technologies (Zeus, Odin, Ra, etc.). If you look at the prerequisites of these technologies the **Age 1 God** technologies each have a different **<civilization>** prerequisite statement, for example, **Age 1 Loki** has:

```
<civilization>  
    <civname>Loki</civname>  
</civilization>
```

- 3) The **<civilization>** variable determines our **Age 1 God** technology and this determine future technology options.
- 4) The **<culture>** then determines the **Starting Units** for the game.

To initiate our own Age technology streams we need to borrow an Age 1 civilization and add a prerequisite that the standard technologies will not meet, but that our technologies will.

To achieve this we need a **unique** proto unit that will be placed in either a single player scenario that we create; or by a modified Random Map script during the game start-up process (refer Chapter 19). This involves three changes to the **TECHTREE(X)** file.

- 1) The first has to **ENABLE** the **unique proto unit** as part of the **Age 1** technology;
- 2) The second has to alter the standard prerequisites for the **Age 1 God** technology we want to borrow to include a check for the absence of this **unique proto unit**;
- 3) The third has to alter the standard prerequisites for the Culture's default **Starting Units** technology to include a check for the absence of this **unique proto unit**.

To make these changes:

- 1) Modify existing **Age 1** technology `<dbid>64</dbid>` (both AoM and AoMTT) to enable the unique unit:

```

..
..
        <effect type="Data" amount="1.00" subtype="Enable" relativity="Absolute">
            <target type="ProtoUnit">Settlement Level 1</target>
        </effect>
//=====
// NEW CIVILIZATION LOGIC
//=====
        <effect type="Data" amount="1.00" subtype="Enable" relativity="Absolute">
            <target type="ProtoUnit">unique unit</target>
        </effect>
//=====
        <effect type="Data" amount="1.00" subtype="Enable" relativity="Absolute">
            <target type="ProtoUnit">Wall Medium</target>
        </effect>
    </effects>
</tech>

```

This is just the standard enable effect we discussed in Chapter 14.4.2.

The game will now recognize our **unique proto unit**.

- 2) Modify the **Age 1 God** technology for the civilization you wish to borrow, for example if we want to borrow **Loki** modify existing **Age 1 Loki** tech `<dbid>320</dbid>` (AoM and AoMTT) to include as a pre-requisite that the typecount for the **unique unit** must to be zero

```

<tech name="Age 1 Loki" type="Normal">
    <dbid>320</dbid>
    ..
    ..
    <prereqs>
        <civilization>
            <civname>Loki</civname>
        </civilization>
//=====
// NEW CIVILIZATION LOGIC
//=====
        <typecount unit="unique unit" count="1.00" state="noneState aliveState " operator="lt">
        </typecount>
//=====
    </prereqs>

```

With this change we have not affected the original logic of the technology and we can still use the **Loki** civilization (unchanged) as long as we do not have our unique unit (for the particular player) on the map – e.g. we can play our new civilization against Loki.

The reason for this is that we are not actually taking over the Loki technology stream, we are just using the value of "**Loki**" in the `<civilization>` variable to initiate our own technology stream.

- 3) Modify the relevant Starting Units technology. If we are borrowing **Loki** we would change the **Starting Units Norse** tech `<dbid>420</dbid>` (AoM and AoMTT) to include as a pre-requisite that the typecount for the **unique unit** must to be zero.

```

<tech name="Starting Units Norse" type="Normal">
<dbid>420</dbid>
..
..
<prereqs>
  <culture>
    <culturename>Norse</culturename>
  </culture>
  <typecount count="0.00" operator="gt" state="noneState aliveState "
    unit="SettlementsThatTrainVillagers">
  </typecount>
//=====
// LOTR LOGIC
//=====
  <typecount count="1.00" operator="lt" state="noneState aliveState " unit="unique unit">
  </typecount>
//=====
</prereqs>

```

This change simply means that any normal Norse startup that does not include our unique proto unit will behave as normal.

Having made these changes, we need to create two new technologies that will start our civilization technology stream – our own **Age 1 God** technology, and our own **Starting Units** technology.

- 1) The new Age 1 God technology will use the same techniques we discussed in Chapter 14. You can either create it from scratch or use The Age 1 Loki technology as your starting point. The new technology will need the following basic attributes:

```

<tech name="Age 1 NewGodName" type="Normal">
  <dbid>last dbid plus one</dbid>
  <displayNameid>last text id plus 1</displayNameid>
  <researchpoints>0.0000</researchpoints>
  <status>AVAILABLE</status>
  <icon>God Major NewGodName Icon 64</icon>
  <rollovertextid> last text id plus 1</rollovertextid>
  <flag>AgeUpgrade</flag>
  <prereqs>
    <typecount count="1.00" operator="gte" state="noneState aliveState "
unit="unique unit"></typecount>
    <civilization>
      <civname>Loki</civname>
    </civilization>
  </prereqs>
  <effects>
    <effect status="unobtainable" type="TechStatus">Age 1 Thor</effect>
    <effect status="unobtainable" type="TechStatus">Age 1 Odin</effect>
    <effect status="unobtainable" type="TechStatus">Age 1 Loki</effect>
    EFFECT 1 etc.
  </effects>
</tech>

```

The basic details need to contain the new technology name, new dbid, text and icon information and must include the `<flag>AgeUpgrade</flag>` statement, the prerequisites

section we must include a check for the existence of our unique **proto unit**; and the effects section should make the other **Norse Age 1 God** technologies **unobtainable** (you can use exclude statements after the <effects> section if preferred).

Otherwise the remainder of the effects statement will include the standard **ENABLE** effects and any basic effects or action effects you want to include.

In the **Age 1 God** technology you do not bother with **texts** or **sounds** as the technology is initiated before the Player sees the game screen. You also use a **Starting Units** technology rather than **generator** statements (this may not be mandatory, but I assume the ES guys did it this way because of some timing issues).

Generally, the initial attributes and actions you defined in the **PROTO(X)** file are the ones you want in Age 1, so usually this file will just be a series of **ENABLE** effects. Just make sure you **ENABLE** everything you require (units and buildings) and remember that your **unique proto unit** along the generally available units (**walls** and **gates**) have already been **ENABLED** in the Age 1 technology.

Finally, you will also need to grant the relevant Age 1 God Power. This can be a standard issue one or one of your own (refer Chapter 18). This effect uses the following syntax.

```
<effect amount="3.00" relativity="Absolute" subtype="GrantedTech" tech="Spy "
type="Data">
    <target type="Player"></target>
</effect>
```

This example grants three (3) uses of the **Spy** God Power

- 2) The starting units technology is just a list of generate effects that create the units you want the Player to commence the game with. It use the following format:

```
<tech name="Starting Units MyGod" type="Normal">
    <dbid>last dbid plus one</dbid>
    <displaynameid>last text id plus 1</displaynameid>
    <researchpoints>0.0000</researchpoints>
    <status>OBTAINABLE</status>
    <delay>0.5000</delay>
    <flag>HideFromDetailHelp</flag>
    <prereqs>
        <typecount count="1.00" operator="gte" state="noneState aliveState "
unit="unique unit"></typecount>
        <civilization>
            <civname>Loki</civname>
        </civilization>
    </prereqs>
    <effects>
        VARIOUS GENERATOR EFFECTS
        <effect type="TextOutput">last text id plus 1</effect>
    </effects>
    <exclude>Starting Units Norse</exclude>
    <exclude>Starting Units Egyptian</exclude>
    <exclude>Starting Units Greek</exclude>
    <exclude>Starting Units Atlantean</exclude>
</tech>
```

It has its own basic details, uses the same prerequisites as our **Age 1 God** technology and excludes all the other **Starting Unit** technologies so that you cannot accidentally trigger these and create unwanted units.

The **generator** effects follow the format discussed in Chapter 14.4.5.

Our subsequent Age technologies can now build on a unique prerequisite:

```
<techstatus status="Active">Age 1 NewGodName</techstatus>
```

or on the availability of any technology or unit that uses this technology as an implied prerequisite and becomes a **unique proto unit** or technology for our new civilization.

In general the only difference between "civilization" technologies and normal technologies is:

- a) They are typically where we grant **God Powers**; and
- b) They are initiated via the Age Upgrade process using **Minor Gods**.

In the next Chapter we will discuss how we make the necessary changes to the **Minor God** files to allow us to use these new Age technologies, followed in Chapter 18, by a discussion on **God Powers**.

17. Minor Gods

As discussed in the last Chapter, **Minor Gods** are just special technologies. They are special because they require a choice to be made within the game.

The game used the `<flag>AgeUpgrade</flag>` statement and the `<prereqs>` to determine what **Minor Gods** (or Age Upgrade Technologies) are available to the Player.

All the **Minor God** file does is provide the relevant information to the Age Upgrade screen so it can display the available choices.

Find the MINORGODS.XMB file in you source \data folder, Copy it to you mods \data folder and then convert it to .XML format. Open the file with Notepad and you will see a list of 36 Minor Gods (dbid 0-35) or 27 (dbid 0-26) if using AoM:

```
<?xml version="1.0" encoding="UTF-8"?>
<minorgods>
  <god dbid="0" nameid="17150" techname="Age 2 Athena" portrait="god minor portrait greek Athena"
position="2" age="2"></god>
  <god dbid="1" nameid="17151" techname="Age 2 Ares" portrait="god minor portrait greek Ares"
position="1" age="2"></god>
  <god dbid="2" nameid="19154" techname="Age 2 Hermes" portrait="god minor portrait greek
Hermes" position="3" age="2"></god>
  <god dbid="3" nameid="17153" techname="Age 3 Dionysos" portrait="god minor portrait greek
Dionysus" position="3" age="3"></god>
  ....
  ....
  ....
  <god dbid="33" nameid="22829" techname="Age 4 Helios" portrait="god minor portrait atlantis helios"
position="2" age="4"></god>
  <god dbid="34" nameid="22830" techname="Age 4 Hekate" portrait="god minor portrait atlantis
hekate" position="1" age="4"></god>
  <god dbid="35" nameid="22831" techname="Age 4 Atlas" portrait="god minor portrait atlantis atlas"
position="3" age="4"></god>
</minorgods>
```

For each culture (Greek, Egyptian, Norse, Altantean) we have 3 Minor gods for 3 Ages.

All the `<god ..>` statements do is provide:

- 1) A unique dbid;
- 2) A reference to the in-game text message in our (xpack)language.dll file;
- 3) The name of the associated technology (in the TECHTREE(X) file);
- 4) A reference to the portrait icon for the Minor God (a bigger icon for use on the Age Upgrade screen);
- 5) A position (1-3) for each culture/age which determines the order it is displayed; and
- 6) The Age it applies to, which the AgeUpgrade function uses to initiate the necessary generic age upgrade technologies and other functions.

Creating a new Minor God simply involves inserting a new entry, for example

```
<god dbid="36" nameid="nnnnn" techname="Age 1 NewGodName " portrait="god minor portrait  
newgodname" position="1" age="1"></god>
```

saving the file and converting it to .XMB format. This file then goes into your production \data folder.

Next create the new entries for the in-game text as described in Chapter 7, all this needs to be is the name of the Minor God.

We also need to create a new portrait icon for this **Minor God**. This icon is a larger version of the icons we have previously created and only part of the area is used so that the image fits into the portrait area available on the Age Upgrade screen (as shown in the example).



To create a portrait icon just copy one of the **god minor portrait name** texture files from your source \textures folder to your mods \textures folder and convert it to a bitmap, remembering the DDT format and mip-map levels.

Replace the picture within the frame with your own Minor God portrait, save the file and convert it back to .DDT format.

Copy this file into the production \textures folder (not it \icons sub-folder).

When you enter the Age Upgrade process you will be presented with one or two Minor God Portraits and selecting one will initiate the appropriate Age Technology.

Normally you would provide two Minor God options at each upgrade but this is not mandatory (for example my LOTR Shire civilization used two options at each level but the Mordor civilization only uses one).

Once you are comfortable with the principles discussed in this and the previous chapter, you are ready to create your own civilization. To this you must create:

- 1) An **Age 1 God** technology (your pseudo Major God) and use prerequisites to enable you to access this technology using an existing **civilization**.
- 2) A **Starting Units** technology to create the initial units.
- 3) One or two **AgeUpgrade** technologies for each subsequent age (2 ,3 and 4) - your **Minor God** technologies.
- 4) An updated **MINORGODS.XMB** file with each of the new **Minor Gods** defined.

5) The technology and portrait icons; and in-game text dictated by the above changes.

Once these changes are installed in your production folder, your civilization will be initiated whenever your **unique proto unit** is present. In the first instance this will only be in situations when you place it yourself using the **Scenario Editor**. In Chapter 19, we will discuss how this **unique proto unit** can be placed automatically.

18. God Powers

Having created some new Minor Gods, you may want to give them God Powers of their own. This is not mandatory and to grant any existing God Power as part of your AgeUpgrade technologies, however adding new God Powers is a nice touch, even if you just rename and existing God Power to give it a name more in tune with your new civilization. For example the Palantir Stone technology in my LOTR civilization is just the standard Spy God Power.

You may also want to change God Powers so that you can get the effect of an existing technology using one of your new proto units. Once you understand the workings of God Powers you may also want to mix and match the functions available to create new effects.

God Powers are a special form of technology that use external command files instead of `<effect>` statements. These command files have access to a wide range of functions not available to standard technologies. When creating a God Power you need to define it as a technology of `type="Power"`, as shown in the following example:

```
<tech name="Prosperity" type="Power">
  <dbid>499</dbid>
  <displaynameid>11190</displaynameid>
  <researchpoints>0.1000</researchpoints>
  <status>OBTAINABLE</status>
  <icon>god power prosperity icon</icon>
  <rollovertextid>10765</rollovertextid>
  <prereqs>
    <techstatus status="Active">Age 1 Isis</techstatus>
  </prereqs>
</tech>
```

and then create a god power script with the same name as used in you technology. When the God Power technology is activated this script will be run.

Let us look at a simple God Power script as used for the "Spy" God Power

```
<?xml version="1.0" encoding="UTF-8"?>

<power name="Spy" type="spy" techname="spy">
  <activetime>-1</activetime>
  <placement forceonmap="1" enemy="" matchtype="LogicalTypeValidSpyTarget">unit</placement>
  <spyproto>spy eye</spyproto>
  <soundset type="StartSound" listenertype="Ally">GodPowerStart</soundset>
  <soundset type="StartSound" listenertype="IfOnScreenAlly">SpyBirth</soundset>
  <soundset type="EndSound" listenertype="Ally">GodPowerEnd</soundset>
  <minimapeventtime sendalertto="ally">10.0</minimapeventtime>
  <messagealertplayerrelation>ally</messagealertplayerrelation>
  <icon>god power spy icon</icon>
  <usedicon>god power spy icon done</usedicon>
</power>
```

This script defines the name of the God Power, the type of God Power and the technology name associated with the God Power (which is the same as you would use when creating the relevant technology).

Of these parameters the `type` is a reference to the game function "spy" which gives you visibility of the line of sight of the unit the God Power is cast on. The `name` parameter is

used by the AI unit and game logic when casting the god power and cannot contain any spaces. It gets setup as a game variable called **cPowerName** (i.e. cPowerSpy in this instance). The techname parameter is the name of the technology as defined in the TECHTREE(X) file. In the Palantir Stone technology used for Age 1 Shire in the LOTR Civ Pack I made, the only changes were `name=PalantirStone` and `techname=Palantir Stone`.

The second line has the statement `<activetime>-1</activetime>`. This determines how long the God Power will remain active. In this case it is -1 (which means indefinitely – until the unit is destroyed). It could however be a positive number that would be interpreted as the number of seconds the God Power will remain active. If it was `<activetime>60.0</activetime>` the "Spy" God Power would remain active for 1 minute.

The third line uses the `<placement ...>` statement. This statement determines type of object the God Power can be targeted on. In this case it can be used on any unit that has the `<unittype>LogicalTypeValidSpyTarget</unittype>` defined in its proto unit definition. This unittype is applied to most units but not buildings. However the matchtype could be a specific proto unit (e.g. `matchtype="Aragorn"`) or a class of units (e.g. `matchtype="Human Soldier"`). The `forceonmap="1"` parameter tells the game it can disregard any placement rules (e.g. obstructionradius etc.) when casting the power (which in this case is attaching a spy protounit to an existing object).

The fourth line `<spyproto>spy eye</spyproto>` is a reference to the proto unit that is used for the Spy Power (the thing that hangs above the unit you cast the power on). This is the Spy Eye protounit from the PROTO(X) file.

The next five lines:

```
<soundset type="StartSound" listenertype="Ally">GodPowerStart</soundset>
<soundset type="StartSound" listenertype="IfOnScreenAlly">SpyBirth</soundset>
<soundset type="EndSound" listenertype="Ally">GodPowerEnd</soundset>
<minimapeventtime sendalertto="ally">10.0</minimapeventtime>
<messagealertplayerrelation>ally</messagealertplayerrelation>
```

tell the game who to inform before and after the power is cast and how, e.g. sound alerts, flares on player minmaps and text messages.

Finally we have some icon information for the unused and used (red and black version) of the God Power Icon. Note that the used icon is automatically generated by the game from the unused icon (so you only need to create one).

So when this power is selected the player will be able to select any unit in its line of sight of `<unittype>LogicalTypeValidSpyTarget</unittype>`. The underlying "Spy" code performs the equivalent of an anim **connect Spy Eye attachpoint** on the unit selected, the players allies are notified of the event (we don't want the enemy to know for this GP) and the attachment remains active until the unit is destroyed. If you note from the details is the PROTO(X) this attachment simply provides a 27.0 LOS from the object it is attached to.

18.1 Placement God Powers

The first common form of God Power is one that allows you to place an object. This is the basic format for the **Nidhogg**, **Sentinel**, **Serpents**, **Skeleton Power**, and **Vision** God Powers and in variants of the tempunit function that support additional unit creation parameters: **Ancestors**, **Dwarven Mine**, **Healing Springs**, **Lure** and **Plenty**. These God Powers use the following format (using the Nidhogg GP \god powers\create gold.xmb as an example).

```
<?xml version="1.0" encoding="UTF-8"?>
<power name="Nidhogg" type="tempunit" techname="nidhogg">
```

The Powers Name, type and techname. Some placements use specific types (see above) but `type="tempunit"` is the base function.

```
<activetime>-1</activetime>
```

How long it will last (in minutes). The value -1 means until destroyed, however you can set a number of minutes as done for the Lure placed in the Animal Attraction God Power.

```
<placement forceonmap="1" losprotounit="Nidhogg">all</placement>
<createunit quantity="1" radius="5.0" delay="0.0" norotate="">Nidhogg</createunit>
```

These next commands tell the game what protounit to create, how many and how to place it. If you are only placing a single unit you do not need to worry about the radius or delay in placement, but if for example your god power created an army you will want to increase the radius so the game can place them over a wider area; or use multiple `<create unit .. >` statements and stagger the creation using the delay parameter (which is in seconds). Look at the **Serpents** or **Ancestors** God Powers for example of this approach.

```
<soundset type="StartSound" listenertype="AllExceptCaster">GodPowerStart</soundset>
```

We then alert all players we are casting our GP (but do not tell them where by setting the minimapeventtime sendalert to none).

```
<icon>god power nidhogg icon</icon>
<usedicon>god power nidhogg icon done</usedicon>
```

These last two lines define our unused and used icons.

```
</power>
```

From this example, we can see the basic format of a Placement God Power:

```
<?xml version="1.0" encoding="UTF-8"?>
<power name="MyGodPower" type="tempunit" techname="My God Power">
  <activetime>-1 for no limit, or n.nn minutes</activetime>
  <placement forceonmap="1" losprotounit="proto unit name">full</placement>
  <createunit quantity="num" radius="0.0" delay="0.0" norotate="">proto unit name</createunit>
  <soundset type="StartSound" listenertype="AllExceptCaster">GodPowerStart</soundset>
  <minimapeventtime sendalertto="none">0.0</minimapeventtime>
  <icon>god power My God Power icon</icon>
  <usedicon>god power My God Power icon done</usedicon>
</power>
```

When we activate the God Power it would allow us to click a valid placement are on the map and then it would create the number and type of units defined.

We would this format to place new proto units (particularly super units later in a game).

18.2 Unit Swap God Powers

That next type of God Power is used to swap one proto unit to another.

Blessing of Zeus, the various **Change Unit** god powers, **Citadel**, **Curse**, **Goatunheim**, **Ragnorok**, **Seed of Gaia**, **Son of Osiris**, **Walking Berry Bushes** and **Walking Woods** God Powers.

As the name suggests these god powers swap proto unit for another.

These God Powers use the following general format:

```
<?xml version="1.0" encoding="UTF-8"?>
<power name="Goatunheim" type="swapunit" techname="goatunheim">
  <activetime>0.6</activetime>
  <radius>20.0</radius>
  <soundset type="StartSound" listenertype="AllExceptCaster">GodPowerStart</soundset>
  <soundset type="StartSound" listenertype="IfOnScreenAll">RagnarokBirth</soundset>
  <placement forceonmap="1">skip</placement>
  <powerplayerrelation>all</powerplayerrelation>
  <overallchance>1.0</overallchance>
  <global></global>
  <abstracttype swapt="Goat">Military</abstracttype>
  <abstracttype swapt="Goat">AbstractVillager</abstracttype>
  <abstracttype swapt="Goat">Huntable</abstracttype>
  <swapdelay>0.5</swapdelay>
  <sfx>Ragnorok SFX</sfx>
  <swapselfonly></swapselfonly>
  <icon>Animal Goat icon 64</icon>
  <usedicon>Animal Goat icon 64</usedicon>
</power>
```

If you remember, this God Power changes some units (types: [Military](#), [AbstractVillager](#) and [Huntable](#)) in the target area into protounit [Goat](#). The power remains active for 36 seconds `<activetime>0.6</activetime>` (units entering the area after the initial cast will also be affected, and operates over a radius of 20 units `<radius>20.0</radius>` from the point the GP was cast. It applies to all units (self, ally and enemy) `<powerplayerrelation>all</powerplayerrelation>` and has a 100% conversion rate `<overallchance>1.0</overallchance>`.

From this example we can see the basic format of a swapunit God Power.

```
<?xml version="1.0" encoding="UTF-8"?>
<power name="MyGodPower" type="swapunit" techname="My God Power">
  <activetime>n.n minutes</activetime>
  <radius>num mins</radius>
  <soundset type="StartSound" listenertype="AllExceptCaster">GodPowerStart</soundset>
  <soundset type="StartSound" listenertype="IfOnScreenAll">RagnarokBirth</soundset>
  <placement forceonmap="1">skip</placement>
```

```

<powerplayerrelation>all, enemy or player</powerplayerrelation>
<overallchance>decimalised percent chance</overallchance>
<global></global>
<abstracttype swaptto="To Proto Unit">From Proto Unit</abstracttype>
<swapdelay>num secs</swapdelay>
<sfx>SFX Proto Unit</sfx>
<swapselfonly></swapselfonly>
<icon>God Power icon 64</icon>
<usedicon>God Power icon 64 used</usedicon>
</power>

```

For a straight single unit swap we would set the `activetime` to 0.1, or we can keep it their for longer if we want to create "upgrade zones" or "no go areas". We vary the size of the area where we want swaps to take place by varying the `radius` parameter, and limit the effect of the swap using the `powerplayerrelation` parameter to limit the power to `player` - our units (for good things); the `enemy` (for bad things) or `all` units.

We can use the `overallchance` parameter to randomize the effect of the God Power, and create `swapdelays` to lull enemies into a false sense of security (we do not want the first enemy entering a no go zone to warn off the others by turning into a goat!).

Finally we can add some embellishment by using one of the many standard special effect (`sfx`) protounits or creating our own.

The `swapunit` God Power is most useful when wanting to swap enemy player proto units, as this cannot be performed with normal technologies, whereas with your own units, most swaps can be achieved using latent PROTO(X) actions, special technologies and their associated anim variations. You tend to use a swapunit on your own characters when these attributes are so different that its is just easier to achieve them with a new proto unit.

The remaining 40-odd God Powers are unique in `type` and use functions specially suited the particular effect they create. I do not intend to discuss all of these but to look at some examples that are representative of a group of God Powers. Once you get the hang of a few examples you should be able to make sense out of the others, and with a working knowledge of the effect they actually have in-game it is not hard to interpret what each script is doing.

18.3 Epic God Powers

I classify Epic God Powers as those that involve a sequence of events (a build up, main attack, crescendo and die-down). They include **Chicken Storm, Earthquake, Implode, Lightning Storm, SPCLightning Storm, SPCmeteor, Tartarian Gate, Tornado, Tornado xp05, Tremor** and **Volcano**. In the game they slowly build up, cause major damage and then dissipate.

To see how this type of God Power works we will look at the SPCMeteor God Power (see next page). The first thing you will notice is that it seems a lot more complex than the God Powers we have discussed to date.

It appears more complex because it is really a series of frames (build-up, main attack, wind down), uses separate damage models for enemies and allies and requires a number of randomizing parameters so that the effect of the God Power is not always the same.

Without this randomization an experience player would soon learn to dodge bolts of lightning and meteors as they would always land in the same relative positions.

So let us look at the God Power Script

```
<?xml version="1.0" encoding="UTF-8"?>

<power name="SPCMeteor" type="meteor" techname="SPCMeteor">
  <builduptime>1.0</builduptime>
  <activetime>5.0</activetime>
  <unitaitype>SPCmeteor</unitaitype>
  <radius>32.0</radius>
  <placement forceonmap="1">full</placement>
  <meteor>SPCmeteor</meteor>
  <splashsfx>Meteor Impact Water</splashsfx>
  <landsfx>Meteor Impact Ground</landsfx>
  <firststrikesoundset>MeteorBigHit</firststrikesoundset>
  <strikesoundset>MeteorSmallHit</strikesoundset>
  <firstapproachsound>MeteorApproach</firstapproachsound>
  <approachsound>MeteorWhoosh</approachsound>
  <fasteststriketime>15.0</fasteststriketime>
  <sloweststriketime>15.0</sloweststriketime>
  <abstractattacktargettype>LogicalTypeValidMeteorTarget</abstractattacktargettype>
  <gpdamageunit>
    <playerrelation>enemy</playerrelation>
    <basedamagepercentunit>1.0</basedamagepercentunit>
    <basedamagepercentvillager>1.0</basedamagepercentvillager>
    <basedamagepercentbuilding>1.0</basedamagepercentbuilding>
    <minhpdamageunit>100000</minhpdamageunit>
    <minhpdamagevillager>100000</minhpdamagevillager>
    <minhpdamagebuilding>100000</minhpdamagebuilding>
    <maxhpdamageunit>10000000</maxhpdamageunit>
    <maxhpdamagevillager>10000000</maxhpdamagevillager>
    <maxhpdamagebuilding>10000000</maxhpdamagebuilding>
  </gpdamageunit>
  <gpdamageunit>
    <playerrelation>ally</playerrelation>
    <basedamagepercentunit>1.0</basedamagepercentunit>
    <basedamagepercentvillager>1.0</basedamagepercentvillager>
    <basedamagepercentbuilding>1.0</basedamagepercentbuilding>
    <minhpdamageunit>100000</minhpdamageunit>
    <minhpdamagevillager>100000</minhpdamagevillager>
    <minhpdamagebuilding>100000</minhpdamagebuilding>
    <maxhpdamageunit>10000000</maxhpdamageunit>
    <maxhpdamagevillager>10000000</maxhpdamagevillager>
    <maxhpdamagebuilding>10000000</maxhpdamagebuilding>
  </gpdamageunit>
  <accuracy>1.00</accuracy>
  <unitthrowchance>0.85</unitthrowchance>
  <minthrowdistance>6.0</minthrowdistance>
  <maxthrowdistance>10.0</maxthrowdistance>
  <minthrowheight>1.0</minthrowheight>
  <maxthrowheight>2.0</maxthrowheight>
  <minthrowvelocity>10.0</minthrowvelocity>
  <maxthrowvelocity>20.0</maxthrowvelocity>
  <camerashake duration="0.50" strength="0.25"></camerashake>
  <icon>god power meteor icon</icon>
  <usedicon>god power meteor icon done</usedicon>
</power>
```

The script is easier to understand if we think of it in three chunks. The visual effect frames and the timing and sequencing of the frames; the macro-level damage we want caused by

the God Power (the <gpdamagemodel>) and micro-level damage response we want from individual units.

So in the first part of the GP script, the information is concerned with timing; the proto unit, sfx proto units, and sound files to use during the execution of the GP; the area impact and frequency of individual projectiles. If you want to see how complex this can get look at the **Implode** GP.

In the second part we set the macro level attack parameters and half the option to set them differently for enemies and allies. We set the maximum unit level damage allowed - in this 100% damage (it is OK to kill them) and set the minimum and maximum range for total hitpoint damage across the various unit types. These will be randomized by the specific GP function so you get varying levels of damage with each cast of the GP.

The last section tells the game what to do with a unit if it gets hit. Again these parameters just randomize the likelihood of the unit being thrown and if they are how high? how far? how fast?.

The actual parameters will vary across different "epic" god powers but the same overall structure applies. Just picture how the god power works during a game and the script becomes easier to read.

18.4 Creating Your Own God Powers

Because you do not have access to the actual God Power code, only the script that calls it using the underlying code to make your own God Powers is always going to be a bit of trial and error.

My advice is to start with the more obvious stuff like tempunit and swapunit and then try and play around with one of the epic God Powers to see the effect of changing the timings, proto units, special effects and damage profiles.

There probably have not been too many God Power mods made, so there is a lot to learn and hopeful scope for some very interesting and visual work.

19. Random Maps

In discussing Random Maps, and in the next Chapter on AI, I do not intend to explain how these work. The Random Map and AI Guides in the standard release and other good guides available on-line have far more detail. My main focus is what needs to be done to these areas to enable you to use your own civilizations and units.

From the point of view of Random Maps the issue is how we place our **unique proto unit** – the one that we use to select our own **Age 1 God** technology into a random map so that we can use this feature of the standard **Single Player** game using our civilization.

To do this we need to create our own version of each random map script and each random map control file we want to use with our civilization.

For example if we want to use the acropolis random map we need to:

- 1) Copy **acropolis.xs** as **myciv acropolis.xs**
- 2) Copy **acropolis.xml** as **myciv acropolis.xml**

Both versions of the random map will be available on the Map Selection screen, but when we have finished the new version will place our **unique proto unit**. In the example we used earlier of "borrowing" Loki, when we use the old random map Loki will play as Loki, when we use the new random map Loki will play as our new civilization.

Sorry if this sounds messy, but until we can work out how to define civilizations it is the only way!

19.1 Enabling a Random Map for New Civilizations

When a random map is created as well as all the resources, trees, wild life, etc., it places a starting settlement and a number of towers (in most cases) for each Player.

How you would need to modify the Random Map depends on whether your civilization replaces one or both of these units or whether you use another **unique proto unit**.

In the two civilizations I have created (the Shire and Mordor), the **unique proto unit** I used was a new town center (a Middle Earth Village for the Shire and a Tower of Isengard for Mordor) that replaced the standard settlement level 1 protounit. I also created a new type of tower for the Shire Civilization. Therefore, to use these civilizations these new proto units must be placed instead of the standard ones.

If your civilization does not replace these, for example your unique unit could be a copy of a **lure** proto unit simply called **MyCivId**, then you would only need to add this unit when laying out.

To explain how this works we will use the acropolis random map. Once you have made the changes once, they can usually be copied to other maps without any further change.

First, let us look at what is involved when we have created our own town center and tower proto units.

- 1) Copy the two files as outlined above and open the **myciv acropolis.xml** file with Notepad. It contains the following:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<mapinfo details = "52258" imagepath = "ui\ui map acropolis 256x256" displayNameID="52259"
cannotReplace=""/>
```

This file contains two text references and an icon reference. All we need to change here is the second text reference so we have the correct label on the map selection screen. Add a new text to your **(xpack)language.dll** file using the next available **id** and just enter **text=MyCiv Acropolis** and then save the file.

- 2) The next file we need to change is the **myciv acropolis.xs** file. Open this with Notepad and scroll down to the line:

```
// -----Define objects
// Close Objects
```

Immediately following this you will see the following definition of the "normal" starting settlement.

```
int startingSettlementID=rmCreateObjectDef("starting settlement");
rmAddObjectDefItem(startingSettlementID, "Settlement Level 1", 1, 0.0);
rmAddObjectDefToClass(startingSettlementID, rmClassID("starting settlement"));
rmSetObjectDefMinDistance(startingSettlementID, 0.0);
rmSetObjectDefMaxDistance(startingSettlementID, 0.0);
```

Immediately after this definition we need to define our own starting settlement, just take a copy of the existing definition and make the changes as follows:

```
int startingMyCivSettlementID=rmCreateObjectDef("Starting MyCiv Settlement");
rmAddObjectDefItem(startingMyCivSettlementID, "MyCiv Town Center", 1, 0.0);
rmAddObjectDefToClass(startingMyCivSettlementID, rmClassID("starting settlement"));
rmSetObjectDefMinDistance(startingMyCivSettlementID, 0.0);
rmSetObjectDefMaxDistance(startingMyCivSettlementID, 0.0);
```

We have created a new variable called **startingMyCivSettlementID** and defined it.

- 3) Now scroll down to the line headed:

```
// Place starting settlements.
```

and you will see the following, placement script:

```
rmPlaceObjectDefPerPlayer(startingSettlementID, true);
```

We need to change this to place our settlement when the cCivLoki condition is met.

```
for(i=1; <cNumberPlayers)
{
    if(rmGetPlayerCiv(i) == cCivLoki)
        rmPlaceObjectDefAtLoc(startingMyCivSettlementID, i, rmPlayerLocXFraction(i),
rmPlayerLocZFraction(i));
```

```

        else
            rmPlaceObjectDefAtLoc(startingSettlementID, i, rmPlayerLocXFraction(i),
rmPlayerLocZFraction(i));
    }

```

So now if civilization is Loki it places our custom built town center otherwise a it places a standard one.

- 4) Now scroll down until you find the line:

```
// Ramp Towers.
```

Immediately following this you will find the definition of the towers.

```

int avoidTower=rmCreateTypeDistanceConstraint("towers avoid towers", "tower", 8.0);
for(i=1; <cNumberPlayers)
{
    int startingTowerID=rmCreateObjectDef("Starting tower"+i);
    int towerRampConstraint=rmCreateCliffRampConstraint("onCliffRamp"+i, rmAreaID("player"+i));
    int towerRampEdgeConstraint=rmCreateCliffEdgeMaxDistanceConstraint("nearCliffEdge"+i,
rmAreaID("player"+i), 2);
    rmAddObjectDefItem(startingTowerID, "tower", 1, 0.0);
    rmAddObjectDefConstraint(startingTowerID, avoidTower);
    rmAddObjectDefConstraint(startingTowerID, towerRampConstraint);
    rmAddObjectDefConstraint(startingTowerID, towerRampEdgeConstraint);
    rmAddObjectDefToClass(startingTowerID, classTower);
    rmPlaceObjectDefInArea(startingTowerID, i, rmAreaID("player"+i), 6);

    /* backup to try again */
    if(rmGetNumberUnitsPlaced(startingTowerID) < 4)
    {
        int startingTowerID2=rmCreateObjectDef("Less Optimal starting tower"+i);
        rmAddObjectDefItem(startingTowerID2, "tower", 1, 0.0);
        rmAddObjectDefConstraint(startingTowerID2, avoidTower);
        rmAddObjectDefConstraint(startingTowerID2, towerRampConstraint);
        rmAddObjectDefToClass(startingTowerID2, classTower);
        rmPlaceObjectDefInArea(startingTowerID2, i, rmAreaID("player"+i), 1);
    }
}
}

```

In the Acropolis script, the random map has a preferred tower placement process (next to each ramp) and a fall back. In other scripts with is not necessary (as discussed below). We need to change this to allow for two tower types and a check on the players civilization.

```

int avoidTower=rmCreateTypeDistanceConstraint("towers avoid towers", "tower", 8.0);
int avoidMyCivTower=rmCreateTypeDistanceConstraint("towers avoid tower", "MyCiv Tower", 8.0);
for(i=1; <cNumberPlayers)
{
    int startingTowerID=rmCreateObjectDef("Starting tower"+i);
    int towerRampConstraint=rmCreateCliffRampConstraint("onCliffRamp"+i,
rmAreaID("player"+i));
    int towerRampEdgeConstraint=rmCreateCliffEdgeMaxDistanceConstraint("nearCliffEdge"+i,
rmAreaID("player"+i), 2);
    rmAddObjectDefItem(startingTowerID, "tower", 1, 0.0);
    rmAddObjectDefConstraint(startingTowerID, avoidTower);
    rmAddObjectDefConstraint(startingTowerID, towerRampConstraint);
    rmAddObjectDefConstraint(startingTowerID, towerRampEdgeConstraint);
    rmAddObjectDefToClass(startingTowerID, classTower);
}
}

```

```

int startingTowerID2=rmCreateObjectDef("Less Optimal starting tower"+i);
rmAddObjectDefItem(startingTowerID2, "tower", 1, 0.0);
rmAddObjectDefConstraint(startingTowerID2, avoidTower);
rmAddObjectDefConstraint(startingTowerID2, towerRampConstraint);
rmAddObjectDefToClass(startingTowerID2, classTower);

int startingMyCivTowerID=rmCreateObjectDef("Starting MyCiv tower"+i);
int MyCivTowerRampConstraint=rmCreateCliffRampConstraint("onCliffRamp2"+i, rmAreaID("player"+i));
int MyCivTowerRampEdgeConstraint=rmCreateCliffEdgeMaxDistanceConstraint("nearCliffEdge2"+i,
rmAreaID("player"+i), 2);
rmAddObjectDefItem(startingMyCivTowerID, "MyCiv Tower", 1, 0.0);
rmAddObjectDefConstraint(startingMyCivTowerID, avoidMyCivTower);
rmAddObjectDefConstraint(startingMyCivTowerID, MyCivTowerRampConstraint);
rmAddObjectDefConstraint(startingMyCivTowerID, MyCivTowerRampEdgeConstraint);
rmAddObjectDefToClass(startingMyCivTowerID, classTower);

int startingMyCivTowerID2=rmCreateObjectDef("Less Optimal starting MyCiv tower"+i);
rmAddObjectDefItem(startingMyCivTowerID2, "MyCiv Tower", 1, 0.0);
rmAddObjectDefConstraint(startingMyCivTowerID2, avoidMyCivTower);
rmAddObjectDefConstraint(startingMyCivTowerID2, MyCivTowerRampConstraint);
rmAddObjectDefToClass(startingMyCivTowerID2, classTower);

if(rmGetPlayerCiv(i) == cCivLoki)
{
    rmPlaceObjectDefInArea(startingMyCivTowerID, i, rmAreaID("player"+i), 6);

    // Check if their are 4 towers and try again if not
    if(rmGetNumberUnitsPlaced(startingMyCivTowerID) < 4)
    {
        rmPlaceObjectDefInArea(startingMyCivTowerID2, i, rmAreaID("player"+i), 1);
    }
}
else
{
    rmPlaceObjectDefInArea(startingTowerID, i, rmAreaID("player"+i), 6);

    // Check if their are 4 towers and try again if not
    if(rmGetNumberUnitsPlaced(startingTowerID) < 4)
    {
        rmPlaceObjectDefInArea(startingTowerID2, i, rmAreaID("player"+i), 1);
    }
}
}
}

```

If you did not really follow this do not worry too much. The code itself works and you just need to ensure that the three references to the custom tower proto unit definition - **MyCiv Tower** and the one reference to the borrowed civilization – **Loki**, are correct for your civilization and then you can just cut and paste this into you random map file. In the other random maps the tower definition is not as complex. In these just after you enter your settlement details you will see the line heading:

```
// towers avoid other towers
```

followed by the following standard tower definition:

```

int startingTowerID=rmCreateObjectDef("Starting tower");
rmAddObjectDefItem(startingTowerID, "tower", 1, 0.0);
rmSetObjectDefMinDistance(startingTowerID, 22.0);
rmSetObjectDefMaxDistance(startingTowerID, 28.0);
rmAddObjectDefConstraint(startingTowerID, avoidTower);

```

Immediately after this you would insert the new tower definition for your civilization:

```
int startingMyCivTowerID=rmCreateObjectDef("Starting MyCiv tower");
rmAddObjectDefItem(startingMyCivTowerID, "MyCiv Tower", 1, 0.0);
rmSetObjectDefMinDistance(startingMyCivTowerID, 22.0);
rmSetObjectDefMaxDistance(startingMyCivTowerID, 28.0);
rmAddObjectDefConstraint(startingMyCivTowerID, avoidMyCivTower);
```

This is a simple tower definition that just tries to spread them evenly around the town center.

- 5) Save the files and put them into your production \RM (AoM) or \RM2 (AoMTT) folders (they do not need converting); start the game and go through the Single Player Random Map set up. Select the appropriate civilization and select your new random map. Press start.

You should be presented with a game screen that has placed your custom town center and towers and your starting units should have been created by the TECHTREE(X).

If something went wrong, you will get a script load failure. So check that the above changes were made correctly, mis-spelling is the usual problem so just make sure all your **MyCiv** changes were correct.

Once you have the first random map working it is relatively straight forward to apply the changes to the other random map scripts and .xml files.

20. Artificial Intelligence

The final area for discussion is Artificial Intelligence. If you create more than one civilization and want them to battle, or if you want to play a standard civilization against your new one, the Computer will need to control one of them.

The AI unit is smart enough to do a lot of things, but others it will need help with. This will mean creating the AI scripts required to support your civilizations.

The extent to which you will need to change the standard AI scripts will be very dependent on the new units you create and the only way to find out is to use the standard AI scripts and see what is happening.

If your characters are equivalent to the standard characters, you may not need to do much. The standard AI can identify your economic units, and various military units and its standard routines generally search for unit types.

On the other hand if you do some slightly off-beat things (for example in my Mordor civilization I used Goblins as Builders and Herdable animals – yes the Orcs eat them), the game has problems, and you need you need to be explicit about who does what. Similarly, in my Shire civilization it uses Hobbits, Elven Artisans, Elven Gatherers and Men of Rohan as economic units but with different building skills. Unfortunately the game seemed to have difficulty with this.

You will also find that the standard AI does some specific things dependent on the civilization and these will no longer work.

If you find it necessary to create a new set of AI files, my only advice is to create a set that is specifically designed for your civilizations, rather than trying to extend the standard ones. If you download my LOTR Civ Pack and look at the AI files included with for the two civilizations, this will hopefully head you in the right direction.

If not post your problems and I may be able to address them.

21. Where to Next?

The joy of AoM modding is that there always seems to be new areas to explore. Hopefully in the near term, a more elegant way to create civilizations will be found and this may unlock more modding opportunities. Creating models for the game is also an area where only a few people are making inroads. After only 6 months of modding, I certainly think there is a lot more to learn.

Hopefully this guide will get you on the start to some modding success and I wish you the best of luck and look forward to seeing your creations.