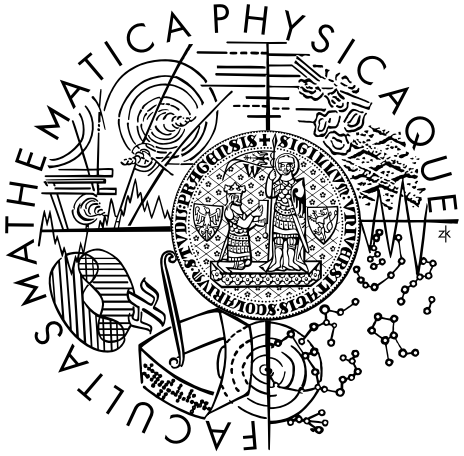
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Jaroslav Jindrák

The Dungeon Throne: A 3D Dungeon Management Game

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Pavel Ježek, Ph.D
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2016

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: The Dungeon Throne: A 3D Dungeon Management Game

Author: Jaroslav Jindrák

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D, Department of Distributed and Dependable Systems

Abstract: Abstract. TODO:

Keywords: dungeon management Lua scripting Ogre CEGUI

Dedication. TODO:

Contents

1. Introduction

Until the release of Dungeon Keeper[1] most well known fantasy video games have allowed the player to play as various heroic characters, raiding dungeons filled with evil forces in order to aquire treasures and fame. In Dungeon Keeper, however, we join the opposite faction and try to defend our own dungeon (along with all the treasures hidden in it) from endless hordes of heroes trying to pillage our domain. Although we can still play the original Dungeon Keeper today, we cannot change its data or game mechanics in any easy way so this thesis aims to recreate the original game with an easy to use programming interface, that will allow such modifications.

1.1 Dungeon Managment Genre

Dungeon Keeper was the first game released in the dungeon management (DM) genre and since our game is going to be based on Dungeon Keeper, we should design it with the elements of its genre in mind. Since the definition of this genre has to our knowledge never been formally documented by the creators of Dungeon Keeper, we are going to create a list of basic elements of the genre based on the gameplay of the original game.

In Dungeon Keeper, the player's main goal is to build and protect his own base, called the dungeon. They do so by commanding their underlings (often called minions), whom they can command to mine gold, which is a resource they use in order to build new rooms and cast spells (in the game's sequel[2], mana was added into the game as a secondary resource used for spell casting), which could be researched as the game progressed. They would then use creatures spawned in their buildings as well as their own magic powers to fight intruders in order to protect their dungeon. From this brief gameplay summary, we can create list of the most basic design elements, which can be found in dungeon management games:

(E1) Resource management

(E2) Dungeon building

(E3) Minion commanding

(E4) Combat (defensive)

(E5) Player participation in combat

(E6) Research

TODO: Think of a way to use the rest of this chapter as a conclusion.

The dungeon management genre combines elements from the construction and management simulation (CMS) genre [1], from the real-time stragegy (RTS)

[1] Bullfrog Productions, 1997

[2] Dungeon Keeper 2, Bullfrog Productions, 1999

genre [2] and from the god game genre [3]. Table 1.1 shows how these elements relate to the three genres the dungeon management genre combines:

	RTS	CMS	God game
Resource management	✓	✓	✓
Base building	✓	✓	✓
Unit commanding	✓	x	x
Combat	✓	x	✓
Player participation in combat	x*	x	✓
Research	✓	✓**	✓

* Not true for all RTS games, e.g. in Age Of Mythology [4], the player has one spell per age, which can be cast in battles.
** Not true for all CMS games, e.g. in Cities: Skylines [5], new buildings are unlocked when the player's city reached a certain population size.

Table 1.1: Dungeon Management elements in other game genres.

As we can see, the dungeon management genre seems more like an RTS/God game hybrid since the elements from the CMS genre are also present in these two. What it takes from the CMS genre is the end goal of the game. In RTS games, the goal generally is to defeat all enemies via combat or politics. In god games, the goal is to convert or kill all infidels. In CMS and dungeon management games, the goal is simply to build and maintain. The player in dungeon management games builds his dungeon and places all kinds of traps and obstacles along with scary monsters to protect his treasure, which is similar to building a city and reacting to natural disasters or power outages.

1.2 Modifiability in Games

One of our basic goals is for our game to be modifiable, which means to provide tools (often called modding tools) to our players that will allow them to create modifications (often called mods) that other players can install and which can change or add elements to the game. This can increase the replay value of the game as after finishing it, more missions, characters, game mechanics, abilities, items or even game modes can be easily downloaded and installed from internet. Since we want our game to be modifiable, we should allow our players to change most (if not all) of the game's data, artificial intelligence and abilities of all minions, creatures and enemies and also give them the ability to edit saved states of the game, which would allow them to create new maps.

Modding tools can generally be of two types: (1) an editor (like a world editor in Warcraft 3 [6] and Starcraft [7]) or (2) an application programming interface (API) that can be used in a scripting language (used in e.g. Starbound [8] allows its players to create modification either by editing JSON[3] files or by using its Lua[4] API).

(1) While an editor that would allow the players to change the game's data and its save files would be easier to use due to its point and click nature, the

[3]JavaScript Object Notation, a lightweight data-interchange format designed to be easy for humans to read and write while being easily parseable by computers.[9]

[4]A fast, lightweight and embeddable scripting language.[10]

development of such tool is unfortunately beyond the scope of this thesis and would probably have to rely on an API (for code generation) anyway. For these two reasons this option won't be used, but creates a possible future extension of the game, which would extend the modifiable part of the game to even bigger audiences.

(2) Creating an API that can be used in a scripting language (e.g. Lua) may, besides making the game modifiable, lead to faster development if the main part of the game is written in a compiled language as parts of the game that are not performance critical can be written in a higher level, dynamically typed and interpreted language with an easy to understand syntax. These characteristics of scripting languages can make mod creation easier to non-programmers [11] and support rapid iteration [12] (as they do not require recompilation) and are the reasong why we are going to use this option.

Now that we have decided on the technical aspect of the modding support in our game, we need to decide which parts of the game we will allow the players to modify. An example of an easily modifiable game is Minecraft [13], a 3D sandbox game in which the player has to survive in a procedurally generated world. They build a house, craft tools, mine for materials and fight off enemies during the night. As we can see the game is very data oriented, with different crafting recipes, enemies, blocks to build with and ores to mine being the main part of the gameplay because the actual goal (to survive) is quite simple. This allows the mod creators to possibly prolong the time spent with the game by several hours (per mod) by adding these new items or by creating new custom maps that offer new challenges to the player.

Aside from adding items and entities, entire new games can be created within a modifiable game. In an interview on the server PCGamer [14] the lead developer of Minecraft, Jens Bergensten, said: *"In the recent snapshot we've added something called the command block. When it's triggered, you can teleport players to a certain position, or change the game mode. So this user called Sethbling has created a Team Fortress 2 map for Minecraft. You can choose a class, it has control points, and everything works like TF2. It's quite amazing."* (More information about Team Fortress 2 can be found at [15])

To allow something the Team Fortress map in Minecraft, our game should have a save feature which will create levels in a format that will allow later modification of the saved state of the game.

In conclusion, we can see that the ability to modify a game can help said game to grow even when its development has stopped or is focused in different areas (e.g. security, stability). Since we want to give this ability to the players of our game, our modding API should allow them to change most of its data, including minions, enemies, buildings, spells, artificial intelligence and others, but the game on its own should be fully featured, offering enough of these entities on its own so the players do not need mods to actually play the game.. Additionally, since our game (like Dungeon Keeper) will have scripted waves of enemies attacking the player's dungeon, the also should allow our players to alter the wave composition and delays.

TODO: Ask PJ: I talk about gameplay, should I explain the term (oxford dict definition, could be in a footer for example)?
(http://www.oxforddictionaries.com/definition/english/gameplay;

"The features of a computer game, such as its plot and the way it is played, as distinct from the graphics and sound effects.")

To create an easily modifiable game we need to realize that both mod creators and mod users do not have to be experienced programmers, so it's very important to have both easy to use modding tools and easy way to install mods.

1.3 Thesis Goals

The main goal of this thesis is to design and implement a modifiable 3D dungeon management game using the design elements **(E1)** – **(E6)**.

In addition to the main goal, the game should complete the following list of goals:

(G1) The game has to be a full competetive product, not a prototype.

- **(G1.1)** It has to be performant, achieving high framerate even on low end computers.
- **(G1.2)** It has to offer full single player experience, with scripted enemies and a chance to both win and lose.
- **(G1.3)** X It has to contain a variety of entities, spells and buildings even without mods.

(G2) X The game has to be highly modifiable, providing an easy to use modding interface for players.

- **(G2.1)** X The mod creators must be able to create new entities, spells and buildings and to change most of the game's data.
- **(G2.2)** X They must also be able to alter the game progression by defining enemies that spawn and delays between them.
- **(G2.3)** The game has to have a save feature which allows the mod creators to change the save files to create custom levels.

(G3) The mods for the game have to be easily installable even by players without any programming knowledge.

2. Problem Analysis

2.1 Programming Language

- talk about the need of fast language that can be used to create 3D games with the ability to be scripted using an embedded language (or itself)
- talk about the benefit of runtime code execution (testing)
- talk about the necessity of scripting without compilation

2.1.1 C++

- industry standard
- powerful and fast
- modifiability can be added through embedded languages
- lots of materials

2.1.2 Lua

- slower than C++, faster than other scripting languages
- can be scripted by itself
- there are no 3D rendering libraries
- high portability

2.1.3 Java

- Minecraft proved that it allows highly modifiable game development
- it also shown that the performance can be a big problem
- high portability

2.1.4 C#

- explain that the only reason this was not picked over C++ is that I did not have much experience with it when I started working on the game

2.2 Engine Design

- maybe mention that the goal to make an engine and that's why UE or Unity weren't used
- say that since the game is to be modifiable and extensible, run time entity creation is a must (both for modding and testing)
- might be good mention the choice before explaining why the different options were/were not chosen so that the reader knows

2.2.1 Inheritance based (wording?)

- explain what is meant by the title (classic OOP design)
- this kind of engine would also have to be planned a lot (to avoid inheritance hierarchy hell)
- ?say that it's not as performant as the other two?

2.2.2 Component based (wording?)

- define the Component design pattern (with references to the gamedesgin-patterns.smth book)
- mention that this is very similar to the component based implementation both Unity and UE4 use
- mention its relationship with both ECS and the inheritance based design

2.2.3 Entity Component System

- define the ECS design pattern
- mention how well it supports run time entity creation and modification
- mention ease of development this model creates
- mention ease of entity access (through ID)
- talk about the difference between Component and ECS patterns
- ?mention cache friendliness? (pro: well, it exists, con: not used that much in this game)

2.3 Scripting Language

TODO: Add more possible scripting languages, possibly C++ using dlls?

- should be fast, easily embedded into C++ and also easily used by modders

2.3.1 Custom Language

- challenge, experience, can be created to suit the project
- slow development, big task
- probably slow and buggy

2.3.2 Lua

- industry standard
- maybe talk about some of the different games using it
- ease of embedding into C++
- no need to create anything, just connect it to C++

Bindings

- talk about most bindings being either dead, obsolete or very archaic
- those that are up to date are mostly focused on OOP in Lua, which isn't needed in this project
- creating new binding just for this project is easy, fast and the resulting binding will only contain the features that are needed

2.4 Libraries

- just mention something about reinventing the wheel and the areas for which 3rd party libraries were used

2.4.1 3D Rendering

TODO: Merge OpenGL and DirectX together as low level API and add more high level libs.

- almost any game will have to use some library for this as going without would take years
- ??I only though about these in the beggining, should I add others just for comparison??
- mention that portability of the rendering library is much more important than for example the GUI library, since it's integrated deep into the engine while the GUI is not

OpenGL

- talk about what it is
- would allow later Linux port
- too low level, would slow the development process

DirectX

- talk about what it is
- similar problem to OpenGL (too low level)
- would prohibit eventual port to Linux

Ogre3D

- talk about what it is
- can wrap both OpenGL and DirectX
- higher level, lots of stuff already implemented
- describe basic usage of the library (entities, meshes, scene graph, ...)

2.4.2 Graphical User Interface

- main requirement is Ogre compatibility
- theoretically does not have to be portable as it can easily be changed (as it's not integrated into the engine)
- other requirements: graphical editor, ease of use, high level, all the widgets needed (buttons, scrollbars, editboxes, labels, ...)

Ogre Overlay

- talk about what it is
- too simple, would require for the widgets to be created during the game development process which would take a lot of time

CEGUI

- talk about what it is
- used to be bundled with Ogre
- easy to use, clean interface, good design
- ??? genrally advised by the Ogre community
- ??? used for Torchlight which proved what can be done with it

- graphical editor and existing skins
- ??I only though about these in the beggining, should I add others just for comparison??

2.5 Game Design

- go over which features of the dungeon management genre will be used and why
- possibly talk about how they could be done? or leave this to implementation?

2.6 Algorithms

- dunno about this section, not many known or big algorithms used in the game

2.6.1 Pathfinding

- talk about the problem of pathfinding
- talk about node graph vs vertex graph (or hows it called)???
- after mentioning node graph, give reason for grid form and 8 neighbours (bcuz of the view)
- also talk about the inclusion of portals in pathfinding and problems that have risen with it

Breadth-first Search

- just mention what it is and why it's not good (uniform cost of graph edges)

Dijkstra

- talk about what it is and why it's not the best alternative
- explain that it might be worth using when performance is not critical and we absolutely need the best path

A*

- talk about what it is (possibly also define heuristic as it should be seen for the first time now)
- mention that it's probably the most used one in games (provide examples?)
- ??mention how it goes well with e.g. portals??

- maybe go into detail about the default heuristic (PORTAL_HEURISTIC) which allows portal pathfinding (include previous iterations, portal component, problem with portal chains and possible fix)

Solution ??

- maybe this should be in the A* part as it's the only one actually implemented?
- describe the generic pathfinding system implemented in the game, switchable algorithms, heuristics and path types (define path type), algorithm and path type interface ...

2.6.2 Level Generation ??

- talk about how complex level generation isn't really needed due to the nature of the game (you dig your own hallways so only need gold distribution)
- mention that similar generic system to the pathfinding one is used
- ??probably go through the algorithm in pseudocode?? or explain it only?? if so, probably go through the part with the gold only and just mention the rest??

2.7 Serialization

- talk a bit about game state serialization in general
- mention that since the game should be extensible, manual save game editing without the need to write big editors would be nice for testing and possibly the creation of simple mod maps (and the save editor - if ever created - would be easy to make)

2.7.1 Binary

- mention that it's the most used (maybe give examples like UE4 save system etc.)
- it's also the most compact as it just meshes data together in binary
- lots of libraries (like boost)
- editing would require complex editor and does not allow code execution in the save file so additional save file functionality (like changing the wave system table, executing custom commands) would need to be made in C++ in a generic way, which would be bad for mods

2.7.2 XML

- mention losts of xml libraries
- talk about how it allows easy data change, which in a data driven game like this one means quite a lot
- but explain that it's bad because it disallows code execution

2.7.3 Lua

- since the game needs an extensive modding api with all sorts of setters/getters, all that's needed is to write serialization of components into a sequence of the API calls
- easy to implement, load is basically executing a script
- easy to modify using a text editor
- !!allows code execution to create special levels
- say that it's limitation is level size, e.g. 256x256 save file can be big (200MB) and very slow to load (10-20sec), but on the allowed level sizes (10-64) the load is almost instant and the resulting files small (16x16 is around 650KB)

2.8 Spells

Here just compare the original idea of having the spell system in C++ and the current version of defining spells fully in lua and C++ just call callbacks basically.

3. Developer's Documentation

- give some basic overview of the project
- say how to tell my functions and Ogre/CEGUI/Lua functions apart
- maybe mention why the camelCase notation of Ogre/CEGUI was not used
- mention what the compiled binary needs to run (?list dlls?)
- say how to change the paths to resources and scripts directories

3.1 Engine (?Layout)

- describe Component - System - Helper - LuaInterface - ... relationship
- possibly create a picture
- describe the "workflow" of the game? or leave it to other sections?

3.2 Game

- !ask PJ: do I have to describe functions even if they have documenting comments?
- describe how the game is initialized
- explain OIS and Ogre callbacks (+ parent classes)
- brief description of its members vs just mentioning them vs ignoring them as they will be explained later?
- level creation (new_level && generate_empty_level)
- throne monitoring? (keeping throne ID so there is only one target of the enemies)
- possibly add a subsections describing Ogre and CEGUI initialisation

3.2.1 State Design Pattern

- describe the pattern
- describe the states of the game
- explain why the pattern is NOT used

3.3 Components

- go through components one by one with UML-like pictures describing theis members and purpose
- explain why strings are moved?
- explain why components need default values for every parameter of their constructor?
- go through some more complex concepts of the components (align states, maybe forward reference blueprints, etc.)

3.4 Systems

- define systems
- mention common base class and the reason for inheritance (list of systems...)
- describe the component-system relationship (none, system-component, system-components, systems-component, ...)

3.4.1 CombatSystem

- describe the update logic
- describe entity querying
- describe effect application
- talk about the fabulousness of the query & effect design!
- talk about the run away mechanism?
- talk about runtime projectile creation?
- talk about line of sight and LoS wrt BB?

3.4.2 EntitySystem

- describe the update logic
- describe cleanup (and its delay and why it's necessary)
- id system (back then vs now)
- entity creation process
- component containers
- describe component manipulation (add, load, set, delete, delete now, ...)
- mention entity registration for the entity creator window

- describe function array (reason, jmp table bonus, etc.)
- explain the macros!
- describe why constructors are delayed (so that the lua function where it's create continues before constructor call)
- maybe talk more about component loading from lua? or leave that to the scripting part? or never mention it?

3.4.3 EventSystem

- describe the update logic
- describe event persistence/destruction
- describe how to make event one (successful) pass only
- targeted/area events and area event expansion
- delete component vs delete entity discussion
- describe handling (fixed C++ vs Lua)
- this should be first time we encounter update periods and time multipliers, explain!

3.4.4 GridSystem

- describe the update logic (+freed/unfreed and Grid relation)
- talk about grid graphics used for debugging?
- explain structure placement
- describe alignment checks

3.4.5 TaskSystem

- describe the update logic
- talk about busy state, processing tasks and checking if task is complete
- probably only forward reference the Lua part? or talk about it right here? !ask PJ!

3.4.6 WaveSystem

- describe states and maybe mention why the State design patter wasn't used (too simple)
- describe the update logic of the different states (-¿ countdown, spawning, chilling)
- talk about the spawning mechanism, blueprint vector, spawn nodes
- talk about wave entity monitoring and wave ending
- talk about the wave table
- talk about the wstart and wend callbacks
- talk about endless mode
- possibly explain how to write a wave? or add that to the scripting part?

3.4.7 Miscellaneous Systems

- here explain in alphabetical order the remaining systems
- say why these are not as important (that is, not important to be described) as the others

AISystem

- describe the update logic
- mention update forcing (in relation to the update period mentioned earlier)

GraphicsSystem

- say how it's supposed to maintain all manual graphics logic but atm only does explosions (since it's the only manual graphics object)
- describe the update logic

HealthSystem

- describe the update logic
- mention configurable regeneration period

InputSystem

- describe the update logic
- explain initial purpose and why it's not used atm

ManaSpellSystem

- describe the update logic (both mana regen and entity spell casting)
- possibly talk about the relationship between player spells and entity spells

MovementSystem

- describe the update logic
- talk aboud move and checked_move (and can_move_to) and why checked is not used but can be from lua (and was intended for 1st person mode)

ProductionSystem

- describe the update logic
- explain how products are spawned and placed

TimeSystem

- describe the update logic (mention the multiple comps updated)
- explain time event handling (and use)
- explain time event advancements

TriggerSystem

- describe the update logic
- explain how triggering works and why it works that way (factions)
- explain the general trap concept?

3.5 Helpers

- describe what helpers are and how are they used
- forward reference the not-singleton status of EntitySystem?
- explain why they were created (use in Lua and optionally in C++ for better code readability)
- explain why they are slower than direct component manipulation
- explain component-helper relationship
- mention the general structure of a helper
- maybe talk about why they are namespaces and not classes? (+ namespace -¿ allows static class without any change basically)

3.6 Script

- talk about how it works (it's basically a wrapper facade)
- mention lpp::Exception
- explain the Lua C API (only informative? add a new section with a tutorial? tell the reader to read the Lua Programming Language 3rd edition book?)
- maybe go through some of the more complex functions?

3.7 LuaInterface

- talk about why it's needed (Lua needs static)
- talk about why it's centralised in one class?

3.7.1 Initialising the API

- explain the C++ function binding process
- explain the Lua module hierarchy
- explain other aspects besides function binding

3.7.2 Extending the API

- explain the general body of the interface functions (stack, return etc.)
- (super) small tutorial on how to add new interface functions

3.8 GUI

- talk about the GUI hierarchy
- explain initialization
- explain CEGUI button binding? maybe in general CEGUI manipulation?
- explain save/load file listing
- mention the GUIWindow class, base class of the following GUI windows
- explain why it has escape_pressed handler and does not use CEGUI handlers for that (would be needed for all subwindows, too much a hassle - also would probably disregard the conditional closing)

3.8.1 Console

- talk about how awesome it is during debugging
- explain how execution and printing works (+ the history concept?)
- explain why it does not execute lines but multi lines after the execute button is pressed

3.8.2 EntityCreator

- mention that currently it is used for entity placement and the creation part is supposed to be a graphical way to make entities that is not currently in the game
- explain how it works
- explain how it's used for testing
- maybe mention that this is where the registered_entity from EntitySystem is used

3.8.3 EntityTracker

- explain how tracking works
- talk about why it's useful
- mention the upgrade, exp convert and delete functionality?

3.8.4 ResearchWindow

- explain initialisation, how it works
- explain dummy_unlock and its relationship with serialization

3.8.5 Miscellaneous Windows

- say that these are generally not that complex and quite straight forward

BuilderWindow

- talk about building registration (how it relates to unlocking)
- talk about the assembly line (also used in spell casting window) and why it's good (lack of images -¿ buttons need to be big and readable)
- don't forget to mention that this is just a graphical front end to EntityPlacer (with price management etc.) as is EntityCreator (though that one ignores price, and has all unlocked even enemies)

GameLog

- just mention that it is basically the ingame chat posting info for the player
- also maybe again mention history (same as console)?

MessageToPlayerWindow

- talk about button renaming and action assignment
- then mention how to use this as a whole

OptionsWindow

- talk about actions, key bindings and video settings (+ how they are done?)

SpellCastingWindow

- talk about spell registration (how it relates to unlocking)
- just mention the assembly line (already explained in BuilderWindow)
- talk about the relationship with the SpellCaster and how spell casting works on this side (simple invoking of the spell caster and marking the spell as active if needed)
- don't forget to mention that this is just a graphical front end to SpellCaster basically

TopBar

- just say something about its purpose (monitoring of resources)

3.9 SpellCaster

- discuss the spell casting concept in the game
- mention the spell types
- explain why it's so damn cool (ease of spell creation, run time spell creation)
- forward reference the Lua spell structure, which is explained later in Scripting

3.10 Utilities

- just say that these are classes that are not directly part of the game world but are used as background tools that support the game

3.10.1 Camera

- talk about how it allows free/nonfree mode, resetting and backups
- maybe talk more about the free nonfree mode and how to toggle them (both keybind and command)

3.10.2 Effects

- talk about how awesome they are and allowed the creation of extensible effect application framework in CombatSystem
- explain how to write one
- mention that they are used as template arguments and as such should conform the given interface (also explain that)
- maybe ilustrate their structure on one of them

3.10.3 EntityPlacer

- explain its purpose and how it works
- explain why only the graphics component data is used (so it's ignored by the game world and serializer, etc.)

3.10.4 LevelGenerator

- explain their purpose and how to write one
- mention the cycle count constructor parameter
- mention the default tables in config (and that it's easy to add new ones)

RandomLevelGenerator

- say that this is just naive algorithm for a pseudo random level generation using number of neighbours that are gold deposits to determine if a gold deposit should be spawned

3.10.5 RayCaster

- explain that I'm a noob that stole other programmer's idea for this from the Ogre wiki
- explain why it's needed (polygon precision ray casting -¿ half cubes, without it only bounding boxes are checked and free space is impenetrable)

3.10.6 SelectionBox

- explain its purpose and how it works
- talk about single/area selection and multi selection using shift

3.10.7 Conditions

- explain their purpose, how they are used in querying and effect application
- mention that it's a good way for extensibility as new can be easily implemented
- mention that they are often used as template parameters and thus need to conform an interface (but not by inheritance)
- talk about the interface
- maybe demonstrate on an example

3.11 Pathfinding

- recapitulate form analysis that a modifiable function was used that allows different algorithms, heuristics and path types
- mentions its parameters (destruction, add path are primary!)
- mention how destruction works
- mentions why path addition is optional (for checks)

3.11.1 Grid

- explain its purpose and implementation
- mention that it only provides a set of grid related operations on a set of IDs it's provided (and assumes those are grid)
- mention random node placement and entity distribution
- mention free node list for ease (and speed) of access
- explain graph creation and linking
- explain general usage

3.11.2 Algorithms

- explain how they should be implemented (what functions, as they are used as template parameters)
- mention that A* is the only currently implemented
- mention portal implementation
- mention difference in complexity (component lookups)
- mention any other differences from a general A*

3.11.3 Heuristics

- explain how they are used and why they are inheriting a base class while other functors are not (state needed for some -¿ run away heuristic)
- maybe demonstrate on an example as the code is small?

3.11.4 Path Types

- explain what these things are and when they are used to check if the algorithm should stop
- explain their effect on the A* algorithm

3.12 Serialization

- talk about how easy component serialization is and how to create template specializations for new components
- show an example?
- explain the whole saving process, ents to be destroyed, unlocks, player, grid etc.
- explain the loading process

3.13 Player

- just say that it is used as a resource bank, keeping track of gold, mana, units etc
- mention that it also holds the starting unlocks used for new games (saved there during the game initialisation)

3.14 Singleton Design Pattern

- explain what this stuff is, its pros & cons
- mention that the main reason for its use in this game is having one instance, not that much global access

Script

- mention how two Lua virtual machines cannot communicate and data are bound to C++ (via EntitySystem) so it's completely unnecessary to have more than one Scripting engine

GUI

- why would we want to have two GUIs?
- mention that here is the global access very good for testing?

Player

- why would we want two sets of player resources?
- serialization preserves them and shown is only the main set (also used is only the main one)

Grid

- explains how it only operates on a set of given IDs that are made during graph creation (+ mentions it would be easy to implement their switch) so there does not need to be more than one pathfinding grid
- also mention that due to the nature of the game levels, it would be nonsense to have two grids

EntitySystem

- explain how even though it's getting passed around a lot (mainly Helpers), there could be a reason to use more than one EntitySystem instance (like a backup for example) and thus the Singleton pattern might not be the best choice
- maybe compare it to the 4 previous classes

3.15 Extending the Game

- small sequence of tasks that are needed for a new feature addition (that is component, system etc.)
- idea: show how a TargetedComponent would be implemented, allowing handlers for targeting/untargeting and how this could be used to create a chess mod

4. Scripter's Documentation

- talk about the API, possibly mention some API reference text file that I should make as an attachement

4.1 Initialization

- talk about the config.lua & init.lua duo
- explain different options in config
- explain how scripts packs are added to init
- talk abut the core.lua file of each script pack and what it should do (probably with an example)
- explain how to add a new script directory
- explain how to override some behaviour
- explain the new level callback and for what it can be used (+ the return value meaning)

4.2 Entity Representation

- explain the structure of a script representing an entity (with a small example)

4.3 Blueprints

- explain what is a blueprint, how to make one and how to use it
- explain why it's a table and not just a function and why the functions in different blueprint should have different names

4.4 Research

- explain how to add new unblocks and modify existing ones

4.5 Spells

- explain the structure of a spell table and how to make new spell
- mention again the spell types and the difference they make in the spell table function implementations
- mention that spells can be made during run time for testing purposes

4.6 Creating a Mod

- provide a simple tutorial on how to make a new mod
- possibly the tower defense one?

5. User's Documentation

TODO: Check if game introduction should be here for the player. Possibly add system requirements.

5.1 Installation

- just simply go over the installing process
- might be nice to create an installer using something like InnoSetup!
- maybe include a note on how to install a mod?

5.2 About the Game

- say what it is
- explain basic elements of the game (building, commanding units, gold, man, spells, research etc.)

5.2.1 Goal of the Game

- explain how waves work and how to win/lose the game

5.3 Controls and Options

- go over default options and how to rebing them
- go over window mode & resolutions
- mentions what the APPLY button does wrt key bindings

5.4 Research

- explain the concept in depth
- ?go over individual research unlocks? (maybe just enumerate each row and say if it's a building, spell, effect etc and mention where are these explained)
- effects (double production, kill all, level up etc) might be explained here as they wouldn't get covered in buildings or spells

5.5 Spells

- explain the concept in depth
- explain how to use the spell assembly line
- explain how to use different types of spells
- ?go over individual spells?

5.6 Buildings

- explain how to build (assembly line already explained in spells)
- go over what the buildings in the game do

5.7 Units

- simply go over the production process
- maybe explain the different types of units in the game

Conclusion and Future Work

Bibliography

[1] Wikipedia. Construction and management simulation genre. https://en.wikipedia.org/wiki/Construction_and_management_simulation. [Online; accessed 2016-04-23].

[2] Wikipedia. Real-time strategy genre. https://en.wikipedia.org/wiki/Real-time_strategy. [Online; accessed 2016-04-23].

[3] Wikipedia. God game genre. https://en.wikipedia.org/wiki/God_game. [Online; accessed 2016-04-23].

[4] Age Of Mythology. http://www.ageofempires.com/news/games/aom/. [Online; accessed 2016-04-23].

[5] Cities: Skylines. http://www.citiesskylines.com/. [Online; accessed 2016-04-23].

[6] Warcraft 3. http://us.blizzard.com/en-us/games/war3/. [Online; accessed 2016-05-09].

[7] Starcraft. http://us.blizzard.com/en-us/games/sc/. [Online; accessed 2016-05-09].

[8] Starbound. http://www.playstarbound.com. [Online; accessed 2016-05-09].

[9] Javascript object notation. http://www.json.org. [Online; accessed 2016-05-09].

[10] Lua. http://www.lua.org. [Online; accessed 2016-05-09].

[11] Garage Games. Why Use Scripting, Torque Engine documentation. http://docs.garagegames.com/tgea/official/content/documentation/Scripting%20Reference/Introduction/Why%20Use%20Scripting.html. [Online; accessed 2016-05-10].

[12] J. Gregory. *Game Engine Architecture*. 1st Edition. A K Peters/CRC Press, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL, 2009.

[13] Minecraft. http://www.minecraft.net. [Online; accessed 2016-04-23].

[14] PCGamer. Future of Minecraft. http://www.pcgamer.com/the-future-of-minecraft/, 2012. [Online; accessed 2016-04-23].

[15] Team Fortress 2. http://www.teamfortress.com. [Online; accessed 2016-04-23].

Attachments