Distributed Systems – Open Project

Samuel Bryner
brynersa
brynersa@student.ethz.ch

Florian Buenzli
fbuenzli
fbuenzli@student.ethz.ch

Frederik Rothenberger
frothenb
frothenb@student.ethz.ch

Simon Wehrli
siwehrli
siwehrli@student.ethz.ch

ABSTRACT
This document describes the process made to build a multiplayer Android game during a project work for the lecture "Distributed Systems" at ETH Zurich.

INTRODUCTION
First the initial ideas and the thereof derived game design are presented. Then we motivate the choice for supporting libraries and the problems occurred during using them. Finally we highlight important design decisions and implementation details.

Basic Idea
The first idea was to create a game in which gravity plays an essential role. Because we wanted it to be simple to play, it seemed natural to use the accelerometer of the device to influence the players gravity, and since it had to become a multiplayer game as the distributed component, we decided to give each player its own gravity. To define a challenge between players, we introduced treasures to combat for, which are also moving to increase the influence of movement of other players on the own task of fetching treasures. This led to the mockup shown in FIGURE 1.

GAME DESIGN

Gameplay
To realize this basic idea we had first to define the gameplay elements. These are static structures like walls and the level and dynamic objects including the players, the treasures and some obstacles. We also had many ideas how to further enhance gameplay by adding additional elements like fog to obstruct the view or special elements like water (which alter movement dynamics). But these were ultimately dismissed due to lack of time.

The static elements are, as the name suggests, not moveable nor do they react to any actions of the player. They are just obstacles and provide structure to the game environment. In some levels there are some dynamic objects bound to such static structures (e.g. the bridge in the *bridge.phy* level).

All other objects are dynamic in respect to position. Levels may have an arbitrary number of these dynamic objects (bound by the performance of the device used). These objects include one player per user and one or multiple treasures. There may also be dynamic obstacles (e.g. the bridge in the *bridge.phy*). Dynamic objects react to collision and gravity. All of them will collide and change movement direction and speed according to the simplified laws of physics implemented in the *Emini*

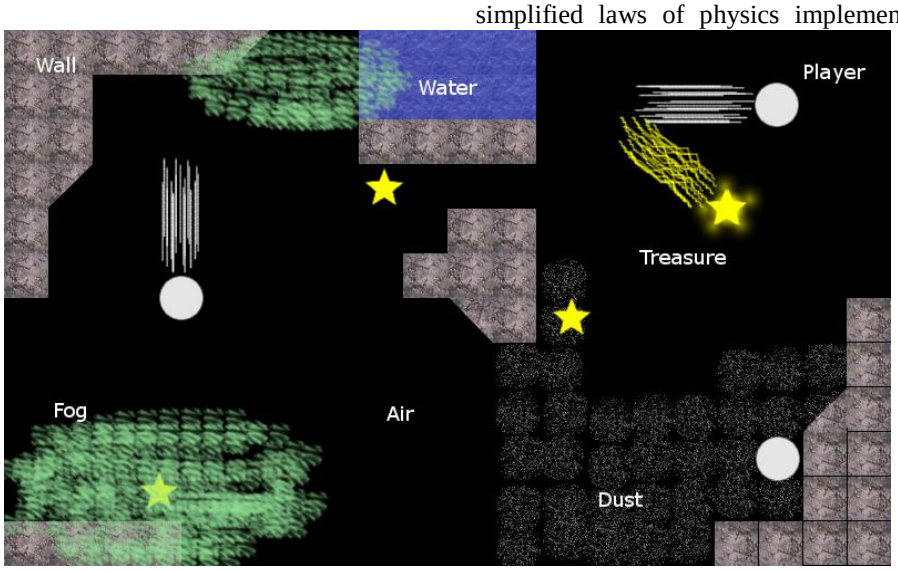

FIGURE 1: Mockup to illustrate basic game idea.

Physics Engine. If a treasure collides with a player it is removed, as this event is interpreted as collecting the treasure. This means that the value bound to the treasure gets added to the colliding players score and a new treasure is spawned at one of the designated spawn points.

Another important element is gravity. In this game there are multiple gravities, one for each player and one gravity shared between all players. The movements of the dynamic objects are largely controlled by these gravities. Each player object is only influenced by the gravity set by the corresponding user. Each users gravity is defined by the orientation of his phone which means that the player object controlled by this user is drawn in the direction that happens to be downwards according to the phone. This also means that "down" is not the same for every player. The other dynamic objects do all share the same gravity. This shared gravity is calculated by summing all individual player gravities and then scaling it to match the strength of one player gravity. Player objects are not affected by the shared gravity.

This means that every user directly controls his player object by changing the orientation of his phone and also influences the movements of all other dynamic objects to some extent.

The goal of the game is to be the first player to reach a certain amount of points. Points are awarded for collecting treasures. The amount of points depends on the collected treasure and may also be negative (these "bad" treasures differ in appearance from "good" ones). Whenever a treasure is collected another one appears at one of the predetermined treasure spawn points. The selection of the spawn point is randomized.

This game is multiplayer only, so there have to be at least to users in order to play this game. To start a game, one user has to host a game server where the other users then are able to join (via the join game dialog). All settings (like the map selection) are set by the host who is also responsible for starting the game once all the users have joined his lobby.

Used Technologies

Physics Engine

In order to be able to meet the deadline for this project and still be able to realize our ideas, we used a third party 2D physics engine (*Emini Physics Engine*) since accurate and fast simulation of physical environments tends to be tricky. This engine provided us with all the features we needed in regard to physics. More precisely, we used this library to calculate the movements of all the dynamic objects (such as players and treasures), detect collisions and design and load levels.

Because physics is a central part of our project and the *Emini Physics Engine* is closed source, we had to adapt and base some design decision on the structure of this library. As a result the extended world object of the physics engine handles most of the game logic and represents the entire game state instead of having these things separated.

Over all, the *Emini Physics Engine* provided us with many useful features and sped up development enormously.

OpenGL

We used *OpenGL ES 1.0* interface provided by the Android framework itself. It was easy to use with Java and takes care about drawing our game elements to the screen. For more information see section *Software Architecture* below.

Software Architecture

We divided our project into 4 major parts. These are graphics, logic, network and util. This division is represented by the package structure.

The **graphic package** contains the classes for the OpenGL renderer (*GameRenderer.java*), a view class (*GameGLSurfaceView.java*) for the renderer to draw into and a sub package primitives which contains helper classes used by the renderer such as *Vertexes.java* (OpenGL helper class to draw lines) or *Skeleton.java* (used to animate players).

In the **logic package** are all the classes which represent game elements. Most importantly, there is the

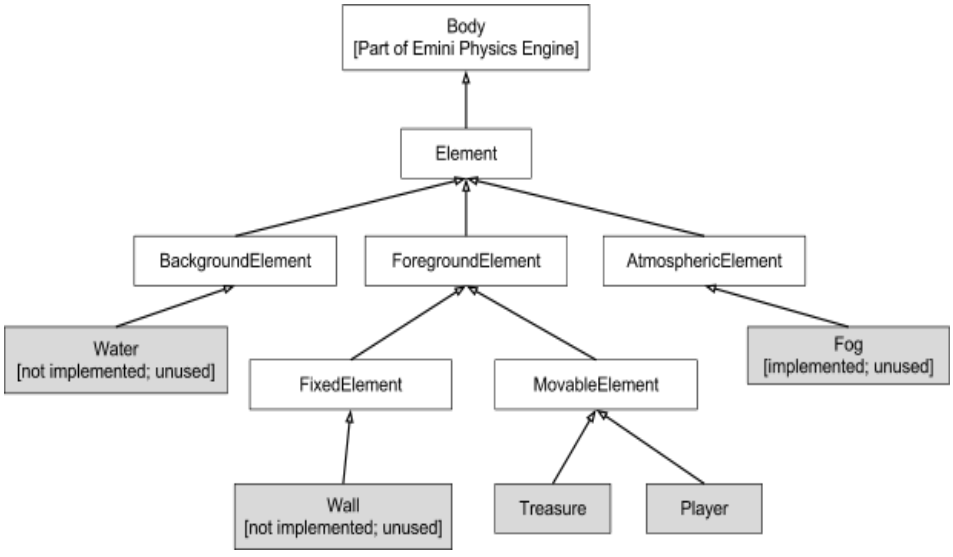

FIGURE 2: Class diagram of logic package.

GameWorld.java which represents the game state including level and any objects (including the players). The structure of the element classes are based on the *Emini Physics Engine*, which represents any object (static and dynamic) as a Body. All the element classes inherit directly or indirectly from Body, but get more and more refined with properties we needed for the game such as indices for players or a value field for treasure. The complete inheritance tree of the element classes is represented in FIGURE 2.

The **Network package** handles the communication between the Users and a Server. The Server, which may be a User itself, processes all user input and computes the complete game state.

To achieve a reliable two-way communication, we set up a persistent TCP connection between the User and the Server. For a detailed view of the network usage see FIGURE 3.

We implemented our own protocol (*GameNetworkCommunicationProtocol*) for sending messages over the TCP connection. The messages are encoded in a binary format to save network bandwidth. Each message consists of a 4-byte header that holds the message's type and the length of payload data.

As a supplement we use UDP multicast functionality to advertise the Server's address in the local network, so the User does not have to enter the address manually.

DIVISION OF WORK

We initially divided the work in four parts: Simon was responsible for the network implementation, Florian did all the graphics related stuff and OpenGL rendering, Frederik was in charge of finding a suitable physics engine and implementing the simulations we needed and last but not least we assigned Samuel the game logic and project management. This strict division did however not prove suitable as the deadline approached nearer. Therefore, and also because the network part did present more difficulties than we anticipated, Florian and Samuel also started working on the network implementation. The documentation and the writing of the report was done together.

DIFFICULTIES

We initially decided to use the well-known and powerful library *AllJoyn* to do all the networking. Since the library documentation is really detailed and not example oriented, we started from the provided example chat program and tried to adapt it to our needs. But later we noticed that the example did not work on every WLAN, which urged us to go deeper in understanding the library structure - without success. While running out of time we have decided to start over and build the networking part ourselves. *AllJoyn* certainly does a good job, but for a quick and dirty approach the chat example was much too complex, also because of a complete event-based message model.

FURTHER IMPROVEMENTS

For now, the network architecture consists of a basic client-server architecture with clients just streaming the data they get from the servers. While fine for local networks and simple maps, this approach quickly runs into lags. Instead of running the simulation only on the server one could also run it on the clients in a 'secondary' mode:

The server can remain unchanged. The clients run their own version of the physics simulation and use it to extrapolate the last received state to the one currently on the server. This would allow to combat lag but requires significantly higher complexity in the clients as conflicts (clients are now in the 'future' and user input could create contradictory situations) and divergence (simulation of server and clients diverge due to numerical errors) has to be handled correctly.

Another thing to be improved is the input which is a bit difficult when the screen is rotated as the left side of the screen always causes the player to walk to the left (even if this is to the right side of the screen).

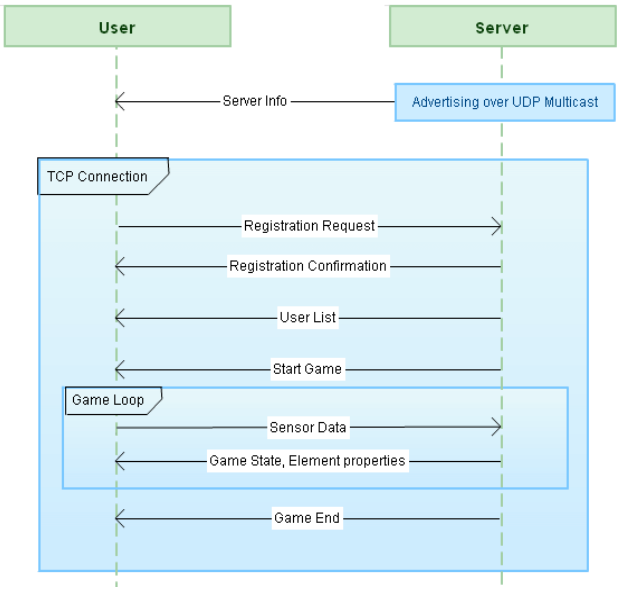

FIGURE 3: Communication between the User and the Server.