Emerging Architecture Design
version 2.0

May 14, 2015

1 Introduction

This document provides an overview of the architecture that is going to be build for this project. The components are covered from a high level view, to give a representation of how the different components work and the dependencies between them.

1.1 Design goals

The following design goals will be maintained throughout the project:

- **Performance**
  The game server will have to serve requests from potentially a couple thousand players at any given moment. To minimise the amount of lag introduced by latency of networks, we have to design the system in such a way that the game experience seems fluent to the users. For this reason we will use non-blocking sockets to handle all requests in parallel.

- **Modularity**
  As explained later, the game will involve three modules that work together. Two of these modules have several sub-components that must act independently when something happens to the game. To support modularity, we will use an event-driven architecture to let the players update game state and also to let sub-components communicate.

- **Code reuse**
  Dividing the different types of logic into modules is good for maintainability, but we will only get this advantage if logic that is similar for both components can be reused. To support this, we have divided the project into three sub-projects: *Backend*, *Desktop* and *Core*. The *Core* project contains all code shared between modules to maximise the possibility of reusing code.

2 Software architecture views

This chapter is about architecture of the system. First the different parts of the system are explained and the relations between them. In the second paragraph the link between the software and the hardware are discussed. The third paragraph illustrates the data management of the system.

2.1 Subsystem decomposition (sub-systems and dependencies between them)

The system is broken down into 3 subsystems with a shared library to enable modularity. This way, all modules can be maintained separately while still being able to reuse shared code.

- **Renderer**
  The renderer connects to the backend via a specialized protocol. Its role is to display the model changes on the screen. For this purpose the renderer uses libGDX's graphical capabilities.

- **Client**
  The client is a static HTML/JS website that the is used to control the game from a smartphone. It send commands to the backend over WebSocket when the user moves his/her phone.

- **Backend**
  The backend runs the game engine and holds the model of the game in memory. Domain events are used to notify the connecting clients if there is any relevant change to the state of the game. The backend hosts separate servers to serve users and to connect the renderer. This creates opportunity for socket-level optimization between the backend and the renderer because these modules are not required to use WebSockets for communication. We have chosen for this set-up because the connection between backend and renderer is likely to become a performance bottleneck. It also leaves the possibility to run the backend server in a cloud environment, which would cut the hardware costs of the game dramatically.

- **Core library**
  The core module contains all classes that can be used by both the renderer and the backend. This involves for example:

  - Protocol code to send and receive game events.
  - Event dispatching code. Both modules use events to communicate and update their internal state.
  - Game entity classes. Both modules need a compatible version of the data model, the backend to apply game mechanics to them, and the renderer uses entities for display.

2.2 Hardware/software mapping

Each module is backed by some hardware. In order to run the backend and renderer, at least one machine is needed, This set-up will practically eliminate latency between the renderer and the backend, but it has to run on a more expensive machine to support high loads.

Another set-up is to run the renderer and backend on different machines. This allows for scenario's where the machine connected to the auditorium beamers does not support a high networking load. The backend can be run on a machine with well performing network hardware and send only relevant events to the renderer.

A third option is to run the backend in a cloud environment and offer a subscription-fee to auditoriums. This reduces upfront costs and eliminates the requirement of high-end hardware inside the auditorium itself.

The client HTML/JS application runs on all smartphone browsers that support WebSocket.