Emergent Architecture Design Document
DuoDrive

Pim van den Bogaerdt, pvandenbogaerd, 4215516
Ramin Erfani, rsafarpourerfa, 4205502
Robert Luijendijk, rluijendijk, 4161467
Mourad el Maouchi, melmaouchi, 4204379
Kevin van Nes, kjmvannes, 4020871

June 17, 2014

TI2805 Contextproject, 2013/2014, TU Delft
Group 5, Computer Games
Version 3 (draft)

Abstract

This document describes the architecture of the system that our product consists of. Firstly, we will describe the design goals for our final product. After that, our Software Architecture Views will be treated. Finally, a glossary can be found at the end of the document. This document will be edited and changed throughout the duration of the project to fit our current system architecture.

Contents

1 Introduction

In this document we will describe the architecture of the system that we will be creating during the Computer Games Contextproject. The architecture will be explained in a form containing high level components with the sub-components and sub-systems. This will give more insight in what the system is composed of and how the several elements of the system work together.

1.1 Design goals

There are four design goals that will be maintained throughout the project:

Availability

During this project our product will be *continuously integrated* with the goal of having a working product running at any time. This allows us and the client to test and work with the product at any time, even before the final release. After the final release we will want to make sure that the server is up as much as possible, especially during the time that people are visiting our client's company or area.

Replayability

A design goal that we consider to be specific for our product is replayability. We want the players to enjoy the game in such a way that the game will be replayable in both the short and long term. We want the players to find our product memorable, so they may even look forward to playing the game the next time. To reach this goal, the game should make a solid first impression, after which elements such as randomness and other unpredictable events will keep the game interesting to play.

Usability

The game is designed in such a way that without, or with few, instructions the player will be able to play the game. After a few minutes, or even seconds, the player will have learned how to work with the system and play the game even better. The physics in the game are set to make the user experience better and making the efficiency of playing the game at its best.

Moreover, with feedback given during the game, the user will be able to see progress and see how well he is playing the game.

Performance

The system will have a central server which is situated in the same room/building/area as the players who are playing the game. Also, all the players should be connected on the same network as the server is. Since the server will be the machine sending and receiving the data of the players, it is necessary that the connection between the player and the server is fast enough. This is needed to ensure that the players will not encounter any *latency issues*.

In a future release we might implement a system where some of the server's calculations will be calculated locally on the player's device. This will take load off the server and may improve the connection speed throughout the game.

2 Software Architecture Views

This chapter will discuss the architecture of the system and how it is decomposed. In the first paragraph the subsystems will be decomposed and we explain each subsystem. In the second paragraph we will elaborate upon the relations and mapping between hardware and software. In the last paragraph the data management will be explained.

2.1 Subsystem Decomposition

The software architecture of the system consists of three different subsystems: the server, the system for the steering player, and the system for the player who controls the throttle. These different subsystems are explained in this section.

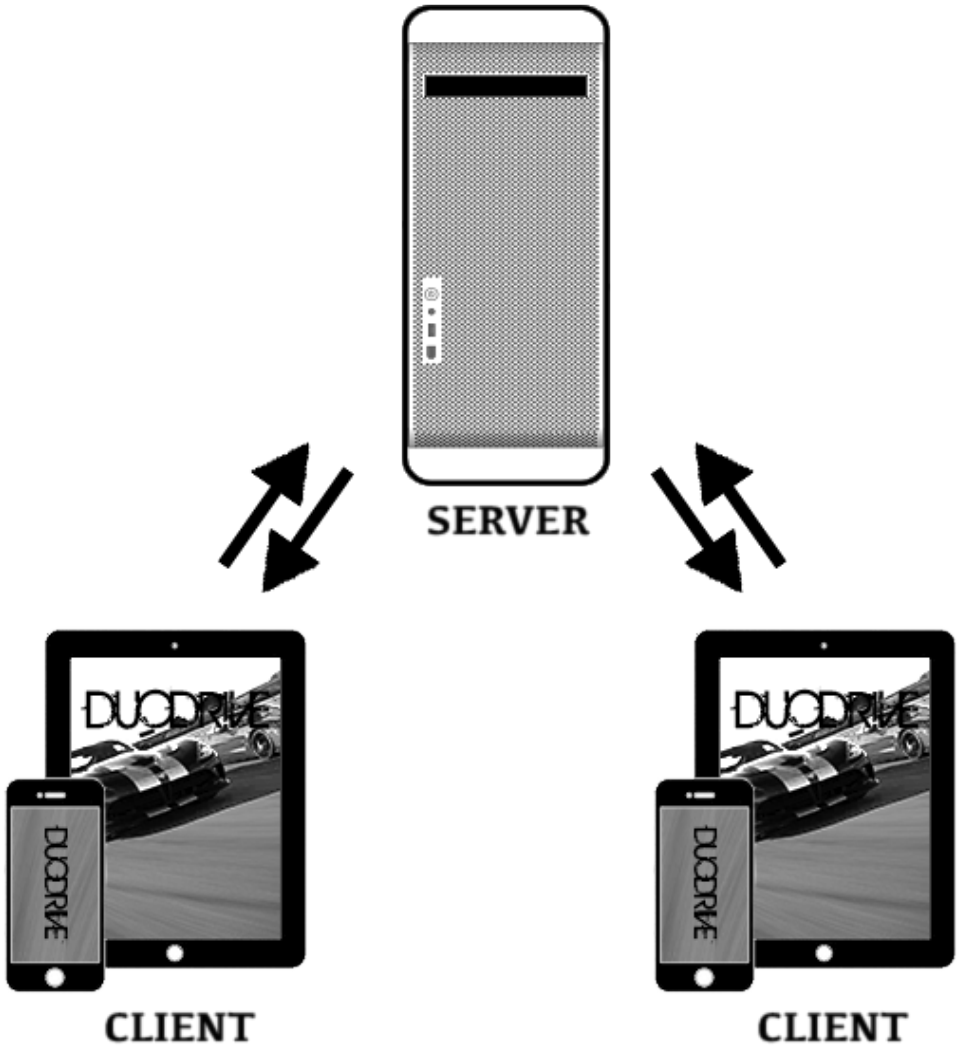
- Server
  The server in the system maintains the data flow. It will send the data of the positions of the cars to all players, so that every player has a near real-time experience, where they can see other players' positions. All data sent by a player will first be sent to the server, which distributes it to the other players. There is no player to player data flow at this moment.

- Steering player system
  The system for the steering player will only contain the view that the steering player should see. This will be a limited view of the track and not being able to control the throttle. It will only have the ability to steer using a *gyroscope* available in the device of the player.

- Throttler system
  In the throttler system the abilities to perform certain actions is also limited. The player will only be able to control the throttle, i.e. accelerating forward or backward, and hitting the brake. By limiting the view and the possibilities to perform actions, communication will be enforced.

2.2 Hardware/Software Mapping

The hardware used is very various in terms of the player-held device. That device must have Android running and must contain a *gyroscope* and *accelerometer*. The only other piece of hardware that will be required is a central computer that will serve as a central server.
The software that is being used on the described hardware is the same. Depending on the functionality, so either being a steering player, throttler, or server, the interface will be different. The server has a simplistic interface which will make it possible to start or end the server. The mapping of the hardware and software can easily be concretised in a small diagram, which is shown on the next page:



2.3 Persistent Data Management

Persistent data management is not needed in our product. There is no data that needs to be saved persistently.

2.4 Concurrency

Concurrency is an important aspect to take into account when dealing with a multiplayer game. Especially if this multiplayer game is one where people interact with each other in real-time.
Concurrency is handled in one specific way in our game, which is by using *Remote Procedure Calls* (RPC).
Using RPC has some advantages, but it also has some drawbacks.

The main advantages are that RPC is easily used within Unity and that RPC allows us not to have to deal with coding the remote interactions.
However, the drawbacks are that RPC waits for a server to process the call before allowing the client to continue. This means that we had to be sparse in using RPC throughout our code, for if we did not do this, the server might get congested with calls and this would influence the output of the game.

3 Architecture

3.1 Introduction

At first a class diagram was created to be able to get a quick view of the architecture. However, during the development of the project, we came to the conclusion that our class diagram would quickly become too messy. In Unity *GameObjects* are used, which have scripts attached to them as components. The interaction of these scripts with other scripts, classes and GameObjects depends on different variables of the GameObjects. This made it very hard for us to create and maintain a class diagram for a clear overview, even at a high level. Instead of a class diagram, we have chosen to dedicate a section to the setup of classes and the interaction between them. In that way it is possible to discuss the hierarchy of the classes and their interactions.

3.2 Classes

Within the whole project we count 41 classes, excluding test classes. Every class has been split up in such a way that it has its own functionality. There where it was needed or useful Design Patterns have been used. In this project we have only made use of the Strategy Pattern. Furthermore, a client-server architecture has been used since this was ideal to maintain and control the new joined players.

3.3 Packages

The classes have been managed to be split into different packages, every package has its own responsibility. In the upcoming descriptions every package and their responsibility have been explained. Furthermore, if a Design Pattern has been used within a package, it has been described with it.

Behaviours

In this package the Scripts have been bundled that belong to certain GameObjects which control behaviours. Such as a Car, has the AutoBehaviour-script attached to it. This script manages how the Car should behave during the game. Furthermore, there are other Behaviour-classes which are ment for the track.

Cars

The Cars-package contains classes which create the Car and the players attached to it. Here we used a Strategy Pattern, this was recommended to us and in practice also seems much better than using simple Inheritance. See figure 1 for the classes in the Strategy Pattern form.

Main

In the Main-package we have four classes: ControlButtonsGUI, DuoDriveGUI, Game and Tutorial. The package contains the main elements for the game which are the GUI for every player and/or the server, the controll buttons which are shown to the player, the actual Game-class which handles the overall game and a Tutorial-class which will show a simple tutorial based on the player role.

NetworkManager

Within the NetworkManager-package, three classes can be found: MainScript, Server and Client.

The Server-class handles all the operations and commands that should go to and/or through the server. The Client-class handles the job that should be assigned to the newly connected player, which is in fact a new client. Moreover, the MainScript-class is intended to do the first steps which are required. The first steps are initializing the Server, Client -and Tutorial GameObjects and initializing the cars.

Utilities

The Utilities-package contains some simple utility-classes which are intended to access several functions with ease and without the need of the special MonoBehaviour class in Unity.

Interfaces

Within this package all interfaces are placed together. Some of these interfaces were not needed for the actual game, but to be able to test several classes we needed to Mock them. In C#, the programming language we chose to program in, Mock-objects can only be made of virtual objects and/or interfaces.

Wrappers

In this package several wrappers are written. These wrappers were needed so that the classes which use the interfaces, as variables, still will have their functionality. However, for the Input a wrapper was also made so that we can simulate certain inputs. Without the wrapper it was impossible to test certain touch-events and thus some methods could not be tested. But with this wrapper that was all possible again.

4 Glossary

Continuous Integration (Wikipedia, 2014)

Continuous integration is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. This type of integration is used to make sure a working copy is always available at any moment during a software engineering project.

GameObject (Unity Scripting API, 2014)

A GameObject is the base class for all entities in Unity.

Latency Issues

In a network, latency is the amount of time a packet needs to get from one point to another, i.e. from a server to a client. Latency issues arise when a packet takes too long to arrive at the endpoint, the fluency of the program may be compromised, because packets may take too long or be dropped completely.

Remote Procedure Call(s) (TechTarget)

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.

References

Continuous integration. (n.d.). In *Wikipedia*. Retrieved May 15, 2014, from http://en.wikipedia.org/wiki/Continuous_integration

GameObject. (n.d.). In *Unity3D*. Retrieved June 17, 2014, from http://docs.unity3d.com/ScriptReference/GameObject.html

Remote Procedure Call (RPC). 2009. In *TechTarget*. Retrieved June 17, 2014, from http://searchsoa.techtarget.com/definition/Remote-Procedure-Call