AS-0.3301 Project Plan: Tower Defense

Joel Pitkänen <joel.pitkanen@tkk.fi> [redacted],
Joonas Nissinen <joonas.nissinen@tkk.fi> [redacted],
Ilari T. Nieminen <ilari.nieminen@tkk.fi> [redacted]

08 November 2010

Contents

1 Introduction

This document is a project plan for a tower defense game, to be developed during the fall of 2010 for the course AS-0.3301 (C++ programming). The following sections describe the high-level features and the approximate architecture of the game.

2 Specification of requirements

This section describes the basic features that the game will have (core features) as well as features that will be added if there is enough time (additional features).

2.1 Core features

The game will fulfill the minimum requirements as specified in the project description as well as some additional requirements:

- Mouse input with keyboard shortcuts
- Multiple enemy types with different properties (hit points, speed)
- Multiple upgradeable tower types: direct, splash and special damage
- Easy-to-read, easy-to-edit format for maps
- Towers can be placed at any time
- Dynamic paths for enemies
- Multiple terrain types
- Player profiles, player progress

2.2 Additional features

A subset of these features will be implemented, depending on how much time the core features will take.

- Multiple entry and exit points for enemies
- Fast-forward: When nothing interesting is happening, the player may speed up the game

- Checkpoints: Player may return to a checkpoint instead of having to start the level over in case of seriously suboptimal tower placement.
- Different firing strategies for towers: Sort the enemies by several properties: shortest remaining path to goal, speed, hit points, maximum hit points.
- Tower statistics: See how much damage each tower has inflicted
- Scrollable maps
- Graphical map editor

3 Program architecture

This section describes the overall structure of the game.

3.1 Class structure

The class structure is shown in Figure 1.

Map

Methods for reading (and writing?) maps, keeps track of towers and waves of enemies.

Tower

Virtual base class for towers

Projectile

Visible graphical ammunition for some towers

Mob

Virtual base class for mobs

Timer

For handling game time / real time (predictable passage of time inside the game)

Player

Player profile stores the progress and settings

GameState

Base class for gamestates. A gamestate is essentially one screen of the game (e.g. MainMenuState, GameplayState)

GameStateManager

Manages a LIFO stack of gamestates. Updates the topmost state on the stack.

Renderer

Draws the map and everything contained in it (towers, mobs etc.)

Camera

A movable camera. Used by the renderer to draw objects relative to camera coordinates.

3.2 Data structures and algorithms

A priority queue to keep track of game events (arrival of enemies, damage to enemies)

The map will have an associated graph, which will be used to determine the shortest path to the goal for enemies using a shortest-path algorithm such as Dijkstra's algorithm.

3.3 Persistence and file formats

Settings and player progress will be stored in an SQLite database.

The maps will be stored in an easy-to-edit format in text files, at least unless a graphical map editor will be implemented.

3.4 Possible problems

If the number of enemies on the map is large, one possible bottleneck is the dynamic path calculation, which would be done for all enemies at the same time (whenever the player places a tower that might interfere with

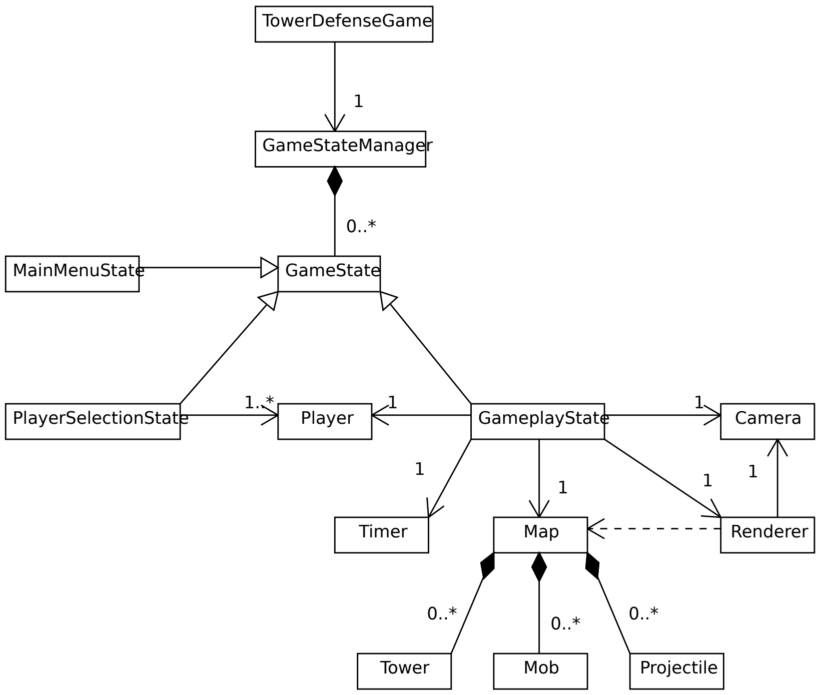

Figure 1: An overview of the relations of the classes

their route); this could cause a noticeable slowdown. The path-finding could be placed in a separate thread to allow faster processing whenever multiple cores or processors are available, or the time allotted to path-updating could be limited in some fashion. Also, the target selection for towers must be implemented in a sane way, so that only a subset of enemies are considered as possible targets.

The role of projectiles is not yet completely determined. For damage calculation, it would be easiest to determine the time of the hit when the projectile is fired and then just update the location of the projectile to match the reality. If projectiles are allowed to miss, the situation becomes a bit more complex, as the aiming has to be done in a smart manner at least if some projectiles are slow.

4 Task sharing

Discussion over implementation issues during coding will be mostly done over IRC. Coding sessions for the whole group will be organized if personal schedules allow.

The rough division of tasks is as follows:

- Joel: User interface, graphics
- Joonas: Graphics, game data systems
- Ilari: Game logic

5 Testing

Automated unit tests will be made for individual parts of the program. These will especially help in making sure that the corner cases are handled correctly. Making sure that individual parts of the program work before proceeding saves time from future tests as you have less lines of code to check for possible errors. Features will be tested in practice after their implementation to make sure that they function as planned.

Certain parts of the program require special attention with testing. For example the map file reading has to be tested rigorously, as users can provide arbitrary files to the program and so it must be able to handle any input gracefully.

As all the individual pieces start to merge together and we have at least a playable prototype, manual testing of different situations in the working program has to be applied.

When the game is nearly finished, we will try to recruit a monkey who wants to test it. The monkey will run as many different scenarios as possible.

6 Schedule

This section describes the planned weekly schedule for the project.

Week 45

During this week, the program structure and the interfaces will be specified in more detail, making it possible to do more individual work. Implementation starts.

Person	Time (h)	Notes
Joel	12	Planning, implementation for the gamestate system.
Joonas	10	Planning, implementation for the map reading system.
Ilari	10	Planning, header files related to game.

Week 46

On this week, we will have a first “playable” (there is something to point and click) protype, making sure that the specifications from the previous week are enough to allow efficient implementation of features for each team member.

Person	Time (h)	Notes
Joel	15	Implementation for Camera and Renderer. Initial implementation for the UI
Joonas	15	Implementation for the player database system. Starting with the graphics.
Ilari	16	Initial tower and enemy implementation, implementation of pathfinding and tower operation.

Week 47

Core features will be finalized during this week. The result should be a reasonably working game which satisfies the minimum requirements specified in section 2.1. The implementation of additional features commences.

Person	Time (h)	Notes
Joel	15	Finalizing the UI
Joonas	14	Finalizing the graphics. Helping with other stuff.
Ilari	14	Implementation of the rest tower and enemy types. Writing the rest of game “rules”.

Week 48

Testing of the whole game, implementation of additional features. The report will be written.

Person	Time (h)	Notes
Joel	15	Documentation and testing
Joonas	15	Additional features, documentation and testing.
Ilari	14	Documentation

Week 49

The demo week. Final bug fixes, aesthetic changes.

Person	Time (h)	Notes
Joel	8	Polishing
Joonas	8	Finalizing what's left, finishing up the additional features and documentation.
Ilari	8	Testing, improving playability

Summary

The total time planned for the project is approximately 60 hours per person. This time does not include the time planning the project before week 45.

7 External libraries

We will use ClanLib, as it provides many useful features for a game: high-level rendering, collision detection, sprites, etc. The Boost libraries can also provide useful tools for many parts of the project, for example Boost Test Library can be used to aid in the testing and Boost Graph Library can be used to help with the dynamic path generation for enemies.

A GUI Sketch

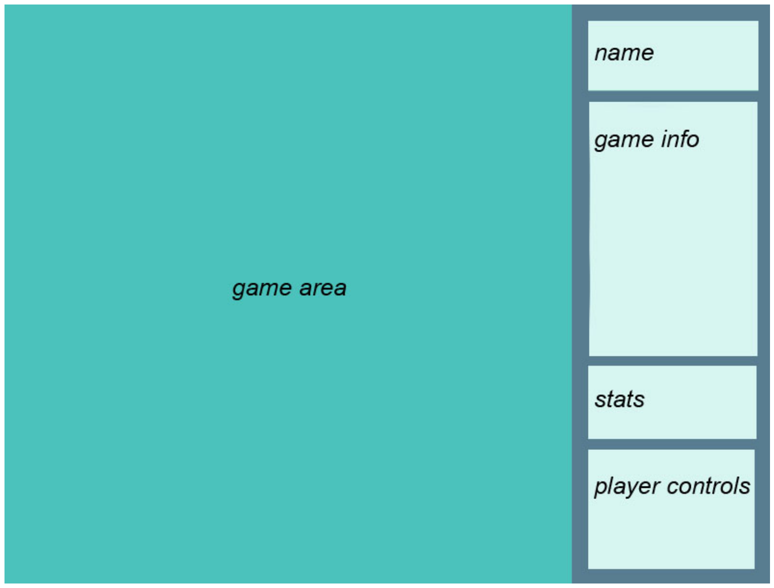

Figure 2: An initial plan for the organization of the graphical user interface

B Map file format

A draft for the map file format.

```
!MAP
LEVEL 1: Level name
#########################
#SSSSSSSSSSSSSSSSSSSSSSS#
#SSSSSSSSSSSSSSGGGGGGSSS#
#SSSSS          GGGGGGGS#
#SR     E       GGGGGSSS#
#SR             GGGGGSSS#
#SR     GRS          RSS#
#SR                  RSS#
#SSSGGGGGGSSWW       RSS#
2                    RSS#
#RRRRRSSGGGS         RRR#
1                   RRRR#
#########################
WAVES:
1: 6 enemy_type_1
2: 4 enemy_type_1, 5 enemy_type_2
3: 6 enemy_type_2, 5 enemy_type_3
.
.
.
n: 1 enemy_type_x, 2 enemy_type_y
```

Map files store the data responsible of level information, i.e. the actual map in ASCII format and the information about the enemy waves. The file always starts with exactly same arbitrary text, "!MAP", which can be used to identify the file as a map file.

Immediately after the identification follows the level information starting with the level number, name and the map layout. The playable area of the map is the area enclosed within the #-characters. Characters like S,R,G and W represent different types of grounds on which you the player can build certain types of towers. Empty characters represent the space in which the enemies can walk in towards their goal, which the letter E is representing in the example. There are also ground types on which the enemies can traverse but can be also built upon by the player. Numbers 1 and 2 in this case represent the entry points for the enemies.

After the map layout comes the information about the waves of enemies. The lines always start with the wave number. After the wave number comes the information of the number of enemies and their type. Different types are separated by a comma. For a variable delays between waves or enemies, some additional information needs to be added.

The information could be stored in a more condensed manner, however it would be nice if the format is logical for the untrained human eye as well. Depending on the features that the maps will be required to have, some human readability might be lost during the project.