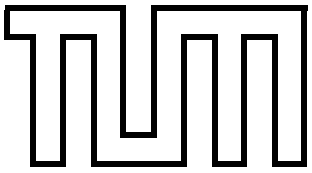


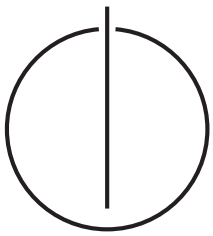
FAKULTÄT FÜR INFORMATIK

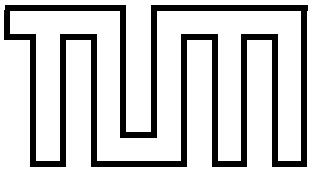
der Technischen Universität München

Bachelor's Thesis in Informatics

Open Innovation in Game Design Using the Example of a Warcraft 3 Tower Defense

Jan Finis





FAKULTÄT FÜR INFORMATIK

der Technischen Universität München

Bachelor's Thesis in Informatics

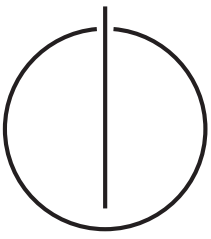
Open Innovation in Game Design Using the Example of a Warcraft 3 Tower Defense

–

Open Innovation im Spieledesign am Beispiel einer Warcraft 3 Tower Defense

Jan Finis

Supervisor:	Prof. Dr. rer. nat. Rüdiger Westermann
Advisor:	Dr. rer. nat. Jens Schneider
Date of submission:	November 9, 2009



Affidavit

I assure the single handed composition of this bachelor's thesis only supported by declared resources

Munich, November 9, 2009

Jan Finis

Acknowledgement

Let me thank Prof. Westermann and Dr. Schneider for giving me the chance to choose my own topic for this thesis in which I could combine my personal interests and skills with my final thesis.
Additional thanks go to the current YouTD admins *tolleder* and *Master Cassim* who helped me administrating the user submitted content and suggested game features.
General acknowledgements go to everybody who submitted content for the game created, thus making the Open Innovation concept possible.

Abstract

This work applies the concept of Open Innovation in the field of game design. Open Innovation is a process which involves customers into the design and creation of products. The aim of this thesis is to create a game with a considerable amount of content created by the players themselves. The game, which is named *YouTD*, is realized as a modification for the game *Warcraft 3: The Frozen Throne*. Reasons for choosing this very platform will be discussed, the steps of the creation processes will be covered, and the results will be evaluated to determine if using Open Innovation in game design is a promising concept.

Zusammenfassung

Diese Arbeit wendet das Open Innovation Konzept im Bereich des Spiele-Designs an. Open Innovation ist ein Prozess, in dem Kunden in das Design und die Erstellung des Produkts mit eingebunden werden. Das Ziel dieser Arbeit ist die Erstellung eines Spiels mit einer großen Menge an Inhalten, die von den Spielern selbst erzeugt werden. Das Spiel, welches *YouTD* getauft wurde, wird realisiert als Modifkation für das Spiel *Warcraft 3: The Frozen Throne*. Gründe für die Entscheidung für genau diese Plattform werden erläutert, die Schritte des Entstehungsprozesses erklärt und die Resultate ausgewertet, um zu entscheiden, ob Open Innovation, angewendet im Bereich des Spiele-Designs, ein vielversprechendes Konzept ist.

Contents

1 Introduction **1**
1.1 Open Innovation in Game Design 2
1.2 Summary 4

2 Tower Defenses and Their Features **7**
2.1 The Tower Defense Genre 7
2.2 Special Abilities 11
2.3 YouTD and its Features 14

3 About the Chosen Platform **19**
3.1 The Graphics Engine 20
3.2 Useful Game Features 23

4 The World Editor **25**
4.1 Terrain Editor 25
4.2 Object Editor 28
4.3 Other Editors 29
4.4 Communities and Services 30

5 Scripting Warcraft 3 **31**
5.1 The Trigger Concept 31
5.2 Script Languages: GUI, JASS, vJASS 34
5.3 Editing maps procedurally using GSL 39

6 Conception of Open Innovation Games **41**
6.1 Roles, Architecture and Use Cases 42
6.2 The Open Innovation Workflow 45

7 The Realization of YouTD **47**
7.1 Development Kit 47
7.1.1 Overlay Engine and API 48
7.1.2 Content Creation Map 52
7.1.3 Export and Import Script 55
7.1.4 Development Kit Bundle and HowTo 57
7.2 Web Site 58
7.3 Game Stub and Build Script 62
7.4 Balancing the Game 67
7.4.1 Tower Balance 67
7.4.2 Creep Balance 70
7.5 Attracting Contributors 74

8 Results, Feedback and Future Work **77**
8.1 Content Received From Users 77
8.2 Problems and Possible Solutions 80
8.3 Success and User Feedback 81
8.4 Conclusion and Future Work 82

Bibliography **i**

List of Figures **iii**

List of Tables **v**

Chapter 1

Introduction

Today, the process of creating a game involves a lot of design work. The days when cheaply designed games like “Pac-Man” or “Tetris” with a few hours of design work were sufficient are long gone. Even if there still exist small and quickly designed games like flash games or java games for mobile phones, the majority of PC games are big projects with dozens of developers and designers and millions of euros of development costs.

This work applies the concept of Open Innovation onto game design in order to drastically reduce the number of developers, artists, and designers required and thus decrease the development costs.

The idea behind Open Innovation in game design is to allow ordinary users, in this case players, to create pieces of content for games and upload them to a web site, for example. This content can then be accessed and rated by other users, and if the user community agrees that a piece of content fits into the game, it will be added to the game procedurally.

The aim of this work is to create a prove-of-concept game where content is created by the players themselves. Afterwards, the success of the concept will be evaluated to see if a sufficient amount of players submitted content with acceptable quality to make Open Innovation viable for game design.

The genre chosen for this thesis are *Tower Defenses* (TDs) [1]. These are games in which one or more human players build towers to shoot at computer-controlled enemies who try to reach a target area. If such an enemy manages to reach the target before he is killed by the towers, the player’s life-count is decreased. If the count drops down to zero, the player loses. If he survives long enough, he wins.

As a basis for the game, the well-known top selling PC game *Warcraft 3:Reign of Chaos* [2] with its expansion pack *Warcraft 3: The Frozen Throne*[3] by *Blizzard Entertainment* was chosen (hereinafter called *Warcraft 3*, *Warcraft* or *WC3*). Thus, the game will not be a standalone executable, but a modification for another game.

Since creating a whole game engine would go beyond the scope of this thesis, this solution was chosen to allow to focus on the topics of main interest, i.e., on the Open Innovation concepts.

Warcraft 3 provides a sophisticated 3D graphics engine, networking, a basis for players to gather for games, and a large Tower Defense-affine community which will speed up the Open Innovation process. It also provides an editor allowing to create own games with its engine, which makes the choice of this platform possible. As setting, the fantasy genre with magic and mythical creatures is chosen, since the models of Warcraft 3 fit best into this genre.

The game to be created is named *YouTD*, which is an allusion to the well-known video portal *YouTube* and other products prefixed with "You-", expressing that the user himself is deeply involved into the product. Since the game will use Open Innovation and user-created content, this prefix fits well into the concept.

1.1 Open Innovation in Game Design

The perception of *Open Innovation* was formed by the American economic scientist *Henry Chesbrough* [4, 5]. He claims that "firms can and should use external ideas as well as internal ideas, and internal and external paths to market, as the firms look to advance their technology". In the field of Computer Science, this means that firms should not only rely onto their developers, but should try to licence inventions of others.
To go even further, companies have started to not only obtain knowledge and design work for money from other firms, but also from the customers themselves, which is often called *Social Commerce* [6]. This means that, for example, customers can design their own products or engage in marketing by including a firm's online shop into their web presence. Programs which allow customers to advertise friends to obtain some presents or amenities and even recommender- and product-rating-systems can also be labeled Social Commerce.
Even if Chesbrough coined Open Innovation to be the knowledge obtained from other firms, Reichwald et al. [7] describe the involvement of the customer into the innovation process as Open Innovation, thus more or less equalizing it to Social Commerce. In addition to the previously mentioned terms, this process is also called *interactive value creation* [7] or *Crowdsourcing* [8]. Many terms seem to exist which overlap in their semantics, but, to simplify nomenclature the process of involving users into product innovation and creation will be called Open Innovation hereinafter.
A prominent example for this concept is the German project *Spreadshirt* [9], where T-shirts can be created and bought by the customers themselves. In addition, users can design their own T-shirts and create an online shop to be embedded into their web sites to receive a share of the profit from the shirts they designed. Spreadshirt does not design any shirts, the whole design work is done by the users themselves. This Open Innovation concept was so successful that, although Spreadshirt was founded with no seed capital in 2004, it has grown fast and now employs 300 people and has registered offices in six countries of the world.

This thesis focuses on the user-generated content and the rating and updating of this content by the community. The generation process can also be collaborative, meaning one user can continue the work of another one. The users are not promised any material profit from their labor, instead the following mechanisms are used to motivate users to create content:

- Users who take part in the creation process of a product are likely to use it themselves later on. So they take part to make the product suit their needs.
- Users get the impression of having created something that they can proudly show to their friends.
- Immaterial profits are promised, e.g. users will be mentioned on the product to gain some sort of reputation.
- For some users, the creation process is fun and they like to do it with no other source of motivation necessary.
- Alternatively, small material profits can be promised, for example, the users submitting the best piece of content will receive an award. However, such profits are kept low in value.

The advantages a company can obtain from using user-created content are obvious:

- Costs for designers are reduced, since the community will design parts of the content.
- The users get involved into the product innovation process, which makes them feel associated with the product since they have participated in its creation. This will make users prefer this product over competing ones.
- The chance that the product does not suit customers' needs is decreased, because if somebody feels that something is missing he can self-create the missing part.

Reichwald et al. [7] and Krempl et al. [10] further investigate the concept of Open Innovation. They all conclude that Open Innovation is a very successful concept in fields where it is applied and can bring substantial benefits for a company using it. However, using this concept in game design is not widely spread at the moment, so this thesis tries to prove the feasibility of this concept when applied in the field of game design.

Allowing users to create their own content for games has a long history. There already existed games for the *C64* like *Mr. Robot and his Robot Factory* [11] which allowed users to create their own levels. However, no scripting was allowed and thus no own games could be created, just levels and environments.

When the first games included script languages which could be used to alter the game rules, users started to create games with their own rules. As an early example, *Starcraft* [12] allowed users to alter the gameplay rules, empowering users to create the first Tower Defense. In fact, the whole genre arose from such Starcraft modifications. However, the script language used in Starcraft did not have access to many gameplay features. Only a few aspects could be altered with it, so some sorts of games could not be created.
In newer games, the script languages are given more and more access to the game engine and thus allow a larger range of games to be created. For example the *Unreal* series with its editor *UnrealEd* [13] allow the users to freely alter most parts of the gameplay and the user interface (UI). Even if Unreal itself is an ego-shooter, many different games varying from racing simulations [14] to simple UI games like the well-known *Tetris* [15] were created by the users.
This thesis uses a game as platform as well: Warcraft 3 as a successor of Starcraft is well applicable for a Tower Defense game.
Even though many games allow the users to create modifications and play them online together, almost none of today's games tries to take advantage from this user-created content. Games with good level editors use these editors as an advertising argument for the game to attract scripting-affine users, but the game itself is still fully designed by employed designers.
Another way how Open Innovation is partly used in today's games is by listening to the needs of the player community. If many players complain about a feature in the game or create third party tools to add a specific feature that they feel is missing in the game, some companies use this to change the feature which was complained about or add the feature which was emulated with third party tools. This can be done by releasing a new patch or add-on or just implementing these features into the next similar game they create.
Companies which use the ideas of their players, however, still design and implement these ideas themselves. In contrast, this thesis applies a system which allows users to create content to be inserted into the game *procedurally*, so no developers are required to implement the players' ideas. A high percentage of the game's content will be created, rated, and updated by the user, so full advantage can be taken from the Open Innovation concept.

1.2 Summary

In this thesis a Tower Defense game called *YouTD* will be created which will make extensive use of the Open Innovation concept. It will be created as a modification for *Warcraft 3: The Frozen Throne*.
The success of the game will be evaluated to check if the concept of Open Innovation is viable in game design.
Chapter 2 explains the basic terms and concepts of Tower Defenses as an entry point to this genre. It discusses features which are included in many Tower

Defenses and their influence on gameplay. Thereinafter, the features chosen for YouTD and the motivation to choose these very features will be covered.
Chapter 3 discusses the choice for the platform Warcraft 3 and its suitability to create modifications.
In the following, Chapter 4 provides an insight into Warcraft 3's editor, which allows terrain creation and scripting of arbitrary game concepts. Chapter 5 further discusses the scripting possible with Warcraft 3.
While Chapters 3 to 5 analyze the platform chosen and its "scriptability", Chapter 6 and 7 covers the actual conception and development of YouTD and thus the main effort of this thesis.
After the explanation of the development process, Chapter 8 evaluates the results of the game's release. The amount and quality of the user-submitted content and the general success of the game is discussed in order to prove that Open Innovation in game design is a viable concept and accepted by the users. Finally, a forecast for possible future research in the field of Open Innovation in game design is given.

Chapter 2

Tower Defenses and Their Features

2.1 The Tower Defense Genre

This section explains the desired genre for the game: the *Tower Defense* (TD) genre. Since very special concepts and terms have risen from this genre, the terms used in the following parts of the thesis are defined here. Afterwards, a few examples of current Tower Defenses are given.

The basic idea behind this genre is that the player or a team of players builds *towers* to shoot enemies. A tower is an immovable structure which can be built or otherwise created by the players and has weapons to shoot at enemies. Most towers only shoot at close enemies, but there also exist towers which do not shoot, but have other special abilities.
To create a tower a player needs resources. In many Tower Defenses there is only one resource: money or something similar like *gold*. However, there exist also Tower Defenses which require other resources. Money is mostly gained by killing enemies. Money gained this way is called *bounty*. There may be other sources of money like a specific amount players receive upon completing a level often called *income*.
Towers cannot shoot at infinite range, but have a maximum attack range which is often only called *range*. Towers can have different reload times. This reload time is called *cooldown*. The reciprocal value of the cooldown is called *attack speed*. I.e., the higher the attack speed of a tower, the lower the cooldown between its shots.
The main term which combines towers and enemies is *unit*. So a unit is basically an object on the battlefield having influence in combat, either as target or as attacker. Opponent units are often called *creeps*. Those creeps spawn at specific locations called *spawns* and try to reach a specific location called goal or *finish*. So, usually, creeps *do not attack the towers*, they just run past them and try to reach the finish alive. On their way to the finish they usually take the shortest possible path. The

path the creeps use is called *lane*.
Alternatively, creeps, instead of walking directly to the finish, pass pre-defined *waypoints* on their path. These are locations creeps try to reach one after another before heading for the finish. Figure 2.1 shows the bird's eye view of a lane from the Tower Defense *eeve! TD*[16], where creeps head for five waypoints before they ultimately walk to the finish. This indirection will give players more strategical possibilities than creeps which head directly to the finish.

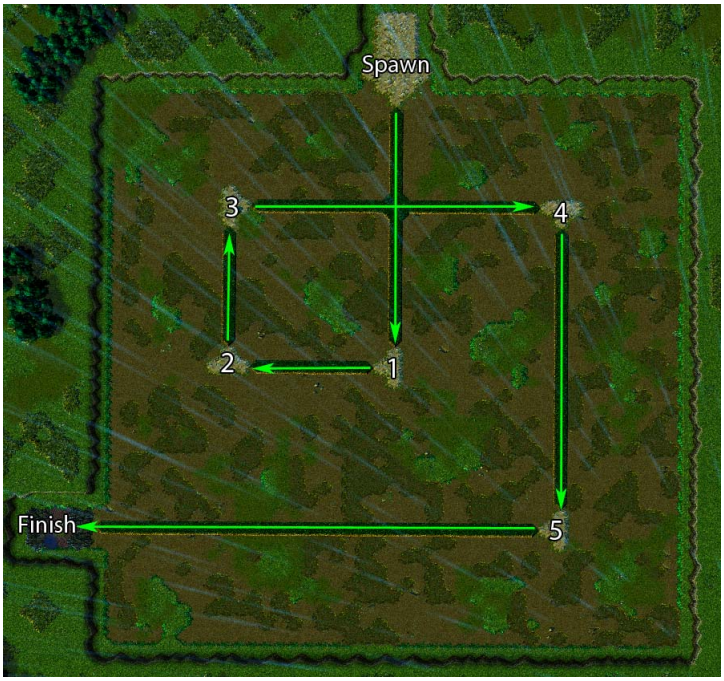
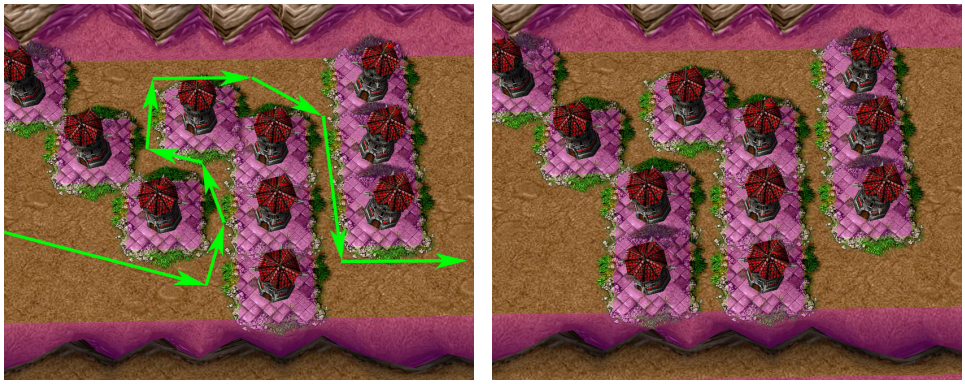


Figure 2.1: *Waypoints in eeve! TD*

In most Tower Defenses, a specific amount of creeps spawns together. Such a group is called *level* or *wave*. After all creeps of a level have been killed or after a specific amount of time has expired, the next level starts. Each new level usually spawns stronger enemies, thus increasing the difficulty with each new level.
Enemies have a specific amount of vital force which is called *hitpoints*. Towers deal a specific amount of damage per shot. If the damage is not modified by effects like *armor*, an attack decreases hitpoints equal to the tower's damage value. If a creep's hitpoints drop to zero, the creep dies.
Players also have a specific amount of vital force which is usually called *lives*, *chances*, or *lifecount*. If an enemy manages to reach the finish, it will decrease the lifecount of the player by a specific value. If the player's lifecount drops down to zero, the player has lost. Not being able to kill a creep and letting it pass to the finish is also called *leaking*.



(a) Valid maze (b) Block

Figure 2.2: Mazing Tower Defenses

A player wins by surviving all levels of a Tower Defense. However, there also exist unbeatable Tower Defenses with an infinite number of levels. In these games, the main goal for players is to reach a level as high as possible or to obtain a good score.

Tower Defenses can be divided into two subcategories: *mazing* and *non-mazing* Tower Defenses. In mazing Tower Defenses, players can build towers directly onto the creeps' paths and thus build a maze of towers which the creeps must pass. The longer a player's maze is, the more time the towers will have to kill the creeps, who need more time for crossing the maze, respectively.

Of course, players must not build a wall of towers without gaps for a creep to pass, so-called *blocking*. This is either forbidden by default, or the creeps will start attacking and destroying towers as soon as a player is blocking.

Figure 2.2(a) shows a valid maze of towers. The areas which are not walkable due to towers or the borders are highlighted in pink. The green arrows show the path the creeps will take, assuming the spawn is at the left and the finish is at the right. In Figure 2.2(b) another tower has been placed in the lower left corner. The creeps can no longer cross the maze, i.e., the player is blocking.

In contrast, in a non-mazing Tower Defense the path creeps can walk and the area where towers can be built are disjunct, so a player cannot influence the way the creeps will walk. Players who prefer mazing Tower Defenses often argue that the mazing component adds some sort of strategy, because a player can decide the shape of his maze. However, common maze forms have evolved which are most powerful in many Tower Defenses, so the strategy component degenerates to building the same maze all over again in every Tower Defense.

As far as the programming effort is concerned, mazing Tower Defenses are more complex, since they require a good pathfinding algorithm to lead the creeps through the maze and to detect blocking. Non-mazing Tower Defenses, in contrast, can contain hard-coded paths the creeps will take.

Towers pick their targets and shoot at them autonomously. A target acquisition algorithm is used to determine which creep a tower will shoot at, if there is more than one in its range. Common target acquisition algorithms are:

- Pick the creep which entered the tower range first
- Pick the creep which entered the tower range last
- Pick the nearest creep
- Pick the farthest creep in attack range
- Pick the creep with the lowest amount of hitpoints

Besides the algorithm used, another parameter for target acquisition is the use of *target locking*. Target locking means that after a target has been acquired, a tower keeps this target as long as it is valid (i.e. not dead or out of range) and only runs its acquisition algorithm again if the target becomes invalid. Not using target locking would make the tower apply the target acquisition algorithm before each shot.
In many Tower Defenses, players can also control target acquisition by selecting a set of towers and ordering them which target to shoot at, which is called *microing*[1]. Even if microing is allowed, the main "work" for a player is to decide where to build his towers and which towers to build when. Tower Defenses do not require fast reflexes, but rather a long term building strategy.
Of course, the above-mentioned characteristics are only the ones which are used by the majority of Tower Defenses. Each Tower Defense has its own set of rules, which may differ from the ones mentioned. For example, in some Tower Defenses, creeps who reach the finish get warped back to the start and will walk the lane again. Another variation is that creeps do not head for a finish, but walk around in a cyclic path. The player loses if more than a specific amount of creeps is in the path at the same time.
There are also games which can be categorized as a Tower Defense combined with some other genre. For example, games exist where players have to build towers *and* spawn creeps at their enemy's position. These games are called *Tower Wars* due to their competitive nature.

There are thousands of Tower Defenses created for Warcraft 3. However, most of them are created by unexperienced creators with just a few hours of effort and thus are not worth being played. Only about 10 to 20 Tower Defenses are played often and appreciated by the community.
To name a few prominent ones, *Element TD* by *E. "Karawasa" Hatampour* [17] currently is the most played Tower Defense for Warcraft 3. It was also copied as a browser game [18] which has less features than the original. Only the theme and the level design were copied.

[1] A short form of *micro managing*

Gem TD by *Brian "Bryvx" K.* [19] also is a very prominent Tower Defense for Warcraft 3 which introduces new techniques for building towers. It was accurately rebuilt as a browser game [20].
Wintermaul [21] by *Duke Wintermaul* was one of the first Tower Defenses for Warcraft 3. It is not being played anymore since it lacks many features compared to today's Tower Defenses. However, since its source code was not protected, it was copied by many people, so its skeleton, which can be recognized by its characteristical environment design, can be found in a great part of newly released Tower Defenses.
Even if most Tower Defenses are either Warcraft 3 modifications or browser games, the company *Hidden Path Entertainment* created a standalone 3D Tower Defense for PC and *XBox 360* called *Defense Grid: The Awakening* [22].

2.2 Special Abilities

All a player has to do in a Tower Defense is to build towers. If all towers shot at the enemy with only different damage, range, and attack speed, Tower Defenses would be very boring, since the player's choice between towers would not matter much. So in almost every Tower Defense, some special abilities exist which distinguish the towers. Often, some of the creeps have special abilities, too.
This section explains the concepts which are being used by many Tower Defenses, to spice up the game and create diversity among the towers available.

- **Types of Armor and Damage**
  Almost every Tower Defense provides for different types of damage a tower can inflict and different types of armor creeps can carry. Each damage type deals a specific percentage of bonus damage against the different armor types and is thus strong or weak against a specific armor type. Often, a *Rock-Paper-Scissors* system is used, i.e. each damage type is strong against one or more armor types and weak versus others.

  For example, a fantasy Tower Defense could contain the armor types "leather" and "plate mail" and the damage types "thrust" inflicted by weapons like spears and "crush" inflicted by weapons like hammers. While thrust could have bonus damage against leather armor because it pierces it, leather armor could in return be more resistant to crushing weapons, since it damps down their force. Of course, Tower Defenses usually contain more armor and damage types and the bonuses are not always backed up by logical reasoning, in contrast to this example. The armor types often do not even resemble real armors but are named completely different. For example, there are games which just use colors as armor and damage types. While towers of a specific color deal low damage to armor of the same color, they deal increased damage to armor of the complementary color.

- **Armor Value**
  In addition to different armor types, creeps often have an *armor value* or *armor level*. The higher an armor level is, the more incoming damage is absorbed by the armor. This grants possibilities for special towers: While *armor piercing* towers could just ignore the armor values and always deal full damage, other towers could decrease the armor value of creeps they attack for a while.

- **Multiple Attacks**
  Instead of attacking only one foe at once, there can be different types of towers with multiple attacks. A tower can use projectiles that explode upon impact, thus hurting enemies in the vicinity of the impact as well. This is often called *splash damage*. Another concept would be a tower which shoots many projectiles simultaneously, or a tower whose shots ricochet from enemy to enemy.

- **Passive Abilities**
  Passive abilities do not have to be activated separately, they are "on" by default. The possible concepts for passive abilities are countless, and many Tower Defenses use several passive abilities. Examples for passive abilities could be:
  Evasion: The creep evades 50% of the attacks targeting him.
  Critical strike: A tower has the chance to cause a multiple of its usual damage on each hit.
  Poisoned weapons: Each attack of a tower deals additional *damage over time* (DoT).

- **Active Abilities (Spells)**
  In contrast to passive abilities, active ones must be activated explicitly, e.g., by pressing an activation button. In fantasy settings, these are often considered spells which are cast when the button is pressed.

  A tower can also use an active ability without the player's command. This is called *auto casting*. Auto casting is often used in Tower Defenses, because explicitly activating abilities is considered microing, whereas towers should usually do their "work" autonomously.

- **Buffs**
  A *buff* is a concept not only used by Tower Defenses, but also by many other game genres. A buff is basically an (often magical) effect on a unit altering it in some way. The unit affected by the buff is considered *buffed*. An example for a buff would be an "armor value reduction" buff which reduces a creep's armor level as long as it is buffed with it.

  Buffs often have a visible effect on the buffed unit so players can see that the buff is on the unit. An example could be a "cold buff" reducing the moving

speed of an enemy. This buff could slightly tint the enemy blue or emit ice particles so players can see which units are affected.

Buffs can have almost arbitrary effects on units: For example, a tower could cast a positive buff spell onto other towers in its surrounding. This buff could for example increase the damage the tower causes or its attack speed. Negative buffs against creeps could slow down the creeps like the mentioned cold buff, reduce their armor, or deal damage over time. A more advanced buff concept could react to certain events, e.g., a buff which, if the buffed creep dies, makes it explode and afflicts damage to surrounding creeps.

- **Auras**
  The aura is also a concept that is used not only by Tower Defenses. An aura is basically a passive ability of a unit, applying a specific buff onto all units that come into a specific range (*aura range*) of the unit having the aura. An example could be a "frost aura" applying the above-mentioned "cold buff" onto each creep which comes too close to the tower emitting it. The buff an aura grants lasts as long as the unit stays within the range of the aura-emitting unit. As soon as the buffed unit walks out of the aura range, the aura buff is removed.

- **Items**
  Towers can have an inventory and carry items. Items can be bought or dropped by enemies upon their death. As long as a tower carries an item, it grants some bonuses or abilities to it, like a buff. Items add a new level of strategy to the game, because the player has to decide which item to allocate to which tower. Some items harmonize well with certain towers and a good player should realize this and allocate the item to this very tower.

  The number of items a tower can carry is limited. Otherwise a player could give all items to his strongest tower to make it even mightier. There may be other limitations, like specific types of towers being unable to carry specific kinds of items.

- **Tower Experience**
  Some Tower Defenses make the towers gain experience when killing enemies. When sufficient experience is gathered, the tower will reach a new level[2] and become stronger. This adds an aspect to these games which is usually found only in Role Playing Games. Often, only the tower which carries out the lethal shot gains experience, so players can try to make one tower always carry out this shot by microing it, to accumulate experience on it and make it exceptionally strong.

With these concepts, very diverse types of towers are possible, and the decisions of a player for one tower or another can strongly influence the result of the game. YouTD will use all of these features to achieve best results.

[2] Not to be mixed up with the levels of creeps that a player faces

2.3 YouTD and its Features

After the previous sections stated which features are commonly found in Tower Defenses, this section shows the design decisions and features included in YouTD and their expected influence on gameplay is discussed.

The first design decision which had to be made was which pieces of content will be user-designed. *Towers* and *items* were chosen, because they have the biggest influence on gameplay and the game gets better with more different towers and items, increasing the range of possibilities a player has. Other candidates for user-designed content would have been the creeps or at least their abilities. However, since creep abilities have less influence on gameplay, this option was dismissed.

When creating the environment, a choice must be made between a mazing or non-mazing Tower Defense. Since both types have advantages and drawbacks, both are good candidates, and non-mazing was chosen randomly.

Concerning damage and armor types, four armor and damage types were chosen to form a rock-paper-scissors-like system: The armor types are *Sol, Hel, Lua* and *Myt*[3], the attack types are *Energy, Decay, Elemental* and *Physical*. In addition, three other special damage types are introduced: *"Essence" damage* causes equal damage to all armor types, *Magic damage* deals 150% to all armor types, thus being better. However, some creeps are immune to magic and cannot be hit by magic damage towers as a drawback. The last special damage type is *Spell*, which is used by many special abilities. It deals less damage to each armor type but, in contrast to all other damage types, is not reduced by armor value. As for armor types, the four standard rock-paper-scissors armor types are backed up by a special armor type called *"Sif"*, which suffers reduced damage from all attack types except Essence. Table 2.1 shows the damage percentage the different attack types deal to creeps with the respective armor. The four rows and columns in the upper left form the rock-paper-scissors system.

Table 2.1: Damage percentages of armor and damage types

	Lua	Sol	Hel	Myt	Sif
Physical	180%	120%	90%	60%	40%
Elemental	120%	90%	60%	180%	40%
Energy	90%	60%	180%	120%	40%
Decay	60%	180%	120%	90%	40%
Essence	100%	100%	100%	100%	100%
Magic	150%	150%	150%	150%	40%
Spell	100%	100%	100%	100%	40%

[3]The names are borrowed from mythological forces mentioned by different ancient cultures

Such a system with rock-paper-scissors elements and special damage and armor types spices up the game significantly, because the player must consider which towers with which damage types to build to beat the upcoming levels. Since every damage type has drawbacks, the player cannot rely on a single type but must find an optimal mix of damage types.
No precise information can be provided about the special abilities of towers and items, since they are user-defined and thus beyond the author's influence. However, the overlay engine provided allows every imaginable type of special ability to be created easily.
Since it is anticipated that many types of towers will be created, a system to categorize and sort tower types has to be applied. Otherwise the player would find himself faced by a wall of hundreds of possible types of towers and have a hard time deciding which type to build. The idea to reduce this huge amount is to present only a limited subset of types of towers to the players in the beginning of the game. The number of available types will constantly grow during the game, giving the player time to learn about the newly available ones.
Different Tower Defenses use different systems to reduce the amount of available towers in order not to overburden the player. The most frequently used system is to divide the towers into *races*. At the start of the game, the player chooses one race and can only build towers of this race.
YouTD will use a similar system with some additions. Towers are divided into seven categories called *elements*, namely, *Storm, Astral, Darkness, Nature, Fire, Ice,* and *Iron*. The elements represent ancient forces a player can control. Each of these elements has its own style, and the users creating towers are advised to make their tower fit into this style. For example, Ice uses snowy and icy models as towers and abilities with cold effects like the frost aura mentioned earlier.
During the game, players receive a number of *force points* which are a currency to gain elements. Players can pay force points to advance in an element of their choice. Each element has 15 levels which a player can "buy". The more points a player has with respect to an element, the mightier towers of this element he can build. So the elements act as races, but a player does not have to decide about the race at the beginning and can mix it with others, or change the direction and focus on another element as the game is in progress.
In addition, towers are divided into four different *rarity grades*, namely, *common, uncommon, rare,* and *unique* (rarity increases from left to right). The rarer a tower is, the mightier and more special its abilities can be. As explained before, only a growing subset of all tower types is presented to the user and can be bought. New tower types are added to this set randomly, with rare towers appearing less often than common ones.

Concerning the levels of enemies a player must face, high diversity should be reached. If the levels are always the same, with only the enemies getting a bit stronger in each level, the player will get bored very fast. Therefore, different types of creeps are created which differ not only in their appearance, but also have

completely different gameplay influence.
The first way to achieve this, which is commonly used by Tower Defenses, is creating enemies of different categories. The first and most common category is simply called *normal creeps* which are 10 ordinary enemies per level. Other special categories are *boss* levels, where only one very strong enemy spawns, *air creeps* that spawn in a pack of five per level and take the direct path to the finish as they are flying, and *mass creeps* which are 20 weak creeps that spawn closely together. These categories will have influence on the towers a player must build to be successful: Against a boss, towers with splash damage are useless, because every tower hits the boss directly. Against mass levels, in contrast, splash damage towers are very useful, because they will hit many enemies at once.
Next, the different creep levels have different armor types and *races*[4]. While armor types modify the damage received from different damage types, races categorize the creeps into *humanoid, orc, undead, magical,* and *natural* creeps. Although this categorization has no direct influence on gameplay, there may be special towers or items which do bonus damage, or other effects against specific races. For example, there could be a "holy tower" which deals bonus damage against undead units like skeletons.
As a last feature that makes each creep level very special, creeps, just like towers, may have special abilities. As an example, some creeps are immune to magic, thus being invulnerable to Spell and Magic damage. Other creeps evade a percentage of attacks or run faster than usual enemies. Around 30 different special abilities for creeps are being used.
To summarize the creeps' attributes: creeps have different armor types, belong to different races and categories and may have different special abilities.
To make matches of YouTD vary from each other, these attributes are chosen randomly for each level, so a player must adapt to the enemies he faces. Without randomization, a player could develop a good strategy which always works and use it over and over again, thus getting bored.
With randomization, no perfect strategy template exists; the strategy has to be reconsidered each time a new level is faced. Thus, the player is kept busy developing strategies.
To allow players to prepare themselves for the upcoming enemies, an ingame scoreboard is created in the player's Heads Up Display (HUD) where he can see the next six levels that will follow after the current one, allowing him to adapt to those levels.
Figure 2.3 shows this scoreboard. In the first column, the level number is indicated, followed by the category in the second column (called size), the race in the third column, the armor type in the fourth, and special abilities in the last column, so all randomized aspects about creeps are displayed to allow the player to consider every one of them when planning his strategy. Beneath the six upcoming levels, additional interesting information is displayed to the player, e.g., his scores, re-

[4]Not to be mixed up with player races represented by elements

maining lifecount, and game time.

Figure 2.3: *The scoreboard of YouTD*

The game difficulty should be adjustable so that new players can choose a low difficulty and thus have fun even if their play is suboptimal. For advanced players higher difficulties should provide a challenge. So at the beginning of the game, the first player can choose between 4 difficulty levels. In accordance with the chosen difficulty, the hitpoints and armor value of the creeps will be adjusted.
Long term motivation has a big influence onto the popularity of a game. Since a Tower Defense usually gets boring as soon as a player has beaten all levels, features have to be included which motivate the player to play another match even if he has already won the Tower Defense once. An approach for this is the use of different elements. If a player has beaten the Tower Defense with one element, he can still try other elements or combinations of elements. In addition, the randomly chosen enemies make matches differ from each other and thus benefit *replayability*[5].
In addition to the element and creep randomization, YouTD will have an infinite number of levels. Of course, the levels become harder and harder, so infinite levels do not lead to an infinitely long match time. A player cannot win the Tower Defense completely, he can only get as far as he survives. This will also benefit the replayability, because players can try to reach a higher level every time they play the game.
To motivate the player even more, a *score and experience (XP) system* is used. *Score* is a counter for points that increases each level. The amount of points gained per level is proportional to the difficulty level chosen at the start of the game, so

[5]Replayability is a neologism. A good replayability means a game can be played often before it gets boring

players playing a higher difficulty will receive more points than players playing on a low difficulty level. After a game, the replay can be saved and uploaded onto the web page (Warcraft 3 allows to save replays of played matches, which can either be watched or parsed to retrieve information like the player score). The score of all players participating in the game will be included into a database. Players can check the best scores ever gained at the *Hall of Fame* on the web site. Since the database also saves the uploaded replay, players can download and watch those replays to check the strategies which led to the best scores, and are thus motivated to play the game again, trying out the strategy they just saw.
The XP system, in contrast, allows the players to accumulate their score over all matches they play. This accumulated value is called a player's *experience*. This will be a motivation to play the game more often, because the achieved score is not lost after a game and can be reused in further matches. Gathered experience can be used at the start of a match to buy small advantages like +10% to all tower damage. The bonuses buyable by experience should be small, so players with much experience do not have a big advantage over new players. If the advantage was too big, new players would be demotivated because they know that they cannot compete with players with high experience. So the bonuses are a small piece of candy for players who play the game often.
Such bonuses should scale logarithmically with respect to experience so some advantages can be bought even with a low level of experience, but players with a very high level of experience will only be able to buy a few more advantages, not granting a very big advantage to "hardcore" players. This way, players will be quickly motivated by receiving some bonuses already on their second or third match. In contrast, if a player has to wait a dozen matches before receiving any bonuses, many less frequently playing users will be disappointed.

Even if the mechanisms shown are not directly linked to the Open Innovation concept, they are still very important for it indirectly: Open Innovation only works if enough people play and like the game. Hardly any player will create content for a game which he considers not worth playing. Since there are thousands of Tower Defenses for Warcraft 3, YouTD must stand out in this crowd to attract enough players and thus content-creators. Chapter 8.3 will show if these features were sufficient to make the game successful when compared to other Tower Defenses.

Chapter 3

About the Chosen Platform

Since creating a whole game engine including networking and graphics would exceed the scope of this thesis, a game that can be modified easily is used as a base for the Tower Defense. This game is *Warcraft 3: Reign of Chaos* [2] including the expansion *Warcraft 3: The Frozen Throne* [3] created by *Blizzard Entertainment*. The basis game was released in 2002, the expansion in 2003. Even though the engine is more than seven years old, it is still one of the most frequently used games for modifications, because it includes a very mighty editor called *World Editor*, a good networking platform, and a sophisticated 3D engine.
The basic format for an own game created as modification for Warcraft 3 is the *map*. A map is basically a terrain environment wherein Warcraft 3 matches with the original Warcraft 3 rules can be played. However, since arbitrary script code to change the game rules can be inserted into such a map, it can become a completely unique game.
The game itself belongs to the *Real Time Strategy* genre. In this genre, players play against each other or a computer-controlled AI. They build a base with different buildings like towers protecting the base against attacks, or buildings training units which can then be used to attack the enemy base. Mission objectives can be set like rescuing specific units or surviving for a certain time. A game is won by achieving these objectives. However, especially in multiplayer, the only objective often is to kill all enemies.
Blizzard Entertainment has also created another Real Time Strategy game called *Starcraft* [12] which was released in 1998 but is still played frequently, especially in South Korea, where Starcraft matches are telecasted and professional players are as prominent as soccer players in other countries. Starcraft was the predecessor of Warcraft 3 and already had a mighty editor. Using this editor, some fans created the first Tower Defenses, thereby giving birth to this genre. The editor was improved in Warcraft 3, affording map makers much more possibilities to completely alter gameplay. *Starcraft 2* [23] has already been announced for 2010 as the successor of Warcraft 3 and promises an even more sophisticated editor which, for example, provides for *first person camera* and *mouse looking*[1], allowing to create an ego-

[1]Mouse looking is a control technique often used in ego-shooters. Here, the players have no

shooter with its engine [24].
Warcraft 3 is set in a medieval fantasy universe called Azeroth, with well-known fantasy creatures like dwarves, elves, gnomes, orcs, humans, undeads, and demons. The game offers a vast variety of fantasy models from the above-mentioned and other races, which can be used for self-made games.

3.1 The Graphics Engine

Warcraft 3 contains a powerful and easy-to-use 3D graphics engine. The battlefield is based on a heightmap up to an extent of 512x512 tiles[2], allowing very big scenarios.
Each tile can be textured with predefined ground textures. There are around 150 predefined ground textures covering almost all common terrains like different types of grass, dirt, and rock textures. More exotic textures like ice, snow, lava, paved road, and fantasy-themed textures are available, too. In addition, players can overwrite textures with self-painted ones, allowing for any desired texture. When two textures collide, they are blended, using predefined blending edge textures. Alpha blending cannot be used, but is promised for Starcraft 2 [24].
Water fields can be applied on parts of the map. Water is lucent if shallow, otherwise opaque. A predefined water texture is used which cannot be changed without small hacks, and predefined wave models are periodically displayed on shores. The water can be colored by the user with an arbitrary RGB color.
Additionally, fog and weather effects can be applied. Fog can be colored arbitrarily, its density can be set, and its start distance and end distance can be set. Objects closer to the near clipping plane than the start distance are not fogged. Objects farther than the end distance are fogged with full density. Inbetween start and end, the fog density is interpolated, using either linear or exponential interpolation.
Weather effects like rain, snow, storm, sun-, and moonlight can be chosen from a small predefined set. Weather effects can either be global or local. They can even be scripted to change in the course of the game.
Environment models like rocks, trees, plants, and buildings can be placed onto the heightmap. These environmental objects are called *doodads* in the editor. For those doodads and for units and buildings, the *mdx* file format is used. This is a propretiary format developed by Blizzard, but it was reverse engineered and is publicly available [25] and tools exist to convert common model formats like the ones of 3DStudio or GMax into mdx models. Since all kinds of files can be imported into maps, arbitrary self-created 3D models can be used in modifications. Extensive databases of freely available mdx models to be used in maps exist on the web.
Named animations can be included in mdx models. The animation techniques are powerful: particles can be emitted, new triangles can be added or removed

mouse cursor, in contrast to strategy games. Instead, moving the mouse will rotate the field of view, i.e., the player virtually "turns his head" by moving the mouse.

[2] a tile is an element of the heightmap which represents the texture around one height value

during an animation, and the model can be transformed using bones and Euclidean transformations.

Predefined unit models often have common animations like *stand*, *walk*, *attack*, and *spell* which are used automatically in-game if a unit stands, walks, attacks, or casts a spell, respectively.

The mdx file format interpreted by the Warcraft 3 engine allows for special features like transparency, but no correct alpha blending is used on transparent parts, leading to small errors when several transparent surfaces overlap. However, these effects are not very disturbing, since the transparent parts are often small and overlaps are uncommon.

The format allows to define *attachment points* in models. These are points where other models can be attached. Most predefined unit models in Warcraft 3 have common attachment points like *overhead*, *head*, *left/right hand*, *chest*, and *origin*. This allows a spell for example to create a halo above a unit's head by attaching the halo model to the overhead attachment point or giving an unarmed unit model a sword by attaching the sword model to the right hand attachment point.

For textures, the special file format blp (*"Blizzard Picture"*) [26] is used. It allows jpeg compression or color palettes. The jpeg-compressed versions contains mipmap pyramides with a maximum of 16 levels. They are automatically used in the game for anti-aliasing.

So basically, Warcraft allows arbitrary models and textures to be used for creating various unique scenarios. However, if own models are imported, they have to be stored in the map itself. No packaging mechanism is used; e.g, if ten different maps use the same user-defined model, it has to be inserted into every single one of them. If a map is played in multiplayer, it can be no larger than eight megabytes. Even staying shortly below this limit is discouraged, as players who do not have the map yet have to download it from people who have it upon joining a game which is using the map. Since most players have asynchronous DSL with low upload ratio, transferring a map between users will take a long time. Most players will leave the game if they have to download for several minutes. So it is encouraged to keep a map below one megabyte. This leaves room only for some imported models with a low number of polygons. Only single player maps can be arbitrarily big.

Since YouTD is a multiplayer Tower Defense, own imported models will not be allowed for towers. Instead, the game will allow users to create new models from predefined Warcraft 3 models. They will be assembled in the game using the script language. The results of these combined models are shown in later chapters. Since Warcraft 3 has a big stock of predefined models, disallowing imported models is not as limiting as it may seem on first sight.

Figure 3.1 shows screenshots of beautiful environments created with the World Editor by a talented artist called *Void*. Even if some models were imported for these screenshots, most models are Warcraft 3 standard models. This shows which beautiful worlds can be created with a seven-year-old engine and a limited stock of predefined models.

Figure 3.1: Art created with the World Editor [27, 28, 29]

3.2 Useful Game Features

Warcraft 3 is a very viable base for making a Tower Defense since it already contains many features which would need much implementation effort if they had to be created by the developer of the Tower Defense. This section sums up the most important features Warcraft 3 provides.
First, the game uses a robust pathfinding algorithm to lead units to their destination even if the path is very complex. Even if the actual algorithm used is unknown, testing showed that it is capable of handling hundreds of units without decreasing the frame rate. This provides a good base for mazing Tower Defenses, since no pathfinding algorithm has to be implemented.
In addition, the target acquisition and shooting is also managed by the engine, so no own target acquisition algorithm has to be developed. Warcraft 3 uses a "pick nearest target" algorithm with target locking (cf. page 10).
The engine allows to create multiplayer games with up to 12 human players. Besides normal LAN games, Warcraft 3 offers networking over Internet using Blizzard's platform *Battle.net* [30]. Using this platform, games can be hosted and then instantly appear in a game list. This list can be viewed by every player, and a game can be joined. If a player does not have a map which is played in the game he joins, the map will automatically be downloaded from other clients who have it. This makes maps spread very fast without the need to advertise them: If people like them, they will be hosted, thus attracting new players.
The game also provides other small beneficial systems, like an inventory system allowing to give items to units to possibly alter their stats or grant other bonuses. For example, a sword in the inventory of a warrior would increase his attack damage.
Warcraft 3 provides three built-in types of resources, namely *gold*, *lumber*, and *food*. However, the appearance and name of the resources can be changed, so the three resources can be used for everything imaginable. Of course, more resources can be added, but only these three are natively displayed in the game's HUD. YouTD will use gold as currency and lumber as force points (cf. page 15).
These features release the creator from the biggest parts of implementational work. A creator does not have to care about networking, pathfinding, target selection, and shooting, but he can concentrate only on the gameplay. This also is the reason why the Tower Defense genre has risen from Warcraft 3 and its predecessor Starcraft, and the reason why this game was chosen for the creation of YouTD.
Of course, all these features are useless if the game cannot be scripted appropriately, but Warcraft 3 also performs well in this discipline. It provides an easy-to-use editor with a powerful script language, and many map-making communities have gathered and provide good code resources, models and other content, as well as tutorials and help for new users. The following chapter will cover the features of the editor and its communities. Chapter 5 will explain the script languages and programming paradigms used in the editor.

Chapter 4

The World Editor

The ultimate reason for choosing Warcraft 3 as platform for the game is its editor called *World Editor*, which allows quick creation of high quality games. It allows to design the terrain, the game objects like units, items, and abilities as well as the scripting, which allows arbitrary games to be created using the Warcraft 3 engine.
The World Editor works on Warcraft 3 maps and can alter all parts of them. The main parts of a map are the terrain, which can be modified in the *terrain editor*, data of in-game objects like units, items and abilities, which can be adjusted in the *object editor*, and the own scripted game rules, which can be edited in the *trigger editor*.
The scripting of own game rules is sourced out into the next chapter. This chapter, in contrast, covers the non-scripting work like the creation of the environment and the game objects.

4.1 Terrain Editor

The terrain editor allows to edit the environment, define specific areas to be used by the script, and set starting units onto the map.
However, most highly scripted maps do not use starting units, they create all units using scripts. So for YouTD as well, the terrain editor serves the sole purpose of creating a beautiful environment and defining areas the script will use to create the enemies, the tower builder, and other features like the waypoints the creep will pass before heading for the finish.
When a map is newly created, it consist only of a flat heightmap, textured with one ground texture. The first change which can be made to obtain a diverse environment is to alter the heightmap by raising or lowering terrain. The editor provides some basic tools like heightening or lowering the terrain, making the terrain level even (called “Plateau”), smoothing the terrain or creating random ripples. All of these tools can be applied using a brush of different sizes to alter bigger or smaller regions at once.

Next, the texture laid over the heightmap can be changed per tile. Again, differently sized brushes are available to alter one or more tiles at once.
The creator can choose up to 13 of the 150 available terrain textures in one map. This limit exists because the heightmap with textures is stored in a very compressed format to save storage space. Only four bits are reserved for terrain textures, so a terrain texture palette can have a maximum of 16 entries. Since three values are reserved, only 13 values remain for the use as textures. Of course, besides choosing between the 150 predefined textures, the user can import his own terrain textures, but, the 150 predefined textures often provide enough options for most settings and no importing is required.
Figure 4.1 shows a hill created by using the heightening tool on the left and the plateau tool on the right side. The ground textures were altered to create a paved road around the hill, grass on top of it and rocks on the steep edges.

Figure 4.1: *A simple hill created with the World Editor*

Besides the heightmap, cliffs can be placed which are steep ascends with special textures. Since a heightmap cannot have infinite inclination (e.g., a 90 degree wall) the cliffs are workarounds to allow for such walls.
While heightmap changes have no influence on gameplay (even a very steep heightmap does not alter pathability or movement speed of units), cliffs are not walkable by default. However, ramps can be placed to travel between two cliff levels. Additionally, water pools can be inserted when creating lowered cliffs. Different cliff models and textures can be chosen. The available models range from natural cliffs to castle or temple walls.
Figure 4.2 shows examples of cliffs. On the left, different cliff levels were combined. In the middle, two ramps were inserted to make passing between the cliff

levels possible. On the right side, a water pool was created, using cliffs. All cliffs use a natural cliff model.

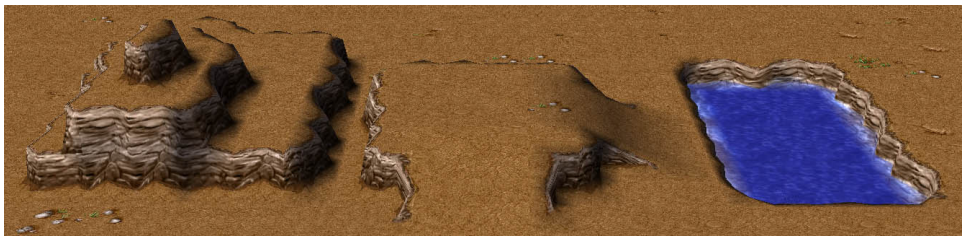


Figure 4.2: *Cliffs in the World Editor*

Next, the terrain editor allows to place environmental models (*doodads*) onto the map, which can also influence gameplay. They can prevent units from crossing them or have other special features. For example, trees can be cut down to obtain lumber in the original game. The creator can choose between plants, rocks, structures, and other special objects. Imported models can be used as well.
Figure 4.3 shows the previously created hill, with plants, trees, and a settlement added. The doodads are placed by picking them from a palette and clicking on the map where the doodad is to be placed. They can be scaled and rotated with short keys to quickly alter their appearance.

Figure 4.3: *The previously created hill with doodads*

While the changes in the heightmap and the doodads create the environment for the game, the terrain editor additionally allows to create units which will be in the game at the beginning and further allows to create so-called *rects*[1]. Rects are definable rectangular areas, which can later be used in the script. A Tower Defense could use such a rect for the creep spawn and create the creeps in the middle of the rect.
Figure 4.4 shows different rects on the left and some starting units on the right, both placed in the World Editor.

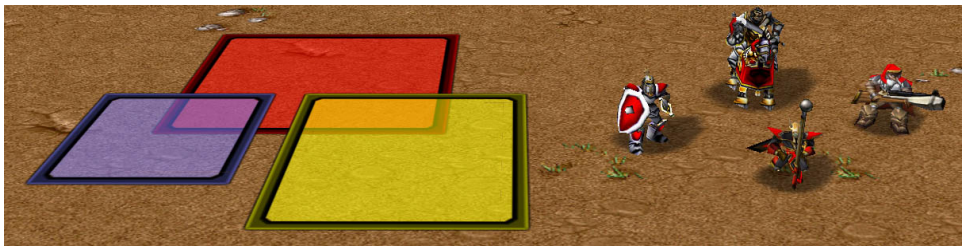


Figure 4.4: *Rects and starting units*

4.2 Object Editor

The object editor allows to define and edit different categories of object like units, doodads, and other features such as items, unit abilities, and buffs (cf. Chapter 2.2). To be more precise, it allows to edit *types* of objects from these categories. Whenever objects in the object editor are mentioned in this section or later on, actually types of objects are meant, not single object instances.
Each unit and doodad placed on the map or created by the script must be of a type defined in the object editor. The type contains various values defining the unit's appearance and its gameplay values. For example, concerning appearance, the model, scaling, and coloring for a unit and the model of its attack projectiles can be set. Concerning gameplay values, the hitpoints of a unit or the amount of damage it will deal in combat, its cooldown, and range can be set in the object editor. In addition, some user interface values like a unit's name and its description can be chosen here.
Figure 4.5 shows a screenshot of the object editor. On the top, tabs for the different categories of objects can be found. The tab currently opened is the one for units. The units defined are shown on the left, and the values of the currently selected unit on the right. The screenshot shows values of the category *Art*, which controls the appearance of the unit. Note the scroll bar on the right. It shows that many changeable values exist and thus a high level of control over the unit's appearance and gameplay can be executed.

[1] Shortform of rectangle

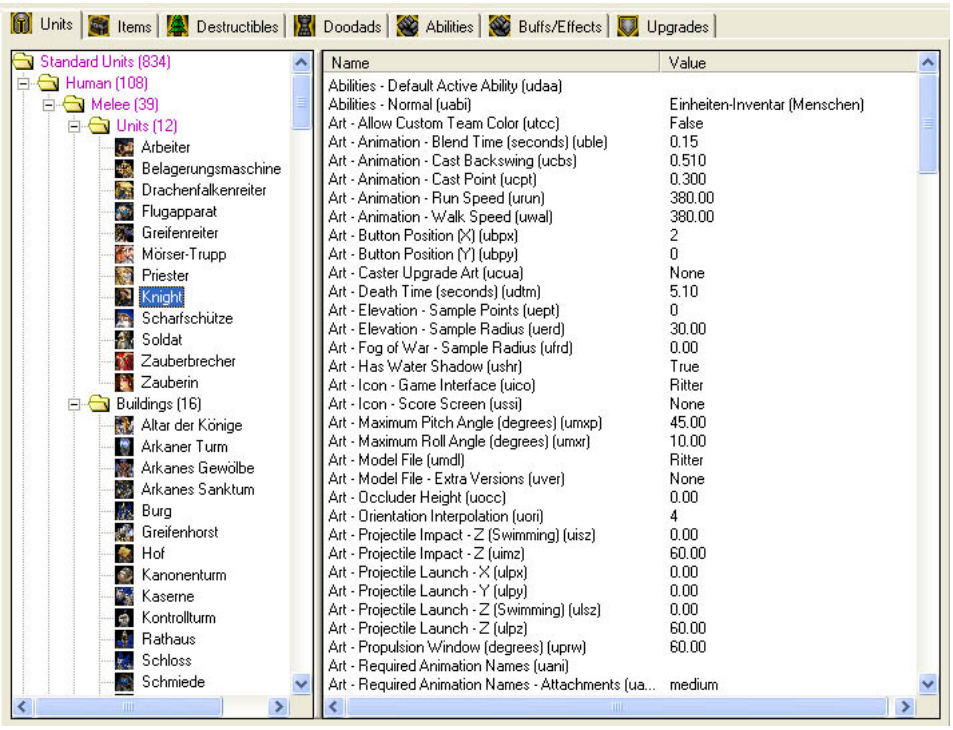


Figure 4.5: *The object editor*

4.3 Other Editors

The World Editor includes some more editors to execute more or less important jobs when creating a map. The most important one is the *trigger editor*, which is used to script the map. Basically, this editor allows to create script sections in the map and enter scripts which alter gameplay. So the trigger editor basically acts as an integrated development environment (IDE) for scripting the map.
When using the Jass NewGen Pack with TESH [31], a non-official extension to the world editor, a user even has advanced IDE features like code completion, syntax highlighting, and syntax folding. In advanced mapping communities the Jass NewGen Pack has therefore become a standard tool and hardly anybody uses the original World Editor alone anymore. Since scripting the map and thus allowing for arbitrary rules makes the World Editor so omnipotent, the next chapter covers working with the trigger editor.
The *Sound Editor* enables the creation of sound variables from imported or predefined mp3 or wave sound files, which can then be used in the trigger editor. A sound variable is a sound file combined with parameters to alter it: It can be pitched, the volume can be adjusted, it can be faded in or out, looped, and it can be set as a 3D-sound with a specific radius. 3D-sounds can be played at specific positions of the map. Only players whose viewport is within the radius of the sound will hear

it, while non-3D-sounds are heard by all players. Finally, there exist further editors like the *AI Editor*, which allows creating tactics for computer players, the *import manager*, which handles sound files, models, and other files imported into the map, and the *campaign editor* to link several maps to one campaign. Since these editors are not used for Tower Defenses, they will not be covered here.

4.4 Communities and Services

Due to the powerful editor and the solid game engine, Warcraft 3 is one of the most widely used games for scripting and creating one's own games and modifications. Many communities have gathered to discuss scripting and upload resources, tutorials, and helpful programs to make scripting for Warcraft 3 easier, faster and more convenient.
One of the most prominent communities is *Warcraft 3 Campaigns* (W3C) found at *www.wc3c.net*. This page features a forum with over 23000 registered users and a large resources and tutorials section. It is the first address for development of Warcraft 3 scripting, and the home of common editing tools like the Jass NewGen Pack and the vJASS language[2]. Users can suggest features for the next release of the vJASS compiler thus taking part in the advancement of the scripting language. Moreover, the resources section contains scripts for basic data structures, like linked list implementations, or complete systems, like a versatile damage detection and reaction system[3]. Warcraft 3 Campaigns is a top address for learning basic and advanced Warcraft 3 scripting, with over 200 tutorials about the World Editor and scripting.
Besides code resources, models, skins, and buttons in Warcraft-readable formats can be downloaded. For example, over 400 models fitting into the Warcraft 3 theme or from different themes like Manga or Science Fiction can be downloaded free of charge.
However, when looking for non-scripting resources like models, *The Hive Workshop* found at *www.hiveworkshop.com* is a bigger address with over 3000 models freely available. With over 53000 registered users, the forum community is more than twice as big as W3C, but W3C is still the home of the most prominent scripters.
There are many more map-making and scripting communities for Warcraft 3, only the most prominent ones were mentioned.
Thus, a big fan base of people scripting and making own games with the Warcraft 3 engine exists who are possible users submitting content for YouTD. These users know the World Editor by heart, so no long tutorials for controlling the editor or using the script language are required, which will speed up the Open Innovation process.

[2] An advanced scripting language for Warcraft 3, see next chapter

[3] Warcraft 3 does not offer a convenient system to react to or modify damage done by units, so such overlay systems were created by users

Chapter 5

Scripting Warcraft 3

The previous chapter provided an overview of the features the World Editor offers. However, without any scripting, users cannot really create their own games within the Warcraft 3 engine. They are bound to Warcraft 3's game concepts; only the environment and values of objects can be changed.
A Tower Defense needs a lot of new rules and thus scripting: The enemies must be spawned and ordered to head to the finish on each level. The kills must be counted, and once all enemies are dead a new level must start after some delay. The player's lives must be counted and decreased if an enemy reaches the finish, and the players must be defeated if their lifecount drops to zero. These examples show only a very small part of the total scope of scripting needed for YouTD.
This chapter shows the concepts which make the World Editor so powerful and allow the creation of unique games using Warcraft 3's engine.

5.1 The Trigger Concept

The basic method to create a piece of code to be executed are triggers[1]. Triggers are data structures which can be registered to react to different game events, and condition functions and action functions can be added.
Whenever the engine creates an event, all triggers which are registered for this event run their condition functions, which have to be functions returning a boolean value. If all condition functions of a trigger return `true`, the action functions of the trigger are executed in the order they were added to the trigger. Even if the engine allows adding as many condition functions and action functions as desired, most triggers have only one condition function and action function which includes all the statements to be executed.
A trigger can register for many different events. Almost anything fires an event in Warcraft 3's engine. This is what makes the engine so useful for scripting. A programmer can react to and thus control almost every event happening during the

[1] for this reason, scripting for Warcraft 3 is often called *triggering*

game. Here are some widely used categories of events triggers can be registered for:

Map initialization: Triggers which register to this event are fired immediately when the map is loaded. Initialization tasks like creating starting units should be executed here.

Timed / Periodic: A trigger can be called periodically or once upon expiry of a specific period of time.

Unit events: This category contains events which fire when something happens to a unit, like when the unit dies, is built, or enters a specific region of the map.

Player events: If a player types a chat message, selects specific units with his mouse cursor, or uses keyboard keys, player events will be triggered.

By using the trigger concept, users can create diverse games. This will be exemplified by showing how the well-known board game *chess* could be implemented for Warcraft 3 by using triggers:
At first, a chessboard environment with black and white ground textures is created and a rect (cf. page 28) is assigned to each chess field. The different chess piece types are created as unit types within the object editor. Their movement speed is set to 0 so they do not move by themselves, since the pieces will be moved by triggers to ensure that only allowed chess moves are made. A targeted ability called "Move Piece" is added to the chess pieces which will be used for moving the piece. Such targeted ability will provide the controlling player of the figure with a button in his user interface whenever he selects the unit. By pressing this button, he will be able to move the piece. Thereinafter, these exemplified triggers are used to implement the chess rules (Note that some special chess rules like *castling* were omitted to keep the example simple):

Trigger 1: Initialization
Event: Map initialization
Condition: None
Action: Create chess pieces and place them on the chess field.

Trigger 2: Chess Piece Movement
Event: A unit uses an ability
Condition: It's the owning player's turn, and the ability being used is the "Move Piece" ability
Action: Check if the target of the ability is a valid chess field for that unit, i.e. whether the piece can move to this position. If not, display a message to the player that the piece cannot move to this field and abort the trigger.
Check if there is a friendly or enemy piece on the target field. If not, move the piece to this field

If there is an enemy, kill the enemy unit and move to the field
If it is a friendly unit, display an error message that a player cannot move onto fields where his own units stand and abort the trigger
Display a message to the non-moving player that it is now his turn and switch to his turn.

Trigger 3: Victory
Event: A unit dies
Condition: Unit type of the unit is "King"
Action: Defeat the player whose King died, display a message showing the game result and end the game.

Trigger 4: Forfeit
Event: A player types a chat message
Condition: The chat message is "forfeit"
Action: Defeat the player entering the chat message, display a win message to the other player telling him that his enemy has forfeited and end the game.

Thus, a game like chess can be created with only a few triggers. Of course, these triggers do not cover all chess rules, and although these triggers seem very simple, the complexity is hidden in the text: Checking if a unit can move to a specific field is non-trivial and requires the implementation of chess rules.
The conditions and actions have to be written in one of the available script languages. For those languages, Warcraft 3 offers a comprehensive application programming interface (API) to access game mechanics. In this example, we would need API function calls for the following things:

- Create a unit, to create the chess pieces.
- Displaying a text message to a player, to display the error or win messages.
- Moving a unit to a target position, to move the figures.
- Killing a unit, if it was captured by an enemy piece by moving onto its square.
- Defeating a player and ending the game.

A problem is that the API is not documented well. There are some small hints for functions in the editor, but no comprehensive documentation exists. However, there exist wikis like the German project mappedia [32] which try to create such documentation for every API function.
The triggers presented in this section were explained informally and cannot be interpreted by the game, of course. The next section will cover the scripting languages which can be used to actually implement the triggers.

5.2 Script Languages: GUI, JASS, vJASS

Warcraft3 provides two ways to create trigger code: The first is using a *Graphical User Interface (GUI)* where the programmer selects the action to be inserted from a drop-down box. If the action is a function which requires parameters, they can also be chosen from a drop-down box. The actions, conditions, and events are named intuitively and inserted into categories to simplify finding a specific action.
Figure 5.1 shows such drop-down box with instructions. The displayed instructions are of the category *unit* and are intended for creating and dealing with units on the battlefield.

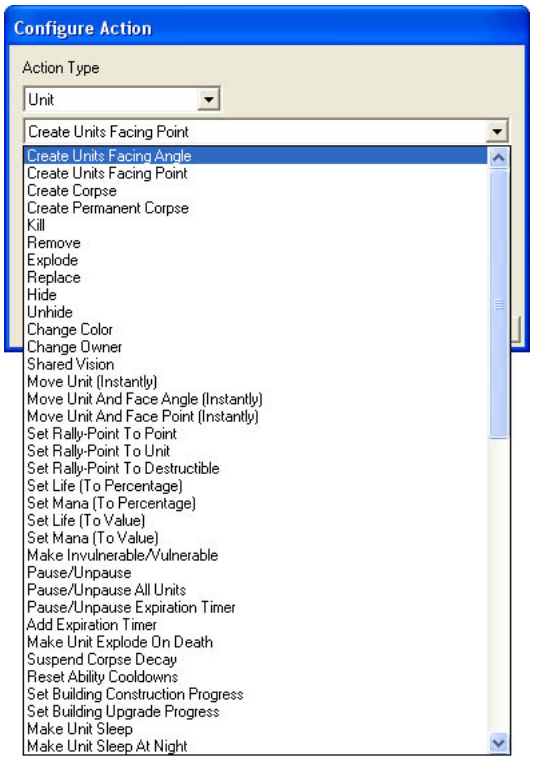


Figure 5.1: *GUI Box to choose actions*

Figure 5.2 shows the victory trigger of the chess game created using the GUI. The trigger fires whenever a unit dies, and if the unit is a king, the owning player will be defeated and the other player will receive a victory message.
GUI triggering was invented for unexperienced map makers, who have no intention to learn a procedural scripting language. Since the majority of Warcraft 3 players are teenagers, Blizzard tried to provide a way to create maps without having to write script code, which would disencourage many teenagers or other players who are not familiar with programming.

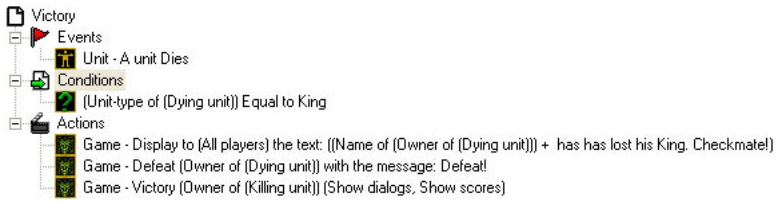


Figure 5.2: *A trigger created using the GUI*

However, in experienced mapping communities and serious projects GUI is not being used for the following reasons:

- Not all functions of the API are available in the GUI. Some are missing in the drop-down box.
- Not all syntax possibilities exist in the GUI. For example, a `while`-loop can only be created using ugly hacks, e.g., by keeping a `for`-loop variable below the limit. In addition, no procedure-local variables can be used, only global variables.
- Writing script code usually is less time-consumptive than constantly choosing entries from drop-down boxes.
- For experienced programmers, writing code is done intuitively so the programmer can focus on what he wants to achieve, not on how to choose the actions to do so.
- The GUI does not allow to define own functions. This is a major flaw which makes the GUI triggers hardly reusable.

The script language which was created by Blizzard Entertainment for Warcraft 3 is called *JASS*. Triggers which were created using the GUI are compiled to JASS code in the background. It is also possible to explicitly convert a GUI trigger to JASS code. Since the possibilities of JASS are a superset of the possibilities when using GUI, converting a JASS trigger back to GUI is impossible.
JASS is a procedural language with syntax comparable to pascal. Its syntax structure is a block of global variables followed by functions which can also contain procedure-local variables.
As to control structures, JASS offers a basic `if elseif else` construct with the common semantics. There exists also one type of loop which always generates an endless loop and a structure similar to `break` used to exit such loops.
Listing 5.1 shows a recursive implementation of the faculty function in JASS to demonstrate how basic calculations are carried out.
Listing 5.2 shows the victory trigger whose GUI version was shown in Figure 5.2 as JASS code. The first function represents the trigger's condition function, which returns a boolean value stating whether the trigger's action is to be executed. The

Listing 5.1: Faculty function in JASS

```
function faculty takes integer input returns integer
    if input < 0 then
        call BJDebugMsg("Negative value handed to faculty function!")
        return 0
    elseif input <= 1 then
        return 1
    else
        return faculty(input-1) * input
    endif
endfunction
```

Listing 5.2: The victory trigger in JASS

```
//Condition
function Trig_Victory_Conditions takes nothing returns boolean
    //Only execute the trigger if the dying unit was the King
    return GetUnitTypeId(GetDyingUnit()) == 'hkni'
endfunction

//Action
function Trig_Victory_Actions takes nothing returns nothing
    local unit dyingUnit = GetDyingUnit()
    local string name = GetPlayerName(GetOwningPlayer(dyingUnit))
    call DisplayTextToForce(GetPlayersAll(), name + " has lost his King.")
    call CustomDefeatBJ(GetOwningPlayer(dyingUnit), "Defeat!")
    call CustomVictoryBJ(GetOwningPlayer(GetKillingUnitBJ()), true, true)
endfunction

//Init
function InitTrig_Victory takes nothing returns nothing
    local trigger v = CreateTrigger()
    call TriggerRegisterAnyUnitEventBJ(v, EVENT_PLAYER_UNIT_DEATH)
    call TriggerAddCondition(v,Condition(function Trig_Victory_Conditions))
    call TriggerAddAction(v, function Trig_Victory_Actions)
endfunction
```

second is the trigger's action function, which will be executed if the condition function returns `true`. The last one is the initialization function to create the trigger, to register it to the event, and to add the action and condition function. The trigger is registered for the "unit-death" event, so it will trigger whenever a unit dies.

As far as data structures are concerned JASS offers usual basic variable types for floating point and integer numbers, boolean values, and strings. Game objects like units or players have their own variable type hierarchy. The root of this hierarchy is the type *handle*, which represents a reference to a game object. JASS has no own garbage collection, so handle types have to be recycled by the programmer.

In addition to the basic types and subtypes of handle, JASS allows onedimensional arrays with some limitations. Arrays always have the size of 8191 entries[2]. They cannot be handed to a function or returned from a function, and no array variable

[2] actually it is 8192, but using the last index creates bugs

can be reassigned (compared to C, these arrays would equal constant pointers). Instead, all declared array variables are implicitly initialized with a new array filled with zeros (or `null`s for handle types and strings).

Along with these limitations, JASS does not offer record types.

To store more than 8192 values with arbitrary indices, JASS offers a `hashtable` datatype, which is a two-way associative hash table. This data structure is quite fast and thus useful for many purposes. Hashing functions for the handle-derived types and strings are included in the API, so every type of variable can be used as a key in hash tables.

As long as the complexity stays low, JASS is a fast tool for creating scripts. However, as soon as the complexity rises or own data structures are involved, JASS becomes very unhandy. Implementing a linked list with JASS would be a cumbersome work, for example.

To break these limitations and to introduce the object-oriented programming (OOP) paradigm to JASS, Víctor "Vexorian" Hugo Solíz Kúncar created an extension language called vJASS [33].

The relationship between vJASS and JASS is similar to the relationship between C++ and C. vJASS Syntax is a superset of JASS, so every JASS trigger is also valid vJASS. Since Warcraft 3 cannot interpret vJASS, this language is compiled to JASS in the background whenever the map is saved.

The main additions of the vJASS language are record types called *structs*. In contrast to structs in C, structs in vJASS can also have methods, data encapsulation, and inheritance, so they are better compared to classes in object-oriented languages. A vJASS struct is compiled to a number of JASS arrays (one for each attribute), so a single struct cannot have more than 8190 instances[3] allocated at the same time.

Even with this limitation, structs allow to create structured code with clean interfaces. This allows to create modular, reusable systems and is thus the first choice for bigger projects, including YouTD and its overlay engine.

Listing 5.3 depicts a struct definition to show the basic syntax and possibilities of vJASS structs. The function names `create` and `onDestroy` are special names which always stand for the constructor and destructor, respectively.

The method `allocate` is a keyword allocating space to an instance and can be compared with `malloc` in C[4]. Note the stub method allowing dynamic binding of methods.

vJASS offers many more syntax extensions besides structs, but the latter do not have so much impact and are used less often[5].

[3] one index (0) is used for the `null` struct

[4] of course, the method of allocation is a totally different one since `allocate` does not really allocate system memory but just an index in the array key space

[5] A full list of extensions and their semantics can be found in the manual which is available at www.wc3c.net/vexorian/jasshelpermanual.html

Listing 5.3: A struct definition in vJASS

```
/*  A block comment, which is not possible in JASS.
    Besides structs and other useful additions,
    vJASS also offers syntactic sugar like this
    (if comments can be considered syntactic sugar)  */

struct AppleTree

    //Static member
    static integer numTrees = 0

    //Normal member
    integer numApples

    //The constructor (a static method)
    static method create takes integer numApples returns AppleTree
        local AppleTree at = AppleTree.allocate()
        set at.numApples = numApples
        set AppleTree.numTrees = AppleTree.numTrees + 1
        return at
    endmethod

    //The destructor
    method onDestroy takes nothing returns nothing
        set AppleTree.numTrees = AppleTree.numTrees - 1
    endmethod

    //A test method
    method dropApple takes nothing returns nothing
        if this.numApples <= 0 then
            call BJDebugMsg("No more apples available on this tree!")
            return
        endif
        set this.numApples = this.numApples - 1
        call BJDebugMsg("Dropped an apple!")
    endmethod

    //Stub methods are equal to virtual methods in C++
    //and can be redefined by structs extending this one
    stub method displayClass takes nothing returns nothing
        call BJDebugMsg("AppleTree")
    endmethod

endstruct
```

To use vJASS along with the World Editor, the vJASS compiler called *JassHelper* [33] has to be injected into the World Editor process. The easiest way to achieve this is using the *Jass NewGen Pack* [31] which includes the latest version of JassHelper, an advanced World Editor, and other improvements like syntax highlighting for vJASS.

Since the verbose syntax of vJASS was disliked by many scripters familiar with C or JAVA, including its creator Vexorian himself, he released another script language called *ZINC* [34] which introduces a C style syntax with OOP additions. This language might be the future of professional Warcraft scripting, but it was not available yet when most parts of this work were implemented. Therefore, it will not be used or explained further.
In addition, a few more non-official script languages for Warcraft 3 exist, e.g., another language with C style syntax called *cJass* [35]. However, they are not widely used.

5.3 Editing maps procedurally using GSL

Since YouTD will include user-created content, a way to procedurally import such content into a map is required. However, injecting something into Warcraft 3 maps is non-trivial, since the map is a packed archive with files in their very own formats. Although the formats were reverse-engineered and thus made available [26], implementing a program which is able to handle them involves a great amount of work.
The author of this thesis already accomplished this, using JAVA, and invented a script language called GSL (Gex's Script Language) and an interpreter therefor called GMSI (Gex's Map Script Interpreter) [36].
GSL is a language with syntax similar to C with some additions of JAVA and Perl syntax. Most important, it contains an API for handling the Warcraft 3 data format and thus allows to write code which reads or writes data from or to a Warcraft 3 map, respectively. Basically, this API allows to read and alter all information of a map procedurally which a human could read and alter using the World Editor.
When the API function for "loading a map" is called, GMSI extracts the data from the map's files and writes it into a GSL struct which can then be altered. Afterwards, this struct instance can be handed back to a "save map" API call which will read the data from the struct and write it back into the different files of the Warcraft 3 map. Thus a programmer only needs the definition of the GSL map struct to find the data from different sections of the map. Since the members of a map struct are named intuitively, altering Warcraft 3 maps can quickly be accomplished by simply altering the respective struct members.
For example, all object editor data is held in the `objects` member. Therefrom, the different objects can be accessed using their editor ID. The object editor fields for such an object can then simply be read or written by inserting their name.

Listing 5.4: Basic map alteration with GSL

```
//Open the map
Map map = loadMap("myMap.w3x", false, false);

//Renaming of the 'hkni' unit
map.objects.hkni.Name = "King";

//Save the altered map
saveMap(map,"myOutputMap.w3x");
```

For example, `map.objects.hkni.Name` accesses the attribute "Name" for the object with the ID "hkni".
Listing 5.4 shows the GSL code for loading a map, setting the name of the unit with the ID "hkni" to "King", and saving it. This job, which would require a big amount of code in an ordinary programming language, is acomplished with three lines of GSL code, since the language abstracts the Warcraft 3 data format and the map archive and allows the programmer to treat a map file like a struct.
Because GSL scripts can be used to easily retrieve and write information from and to maps, respectively, they will be used for YouTD to export content from the map, where the users created it, and import it into the final game and the test map.

Chapter 6

Conception of Open Innovation Games

The previous chapters described the platform for YouTD, the scripting for this platform, and Open Innovation concepts. This chapter proposes some basic concepts, architectures, and workflows for designing Open Innovation games in general.
To give a short overview over the topics which have to be addressed, here are the main functional requirements for any Open Innovation game:

- A game stub[1] must be provided.
- Users must be given a development environment to create content for the game.
- Users must be enabled to submit the created content to the game's publisher.
- After having been submitted, the content must be rated and checked for mistakes to decide whether it will be accepted and added to the game.
- Accepted content has to be imported into the game stub to create a runnable game which can be released.
- Whenever a considerable amount of new content has been gathered, a new update of the game must be released.

This chapter suggests ways to meet these requirements by proposing an architecture and example workflows. The next chapter will describe the realization of YouTD, applying the concepts of this chapter. While the next chapter explains details for creating a Tower Defense and is thus not generally applicable, this chapter proposes concepts to be used in any Open Innovation game.

[1] It is called *stub* because it is not a complete game, but a game missing the user-created content. In this thesis, every map or game containing everything but the user-created content to be perfected and playable will be labeled *stub*

6.1 Roles, Architecture and Use Cases

An Open Innovation project includes different types of developers and voluntary contributors. Before explaining the composition of the software architecture, roles interacting with the different software components are defined:

- **Author**
  The term author refers to the person or group of people developing and publishing the game. This can be companies, open source teams, or, like in this case, a single person. His job is to create the non-user-created content, the Open Innovation architecture, and to release new versions of the game.

- **User**
  Any person joining the community of the Open Innovation game is called user. These users help perfecting the user-created content by discussing and rating it. Even if many users do not contribute content themselves, they are very valuable for the Open Innovation process. Most users frequently play the game to be created[2] and other similar games, so they know the game's mechanics and thus have the knowledge to rate the balance and concept of uploaded user-created content.

- **Contributor**
  Contributors are users which create and submit content for the Open Innovation game and thus are the backbone of the concept. They can also modify the content of other users, if they detect a flaw or imbalance in it.

- **Administrator**
  Administrators are persons (either voluntary users or employed persons or the author) who have a deep knowledge of the game. Their job is to decide whether submitted content is accepted and makes its way into the game, or is declined because of flaws or imbalances. Therefore, they are the control instance to protect the game from low quality or malicious content. They assure the quality of each new release.

Since contributors, users and partly also administrators are volunteers who like the concept and want to help improving the game, it is very important to release an early beta version with a low amount of content. This will attract users and start the Open Innovation process.
In contrast to the volunteers, the author usually has economic interests and wants to increase his profit by using Open Innovation.

For the basic architecture of an Open Innovation game, the following three main parts are proposed:

[2] Of course, only few users are attracted until a first beta version is released

- **Game Stub**
  As already mentioned, the parts of the game which are not user-created have to be developed by the author. The game stub is just a usual, non-Open Innovation game with some parts missing. Instead, interfaces to add the missing user-created parts have to be included. In addition, special software has to be developed which imports all accepted user-created content into the game stub to form a new release of the game. This software is called *build script*, even if it does not necessarily have to be a script.

- **Web site**
  Most games which are released already have official web sites with information about the game and a forum to discuss it or report bugs. For Open Innovation games, such a web site is even more important. Besides the features found in usual game web sites, it allows users to upload their created content and hosts a database of all uploaded content. Users can search or browse the uploaded content, discuss it, rate it or download it to use parts of it for their own contributions. Admins check the uploaded content and accept or decline it. So the web site is the core place for the Open Innovation to deploy.

- **Development Kit**
  The development kit is a bundle of software which is available for download by users who want to contribute content. It includes all software necessary to create or modify content and save it in an uploadable format. In addition, it should include a testing environment where users can check if their content is working as intended before uploading it.

Figure 6.1 displays a UML use case diagram showing the roles and their use cases. The use cases are categorized in the three parts of the architecture. The use cases only reflect the Open Innovation process. Prior tasks like creating the game stub, development kit and web site are not shown.
The ordinary user can browse, rate and discuss content on the web site. If a user wants to contribute his own content, he can download the Development Kit and create or modify a piece of content, which can then be uploaded to the web page. Upon uploading it, it will be stored in the database and users can start rating and discussing it on the web site.
Administrators can decide whether the uploaded content is accepted for the game, which is called *administrating* the content. Administrators use a special administration environment to browse for and detect unadministered content. The browsing process is included in the administration use case and not explicitly mentioned in the diagram. The administration process is explained in detail later.
The author has only a little amount of work in the Open Innovation process. This is fully intended, since the Open Innovation process should run autonomously to save costs.

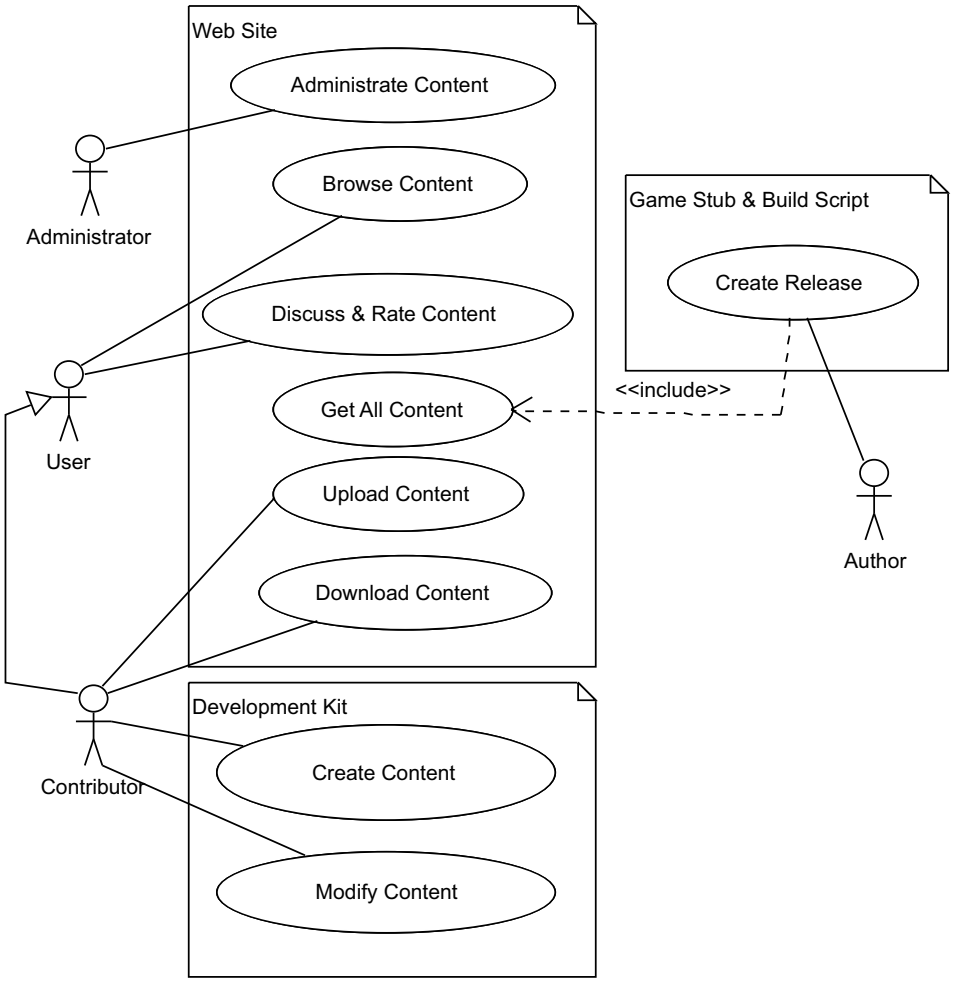


Figure 6.1: *Use cases for creating an Open Innovation game*

To create a new release of the game, the author has to start the build script which inserts all uploaded and accepted content into the game stub. This includes of course to obtain all the content from the web server first.

Of course, the whole architecture could also be included in one application. In this case, no web site would be provided, but an application which features a development environment and a browsing, discussion and rating environment in the game itself. However, the aforementioned architecture with a dedicated web server was chosen so even users who do not have the game installed on their local machine can browse and discuss the content. In addition, since YouTD uses Warcraft 3 as a base for the game stub and the development kit, it is not possible to include an online browsing and discussion feature into the game without hacking Warcraft 3.

6.2 The Open Innovation Workflow

This section shows the example workflow from a user contributing a piece of content to the author releasing a new version of the game. An overview of this workflow is shown in Figure 6.2.
The workflow starts with a contributor who creates content for the game. After testing the content, the user uploads it to the web site. Once the content is uploaded users can rate and discuss it. Even if not shown in the diagram, this is of course an iterative process where the content is discussed, rated and updated by the user multiple times. An admin can wait for the result of the discussion or start right away with checking the uploaded content. If the content is malicious (e.g., it contains racist or offensive content), the admin can completely delete it from the database. Otherwise, he checks whether the content is well-made and working properly. If it is not and the user does not update it, the content is declined. Declined content can be modified and updated by any user. Of course, the user is informed about the flaw and has some time to update it before it is declined (omitted in the diagram for simplicity reasons). If the content has become acceptable, an admin will approve it, so it will be in the next release of the game. Once enough newly approved content was collected, the author will run the build script to create and release a new version of the game.

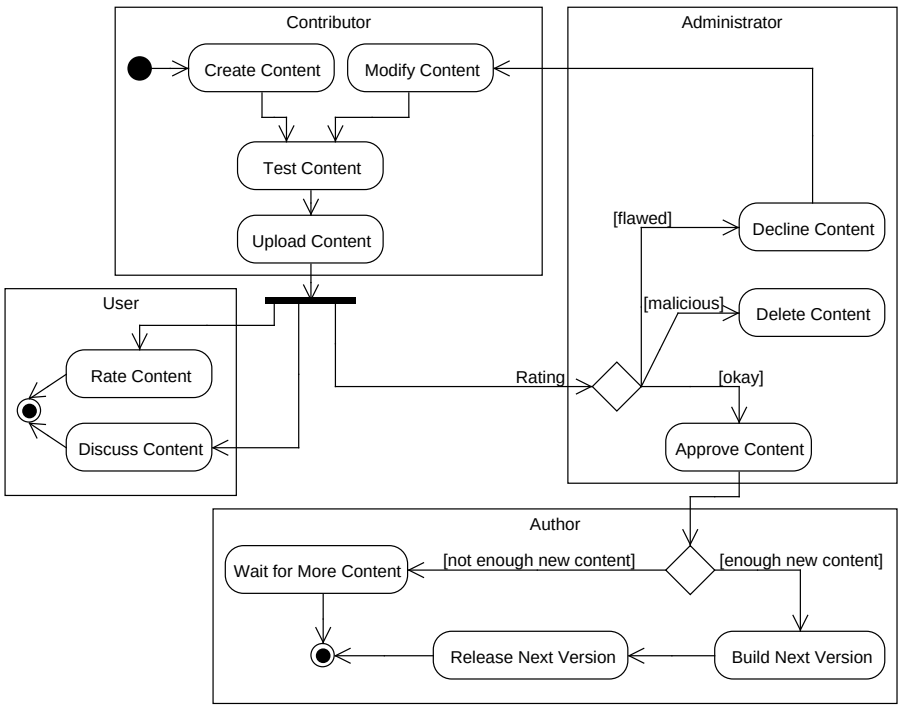


Figure 6.2: *Example Open Innovation workflow*

The explained process demonstrates the main workflow applicable to any Open Innovation game. It is just an overview with many design details omitted. The next chapter will show the realization of YouTD, applying the concepts from this chapter and showing detailed design and implementation decisions.

Chapter 7

The Realization of YouTD

This chapter explains the implementational and design details of YouTD, using the conception from the previous chapter. Mainly, it describes how the three main tiers of the Open Innovation architecture (web site, build script and game stub, and development kit) are implemented. Finally, it shows further details like the mechanisms used to balance the game and how users get attracted to become contributors.

7.1 Development Kit

The main task of the Development Kit is to provide an IDE for contributors. Since YouTD is a map for Warcraft 3, the user-created content will be Warcraft 3 content. Warcraft 3 already provides an IDE for this type of content: the World Editor. The contributor will create the tower or item in the World Editor and then save it as a Warcraft 3 map. A GSL script called *export script* will open this map, extract the tower or item from it and save it in a portable format (XML) which can be uploaded to the web site.
The Development Kit consist of the following main parts:

- **Content creation map**
  The contributor will design the towers or items in the World Editor. A Content creation map (CCM) is provided, which is a normal Warcraft 3 map to be opened with the World Editor. It allows the contributor to design one piece of content. Since items and towers should be designable, a map is needed for each of these two types of content. The CCM already contains a predefined tower or item without any abilities, respectively. The contributor only has to load it with the World Editor and fill the tower or item stub, respectively, with values, model and code to create his very own piece of content.

- **Overlay engine with API**
  An overlay engine is written in the scripting language vJASS, which makes scripting easier for the contributors. It will hide complicated things and replace them by simple functions allowing the modification of most features

in one line of script code. The set of functions accessible by the contributors (the API) will be described in the "HowTo" which is to be created.

- **Test map stub**
  The test map stub is small map which includes the overlay engine and functions to test content elements. The export script will automatically inject created content elements into this map, making them testable by the contributor before submission. This map contains a small piece of land, where the created towers can be built, and waves of different enemies can be spawned to test the strength of the created tower or item against different types of enemies. The test map stub was separated from the CCM, so that the CCM stays small and clearly arranged. The CCM only contains the triggers and objects necessary to create a tower, not the triggers needed to spawn enemies and do the testing, which could confuse the tower creator.

- **Export script**
  A script written in GSL is used to export content from the CCM. The script opens the CCM, extracts the tower or item and all other data required (like abilities used by it) and writes this information into an XML file. It then packs this XML file into an archive which can be uploaded to the web page to publish the content.

- **Import script**
  The import script takes the XML file created by the export script and uses the information to inject the content into another map. This map is the test map stub where contributors will test their content. Finally, this script will be used to inject all content elements into the game stub to created the final game. Like the export script, it is written in GSL.

- **Comprehensive HowTo**
  A comprehensive HowTo is written, which explains every single step required to create a tower or item. It also describes the engine API and gives hints how to balance a content element and how to make it fit into the game concept. It will be the documentation for users who want to create content.

These main parts will now be explained in detail.

7.1.1 Overlay Engine and API

Warcraft 3 already provides an API of JASS functions which can be used to influence gameplay. However, these functions are often not satisfying, the API is not documented very well and it is not object-oriented, with functions not being grouped reasonably. Therefore, an overlay engine created in vJASS will straighten out these inconveniences. It is object-oriented so functions are grouped into struct

methods, and these methods are well documented. While some of the struct methods are only object-oriented wrappers of JASS API functions, many others add functionality missing in the API.

The overlay API was designed to execute all methods it provides extremely efficiently. For the majority of problems, the API uses the fastest and most sophisticated algorithms for solving them. Since it covers the most common and complex topics, it even allows unexperienced users to create high-performance code. Without this engine, many contributors would try to code the methods themselves, which might lead to duplicated and slow implementations. Users may request new features for the overlay engine, which will, if accepted, be included into the next update of the development kit.

Figure 7.1 shows an UML class diagram as an overview of the most important struct types used in the overlay engine.

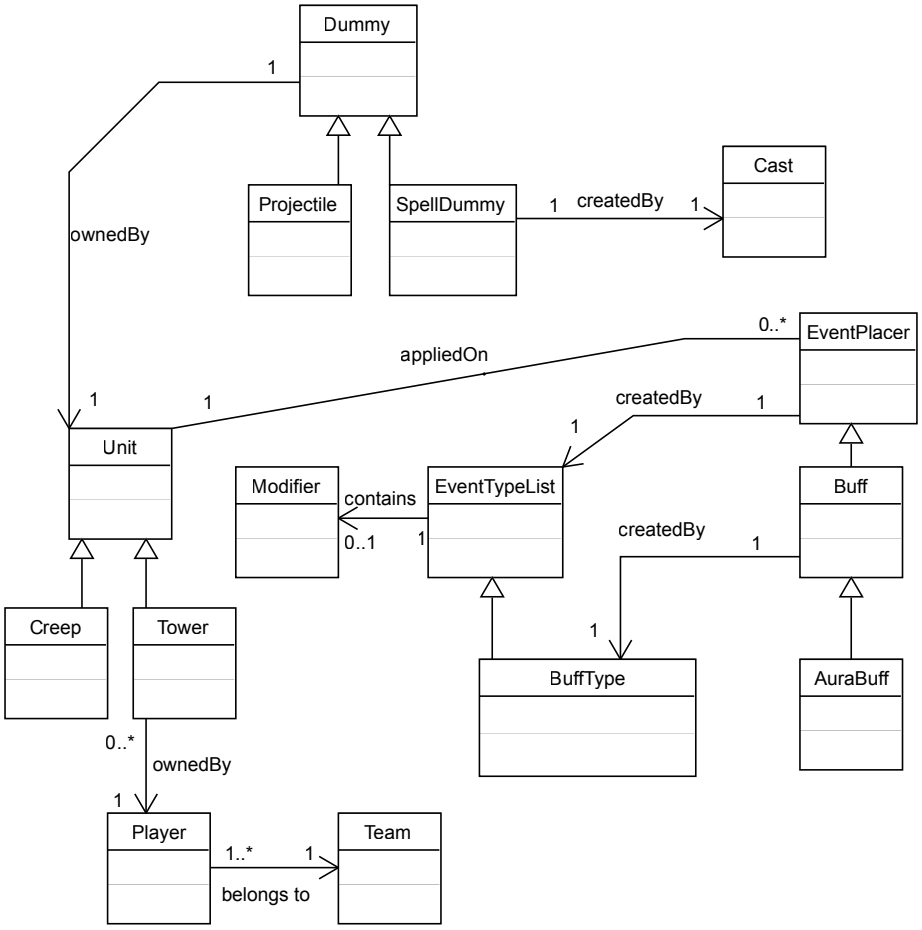


Figure 7.1: *The most important structs in the overlay engine*

The central role in the engine is played by the struct `Unit`, which represents an in-game unit, either a `Tower` or a `Creep`. These structs contain methods for obtaining information about units and altering them. For example, stats like "damage" and "attack speed" of a tower can be adjusted using those methods, or *Area of Effect*[1] (AoE) damage can be dealt by a specific tower by calling its respective method.

As can be seen from the class diagram, a `Unit` is linked to many other classes. The first one is `Dummy`. A dummy is a Warcraft 3 unit which was altered so it no longer is a selectable and visible in-game unit, but an invisible object which can perform tasks for the programmer.
Two structs are extended from `Dummy`. The first one is `SpellDummy`. Spell dummies solve a prominent problem in the Warcraft 3 engine: Abilities (also called spells) defined in the object editor always have to be cast by a specific unit. However, for some scripted abilities, a contributor would want a spell to be cast "out of the blue". A `SpellDummy` is an invisible unit existing only for a short time for the sole purpose of using an object editor ability. Contributors can use such a dummy to cast spells from any desired position.
For example, a contributor might want to create a "bouncing fireball" for his tower: The tower throws a fireball at an enemy unit, dealing damage to it upon impacting. If the fireball kills the creep, it should bounce to the next creep in range. There exists an ability in the object editor which shoots a fireball, which is a convenient base for this ability. Thus, all the scripting a player would have to do is make the tower use this ability on the creep and, if it dies, cast the spell from the creep's position to the next creep. Since there is no unit in this position[2], a dummy has to be created to cast the second fireball to emulate the fireball bouncing from the creep.
The `SpellDummy` from the overlay engine hides most cumbersome parts of the dummy-creation. The contributor only has to indicate which ability should be applied from which position to which unit or location. The engine will create an invisible dummy, grant it the chosen ability and order it to use the ability on the desired unit or location. Usually, a dummy should be removed after having applied its ability. However, since unit creation and removal is a time-consuming job, the dummies are recycled instead: A free stack is administered where unused dummies are stored and reused for other spells later. This is an example of the engine increasing performance, since an average contributor would have most likely created and deleted a dummy every time he needs one.
The second extended struct is `Projectile`. A projectile is a graphical object which usually flies from a unit to another one and has some effect like dealing

[1] AoE damage is damage that hits a circular area, not only a single creep.
[2] the dead creep cannot use an ability anymore

damage to its target. Example projectiles would be arrows from archers or cannonballs from cannons. Besides the hard-coded attack projectiles of units, the Warcraft 3 engine has no functions for creating specific projectiles. However, contributors might want to create own non-attack projectiles, e.g. ice shards falling down from the sky for a blizzard spell. This gap is filled by this struct. Projectiles are emulated by dummies, which are given the model of an attack or spell projectile. They are moved by the engine using different movement modes. The contributors can create a projectile, set parameters for it and let it fly over the map.
YouTD's engine provides many sophisticated ways of moving a projectile conveniently:

- Projectiles flying in one direction or a curve, with constant velocity or acceleration
- Projectiles flying "physically", i.e. describing parabolas, and once they reach the ground they either impact or bounce off
- Projectiles following a target unit
- Linear interpolated projectiles between two locations
- Bézier interpolated projectiles with four control points
- Cubic Spline interpolated projectiles which can take an arbitrary number of control points and thus follow very complex trajectories

In addition to the various ways of movement, event handling functions can be written for a projectile to make it influence gameplay. For example, a projectile produces an *onHit* event when it hits a creep. A contributor could react to this event by making the projectile deal damage to this creep, thus creating a damaging projectile.
All dummies are created by a unit (the unit casting the dummy-supported spell, called "owner" of the dummy) and store a back reference to it. This is important, because if the dummy kills a creep, the credit for the kill (i.e. the bounty and experience) should be granted to its owning unit.

The second big struct complex in the overlay API besides dummies is the `EventTypeList`. It is used to create permanent and temporary changes and events on units. An event type list is basically a list of event handling functions reacting to different events. When it is applied onto a unit, it will permanently add these event handlers to the unit. The unit onto which such a list is applied is considered *"buffed"* by this event type list.
An example will make the usage clearer: An `EventTypeList` is created, and a function which makes the buffed unit deal 200 points of damage to the target is added as an *onAttack* handling function. If this `EventTypeList` is now applied onto a tower, the function will be called on each attack and thus the tower will deal 200 additional points of damage whenever it attacks a creep.

In addition to adding events, an `EventTypeList` can also contain a `Modifier`, which modifies values of a unit (like damage or attack speed) as soon as the `EventTypeList` is applied.
In addition to applying an `EventTypeList` onto a unit, which permanently adds the event handlers, an `EventTypeList` can also be used to create an `EventPlacer`[3] and apply it onto the unit. The difference is that a reference to the `EventPlacer` is stored and the `EventPlacer` can be removed by calling its remove method. Thus, the events can be on the unit temporarily, while the direct application of an `EventTypeList` is always permanent.
The types `BuffType` and `Buff` extend `EventTypeList` and `EventPlacer`, respectively. They are an implementation of a buff system (cf. page 12). A buff adds events or modifications to a unit as long as it is applied on it, so the event placer concept is already an implementation of these effects. The only thing missing on an EventPlacer is that it has no visible graphics effect on the unit and no expiration timer. So the struct `Buff` adds the graphical effect and makes the buff expire after a specific duration. The duration and effect can be set in the buff's `BuffType`.
The struct `AuraBuff` is an implementation of an aura system (cf. page 13). Since an aura is an effect which applies a buff onto every unit coming in range of the aura-emitting unit, this system can be implemented by extending the struct `Buff` and adding aura-specific features: The buff must be applied onto every unit coming in range, and if a buffed unit leaves the aura range, the buff must be removed.
Since there can be more than one unit which emits an aura with different powers[4], a priority queue must be managed for each aura buff to track all aura-emitting units currently in range. If a buffed unit leaves the range of the current aura-emitting unit, the next emitting unit is taken from the priority queue to check if the buffed unit can instead receive the aura from this unit.

The engine provides many more structs, but only the most important ones were mentioned. It provides systems and structs to easily use *almost all* common Tower Defense features with only some overlay API calls. The overlay engine currently contains about 11500 lines of vJASS code. For more information and description of the methods, please refer to the created HowTo, which is a comprehensive manual for the overlay API.

7.1.2 Content Creation Map

The *content creation map* is a Warcraft 3 map which, when loaded with the World Editor, serves the contributor as an editor for creating a tower or item. This section explains the basic approach for a contributor to create a tower with the content creation map. The approach for items is very similar, so it will not be explicitly described.

[3] This term was coined since it places event handlers temporarily
[4] Of course, the currently most powerful version should always be applied

The content creation map for towers contains a predefined tower type in the object editor. This is the type of the final tower which has to be edited by the contributor to customize his tower's object editor values (e.g., name, description, range, attack projectile model or attack speed).
There is a rectangular region in the middle of the map where a sample of the tower to be created is placed to show the contributor the appearance of his tower. Since the final file size for YouTD must not exceed a certain limit, no custom-imported models for the tower's appearance are allowed. To increase the number of models in comparison to the number of predefined models in Warcraft 3 without allowing imported models, a *model combination system* is used. This means, the player can create model parts in the object editor and choose between the predefined models for each of them. The model parts can then be rotated, colored, scaled, and placed in the map around the tower model in the rectangular region. The export script will combine these models into one, so whenever the tower is built in the game, not only the main model will be visible, but also the model parts which were placed.
Figure 7.2 shows three exemplary steps for creating an assembled tower model. First a predefined base tower model was chosen on the left. The rectangular region in which effects can be placed to create an assembled tower model is shown in light blue. In the middle, four brazier effect models were placed around the tower and on the right, an archer model was placed on top of it. Even if this tower does not look very aesthetic, it shows the theoretical concept for assembling a tower, allowing to create very unique towers.

Figure 7.2: *Three steps of creating an assembled tower model*

Figure 7.3 shows three contributed tower models from YouTD where the assembling concept was used to create innovative and beautiful models. On the left, 6 arcs were rotated and combined and a fire model was put in their middle to create the "Caged Inferno" tower. In the middle, a fantasy creature was put on a rock and surrounded by plants. On the right, a "fire elemental" model was put inside of a volcano model and surrounded by fire and skull models. The tower was fittingly named "Living Volcano". Like in most Tower Defenses, these models no longer resemble real towers. Instead, any model that fits the genre can be used.

Figure 7.3: *Assembled tower models from YouTD*

To create unique tower abilities using scripting in vJASS and the overlay API, the contributor must enter the trigger editor, where he will find predefined triggers for events he can react to. Such an event is, for example, when the tower kills a creep or damages a creep, or when the tower is built. Each trigger contains an empty function and some settings in the trigger's comment area. The contributor can insert script code in the empty function's body to create a handler for the event. Then, he can name and explain this special ability in the comment area. These comments will automatically be inserted into the tower's description by the import script and will be shown in-game (when clicking onto the tower) and on the web site (when showing the tower's details).
In addition to the event handling triggers, there exists a *header trigger* which contains no event handler but is plainly copied into the final game and can contain user-defined functions, structs, and global variables which then can be used in the event handlers.
By reacting to the events, very unique towers can be created. For example, one contributor created a teleport tower which reacts to the *onDamage* event by saving the enemy's current position, waiting three seconds and then restoring the enemy's initial position.
After the event handlers have been scripted by the contributor, he is basically done with the creation. Now the map can be saved, GMSI started and the export script executed. This will create an archive which contains all necessary information and can be uploaded to the web page. In addition, the tower will be inserted into a test map stub, which can already be played by the contributor. This test map features an in-game testing environment for the tower. The tower can be built and creep levels of different strength and categories can be spawned. This way, the contributor can test the abilities of his tower against different types of enemies, find bugs in his scripts and balance his tower before submitting it to the web page.

Listing 7.1: Tower Struct in GSL

```
typedef Tower struct{
        int scriptVersion = 0;
        array<array<var>> oeValues = array();
        array<array<var>> settings = array();
        array<TowerEffect> effects = array();
        array<TowerTableEntry> dmgTable = array();
        array<TowerTrigger> triggers = array();
        array<TowerObject> abilities = array();
        array<TowerObject> buffs = array();
        array<TowerObject> units = array();
        array<Reference> references = array();
}
```

7.1.3 Export and Import Script

The previous section provided an overview of how a contributor can create a tower using the content creation map. This section explains the mechanics behind the export and import script which extract the tower from the map, create the uploadable archive and insert the tower into the test map.
Both scripts are written in GSL to allow reading data from and writing data to a map, respectively.
The *export script* loads a map, finds the content element in it and extracts its information into an XML file. The extraction is done by gathering data from different parts of the map (the assembled model from the effects in the terrain rect, the data values from the object editor file, and the triggers from the script file). Before the extraction, the script will carry out some preprocessing on the tower and check it for some common mistakes.
First, the export script will extract the information into a GSL struct which can then be serialized to XML. Listing 7.1 shows the GSL code for the struct definition of a tower. On top, it consists of a version control integer, an array (`oeValues`) where the tower's object editor values will be stored and a `settings` array where additional settings are stored. The next member is the `effects` array, which contains the effect units placed in the rectangular region to create an assembled model. The `triggers` array is used to store the script code of the tower's event handlers. The other values are of minor significance and will not be explained here. Note the types in the angle brackets, denoting the types of values to be put into the respective arrays. These are struct types to represent different data, which will not be further explained here.
The export script does some checking to validate the tower and warn the contributor if a mistake is revealed. For example, a lightweight spellchecker checks for capital letters within words and lower case letters at the beginning of ability and tower names, which are probably mistakes, and reports this as a warning to the contributor.
After the checks and the insertion into the struct, the struct is serialized into an

XML file. For this purpose, an XML parser and generator was written in GSL which uses reflection to turn arbitrary structs into XML files. These files can then be re-read and deserialized back into a struct instance. This is the basic mechanism for making content portable. Additionally, the XML file is interpreted by the web script to insert the tower into the database with appropriate parameters.
To test if the XML file can properly be read and the tower contained in it can be injected without errors, the export script runs the import script as soon as the XML file is created and tries to import the tower in the XML file back into a test map. If an error occurs during this import, the contributor is informed and the script is aborted. This way no towers can be submitted which contain errors and would crash the build script assembling the final map. In addition, this import into the test map allows the contributor to test his tower.
Next, the script asks the contributor to provide a screenshot of his tower to be displayed in the web script. As soon as the contributor specifies one, the XML file, the content creation map and the screenshot image are packed into an archive. This archive can be uploaded to the web page and thus the tower can be submitted and added to the next release of the game.

The *import script* inserts the tower into the test map or the final map. It reads an XML file which should, of course, represent a tower or an item, respectively. It first uses the XML engine to deserialize the XML data back to a struct. Next, it checks the script version in the tower struct against its own script version. This prevents that towers with an outdated incompatible version of the export script are imported with this version which could lead to wrongly injected towers, because not all early versions were backward compatible.
Following, the script checks the tower data for validity. This validity check tests values which are known to cause bugs or make the game crash if a wrong value is inserted, and controls if all necessary data is contained in the script. For example, the alpha value for the tower model must be between 0 and 255. Other values would make the game crash and are thus forbidden and will cause the import script to stop.
Next, the script creates the tower unit and objects used by the tower (like abilities or buffs) in the object editor and copies the object editor values stored in the struct to the respective objects.
Another important step carried out by the import script is to balance the tower damage. Since the level of damage a tower deals is the most important factor for the tower's balance, this burden is taken from the contributor and executed automatically. More information about the way the script balances the damage can be found in Chapter 7.4.1.
The final data which has to be inserted into the map is the vJASS script code to initialize the tower and the contributor's script code used in the event handling triggers. The contributor's script code is simply copied into a special trigger in the map and surrounded by a scope block. A scope is a vJASS construct creating a namespace for private functions and variable names used therein. This is necessary

because two users could use the same name for their functions or variables. With the scopes, these names are put into different namespaces and no collision can occur.
There is an initialization trigger called at map start, in which each tower must be registered. The code for doing this is completely created by the script, no line has to be written by the contributor. The effect units from the assembled tower model must be registered, because the tower model is assembled by a trigger in the game by spawning dummy effect units with appropriate models at the correct positions relative to the tower's position whenever a tower is built. To achieve this, code for registering each effect is added to the initialization trigger. Next, the events which the contributor handles in his triggers must be registered with the tower type so the appropriate function is called whenever the event occurs for this tower.
Lastly, the script registers some more features for each tower like the tower's type so it appears in the "buildable tower" list.
Now the tower is available in the object editor and all necessary vJASS code is created or copied into the map, so the tower is successfully injected and can be built in the map. For the final map, the import script just has to be executed onto the game stub for every tower. Then the game will be ready for the next release.

7.1.4 Development Kit Bundle and HowTo

The export and import script, the content creation map, and the test map are packed into the development kit which can be downloaded by the contributor. The kit also contains GMSI, the interpreter for GSL which is used for the export and import scripts, since most users do not have it.
Since the overlay API is written in vJASS, which needs a compiler not included into the basic World Editor, a contributor needs the Jass NewGen Pack including the compiler for vJASS. To simplify the installation process, a further version of the development kit including the Jass NewGen Pack is released. Since the Jass NewGen Pack is free software, no copyright issues are to be taken into consideration. In addition, the Jass NewGen Pack World Editor contained in the development kit has a modified editor with a GMSI menu injected, so the contributor can start the export script conveniently from the World Editor, where he created his tower or item.
In addition to the previously mentioned content, a comprehensive HowTo is written and added to the development kit and published on the web page. The HowTo explains in detail how the development environment is set up and how exactly a tower and item can be created. It describes all possibilities a contributor has when designing a content element.
In addition, it contains a comprehensive documentation of the overlay engine API with examples for each topic. So it serves as a tutorial to learn about the API and as a reference work for contributors who are already familiar with the API. The last chapter of the HowTo gives hints for balancing towers and items and making them fit the different elements.

The HowTo is intentionally written informally and humorously, so that it does not evoke the impression of a boring manual, but rather of an interesting helper for creating content. Of course, it still explains every detail of the creation process which might be boring for young contributors, but since it should be usable as a reference guide for the API, it must cover every little topic.

7.2 Web Site

A web site is set up at *www.eeve.org*, allowing contributors to submit content and browse and discuss content submitted by others. Again, this section only explains the procedure for towers, the one for items or other imaginable user-created content is similar.
The first important feature is to upload tower archives, which were automatically created by the export script, and tag them with a small comment. The web script, which is written in PHP and linked to a PhpBB [37] installation, which is an open source PHP bulletin board, will extract the archive and parse the XML file which stores the tower data. After doing some validity checks, necessary data like the tower's name and its description is taken from the XML file and stored in a SQL database. A folder is created, the extracted XML file and content creation map is copied to it, and a discussion thread is created in the PhpBB board which will cover the discussion about this tower. The folder's name and the discussion thread ID are also stored in the database, so the thread can be linked to the tower, and the content creation map can be downloaded from the folder.
In the same way, a tower can be updated by a contributor by specifying which tower to update and submitting an updated version to the web site.

Of course, a control mechanism must be set up to keep malicious and low-quality content from entering the final game. Otherwise, towers which do not fit into the game, or contain bugs in their scripting or offensive or racist content, might enter the map and ruin the game.
This is achieved by an approval and decline system. Administrators are nominated who have special functions enabled in the web script to administrate uploaded content. Of course, the game officials[5] are the first administrators, but users who have proven to be reliable and have sufficient knowledge to find bugs in the script are nominated, too. This benefits Open Innovation by outsourcing even the control mechanism to the users.
The administration process controls which towers will enter the map. When a tower is submitted to the page, its status is *pending*, which means that the tower has not been reviewed by an administrator and will not enter the map yet. Administrators should review these towers and either write a comment in the tower's discussion thread if they find bugs, typos or balance issues, or approve the tower, thus setting the tower's status to *approved*.

[5] In this case only the author since this is a one man project.

Approved towers will be added to the next release of the game. If the tower is not approved and the contributor does not fix the flaws within an adequate period of time, an admin will set its status to *declined*. Declined towers can be updated by everyone, which allows other users to continue the work of people who ceased to update their tower.
As long as a tower's status is pending, the contributor who submitted it can submit updates any time. When the tower becomes approved, the contributor and other users can suggest updates, but these will not immediately overwrite the old version. Instead they will receive the status *update pending*, and an admin must approve or decline them like if they were newly uploaded towers. This prevents users from introducing malicious content into the game by first creating a flawless tower which is approved and then updating it with a malicious version.
An admin also has the right to *revoke* already approved towers. This might be necessary if a tower is approved and later it turns out that it contains bugs or balance issues. By revoking it, the tower status will be set back to *pending* and the contributor can fix it.
Lastly, admins can *delete* towers which are offensive, racist, or consciously malicious.
Besides this authoritarian control mechanism, a community-based control mechanism is used. This is achieved by allowing every registered user to rate uploaded content. By checking this rating, administrators can decide about towers to be accepted or declined. If the community dislikes a tower, it is not worth being accepted even if its balance and coding is flawless.
The amount of towers in the map could be restricted by allowing only a specific number of towers in the map and choosing those with the highest rating. However, this approach is not used at the moment. All towers which get approved are added to the map.
The aforementioned rating system also encourages contributors to create high quality content and to perfect already uploaded content and thus benefits the Open Innovation idea.

A content browser is created which allows users to browse the uploaded towers and items. The content can be sorted and filtered to find a specific set of towers, or a certain tower can be searched by name. For example, only towers of a specific element, in a specific price range or submitted by a specific contributor can be displayed.
The towers matching the filters will be displayed with some details like their name, image, description, combat attributes, and submission details like the author, the submission date, and last update date.
Figure 7.4 shows a tower in the web interface. On the left, the price, icon, and screenshot of the tower can be seen. The next two columns contain more details about the tower's attack features, the submission details, and other miscellaneous information. The last column contains the description assembled by the export script from the different ability, trigger, and tower descriptions. It was converted

to HTML to allow the text highlighting. In addition, the rightmost column contains buttons for displaying more tower details, downloading the tower's content creation map and for suggesting an update for this tower. If the browsing user is an administrator, he will also see buttons to administrate the tower.

Figure 7.4: *A tower in the web interface*

Users can download the tower's content creation map to test towers with the test map or to copy code for their own towers. This benefits the Open Innovation idea, since users can learn from and reuse the work of others.

If a user clicks on the image or the name of the tower, he will be led to a page where the tower details are shown again, followed by the discussion thread of the tower. In the discussion thread, users can share their opinion about a tower or suggest changes. In addition, administrators can indicate here, if the tower still has flaws and is thus not approved until the user changes them.

If a user pushes the "tower details" button, he will see implementatory details of the tower. For ordinary players, the information already displayed is sufficient. Only people who want to review the tower or copy code from it will be interested in the information shown by this button.

The web script must also be able to gather all approved towers and send them to the author so the import script can be run on them to create and release a new version of the game. For this purpose, the script queries the database for all approved towers and packs their XML data into an archive. In addition, it creates a GSL script (the build script), which calls the import script for every tower and item, and adds this script to the archive. The archive is then sent to the author.

All that has to be done to release a new version is extracting the archive and running the created GSL script onto the final map stub to insert all user-created content.

As an additional motivation to create content and as an overview for the submitted content, a statistics page is created. This page contains different tables on the uploaded content. A table indicating how many towers have already been submitted

for the different elements and the respective rarity grades shows users who want to create towers which elements and rarities still miss towers and thus should be adressed (see Figure 7.5).

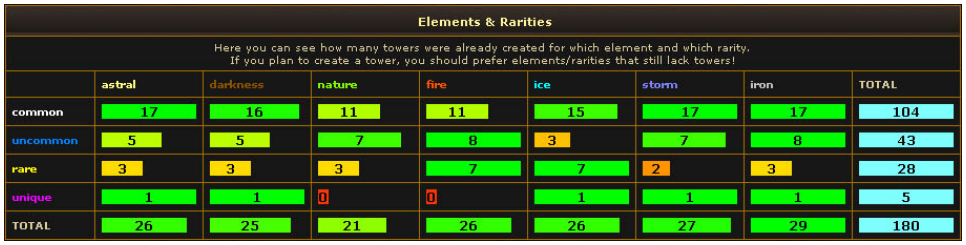


Figure 7.5: *The element and rarity distribution*

Another table shows the ten users who have the highest contribution score (i.e. they have created the most content, cf. Chapter 7.5) This will encourage users to submit more content in order to appear in this list.
The recently submitted towers and items are also displayed on this page so users can check the latest changes and new content in the map (see Figure 7.6).

Figure 7.6: *The recently submitted towers*

Besides these tables, more statistics are planned to provide even more information on this page and summerize the development process and current status of the game.
The pages already mentioned contain the core web interface for the YouTD project. These pages are sufficient to create an Open Innovation game and allow users to upload, discuss and rate content.
However, some more pages exist, like a forum with tutorials about scripting and where users can ask questions about the game. Such a forum is important to receive feedback for the project and to create a community. Another page covers a download section for the latest release of the game and the development kit.
With all these pages, the web interface is the backbone for this Open Innovation game and thus one of the most important parts of this work.

7.3 Game Stub and Build Script

Not the whole game is user-created, but only towers and items. The rest of the game is designed by the author and will be explained in this section.
The game is created as a normal map for Warcraft 3 with only the towers and items missing ("game stub"). The build script will later insert the user-created content into this map, to create a release of YouTD.
The design decisions and features to be included into the game were already discussed in Chapter 2.3. They are all implemented in this game stub.
The biggest parts of work for the game stub are creating the triggers for the game's rules and designing a basic environment layout for the Tower Defense (i.e., the lanes the creeps will walk and the areas where players may build towers). The basic environment is then beautified with environmental objects to make the map more appealing.

Concerning triggers, the game stub contains a complete implementation of a Tower Defense trigger framework. This framework must satisfy the following requirements:

- A copy of the overlay engine must be included to allow imported tower and item code to call its API functions.
- The host must be able to choose game difficulty and other game settings at the start of the game.
- Players must be able to choose between elements and must get new buyable tower types randomly (cf. page 15).
- Players must be able to place towers they bought on the battlefield, but only in specific areas.
- Different score boards must be provided, e.g. for displaying the upcoming levels (cf. Figure 2.3) or the players' scores.

- Random creep levels must be generated at the start of the game.
- The living creeps must be tracked and a new level must be started with some delay as soon as the last creep of a level dies.
- The respective creeps must be spawned for each player whenever a new level starts.
- Per-level-tasks, like assigning income to the players, must be executed after each level.
- Creeps reaching the finish must decrease the player's lifecount and a player must be defeated when he has no lives left.
- Killed creeps must grant bounty to the player and must have a chance to drop a random item.
- The score and XP system must be included (cf. page 18).

The implementation of these systems is not further explained here, it can be found in the map's trigger source code. Even if the overlay engine already includes many tasks, thus reducing the amount of code, the mentioned points excluding the API still take another 4000 lines of vJASS code, resulting in over 15000 lines of code in the game stub.

Besides the scripting, an environment for the Tower Defense has to be created. Since a non-mazing style was chosen, the environment can be beautiful with many environmental objects spicing up the scene (in contrast to mazing Tower Defenses, which often have only plain terrain with no environmental objects since the player needs such a plain territory to build his maze on).

Before adding a beautiful environment, the general shape of the lanes and the areas where towers can be built has to be designed. YouTD will support up to eight players in a match. As setting, an ancient ceremonial pyramid is chosen. The creeps start at the bottom of the pyramid and take paths (lanes) to the peak of the pyramid where the finish is located. Each of the four sides of the pyramid contains two partly overlapping lanes. This allows players to play solo, together with the player on their side (also called "lane partner"), or together with all other players in a team. When playing with a lane partner, the two partners can help each other: Since the lanes are overlapping, there are regions where the towers of one player can also hit the creeps of the other player.

Figure 7.7 shows the bird's eye view of the map. The areas relevant to gameplay are displayed for the two lanes on one side of the pyramid. The creep spawns are denoted with *S*. The arrows show the lanes the creeps will take to reach the finish (*F*). Depicted in red and green are the areas where the respective player can build his towers. The creeps' ways cross at the middle path. Here, both players can hit their own creeps as well as the creeps of their partner. After the paths have crossed, the creeps even proceed into the area of the other player, which then allows a player to compensate his partner's misses. The other sides of the pyramid contain equally shaped lanes. Such a symmetry is very important for a Tower Defense to keep the lanes equally challenging.

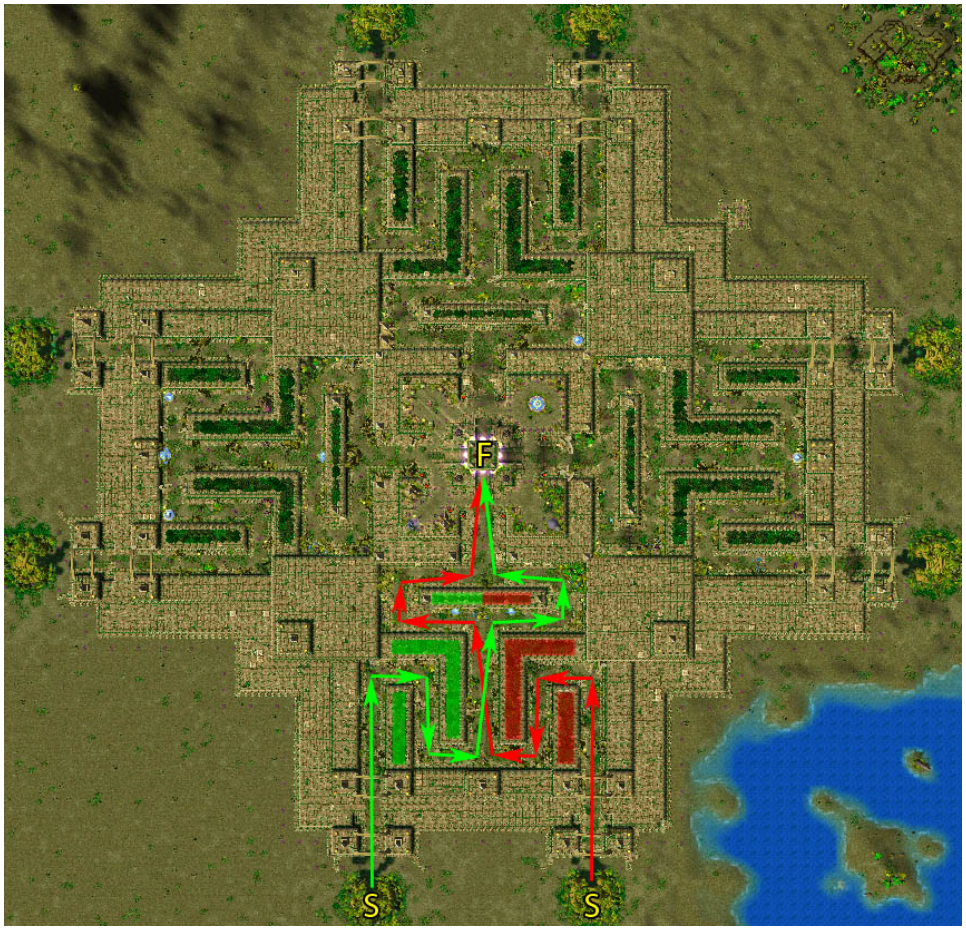


Figure 7.7: *Bird's eye view of the map*

When the shape of the lanes is determined, the maps can be filled with a beautiful environment. The lanes are filled with environmental objects like plants, ruins, buildings, and other objects fitting the setting. Figure 7.8 shows a part of a lane decorated with plants, ruins, statues, and a throne in the middle. Of course, these objects must leave enough space for the creeps to pass. Figure 7.9 shows the finish at the pyramid's peak and two sides of the pyramid with their lanes in the background. The finish was designed as a building with magic portals which the creeps try to reach. Note the many environmental objects like the settlements around the pyramid, which have no gameplay influence, but benefit the atmosphere.

Figure 7.8: *A lane with environmental objects*

Figure 7.9: *The finish at the peak of the pyramid*

After the triggers are inserted and the landscape is designed, some minor necessary features like, e.g., a loading screen, are added. Now the map is ready to receive the user-created content and to be released. Figures 7.10 and 7.11 show in-game scenes from the final game. The numbers which are displayed provide information like critical hits and their amount of damage, or the gold gain from killing a creep (bounty). On the enemies, some buff effects are visible like the blue effect at the enemies' feet representing an ice buff which slows down enemies or the fire on their heads for a "burning buff" which deals damage over time.

Figure 7.10: *Towers killing an enemy*

Figure 7.11: *Towers engaging a boss*

The build script that assembles the final map is very trivial since the import script is already able to insert a tower or item into a map. So the build script just has to call the import script for every tower and item, to assemble a new release. As mentioned in Chapter 7.2, the build script is assembled by the web site automatically whenever the author downloads all approved content to release a new version.

7.4 Balancing the Game

After the previous sections described how the Open Innovation process was realized, this section covers the mechanisms used to make the game balanced.
Balancing a game is the process of achieving *game balance*, which means that no player has unfair advantages and the game is neither too easy nor too difficult at any stage. Aspects that are not balanced are considered *imbalanced*.
For a Tower Defense this means that different towers should not differ in strength when compared on a *strength-per-money-spent* basis. So a tower which is comparable in price with another one should not be significantly stronger to prevent players from gaining an unfair advantage by choosing the stronger tower over players choosing the weaker tower at a similar cost.
In addition, the creep strength must be balanced so the creeps become stronger each level, but in a way that allows good players to keep up and still kill them. If the creep strength grows too slowly, the game will get too easy and players will not have fun playing it, because it represents no challenge. On the other hand, if the creep strength grows too fast, the game will become invincible.
Since most of the content is created by contributors, and the game contains an above-average amount of content, it is harder to be balanced than a usual Tower Defense. For this sake, well thought-out balance mechanisms have to be applied on towers, items, and creeps.
Achieving a high level of balance is very important for a game's success. Even if only small aspects are imbalanced, players will sooner or later find them and abuse them and the gameplay will be reduced to knowing and abusing all imbalances. If such imbalances are fixed later by releasing a new patch, many players will be disappointed because the balance has shifted and many previously viable strategies will not work anymore.
The balance mechanisms for items will not be explained in this thesis. They are very similar to those for towers, despite that no damage output is balanced by the script. Instead, the script will rate the item's power and assign a *starting level* (the first level in which creeps can drop the item upon death) to it. More powerful items will receive a higher starting level and thus will appear in higher levels only.

7.4.1 Tower Balance

The towers are created by contributors, so the author has no direct control over their balance. Since most contributors do not have the experience in game creation

to balance a tower, some arrangements for ensuring the balance have to be made. The tower's damage output, which is the value with by far the highest influence on its balance, is calculated by the script; the contributor has only indirect influence on it.
First, a basic value must be set how the tower's damage output is correlated to its price. The simple value *one damage per second per gold spent* was used, so a tower which costs 100 gold should have 100 damage per second (DPS). Since the tower has a specific attack cooldown with which the damage has to be scaled to get the desired DPS, the tower's damage output is calculated in the following way:

$$damage = cost \cdot cooldown$$

Thus, if a tower has an attack cooldown of 4.0 seconds (i.e. it shoots once every four seconds) and costs 100 gold, its damage is set to 400. This formula makes the greatest contribution to a tower's balance. Making damage directly proportional to a tower's cost ensures that neither expensive nor cheap towers have an advantage concerning damage output per gold spent. Thus, many cheap towers are basically as strong as one expensive tower, if the overall cost is the same.
For additionally fine-tuning the damage output, some more values are incorporated into the final calculation of damage a tower deals. The first value to be considered is the tower's range. Of course, a tower with a longer range is stronger than one with a shorter range, since it has more time to shoot at its enemies before they run out of its range. To find good balance values for different ranges, the time towers with different ranges can shoot at enemies was measured. Then, a function was fitted to the measured values. The amount of damage is divided by this function, because the damage should scale indirectly proportional to the time the tower can shoot at a level, so each tower will do the same damage in a level, irrespective of its range.
A function that fitted the values well was a power function with an exponent of 0.6. In addition, a proportionality constant was inserted to normalize the influence of range on towers with 800 range[6] to 1.0, because 800 is assumed to be the "standard" range. So, the damage of towers with 800 range will not be changed, but towers with a higher range will deal less damage and towers with a lower one will inflict an increased amount of damage. To nullify the influence of the range modifier for towers with 800 range, the proportionality constant must be $800^{0.6} \approx 55$. This formula calculates the damage modified by range:

$$damage = cost \cdot cooldown \cdot \frac{55}{range^{0.6}}$$

Even if the attack speed is already included in the formula by multiplication with the cooldown, faster towers should get a slight damage penalty, because a faster tower is usually better than a slowly attacking one due to "percentage based on

[6]The range is measured in Warcraft's length units. For example, a tower is 128 units wide.

Table 7.1: Rarity damage multipliers

rarity	multiplier
common	1.00
uncommon	1.05
rare	1.10
unique	1.20

attack" abilities: There might be items or buffs granting an ability to a tower which has a chance to trigger on each attack. For example, an item called "stunning hammer" could grant the tower a 10% chance to stun its target for one second. Of course, giving such an item to a fast-attacking tower will trigger more stuns than on a slowly attacking one. To compensate this, the fast-attacking tower will receive a minor damage penalty. The formula for this was chosen empirically and ensures that the penalty is not too severe, which would make fast-attacking towers useless. Again, a power function with a low exponent (0.2) was chosen to be applied to the cooldown. This low exponent, which is basically a fifth root, ensures that the penalty for fast-attacking towers will not be too severe.
Next, rare towers should have a small damage advantage over more common towers to reward the luck of having them. Nevertheless, the rarity bonus should not be too high, so getting a rare or even unique tower is no game-breaking advantage. Table 7.1 shows the damage multipliers for the different rarity grades. The highest damage boost achievable through rarity is 20%.
Lastly, the kind of attack the tower uses is considered. A tower with a usual attack, which only hits one target, will not be modified. However, there are towers with splash attack or bouncing attack, and towers which shoot several projectiles at once. Of course, such towers should cause lower damage than towers with a normal attack, since they can hit several targets at once, thus increasing the total amount of damage per shot[7].
Functions to achieve balance for each of the mentioned attack types are included into the import script. Since explaining all of them here would go beyond the scope of this thesis, they will not be mentioned further, but can be found in the source code of the export script. The return value of the attack type balance function will be called *attack type modifier*.
Besides the normal attack, towers can have special abilities as mentioned in Chapter 2.2. It is not possible to balance them procedurally, as a contributor might insert arbitrary code into the event handling triggers, whose strength cannot be evaluated by a program. Consequently, these abilities will be balanced by the contributors themselves by setting an *ability factor*, which indicates how much of a tower's strength is used by its attack.

[7] Assuming that each enemy hit receives the full damage, the damage is *not* divided among the targets

A tower without any special abilities has an ability factor of 1.0. An ability factor of 0.7, for example, means that 30% of the tower's strength is used by its abilities, only 70% are used by its attack. The ability factor is directly multiplied into the formula when calculating the damage. So the stronger the abilities of a tower are, the lower the ability factor and thus the tower's attack damage is. In the above example, the tower would only deal 70% of its normal damage. The other 30% of its price are used to pay for its abilities.
The HowTo contains chapters explaining how to set the ability factor for different abilities. Of course, the HowTo cannot capture all thinkable special abilities, only the ones that are used most frequently, and some basic concepts for choosing the factor. For very complex abilities, the ability factor has to be discussed in the tower's discussion thread and playtested in the test map to achieve balance.
Summing up all prior paragraphs, this is the final formula used to calculate a tower's damage, where M_r is the rarity modifier from Table 7.1, M_{at} is the attack type modifier from the previously mentioned balancing function, and M_{af} is the aforementioned ability factor:

$$damage := \underbrace{cost \cdot cooldown}_{basic\ damage} \cdot \underbrace{\frac{55}{range^{0.6}}}_{range\ modification} \cdot \underbrace{cooldown^{0.2}}_{speed\ modification} \cdot M_r \cdot M_{at} \cdot M_{af}$$

No perfect tower balance can be achieved by this formula, especially for the special abilities, but the current beta version of YouTD has proven that at least no obvious balance flaws could be found, and, from a subjective point of view, the towers seem balanced.
If an imbalanced tower is detected, users can suggest an update to this tower to fix the imbalance, so the Open Innovation concept is helpful again.

7.4.2 Creep Balance

Balancing the creep levels is primarily achieved by setting their hitpoints correctly so that the levels stay challenging and the difficulty rises with each level. Since YouTD is an open-end Tower Defense with an infinite amount of levels, the balance has been chosen in a way that the creeps get very strong at a certain level range, slowly making surviving a level impossible from a certain point on. Otherwise players could play forever and get bored. Most Tower Defenses take about one to two hours to be finished if the players survive long enough. The balance will be adjusted in a way that YouTD will also fit this scope of time. Since a creep level lasts about one minute, the game should become impossible at level 120 at the latest, which equals approximately two hours of game time.
To balance the creep hitpoints and to make the difficulty grow with each level, the first parameter to be calculated is the average amount of money a player has gathered when reaching a particular level. Since the amount of money is directly proportional to the damage output which can be bought with it, a suitable formula for measuring the difficulty of a level is this ratio:

$$difficulty := \frac{creep\ hitpoints}{available\ money} \tag{7.1}$$

The higher the ratio is, the longer the creeps will survive, because their hitpoints exceed the buyable damage output, and thus the more difficult the game will be.
The money a player has on average can be calculated from all the money he has gained up to that level, which is the sum of his starting money, the bounty the player was afforded for killing the creeps, and the income that is paid to the player after each round. These values are are well-known, so the available money can be calculated easily:

$$available\ money(level) := start\ money + \sum_{i=0}^{level}(bounty(i) + income(i)) \tag{7.2}$$

Next, the modality of the increase of ratio (7.1) per level must be determined. If the ratio stays equal all the time, the game does not become more difficult, since the growth rate of the creeps' hitpoints is fully compensated by the gold the player earns and can buy towers with, which increases his overall damage output. The game will even become easier, since, over time, clever players will build towers which harmonize with each other, thus increasing their effectivity. Items players give to their towers can also increase their effectivity.
Thus, for the game to become more difficult with each level, ratio (7.1) must increase with each level. It has to grow faster than the towers' effectivity increases due to items and tower combination.
For a game that becomes gradually more difficult, a linear growth could be suggested, but, as discussed before, the game should become impossible at level 120. Therefore, linear growth is not sufficient for later levels, when the game difficulty should rise drastically to end the game at a certain point. Consequently, higher order polynomials will be used.
Since the money a player gains per level grows linearly (the bounty a creep grants rises linearly and so does the income) the accumulated money a player can dispose of growths squared, as it is the sum of the received money. Consequently, to ensure a linear growth of ratio (7.1), the hitpoints must grow at least cubically, because the denominator growths squared. For more than linear growth of the ratio even higher level polynomials must be chosen.
For YouTD, fifth order polynomials were chosen to allow a steep growth at later levels. The only missing action is to set the six coefficients of the polynomial. This was achieved empirically by playtesting the game with some towers and checking at which point the game becomes really difficult and at which points the game was too easy. This is the polynomial for setting the creep hitpoints of a specific level:

$$hitpoints(level) := a \cdot level^5 + b \cdot level^4 + c \cdot level^3 + d \cdot level^2 + e \cdot level + f$$

Since four different difficulty levels can be chosen at game start, there must be different polynomials for each of them. Table 7.2 shows the coefficients that were used for the difficulties.

Table 7.2: Coefficients used to calculate the creeps' hitpoints

difficulty	$f\ (const)$	$e\ (x)$	$d\ (x^2)$	$c\ (x^3)$	$b\ (x^4)$	$a\ (x^5)$
beginner	34.8	42.0	3.42	0.0220	0	0.000022
easy	42.0	43.2	3.61	0.0235	0	0.000022
medium	51.6	54.4	3.80	0.0250	0	0.000022
hard	62.4	65.6	3.99	0.0265	0	0.000022

As can be seen from the table, the difficulty influences the lower order coefficients most. On the fourth and fifth order coefficients, difficulty has no influence at all. The first to third order coefficients make the game difficulty increase linearly. The fifth order coefficient is the one that will end the game at some point between level 100 and 120. It is constant throughout all difficulties because the game should end around these levels irrespective of the difficulty chosen.
Thus, playing a low difficulty level provides most advantage at the start of the game, when low order coefficients have the greatest influence. This choice was made so new players have an easy start to learn the game and survive even with suboptimal play. In later levels, the difficulty level matters less, because if a player reaches a high level, he has probably already learned how to play the game and can handle an increased difficulty.
The chart on the left of Figure 7.12 shows the difficulty value according to definition (7.1), which was calculated using the chosen polynomials. This shows the desired difficulty progress: In the first forty levels, difficulty rises only slightly. From level 40 to 100, the difficulty level grows more noticeably. Players will have to use good tower combinations to manage this phase of the game, and it will be the point where less able players will be "sorted out". At level 100, the game is already over four times as difficult as in the beginning. After level 100 the game should sooner or later come to an end. At this point, the fifth order factor takes control and makes the graph rise very steeply. At level 120 even the best players will probably have lost.
However, there is no harsh end, ensuring that a very good player can always try to reach one or two levels more than his current record. It only becomes harder and harder the further beyond level 100 he gets.
The chart on the right of Figure 7.12 compares difficulty levels with the "hard" difficulty level and also shows the desired result: At the game's start, players playing the lower difficulty levels have a significant advantage compared to "hard": When playing beginner difficulty, for example, the starting creeps have only approximately 50% of the hitpoints that "hard" creeps have. The higher the level, the more equal the difficulties become. At level 100 even "beginner" hitpoints will have reached over 95% of the "hard" hitpoints.

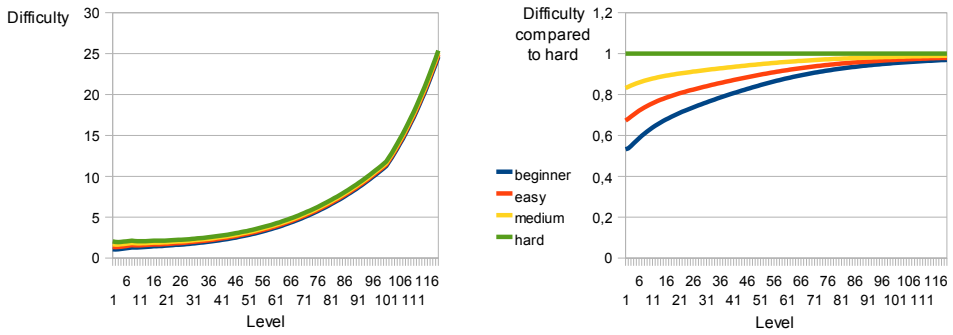


Figure 7.12: *Resulting difficulty per level and its comparison to "hard" difficulty*

After balancing the general creep hitpoints, the different creep categories (cf. page 16) have to be balanced so no player has a disadvantage by having to face a specific creep category.

A basic way to do this would be to distribute the level hitpoints evenly on all creeps of a specific level. Thus, for example, a boss would receive the full amount of hitpoint, whereas normal creeps, of which ten spawn in one level, would receive one tenth of the hitpoints. This value is a good base for balancing, but has to be adjusted further, because different categories influence the difficulty of the level. For example, flying creeps take the direct path to the finish, so towers have less time to shoot at them. Their hitpoints have to be lowered or else they will be more difficult than ground creeps. Also bosses present more difficulty than normal creeps, because normal creeps spawn one after another, thus ensuring that some creeps will always be within a tower's range even if others already walked out of it. If, in contrast, a boss walked out of the tower range, there will be no other creeps for the tower to shoot at.

Rating the different creep categories mathematically is very difficult, so playtesting was used for balancing. From the above-mentioned examples it was clear in which direction the hitpoints of different categories had to be adjusted, only the fine-tuning of the values had to be carried out by testing the game over and over again and adjusting the values if one category seemed imbalanced.

Table 7.3 shows the values which were chosen in accordance with the playtesting. It was established that a boss, for example, would be attributed 60% of the basic HP of a level. Thus he is weaker than the naively suggested 100%, since he presents more difficulty than creeps spawning one after another.

The last attribute of creeps which has to be balanced are their special abilities. Of course, even creeps with special abilities should not be stronger than creeps without abilities, or else a player who receives creeps without abilities randomly will have an advantage. This is achieved by assigning a *balance value* to each ability. This

Table 7.3: Category hitpoint modifiers

category	#spawned	hitpoint ratio per creep (HPratio)	accumulated ratio (#spawned · HPratio)
normal	10	0.1	1.0
mass	20	0.03	0.6
air	5	0.1	0.5
boss	1	0.6	0.6

value will be multiplied with the creep's hitpoints. If the ability is beneficial for the creep, it will be in the interval $(0, 1)$ thus reducing the creep's hitpoints to achieve balance. If the ability has a negative effect for the creep, the balance value should be above 1 so the creep's hitpoints will grow in exchange.
Since the abilities can have complex influences on gameplay, using mathematics will not always be suitable to determine the balance value. Often, the value of an ability has to be estimated or adjusted by playtesting. Nevertheless, some ability values can be calculated. For example, an ability which makes the creeps move twice as fast should yield a balance value of 0.5, because the towers will only have half of the usual time to kill the creeps. Therefore, the creeps should have only half of their usual hitpoints in compensation.

Even if the above-mentioned formulas and concepts were used to balance the game, the balance is still not perfect. This shows that balancing a game is very difficult and complex work. However, according to the players' feedback, the game created within the framework of this thesis seemed quite balanced in comparison to other Tower Defenses. Of course, the feedback reflects a subjective point of view, but objective results are very difficult to obtain. On the other hand, if the players *think* that a game is balanced, they will like its balance, even if objectively it is imbalanced.

7.5 Attracting Contributors

The backbone of any Open Innovation game are contributing users. Therefore, players who like the game should get encouraged to create content. The basic mechanisms to attract contributors were already explained in Chapter 1.1. Especially the point "Immaterial profits are promised, e.g. users will be mentioned on the product to gain some sort of reputation" must explicitly be addressed. YouTD uses various immaterial profits to encourage users to create content:
Contributors get rewarded *contribution score* for uploading new content or perfecting the content of others. The precise numbers rewarded for creating or updating towers or items are subject to changes and thus not mentioned here. Instead, only

the comparisons are explained: Towers grant more score than items since they need more work (a tower model must be assembled). Updating the content of somebody else grants less score than creating an own piece of content. However, updating should still grant enough score so users also get encouraged to perfect the flawed content of others. If too many towers stay flawed and nobody updates them, the score for updating can temporarily be raised.
The contribution score of a user is displayed at the web site next to the user's posts in the forum and in the content discussion threads. There is also a ranking displayed at the statistics page showing the users with the highest contribution score. All contributors are mentioned in the map's loading screen, sorted by contribution score. Consequently, users contributing the most content will get mentioned on top of the screen (also using a bigger font size). This encourages users to contribute more content to get a higher contribution score than others. It also attracts people playing the game to create content, since they want to be mentioned in the loading screen, too.
The contributor's name is also shown in-game in the description of his pieces of content. This attracts new contributors since people playing the game always get reminded that each tower and item was created by players like them. They might finally come to the conclusion that it would be great to create their favorite tower or item, so that their friends see their name when playing the game and building or getting the tower or item, respectively.
The contribution score and the contributors being mentioned in the game are the biggest sources of motivation. Additional promotion campaigns can be started, e.g., naming specific creeps after users that contribute most in a specific period of time.
The next chapter will show if all these mechanisms suffice to attract enough contributors and make YouTD a successful project.

Chapter 8

Results, Feedback and Future Work

At the moment, the project is in its open beta stage. A great amount of content has already been created, many matches have been played and many users have provided feedback. Even if some small features are still missing, most parts of the game are finished or at least in a playable state.
This chapter describes the success of the project as well as the amount and quality of the user-created content and will try to determine if using Open Innovation in game design is a promising concept. Finally, possible future research topics will be discussed.

8.1 Content Received From Users

Even if the project currently is only at the beta stage, a lot of content was uploaded from users. To assess this amount, the number of towers other popular Tower Defenses feature is discussed first. Many badly-made Tower Defenses have only a dozen or less towers to choose from, better games provide approximately 50 to 100 towers. The currently most popular Tower Defense *Element TD* [17] contains 124 towers. Most Tower Defenses stay below 100 towers, some have 100 to 200 towers but hardly any Tower Defense has more than this.
The question can be asked if more towers are always better. On the one hand, more towers will confuse the user, so mechanisms have to be set up to limit the amount of towers presented. Such mechanisms were included into YouTD, too (see Chapter 2.3). On the other hand, more towers usually offer a greater choice of tactics a user can try out and more tower combinations that could lead to nice *combos*[1]. So basically, more towers means more long-term motivation for players, since they

[1] a combo is usually a situation where a combination of towers is much stronger than the sum of their individual power was if used separately. A simple example for a combo would be a tower which deals spell damage and another tower which increases the amount of spell damage the enemies receive.

can try out new tower combinations in every new match. Long-term motivation is a key to game success, so many towers are generally beneficial.
Even if too many towers made the game worse, the total amount of towers could still be limited by picking only the best-rated towers. In this case,"more towers" would mean "more towers to choose from". This would increase the average quality of towers in the game, since more badly-rated towers could be replaced by better ones.
At the current beta release *YouTD beta v0.4*, the game already contains 180 towers and thus more than almost all other Tower Defenses. 377 towers have already been submitted. Since the author was occupied with writing this thesis and working on the missing features and only two other administrators were found, the time was lacking to review and approve more towers yet. As soon as the thesis is finished and the missing features have been added, there will be time to approve the pending towers, thus raising the amount of towers further. Consequently, the Tower Defense will feature significantly more content than *any other* Tower Defense project.
When evaluating these figures, it has to be taken into consideration that this is a non-commercial project without any advertising. An advertised project can attract a greater number of users and thus contain more uploaded pieces of content. In addition, the first beta version of the map was released only three months ago. In a few months, more towers will most probably be uploaded.

The evaluation of the amount of content submitted showed that Open Innovation is a suitable means to create a great amount of content. Consequently, for genres in which the amount of content benefits the player motivation, Open Innovation can be a significant advantage.

Concerning the quality of the uploaded content, most towers met the high quality demands that were set for this project. Only approximately 10% of the submitted towers were of low quality and not further improved by the users so they had to be declined. Many towers had significant flaws, such as major imbalances, or bugs in the script code, when they were submitted for the first time. But the community and the discussion for each tower helped the users to realize and fix these flaws in most cases. Every flawed tower was discussed by the community, no flawed tower ended up without any discussion. The only case towers had to be declined was if the user did not want to invest any more time in fixing and updating them. Some of the declined towers were continued by other users and still made it into the game. This leads to the next fact that this project has proven:

The community helps to perfect the submitted content, even if the quality is low when first uploaded.

It was observed that most users who had created one tower created many others afterwards. The 377 towers submitted were created by only 21 users, which is an average amount of approximately 18 towers per user. Obviously, creating a piece

of content motivates the author to create more content, which is beneficial for the Open Innovation concept.
Finally, the amount of design time "saved" by the submitted content is to be estimated. The creation process of a tower including scripting, assembling the model and introducing necessary changes can be estimated at around 0.5 - 5 hours, depending on the complexity of the tower's concept and model. As a conservative estimate, 1.5 hours is chosen per tower, which would equal around 560 design hours for 377 uploaded towers. For a big project, this is not a huge amount of design time saved.
However, a tower also has to be balanced and perfected. This process can be interminable, and a lot of playtesting might be required to assess and tune the strength of a tower. At this point, the user-created content combined with the community discussing it really saves time.
The average time the perfection of a tower takes is difficult to estimate. From the author's experience[2], many towers need only a few test games and a few hours to be perfected, but there are also some towers which take days and are still not balanced or contain bugs in their script code. As a rule of thumb, perfecting a tower takes at least twice the time needed for its initial creation. This does *not* include playtesting, only the adjustment work. Playtesting often involves playing the game ten or more times and focusing on a small set of towers which have to be balanced. In this case, the testing and perfecting process is entirely accomplished by the community.
Approximately 10 hours are estimated for testing and perfecting a tower, which would equal 3770 hours saved. For small to medium sized projects, 4330 hours of work saved, which equals approximately 2.5 man-years[3], may already be a significant percentage of the development costs. In addition to these figures, the fact that the game is still in its beta stage has to be considered. There are many more towers yet to come and thus the total amount of time saved could turn out to be significantly higher.
Another approach for measuring the amount of work acomplished by the users is accumulating the lines of code of all user-created content. At the moment, around 6000 user-created lines of code are injected into the map, which equals around 33 lines of code per tower. However, since this value does not reflect the work for designing the tower model, balancing the tower and playtesting it, this approach will not be analyzed further.
Note that these figures are only very rough estimates, the actual figures can deviate much. It is not the focus of this thesis to exactly determine the time saved, but rather to demonstrate the dimension.
The advantages of Open Innovation are not limited to saving design time. As the name states, the *innovations* that users contribute are also one of its significant

[2]The author has created and balanced two Tower Defenses before YouTD and has created and balanced the first 30 towers (seed) for YouTD and therefore knows how cumbersome and time-consuming the perfection of a tower can be.

[3]calculating with 220 man-days per year and eight man-hours per day

advantages. For the current project, contributors created towers with very special and fascinating abilities, which the author, who can be considered an experienced tower designer, would have never thought of. Generally speaking, a small group of designers has a much more limited pool of ideas. An open community, in contrast, might come up with completely new ideas, which will make the game really special, presenting a unique selling proposition for a commercial game and thus decide about its success.

No precise observations could be made with respect to items, since designing them was not yet possible in the early beta stage. Uploading items is possible for two weeks now and 43 items have been submitted. The calculations on items are similar to those of towers and not performed here.

8.2 Problems and Possible Solutions

Although Open Innovation proved to be an advantage when designing a game, this concept also created some problems, which are discussed in this section. In addition, possible solutions for these problems are proposed.
The first problem was a "chicken or the egg" dilemma: The game cannot be released without any towers, but most players will only be willing to create towers if they have already played and liked the game. This problem was solved by self-creating a small set of around 30 towers[4] so a first beta version could be played. In addition, three users uploaded about 30 towers before the first version was released. When these persons were asked about their intention, they replied that they liked the Open Innovation concept so much that they created a tower before they knew whether they would like the game or not. In addition, two of them already knew the author as being an author of well-known, high-quality modifications, and thus thought that this game will also become a good one. Consequently, the solution for the "chicken or the egg" dilemma was solved by creating a seed of towers and advertising the game before the first public version was released, since some users are ready to create content without knowing the future game. In addition, being a company with a good reputation among the target audience will help, since people will hold that the upcoming game will be as good as its predecessors and thus create content before a beta version is released.
The next problem is the complexity of the editor. Since YouTD allowed arbitrary scripting, using an object-oriented scripting language, many users without programming knowledge were disappointed. In addition to the scripting, users also had to learn to use the World Editor, which is not a trivial tool either. The problem was partly solved by choosing Warcraft 3 as a platform, since map-making communities already exist for this game with many people who have the know-how to handle the World Editor and the script language. If a standalone game is created,

[4]These towers were called *seed* because they can be considered the seed necessary to "grow" more towers from

the problem can be solved by providing an easy-to-use editor, which a user without any scripting knowledge can understand quickly. This shows that a good and easy-to-use editor is crucial for the success of an Open Innovation game.
Handling low-quality content was also an issue which came up when creating the game. Since the target audience of such a game are young people, many of them might not have the skills to create high quality content yet. Fortunately, it turned out that the community fixed this by itself by discussing and rating the content of others. This ensured that most content was perfected and could be approved. Of course there were some towers that were unacceptable and even after the user was told so, he did not change anything or introduced changes which made the result even worse. There will always be people who do not have the talent to produce a good piece of content, no matter how long they try. Fortunately, only less than 10% of the submitted content was not suitable and was declined based on the negative rating of the community. Although this might have disappointed the creators, it was necessary to maintain the high quality of the game. To solace these users, it was suggested that they post their ideas in the forum, and another user with more talent will implement them. This way, their ideas will not be lost.
A big problem was the administrative work necessary to review and accept or decline the towers. Since the users could create their own script code, which often was hard to understand, reviewing a tower took a lot more time than expected. This is the reason why the two administrators and the author were able to approve only 180 of the 377 towers until today. This problem might be solved if the community grows and more experienced users agree to become administrators in the future.
For future projects, this problem will be solved by creating an advanced editor. Besides being easy to use, such an editor should not allow arbitrary script code, but provide a framework which allows easier checking if the content works as intended. The best solution would be an editor allowing only content which is so trivial that its correctness can be checked by the editor itself. However, this would allow only very simple content, which is not desirable, either. It is obvious that creating a good Open Innovation editor allowing complex content which is still easy to review is highly non-trivial and a possible topic for future research.
A general problem with Open Innovation is that it has to fit the genre, because not all genres benefit from great amounts of content. Before using Open Innovation, it has to be considered if the game concept requires great amounts of content. If not, game creators are better off creating the content themselves, because the workload for providing the Open Innovation framework will exceed the workload for self-creating the content. However, this problem did not affect YouTD, since the Tower Defense genre rather benefits from a great amount of content.

8.3 Success and User Feedback

While the previous chapters featured only the creation process of YouTD, this chapter discusses how people liked the game when playing it. The web site

www.mapgnome.org tracks all games played in Battle.net, so this site can check how often YouTD is hosted, and the figures can be compared to those of other Warcraft 3 maps, particularly other Tower Defenses.
According to mapgnome.org, the game was hosted 513 times per day which is quite a high value for a Tower Defense. It was ranked 17th in Europe and 50th in the world, respectively, in the category "most frequently played". As for Tower Defenses, only Element TD was hosted more often in Europe and only three Tower Defenses were hosted more often worldwide.
The only actions undertaken to advertise the game were presenting it in two community forums (www.wc3c.net and www.hiveworkshop.com) and putting it onto the author's web page www.eeve.org, which also houses a community with around 1000 visitors per day. With no further advertising, the game spread merely over community portals and by players hosting it. This is a very good result, taking into consideration that YouTD is still in a beta stage, and many towers and some features are still missing.
The map also received very positive player feedback in the forum. In addition, some games were hosted with a program tracking the player chat during the game. Some players gave negative feedback because they disliked parts of the game's concepts, but hardly anybody criticised the Open Innovation concept and the user-created content.
In total, 296 feedbacks were tracked. Only three persons doubted that the user-created content would benefit a map, all others either agreed that user-created content is a very good concept or did not comment this issue.
Many players explicitly stated they liked the user-created content and argued that this would bring the game closer to the players and their needs, in comparison to other closed-innovation games where players have no control over the content.

The conclusion is that the concept of Open Innovation earned a high level of player acceptance.

8.4 Conclusion and Future Work

This thesis covered the creation of a prove-of-concept game for using Open Innovation in game design. The genre chosen for this game was the Tower Defense genre. Its basic concepts and common features, combined with features selected for the game, were explained. The decision for using the platform Warcraft 3 and its features were covered and scripting concepts and languages which can be used to create games for this platform were discussed. The creation process of the game and its Open Innovation framework were illustrated.
The results after releasing the game and the development kit to the public were evaluated. It was shown that Open Innovation can be used to gather great amounts of content. Most of the submitted content was of sufficient quality or was perfected by the community to reach sufficient quality. The players liked the concept and

rated user-created content as a generally good idea, since it makes players feel more involved into the game and its innovation process. In addition, the amount of design time that could be saved by Open Innovation was estimated and the conclusion was reached, that much time and thus development costs can indeed be saved.
The general conclusion concerning the game success and the feedback of players and submitting users is that Open Innovation is highly beneficial for game design if applied correctly.

Since this work was a successful prove of concept for using Open Innovation in game design, games using Open Innovation with a great amount of user-created content could be created. Taking into consideration that YouTD is only a modification of an existing game and was created by one person, the results are, of course, worse than for a commercially created game with many developers and serious advertising involved. The success of a game of commercial scope could be evaluated to test if an Open Innovation game can stand its ground against other commercial products. In such a setting, it could also be estimated how much design work and thus how much money was saved due to the user-created content.
In addition, more research could be done to find a suitable editor, allowing users to easily create unique and diverse content, but at the same time enabling administrators to quickly check if the submitted content is working as intended. The ultimate goal would be to find an editor which allows complex content to be created easily and is able to decide completely autonomously if the content is working correctly. However, this would result in significant limitations for the freedom of creation.
For example no Turing-complete script code could be permitted, because no machine would be able to check if the code is working as intended and without bugs. This is proven trivially:
If an algorithm could decide whether the code works as intended, it would also have to know if the code actually finishes execution in finite time. This algorithm would thus solve the *Halting problem* for this code, which is proven to be undecidable for Turing-complete code. □
Even if verifying the code of an arbitrary tower is not possible, at least its balance could be calculated procedurally: A program "plays" the tower's test map, builds a few instances of the tower, and spawns test creeps. It then determines the damage done to the creeps, divides it by the cost of the tower, and compares it to a reference damage-per-gold-spent value, to check if the tower is balanced. Since the tower could be a support building that just makes other towers stronger, the test script should build other towers along with this tower to assess the supportive aspects of the tower correctly. However, such a script would be unable to find hidden combos: There might be a small set of other towers that harmonize so well with the tested tower that it gets way too good. Therefore, such a script would only give a hint for a tower's balance and do the major part of the balancing work, but it could not replace human interaction completely.
Another topic of future research could be the processes running in communities gathering around Open Innovation games. An example would be the question

whether enough administrators can be found in the community to automate the content review and approval process without any need for additional personnel.
Generally speaking, research into the optimization of the Open Innovation processes could be done to attract more users to create more content. A survey and evaluation of players' motivation for creating and uploading content could be beneficial.
A further idea would be using Open Innovation in terrain and game universe creation. For this purpose, a collaborative editor and algorithms allowing to merge the terrain created by different users would have to be developed. In combination with user-created non-player characters and enemies, worlds could be generated with users deeply involved. Such worlds could be used for *Massive Multiplayer Games* where all players act together in one big world.
Another approach could be to allow the users to create content while playing the game. For example, a *Role Playing Game* could allow users to alter their environment while they are passing it.

Using Open Innovation in game design is still an unexplored field of research, with many topics for research remaining uncovered. If this concept proves to be as successful as this thesis gives reason to assume, it could deeply revolutionize the process of designing a game.

Bibliography

[1] Wikipedia, The Free Encyclopedia. *Tower Defense*. http://en.wikipedia.org/wiki/Tower_defense [accessed 12-August-2009].

[2] Blizzard Entertainment. *Warcraft 3: Reign of Chaos*, 2002. http://www.blizzard.com/us/war3 [accessed 15-August-2009].

[3] Blizzard Entertainment. *Warcraft 3: The Frozen Throne*, 2003. http://www.blizzard.com/us/war3x [accessed 15-August-2009].

[4] H. W. Chesbrough. *Open Innovation: The New Imperative for Creating and Profiting from Technology*. Harvard Business School Publishing, Boston, Massachusetts, 2003.

[5] H. W. Chesbrough, W. Vanhaverbeke, and J. West. *Open Innovation: Researching a New Paradigm*. Oxford University Press Inc., New York, 2006.

[6] Wikipedia, The Free Encyclopedia. *Social Commerce*. http://en.wikipedia.org/wiki/Social_commerce [accessed 11-August-2009].

[7] R. Reichwald and F. Piller. *Interaktive Wertschöpfung*. Gabler, Wiesbaden, second edition, 2009.

[8] J. Howe. The rise of crowdsourcing. *Wired Magazine*, June 2006.

[9] sprd.net AG. *Spreadshirt*. http://www.spreadshirt.net [accessed 11-August-2009].

[10] S. Krempl, A. Poltermann, and O. Drossou. *Die wunderbare Wissensvermehrung: Wie Open Innovation unsere Welt revolutioniert*. Heise, Hannover, 2006.

[11] Rosen, R. *Mr. Robot and his Robot Factory*, 1983. http://www.c64-wiki.de/index.php/Mr._Robot_and_his_Robot_Factory [accessed 15-August-2009].

[12] Blizzard Entertainment. *Starcraft*, 1998. http://www.blizzard.com/us/starcraft [accessed 15-August-2009].

[13] Epic Games. *Unreal*, 1998. http://unreal.com [accessed 15-August-2009].

[14] K. "Shrimp" Watson. *UnWheel :: Unreal Tournament 2004 Driving*. http://unwheel.beyondunreal.com/ [accessed 19-August-2009].

[15] JbP. *Tetris mod for Unreal Tournament*. http://www.moddb.com/mods/tetris [accessed 19-August-2009].

[16] J. "geX" Finis. *eeve! Tower Defense*, 2007. http://www.eeve.org/board/viewtopic.php?t=492 [accessed 2-September-2009].

[17] E. "Karawasa" Hatampour. *Element TD*, 2006,2009. http://www.eeve.org/eevetd.php [accessed 2-September-2009].

[18] D. Scott. *Flash Element TD*. http://novelconcepts.co.uk/FlashElementTD/ [accessed 2-September-2009].

[19] Brian "Bryvx" K. *Gem TD*, 2008. http://www.epicwar.com/maps/75783/ [accessed 2-September-2009].

[20] P. Holko and D. Cox. *Gem Tower Defense*. http://www.gemtowerdefense.com/ [accessed 2-September-2009].

[21] Duke Wintermaul. *Wintermaul*, 2005. http://www.epicwar.com/maps/1/ [accessed 2-September-2009].

[22] Hidden Path Entertainment. *Defense Grid: The Awakening*, 2009. http://defensegrid.hiddenpath.com/ [accessed 5-September-2009].

[23] Blizzard Entertainment. *Starcraft 2*. http://www.starcraft2.com/ [accessed 15-August-2009].

[24] Eredalis. *BlizzCon 2009 - Der leistungsstarke Editor von StarCraft II*. http://starcraft2.ingame.de/content.php?c=93964&s=916 [accessed 9-September-2009].

[25] Magos. *The MDX file format*, 2008. http://home.magosx.com/index.php?topic=20.0 [accessed 8-September-2009].

[26] Zipir, BlackDick, DJBnJack, PitzerMike, StonedStoopid, Ziutek. *W3M and W3X Files Format*, 2006. http://www.wc3c.net/tools/specs/index.html [accessed 8-September-2009].

[27] Void. *Tower of Dawn*. http://www.wc3c.net/showthread.php?t=105951 [accessed 7-September-2009].

[28] Void. *Winter Sunrise*. http://www.wc3c.net/showthread.php?t=100832 [accessed 7-September-2009].

[29] Void. *Fjord Kingdom*. http://www.wc3c.net/showthread.php?t=102274 [accessed 7-September-2009].

[30] Blizzard Entertainment. *Battle.net*. http://www.battle.net [accessed 15-August-2009].

[31] MindWorX and many more. *Jass NewGen Pack*. http://www.wc3c.net/showthread.php?t=90999 [accessed 17-September-2009].

[32] L. "Die Backe" Drögemüller. *Mappedia*. www.mappedia.de [accessed 16-September-2009].

[33] V. H. S. "Vexorian" Kúncar. *JassHelper - A vJASS 2 JASS compiler*. http://www.wc3c.net/showthread.php?t=88142 [accessed 17-September-2009].

[34] V. H. S. "Vexorian" Kúncar. *ZINC*. http://www.wc3c.net/vexorian/zincmanual.html [accessed 31-October-2009].

[35] ADOLF, Van Damm. *cJass*. http://cjass.xgm.ru/ [accessed 31-October-2009].

[36] J. "gex" Finis. *Gex's Map Script Interpreter (GMSI)*. http://www.eeve.org/board/viewtopic.php?t=1102 [accessed 7-September-2009].

[37] phpBB Group. *phpBB*. http://www.phpbb.com [accessed 4-September-2009].

List of Figures

2.1 Waypoints in eeve! TD 8
2.2 Mazing Tower Defenses 9
2.3 The scoreboard of YouTD 17

3.1 Art created with the World Editor [27, 28, 29] 22

4.1 A simple hill created with the World Editor 26
4.2 Cliffs in the World Editor 27
4.3 The previously created hill with doodads 27
4.4 Rects and starting units 28
4.5 The object editor 29

5.1 GUI Box to choose actions 34
5.2 A trigger created using the GUI 35

6.1 Use cases for creating an Open Innovation game 44
6.2 Example Open Innovation workflow 45

7.1 The most important structs in the overlay engine 49
7.2 Three steps of creating an assembled tower model 53
7.3 Assembled tower models from YouTD 54
7.4 A tower in the web interface 60
7.5 The element and rarity distribution 61
7.6 The recently submitted towers 61
7.7 Bird's eye view of the map 64
7.8 A lane with environmental objects 65
7.9 The finish at the peak of the pyramid 65
7.10 Towers killing an enemy 66
7.11 Towers engaging a boss 66
7.12 Resulting difficulty per level and its comparison to "hard" difficulty 73

List of Tables

2.1 Damage percentages of armor and damage types 14

7.1 Rarity damage multipliers 69
7.2 Coefficients used to calculate the creeps' hitpoints 72
7.3 Category hitpoint modifiers 74