NTNU

Det skapende universitet

Department of
Computer and Information Science

A Component based Game Architecture for Unknown Horizons

Thomas Kinnen

TDT4570 - Game Technology, Specialization Project

Table of Content

1 Introduction

1.1 Motivation

Unknown Horizons[1] is an open-source real-time strategy game developed by a team of programmers, artists, game designers and many more around the globe. The first revision was committed in late 2007[2].

As the project evolved the games's code architecture grew dynamically, without much planned structure or designed architecture. This resulted in a very tight coupling between the different components inside the game, making it difficult to add/change certain functionalities in the game. This became clear when adding the boat builder building a while back, which resulted in months of fixing introduced bugs.

Unknown Horizons uses the outdated idea of making use of multiple inheritance to compose its in-game objects. Besides introducing very tight coupling between the different classes the current approach also does not allow non programmers to add new assets to the game. For an open-source project this is clearly not ideal, as user contributions would add great value to the project and save valuable programming time.

The idea for this project is to research how this problem is solved in similar open-source games and to transfer the results to the *Unknown Horizons*source-code. The following games have been chosen to be researched:

- *Battle for Wesnoth*[3]
- *0 A.D.*[4]
- *Mega Glest*[5]

[1]Unknown Horizons website: `http://www.unknown-horizons.org`

[2]First commit to *Unknown Horizons*: `https://github.com/unknown-horizons/unknown-horizons/commit/53eec12fd8bb52ac1a6ccfdb097296c479499dfd`

[3]Battle of Wesnoth website: `http://www.wesnoth.org`

[4]0 A.D. website: `http://wildfiregames.com/0ad/`

[5]Glest website: `http://megaglest.org/`

1.2 Problem Statement

Two main questions should be answered by this project:

- Which architecture do open-source games similar to *Unknown Horizons*use to model their ingame objects?
- Can users add objects without modifing the game's code and if yes – how?

1.3 Project Context

This project is conducted for the course *TDT4570 - Game Technology Specialization Project*[6] which is part of **NTNU**'s computer science master program.

2 Research Methods and Questions

In this section we present our research questions and methods used in this work.

2.1 Research Questions

We work on a set of four main research questions:

- RQ1: Which architecture is used to describe objects in-game?
- RQ2: How are new objects added to the game?
- RQ3: Can existing objects easily be modified?
- RQ4: Are tools available to help with adding/modifing objects?

2.2 RQ1

Which architecture is used to describe objects in-game?
The goal of this question is to find out if the game uses an inheritance based approach, a component based approach or some other design to describe objects in game.

2.3 RQ2

How are new objects added to the game?
With this question we want to find out if many changes have to be made to the code to add new objects. We also want to know if the objects are data or code driven. If they are data driven, we research which technology is used. Our goal is to find the easiest methode of adding objects to the game.

[6]TDT4570 Project description: `http://www.idi.ntnu.no/emner/tdt4570/`

2.4 RQ3

Can existing objects easily be modified?

Our goal is to assess if existing objects are easily changable or if code has to be changed to modify them. We research what the possible implications of changing objects are.

2.5 RQ4

Are tools available to help with adding/modifing objects?

As creating game content is usually done by non programmers, we want to assess how easy it is for them to add content to the game and if there are tools to support them in the progress.

2.6 Research method

The first part of this work is four case studies in which we research four existing open source games. All games are mainly real-time strategy games, so their implementations face similar problems and are comparable to some degree. We will then use the gained knowledge to improve the handling of objects in *Unknown Horizons* by designing and implementing a system combining the best practices we found in the case studies. Literature research is not a big part of this project, as there is almost none available on the topic of game architectures, besides from massiv multiplayer online role playing games.

3 State-Of-The-Art

3.1 Related Work

3.2 Literature

4 Own Contribution

In this section we present four different case studies to answer our research questions. We begin with presenting *Unknown Horizons* as it is the project which focus our efforts of improvement on. It is followed by *Battle for Wesnoth*, *Mega Glest* and *0 A.D.*. The results are then evaluated and transferred to *Unknown Horizons*.

4.1 Unknown Horizons

Unknown Horizons as described on the project website:

> *Unknown Horizons* is a 2D realtime strategy simulation with an emphasis on economy and city building. Expand your small settlement to a strong and wealthy colony, collect taxes and supply your inhabitants with valuable goods. Increase your power with a well balanced economy and with strategic trade and diplomacy.

RQ1

Unknown Horizons uses a largely inheritance based approach to describe ingame objects. As the game is programmed using the Python[7] programming language it is possible to use multiple inheritance. The project makes great use of this ability, resulting in large inheritance trees. To illustrate this we have generated an inheritance diagramm for the *Settler* class in Figure 1. The tree consists of 16 classes including many cases of multiple inheritance.

Experience in working on this project has shown that making changes to any of the classes included in this tree is often a very big task and comes with a great risk of introducing bugs into the code. It is also very difficult or even impossible to write unit tests for these classes, as they are so dependent on each other and the game core, that it is almost impossible to create the needed environment synthetically.

Settler Explained The *Settler* class is comprised of 4 basic classes: *BasicBuilding*, *SelectableBuilding*, *BuildableSingle* and *CollectingProducerBuilding*. This is how most buildings in *Unknown Horizons* are constructed.

BasicBuilding is a base class for every building, it loads graphics and provides basic information like the name, position, owner and functionality for running costs and level upgrades.

SelectableBuilding is a decorating class, that implements functions for selecting the building ingame. It manages showing ingame menus and outlines. If a building is not supposed to be selectable, this class should not be inherited.

BuildableSingle is a decorating class which is used when building new buildings. It tells the game that it can only be built as single instance, so there is no building of multiple instances at once. For this purpose the code provides the *BuildableLine*, *BuildableRect*, etc. classes which can be used if needed.

[7] Python website: `http://www.python.org`

CollectingProducerBuilding is a collectiv class to make the Settler have collecting units which pick up resources for usage and then produce something from it. This is easier to demonstrate on a *LumberJack* for example, he picks up trees and produces planks from it. The Settler consumes resources (food, textiles, etc.) and in turn produces the abstract resource happiness.

Datadriven? *Unknown Horizons* uses a SQLite[8] database to save parts of the object's attributes. For example the size, health and name are saved in the database. This is necessary to make the highler level classes in the architecture reusable for subclasses. All buildings have a size, but it may be different from building type to building type. It is saved to an external file to make it easily editable by non programmers.

In summary we can say that the objects are partly datadriven, but usually it is not possilbe to add new buildings without writing new code.

RQ2

In order to add a new building to *Unknown Horizons* one has to look at the characteristics the building should have and then find the appropriate classes from the *Unknown Horizons* building classes collection. Those can then be combined to form new buildings.

For example to create a settlement wall one could use the classes *BuildableLine* and *BasicBuilding*. This is a very simple example which does not need to inherit many classes, as its functionality is very limited. All attributes of this building can then be added in the database by using any SQLite database manager.

RQ3

Modifying existing ingame objects in *Unknown Horizons* can be easy and very difficult. This depends on the degree of change that is to be made. If only basic attributes like health, production time or similar are to be changed, then it can easily be done by someone who knows their way around the database. If however new functionality is required, for example an building which previously did not collect resources needs to collect resources, a change in the games code is most certainly required. Again sometimes if the functionaliy exists, this can be easy by just adding another class to the hierarchy of the building or it can be very difficult if new functionaliy in the existing classes is required.

A good example for this is the boatbuilder, which is mainly a *CollectingBuilding* which produces units instead of resources. The building has been implemented for over a year now and the team is still not certain if it works bugfree or not, as it required huge modifications to the production classes to be able to produce units instead of resources.

[8] SQLite website: `http://www.sqlite.org/`

RQ4

There are no tools available to help with the addition/edition of content at this point. A map editor is planned for future versions, but it is not yet in a working state.

4.2 Battle for Wesnoth

Battle for Wesnoth as described on the project's website:

> *The Battle for Wesnoth is a Free, turn-based tactical strategy game with a high fantasy theme, featuring both single-player, and online/hotseat multiplayer combat. Fight a desperate battle to reclaim the throne of Wesnoth, or take hand in any number of other adventures...*

RQ1

Battle for Wesnoth comes with its own markup language, *WesnothMarkupLanguage - WML*, to describe units, campaigns, AIs, missions, maps, sounds, etc. *WML* is similar to other markup languages like *XML*, but more human readable and provides some basic functions to create logic - like basic if-clauses and variables. Entities are described in a component like way, not all clauses are components though. For most clauses like skills, attacks and races components are used in code. Other attributes such as *[portrait]* are not mapped to their own component, but are just read as data for the basic unit class. *Battle for Wesnoth* parses all the tags in a config file into the config class where the single tags can be easily accessed by code. The actual use of the tags is left to the code using the config. Classes like unit and race read values from the config class.

To ease development it is also possible to access many internals using a lua API. This is used for reading in tags for maps and units. According to the development team the lua API can be used for modding as well, this behaviour is not documented anywhere though.

Datadriven? As all game content is described using *WML*, *Battle for Wesnoth* can be sean as completely datadriven. Wesnoth contains an in-game option to download other mods, containing new units, campaigns, etc. Within the concept of a round based strategy game the engine completely independent of the content.

RQ2

New objects are added to the game by adding a new file containg basic *WML* with the description of the unit. Sinc ethe game is fully modable all units, maps and campaigns can be replaced just by editing and adding new files into the basic directory structure Community [2011].

In order to add new *WML* attributes, the c++ sourcecode has to be changed to recognize them. The *Battle for Wesnoth* parser does not know anything about the tags it loads, therefore the only constraint is that the new tag should be used/accessed in another c++ class, like the unit class, or lua helper code.

Listing 1: A basic (shortened) *Battle for Wesnoth* unit definition in WML

```
[unit_type]
    id=Elvish Lady
    name= _ "female^Elvish␣Lady"
    gender=female
    race=elf
    image="units/elves-wood/lady.png"
    profile="portraits/elves/lady.png"
    {MAGENTA_IS_THE_TEAM_COLOR}
    hitpoints=41
    movement_type=woodland
    movement=6
    experience=150
    level=3
    alignment=neutral
    advances_to=null
    {AMLA_DEFAULT}
    cost=10
    usage=null
    description= _ "Elves␣choose␣their␣leaders␣for␣their␣wisdom␣and␣sensitivity␣to␣
        the␣balance␣of␣universal␣forces;␣foresight␣is␣what␣has␣protected␣them␣in␣
        times␣of␣uncertainty.␣Their␣just␣reign␣is␣rewarded␣by␣the␣unflagging␣fealty
        ␣of␣their␣people,␣which␣is␣the␣greatest␣gift␣for␣which␣any␣ruler␣could␣ask.
        "
    [portrait]
        size=400
        side="right"
        mirror="true"
        image="portraits/elves/transparent/lady.png"
    [/portrait]
[/unit_type]
```

RQ3

To edit a unit in *Battle for Wesnoth* the *WML* files have to be changed, nothing else has to be done. In order to change the units behaviour the c++ code has to be edited.

RQ4

Battle for Wesnoth comes with a set of tools to help developers and content creators. Wenoth comes with a map editor, tools to validate user created *WML* and provides an

Eclipse[9] plug-in, called *"The Battle for Wesnoth UMC Development IDE"*[10].

It can setup campaigns, races and more for the user, provides syntax highlighting for the WML and also some means of auto-completion. It can also launch the map editor and game with the specified campaigns and maps the user is working on and provides means of using the WML validation tools. A screenshot of the WML editor is provide in Figure 2.

4.3 Mega Glest

Mega Glest as described on the project's website:

> *MegaGlest is a free and open source 3D real-time strategy (RTS) game, where you control the armies of one of seven different factions: Tech, Magic, Egyptians, Indians, Norsemen, Persian or Romans. The game is setup in one of 16 naturally looking settings, which -like the unit models- are crafted with great appreciation for detail. Additional game data can be downloaded from within the game at no cost.*

RQ1

Mega Glest uses a mixture of inheritance and component-based object description. Basic things are set using inheritance, for example the *UnitType* class inherits from the *ProducibleType* class, as every unit in the game is producable. More advanced things are added to the unit as components, for example the *UnitType* has *Level*,*SkillType*, *Resource*, *CommandType* and *UnitParticleSystemType* components. Units themselves are part of a bigger compontent hierarchy: Units are part of a *FactionType*, which is part of a *TechTree*. See Figure 3 for a detailed structure analysis. A class ending in **Type* is used to represent prototypes for the actual instance classes. For example the *UnitType* class loads all necessary data from the XML definitions. Ingame a *Unit* instance is used, which itself contains a *UnitType* as information base.

Datadriven? *Mega Glest* is fully datadriven. All information needed for ingame objects, scenarios and campaigns is stored in XML files. This makes *Mega Glest* more of a

[9] Eclipse Project Homepage: `http://www.eclipse.org`
[10] UMC Plugin Website: `http://eclipse.wesnoth.org/`

game engine with a focus on real-time strategy than only a game. Several[11] mods exist, proving that it is indeed possible to create new games using *Mega Glest* as a game engine.

4.4 RQ2

In order to add new objects to *Mega Glest*, new XML files have to be created. It can contain the definition of a unit, campaign, tech tree or similar. See Listing 2 for a shortened example of a basic unit definition. Every class has many parameters which allow the user to specify many details for every unit.

Listing 2: A basic *Mega Glest* (shortened) unit definition in XML

```
<unit>
  <parameters>
    <size value="1"/>
    <height value="2"/>
    <max-hp value="450" regeneration="5"/>
    <max-ep value="3000" regeneration="30"/>
    <armor value="15"/>
    <armor-type value="leather"/>
    <sight value="12"/>
    <time value="200"/>
    <multi-selection value="true"/>
    <cellmap value="false"/>
    <levels>
      <level name="expert" kills="5"/>
      <level name="master" kills="15"/>
      <level name="legendary" kills="30"/>
    </levels>
...
...
</unit>
```

RQ3

In *Mega Glest* editing objects is easy: Simply change the XML files to match the new requirements. Changing things in the source code seems nicely doable as there are not too many inheritance based objects, instead composition is preferred.

RQ4

Mega Glest comes with a map editor(Figure 4) to help with creating new maps to use ingame. It allows editing every possible detail of the map and is thus of great help for content creators. *Mega Glest* also comes with a model viewer, with which the custom *g3d* 3D model file format can be opened. It allows to view basic models and particle effects.

[11] List of some mods: `http://www.moddb.com/games/megaglest/mods`

4.5 0 A.D.

0 A.D. as described on the project's website:

> *0 A.D. (pronounced "zero ey-dee") is a free, open-source, cross-platform real-time strategy (RTS) game of ancient warfare. In short, it is a historically-based war/economy game that allows players to relive or rewrite the history of Western civilizations, focusing on the years between 500 B.C. and 500 A.D. The project is highly ambitious, involving state-of-the-art 3D graphics, detailed artwork, sound, and a flexible and powerful custom-built game engine.*

4.6 RQ1

0 A.D. uses a completely component based approach to describe entities in the game. Units are described using a simple XML based format, making up single units of many components. The engine provides components like *Attack*, *Cost*, *Position* or *VisualActor*. See 0A.D. [2011a, Entity Component Documenation] for a complete list and details.

It is possible to implement components in c++ and javascript. The general idea is to use javascript where possible and only use c++ if necessary for performance or communication with the game engine, for instance for rendering. The components communicate with each other using a message based system or calling methods on other components directly. The first approach is to be prefered, as the implementations remain seperated from each other this way. A component can send messages directly to a specific component type, or broadcast it to everything listening to the specific message. A entity in the game is basically a number associated with a set of components. There is no inheritance involved in creating an entity, other than inside a single component.

The engine allows to *hotload* components implemented in javascript, meaning changes in the javascript files will be detected while the game is running and loaded into the engine. This enables the developer to make changes to the components behaviour while the game is running and directly seeing his results in-game.

Datadriven? *0 A.D.* is completely datadriven. All information needed to construct a unit is saved in the basic XML format. The XML format is specified by the components directly, making it easy to extend the markup language by new components. As the game logic is implemented using these external components, it is easy to exchange the complete gameplay logic, thus enabling very wide modding support and seperating the gameplay logic from the main engine.

4.7 RQ2

New objects can easily be added to the game by adding new XML unit entity definitions. No extra source-code is necessary. A shortened sample unit definition is given in Listing 3.

Listing 3: A basic *0 A.D.* (shortened) unit definition in XML

```
<Entity parent="units/cart_cavalry_spearman_b">
  <Attack>
    <Melee>
      <Hack>6.0</Hack>
      <Pierce>16.0</Pierce>
    </Melee>
    <Charge>
      <Hack>18.0</Hack>
      <Pierce>48.0</Pierce>
    </Charge>
  </Attack>
  <Health>
    <Max>140</Max>
  </Health>
  <VisualActor>
    <Actor>units/carthaginians/cavalry_spearman_a.xml</Actor>
  </VisualActor>
</Entity>
```

4.8 RQ3

All units are defined using basic XML based definitions, making it easy to edit them with a normal text editor. No changes to the code are necessary to edit any unit.

4.9 RQ4

0 A.D. comes with an editor called *Atlas*. It is a map and scenario editor and is not meant for editing unit definitions. The editor is very similar to *Mega Glest*'s editor in terms of features. It allows the user to edit any detail of a map and comes with a unit viewer, that previews animations. A detailed user-manual is provided on the project's developer pages 0A.D. [2011b].

Tool support for editing unit entity descriptions is not provided.

References

0A.D. Entity component documentation, 2011a. URL `http://svn.wildfiregames.com/entity-docs/#component.VisualActor`. [Online; accessed 08-November-2011].

0A.D. Atlas (scenario editor) manual, 2011b. URL `http://trac.wildfiregames.com/wiki/Atlas_Manual`. [Online; accessed 08-November-2011].

Wesnoth Community. Editingwesnoth, 2011. [Online, accessed 08-December-2011].

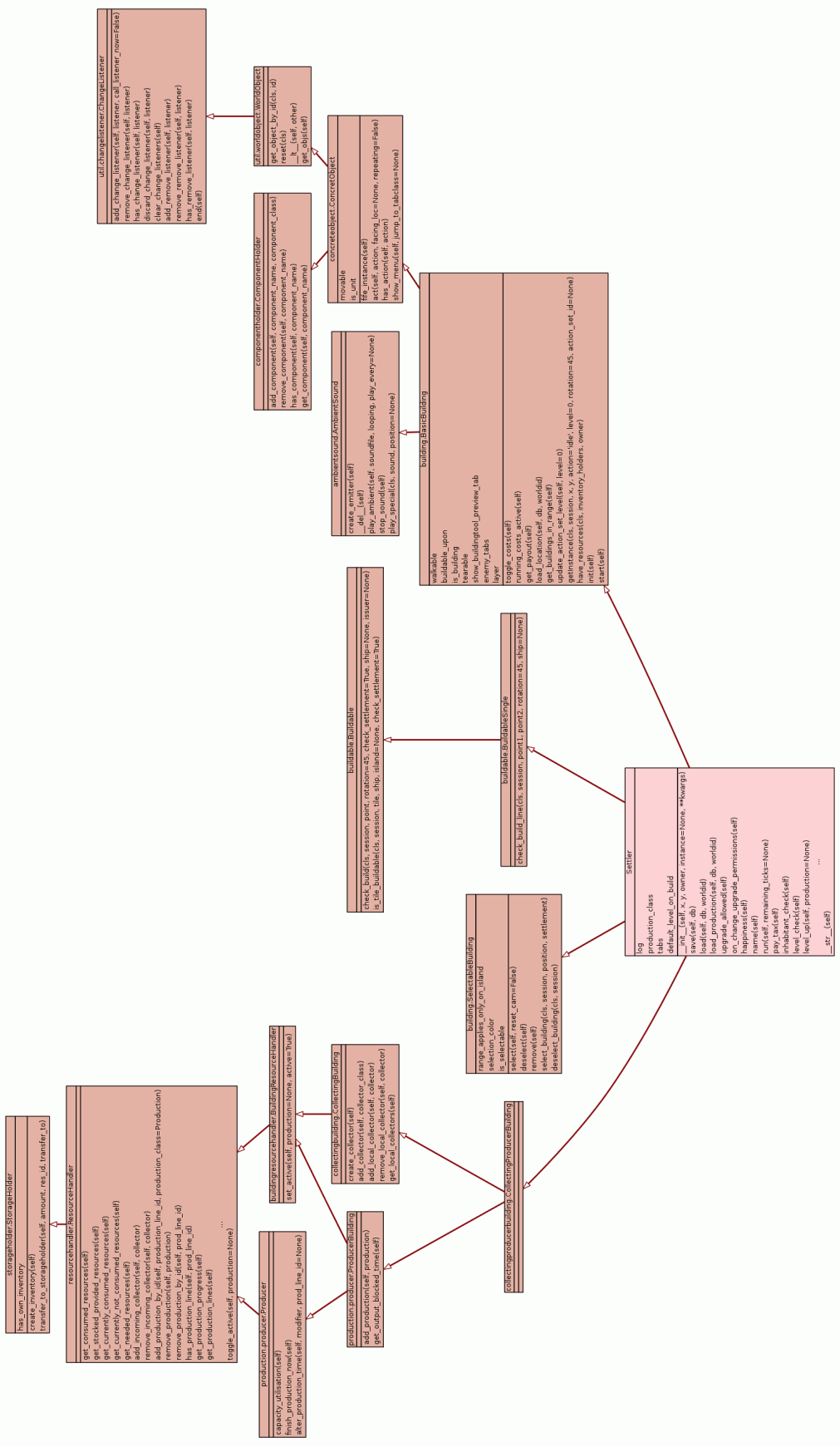

Figure 1: Inheritance tree for the *Settler* class in *Unknown Horizons*

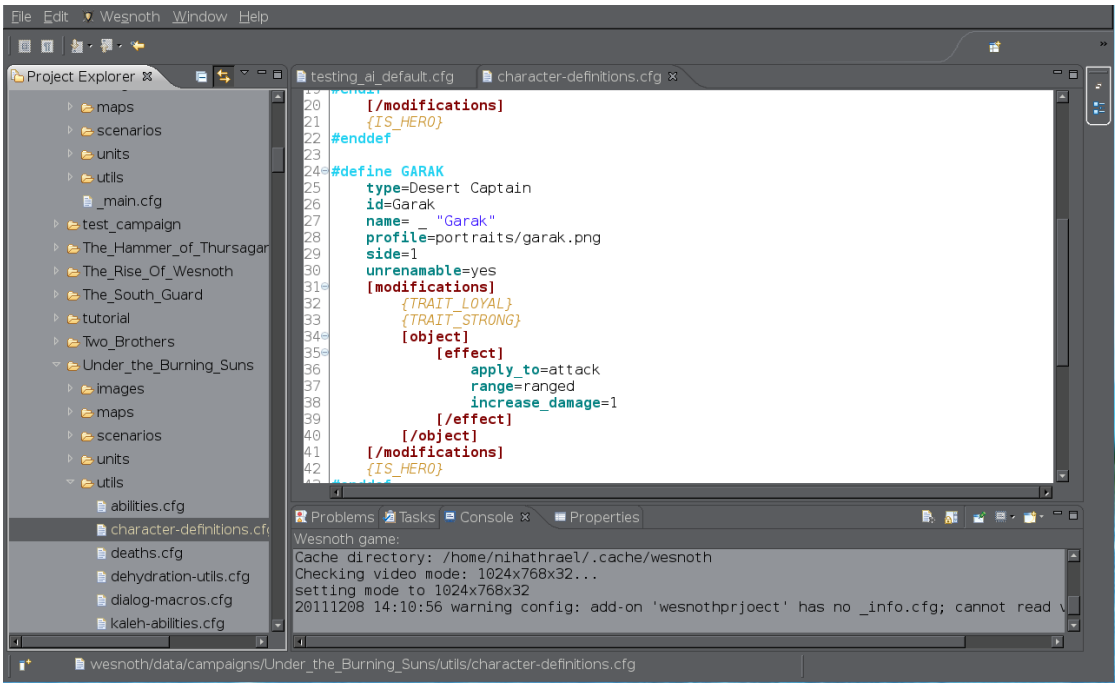

Figure 2: Wesnoth UMC Plugin WML editor

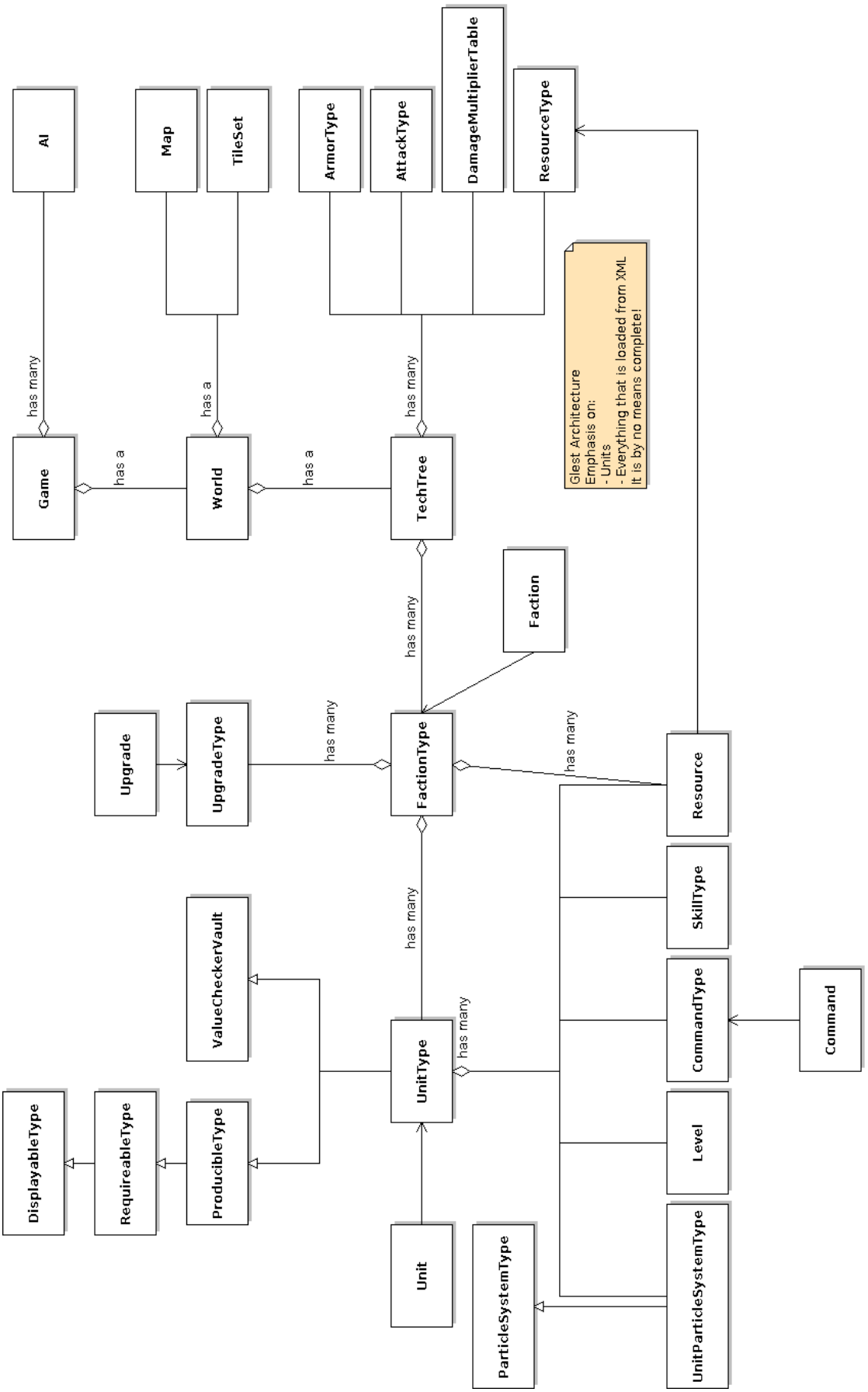

Figure 3: MegaGlest class hierarchy diagram

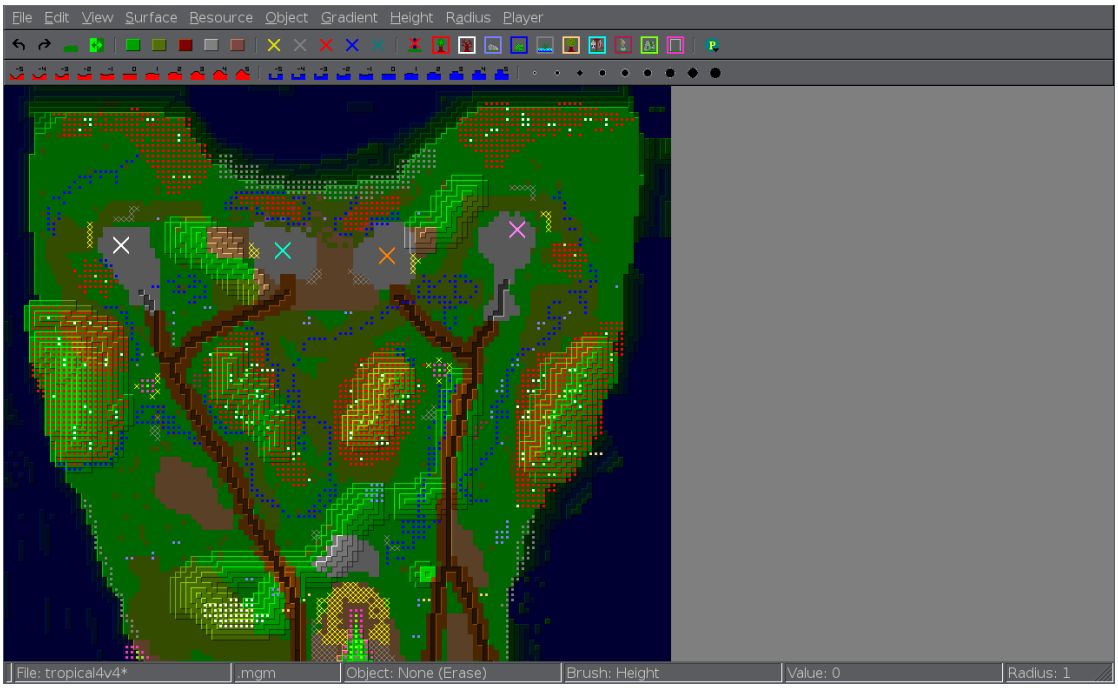

Figure 4: *Mega Glest* map editor

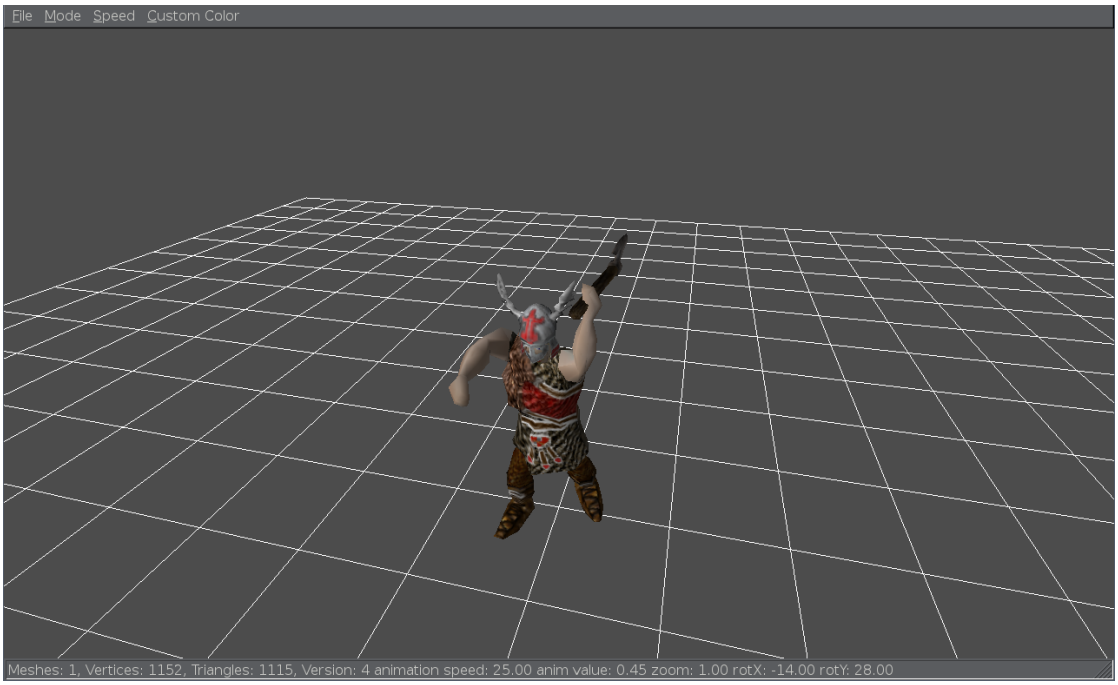

Figure 5: *Mega Glest* model and particle viewer