Architecture Design - Carried Away

1. Introduction

In this document you can find a sketch for our context project on computer games. We will tank about the design goals and the software architecture. This document will be modified when needed.

1.1 Design goals

Availability
Following the *Scrum* principle, at the end of each *sprint* a working product should be ready for users to try out. This way testers can give us feedback on the current system each week and the developers can adjust the system accordingly. It also makes sure that the game studio doesn't deviate the game from what the users actually want.

Performance
What every user wants in a game is a very good performance. The Oculus Rift can be quite performance intensive for older systems and non gaming laptops. Therefore it's imperative that we focus on higher performance to make sure the game runs very smoothly.

Reliability
Just like performance, running into bugs during their gameplay experience ruins the fun. With each new feature, the amount of entropy in the system increases drastically. It should be the goal of the developers to keep that as low as possible. Every developer is responsible for their own code, so each of them will have to make sure that all their code is nearly bug free.

Manageability
When we want to change a part of the game, we want to be able to do this without any problems. To accomplish this, we want our game to be playable after every sprint, as well as keep our code modifiable. Every developer is responsible for their own code, so they'll each have to make sure all their code is kept manageable.

Scalability
As new features are added, the program not only becomes less and less manageable, the scalability is also compromised. When a developer adds a new feature, he or she should keep in mind if it affects the program as a whole for later expansion. Sometimes spending some more time on a feature to make the program future proof is very much worth it in the long run. Developers should consider this when implementing a feature and sometimes put in some extra work.

Securability

Since our game will be played on LAN only, we don't have to secure our game. When the internet connection that our game will be played on is safe, our game is safe. *Or is it?*

2. Software architecture views

In this chapter we will explain the architecture in the form of components of the system, which will be split into sub components and subsystems.

2.1. Subsystem decomposition

The system has been divided into three subsystems. The Client Interface, Server Interface and Server Resources. The interaction between these systems is illustrated in figure 1.

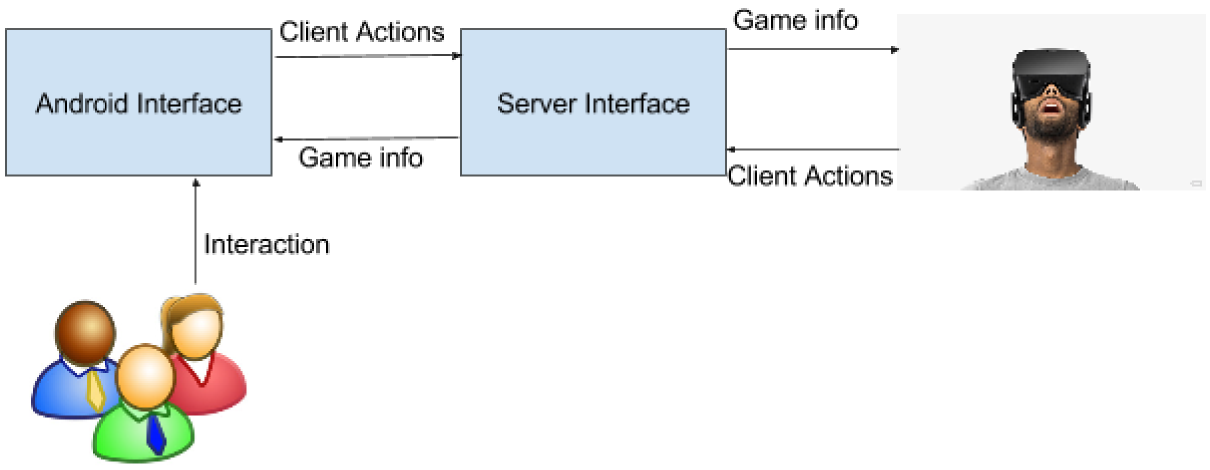

Figure 1. Architecture Diagram

In this section the subsystems are described in more detail.

- The Android Interface allows its users to interact with it. The actions the users commit are send to the server in order to influence the world.
- The Server Interface is used to connect every client with each other. It is responsible for handling the entire game. It sends information of the game (for example the location of the player in the world) to the Android users and the Oculus Rift user. What information the server sends to who depends on the game design.

The Oculus Rift should be directly connected to the machine where the server interface is running on. So everything done by VR-user is directly being handled by the server interface, instead of through an extra interface like the Android interface.

2.2. Hardware-Software mapping

Both the hardware and the software for the server are not the same as the software and hardware for the Android devices. The Oculus Rift should be directly connected to the server using a HDMI and a USB cable. Through the Server UI (which can be operated by any user) a lobby can be started, to which the Android users can connect using the Android UI and the Local Area Network where everyone is connected to.

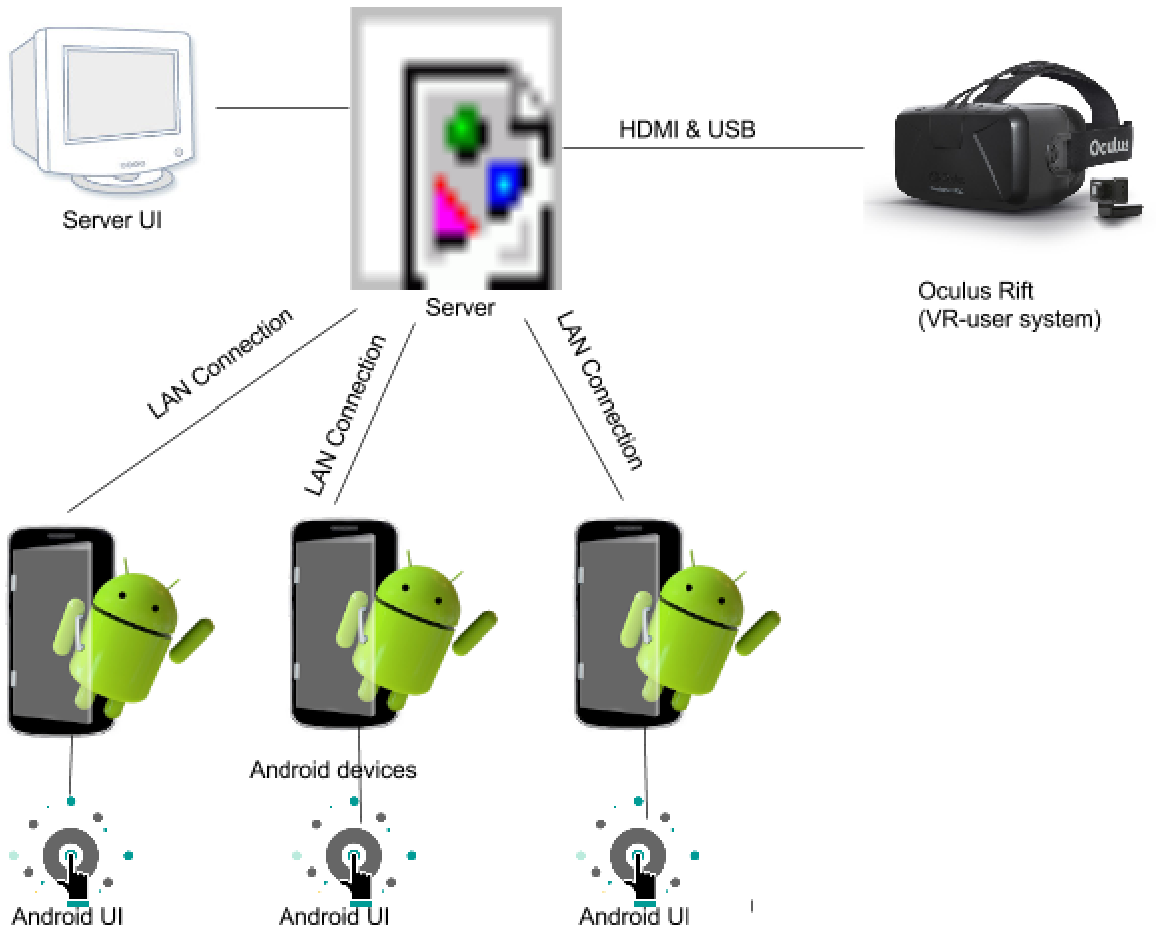

Figure 2. Illustration of the Hardware-Software mapping

2.3. Persistent data management

We won't use large amounts of memory, but we will keep our models, textures, animations, etc. in an assets folder. The highscore feature, if implemented, will require a file where the statistics are saved. This and possible other statistics that need to be saved will all be contained in a *resource* folder.

2.4. Concurrency Control

The Android devices repeatedly send messages to the Server and vice versa. This may cause some concurrency problems if the Server received many messages at once.
To prevent this, a message coming from one device should not depend on message sent by another device. This can be achieved by storing information sent from the clients separately. So a information from a client *A* should be stored on a different location than a message from client *B*. The information should then be processed by one single thread.

3. Project Modules

Because of the clear distinction between the desktop application and the Android application, the project has been split up into three modules. These modules and their relations to each other are discussed in this section.

The modules are contained within their own separate Maven projects. A third Maven project has both of these projects in its project folder and builds them consecutively. This setup is very simple and easily allows the addition of more modules.

The Desktop Module
The Maven module ContextDesktop is meant for the server. So this is what the computer that has the Oculus Rift connected to it must have.

The App Module
The ContextApp module is meant for Android devices. This module is used to deploy the application on an Android device.
Unlike the Desktop module, this one runs on *Java 1.7* instead of *Java 1.8*, because many Android devices do not support *Java 1.7*.
In order for this module to be installed, you need the Android SDK on your computer (which is automatically installed when also installing Android Studio).

The Messages Module
The ContextMessages module, unlike the other two modules, is more a library than an application. Both the Desktop and the App module require the Message classes stored in here. That’s why those module have this module as a dependency. So this module requires to be installed for the other two to even work.

The jme3-networking library requires Message classes to be in both the server and the client, even though they’re two completely different application. So instead of copy and pasting Message classes in those modules, having them in this module is a much cleaner way.

4. Class structure of features

In this section, the structure of the implementations of many features of the game are described.

4.1 General structuring of the project

The whole project actually consists of two applications, one which runs on the desktop for the oculus rift user, and one that runs on mobile android devices for the carriers. These are two modules of the entire project. Finally there is a message handler module which is used by both applications for communication between the applications. Below we go into more detail of each module.

4.2 The desktop module

4.2.1 General structuring of the desktop application

The main method creates the virtual reality app. The desktop application consists of multiple states, which are created when the virtual reality app is created. The current states are the environment state, which is the game itself, and the main menu state. The main menu is booted up right after the states are initialized. Pressing the start game button attaches the environment state to the game screen and detaches the main menu screen.
The simpleUpdate method calls all update methods, which are the environment updates and polling for user input. When the user enters the lobby, the game will try to connect to its mobile device users.

4.2.2 Main

For the main game, we need to configure the main to be compatible with VR, this is done in the VRConfigurer class.
We also need to handle key events, for example when switching to the debug camera, moving this camera around or exiting the game. This all is done in the InputHandler class, which is called each update from the Main class.

4.2.3 Screens

The desktop module contains two screens, the main menu screen and the lobby screen. Both of these methods are created with the NiftyGUI library:

- The main menu screen contains two buttons, a “Start game” button and a “Quit game” button. The “Start Game” button which sends the user to the lobby screen, and the “Quit game” button quits the game.
- The lobby screen contains two buttons, a “Start game” button and a “Back to main” button. The “Start Game” button starts the game. At this point in the project, the game can start under any condition. In the end this start button should only enable

once exactly four players have joined the lobby, but this is not implemented yet. The "back to main" button sends the user to the main menu screen.
This menu also contains four fields. These four fields are empty at the moment, but in the end we want these four field to contain the four distinct players of the game.

4.2.4 Environment

This section talks about the Environment class. This class is responsible for setting up the game screen. Most of its functionality has to do with interaction with the AssetManager, the RootNode and a list with all Movable objects.

4.2.5 MainEnvironment

The MainEnvironment class, which is a subclass of the environment, handles the initialization and updating of all objects in the game world. For initialization, the methods *initLights(), initShadows(), initSpatials()* and *initCameras()* are called. The lights, shadows, Spatials and Cameras that are produced by this method are attached to the rootNode, which is located in the superclass. This class is also responsible updating. While updating, objects can be created, deleted or modified. The MainEnvironment class is also responsible checking collision. Collisions work like this: every collidable object checks each update whether it is colliding with the players. If it does, a message is sent to the CollisionResults object. From here it is read out and the wanted actions are performed on the two objects.

4.2.6 Level piece generation

A level piece is constructed from a model that is made in blender. In each level piece, there are two lights added to give the environment a better atmosphere. Due to the usage of these lights, we also render the shadows that should be generated.

4.2.7 Level generation

The following steps show the process of creating a world:

1. Initially, five random level pieces are generated.
2. The player is generated in the first level piece in line. In this context, the player is defined as the commander, the camera attached to the commander and the platform the commander is standing on.
3. The fly cam is generated for debugging purposes. The fly cam is explained in the subsection below.
4. The game starts.
5. While the game is running, the player will move forward at a constant speed. Whenever the player reaches the middle of the second level piece in the

environment, the first piece is removed and a new random level piece is added. This way, there will always be five level pieces in the game.

4.2.8 Interfaces

To describe behaviour for objects within our game world we have created multiple interfaces. This section briefly describes them.
IDisplayable: Implemented by all objects that are visible in the game world.
IKillable: implemented by all objects that need a health attribute and can die.
IMovable: implemented by all object that need to be able to move within the game. These objects have a particular moveBehaviour.

4.2.9 Abstract superclasses

Because some game objects have particular features in common, some superclasses were made to describe them partly. In this section they will be described briefly.
Entity: This superclass provides a template for all object that are movable and collidable. Examples of such objects are Commander and Carrier.
MoveBehaviour: Provides a template move vector. Each update this move vector is added to the location of the object.

4.2.10 MessageListeners

The AttackMessageHandler receives an AttackMessage and processes the data it holds:
- A PlatformPosition, which is used to determine which carrier performed an attack
- The direction (a string: left, middle or right) which is used to determine in which direction the player performed the attack.

This initiates an attack function, which in turn sends a HitMissMessage using the HitMissMessenger,

4.2.11 Messengers

The HealthMessenger creates and sends a HealthMessage every time a carrier's health is changed. This message contains:
- A PlatformPosition, used by the clients to decide whether or not that HealthMessage is meant for them.
- A health value (integer) which is used by the app to update its hearts.

The HitMissMessenger creates and sends a HitMissMessage when ever a carrier performs an attack. This message contains:
- A PlatformPosition, used by the clients to decide whether or not that HealthMessage is meant for them.
- A boolean called hit, which tells the receiver whether or not the carrier successfully hit an enemy.

4.3. The android module

This section will talk about the structure of the Android application.

4.3.1 The android Manifest

This file, located in *ContextApp/src/main* is used to configure the android application. So far, it is only used to define the class that is launched when the application is deployed, which right now is *MainApplication*.

4.3.2 MainActivity

This the activity that is started when the application is deployed. An activity is a class that implements the Activity superclass, which is imported. Activities are responsible for visualization and interaction with the hardware. *MainActivity* is a special activity, since it implements the *AndroidHarness* class, which is provided by jMonkey. This allows the activity to serve as a wrapper for a jMonkey application, which can be defined. MainActivity also projects the UI on the screen and contains the logging function for the buttons.

4.3.3 AccelerometerSensor

This activity is created by the *MainActivity* and serves as a sensor, which senses sensor changes that are reported by the accelerometer of the smart phone.

4.3.4 UI

The UI consists of multiple screens which all have their own XML file. A few screens are dedicated to loading and will show the player what is happening while they have to wait. The UI screens that are projected during gameplay show which position the player has carrying the platform, contain buttons used to defend against enemies, and have three hearts that represents the player’s health. There’s one in-game UI screen for each player/position.

4.3.5 Automatic Connection on LAN servers

One of the features on the Android, is that it can automatically connect to an open server lobby, that is running on the same LAN. Three classes are responsible for this. On the server side, there is the *ClientFinder* class, and on the client side there is the *ServerFinder* and *AutoConnector* class. Running both "*Finder*" classes (on different instances of the application) will make the *ServerFinder* find the ip addresses of every server that is currently running the *ClientFinder* class.
The following steps show the process of the client finding a server:

1. (ServerFinder) Sends a message with a certain password to every device on every network interface.
2. (ClientFinder) Receives a message and verifies using the password that the message really came from a *ServerFinder*.
3. (ClientFinder) Sends a message with a certain password to the device that sent the previous message.
4. (ServerFinder) Receives a message and verifies using the password that the message really came from a *ClientFinder*.
5. (ServerFinder) Can now do something useful with the ip-address of the server, such as connecting to the lobby!

So the responsibility of the *Finder* classes is finding ip addresses of servers that are running on the same LAN.

If an android device can't connect to the server within about 5 seconds, the app will be turned off by the alert function in MainActivity, which also shows a popup telling the player that the game should be turned on before trying to access the app.

The *AutoConnector* uses those two classes, to find the ip address of a running server. Then it uses that ip address to automatically connect to them.

4.3.6 MessageListeners

Whenever a carrier takes damage or his/her health is set by the game, the HealthMessenger on the side of the server will send a HealthMessage to the android devices. The HealthMessageHandler receives the message, uses its PlatformPosition to check whether or not that message is targeted at them and sends the health value (an integer) to MainActivity, which calls functions that update the hearts shown at the bottom of the UI.

When a carrier performs an attack, the carrier's android device receives a HitMissMessage from the server. This message has a boolean, called hit, which tells the device whether or not he/she has successfully hit an enemy. This determines which sound effect will be played and whether or not there will be cooldown period for the player, which makes the player wait for a few seconds before he/she can perform an attack again.

4.3.7 Messengers

When a player presses one of the attack buttons on their screen, the AttackMessenger creates an AttackMessage, containing the PlatformPosition of the player and the direction of their attack (left, middle or right, corresponding to the three attack buttons), which is sent to the server.

4.4 The Message Module

4.4.1 Traffic between server and client.

The jme3-networking library is mainly used for the sending of messages between a server and client. The following steps need to be taken in order to send a certain message, the *AccelerometerMessage* is used as an example here:

1. Create a Message class that extends *AbstractMessage*. This class requires an @Serializable annotation and a public empty constructor (other constructors are still allowed, but this one is required).
2. Add any kind of attributes that serve as data that has to be sent. For example the *AccelerometerMessage* has x_force, y_force and z_force as attributes.
3. In both the *ServerWrapper* and *ClientWrapper* class (in the Desktop and App module respectively), the created message needs to be registered to the *Serializer.* This is done in the static initializer (*static { /* code here */}*) of the classes. The *AccelerometerMessage* is already there if an example is needed.
4. Now an instance of the message class can be sent from anywhere using the availible methods in the server (in the Desktop module) or in client (in the App module), which are both accessible from their respective wrapper classes. *AccelerometerSensor* for example repeatedly sends AccelerometerMessages.
5. Now a *MessageListener* is needed in the module that receives the message. To do this, the abstract *MessageListener<T>* class needs to be extended. *AccelerometerListener<AccelerometerMessage>* is such a class. The *messageReceived* method gets called every time it receives a message from a connection. Then the contents of that message can be used to do some intresting things, such as moving the platform.

There is a reason that this module contains a *MessageListener<T>* class while the jme3-networking library already contains such a class. The problem with the one from jme3 is that the *messageReceived* method gets called when ANY type of message is received. So an *instanceof* statement and a cast is required to process this message. This is a very bad code practice. This *MessageListener<T>* class however, only listens to one specific message.

4.5 Several class diagrams

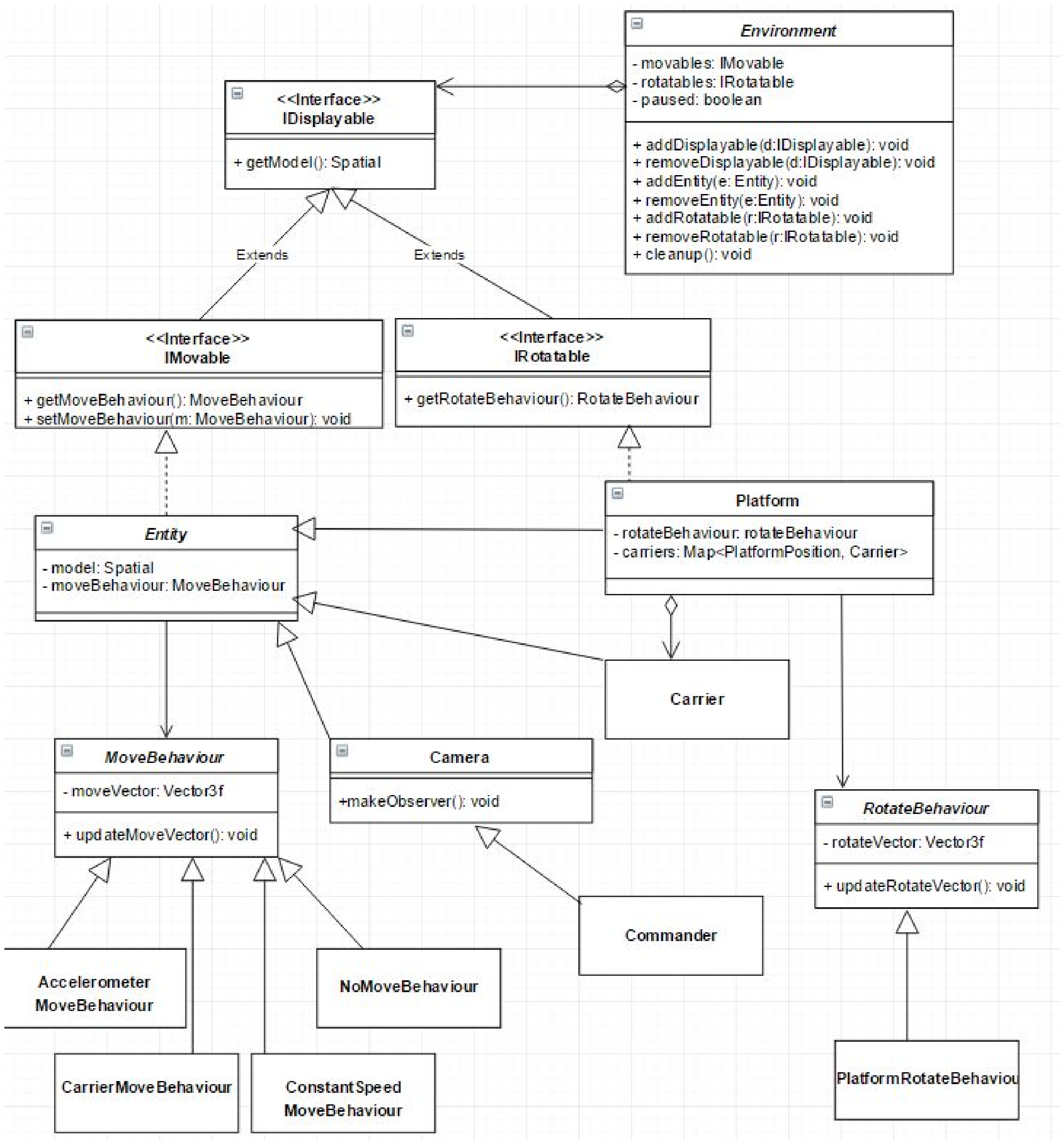
Environment
- movables: IMovable
- rotatables: IRotatable
- paused: boolean
+ addDisplayable(d:IDisplayable): void
+ removeDisplayable(d:IDisplayable): void
+ addEntity(e: Entity): void
+ removeEntity(e:Entity): void
+ addRotatable(r:IRotatable): void
+ removeRotatable(r:IRotatable): void
+ cleanup(): void
<<Interface>>
IDisplayable
+ getModel(): Spatial
Extends
Extends
<<Interface>>
IMovable
+ getMoveBehaviour(): MoveBehaviour
+ setMoveBehaviour(m: MoveBehaviour): void
<<Interface>>
IRotatable
+ getRotateBehaviour(): RotateBehaviour
Platform
- rotateBehaviour: rotateBehaviour
- carriers: Map<PlatformPosition, Carrier>
Entity
- model: Spatial
- moveBehaviour: MoveBehaviour
Carrier
MoveBehaviour
- moveVector: Vector3f
+ updateMoveVector(): void
Camera
+makeObserver(): void
RotateBehaviour
- rotateVector: Vector3f
+ updateRotateVector(): void
Commander
Accelerometer
MoveBehaviour
NoMoveBehaviour
CarrierMoveBehaviour
ConstantSpeed
MoveBehaviour
PlatformRotateBehaviou

Diagram 1. The Environment and the strategy pattern.
This diagram shows the basics of how the Environment is set up.
A *strategy* design pattern is also hidden in this diagram. Each *Entity* contains an abstract *MoveBehaviour* (the strategy), which contain some concrete strategy implementations (*AccelerometerMoveBehaviour, NoMoveBehaviour, etc*). The same applies to the *RotateBehaviour*, which is another strategy.

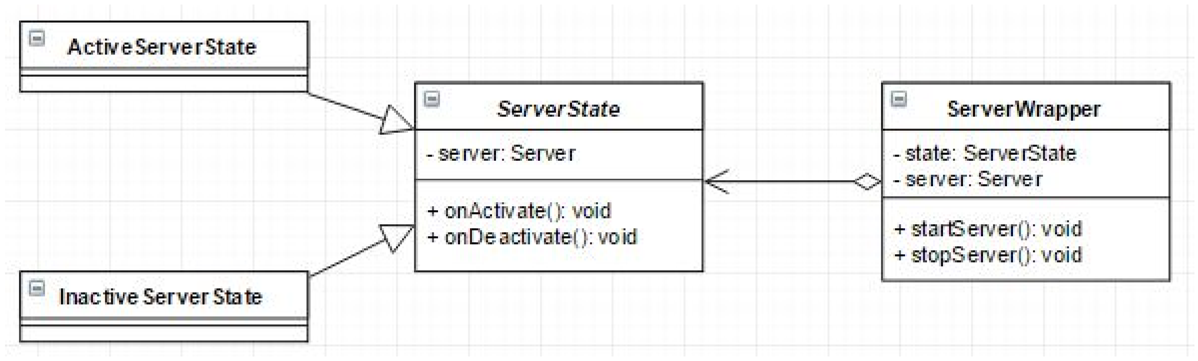

Diagram 2. The Serverwrapper and the state pattern.
This diagram shows the basics of the ServerWrapper and its states. The *ServerWrapper* contains two different states, the *ActiveServerState* and *InactiveServerState*, both which have their own implementation of what happens when it’s activated and deactivated.
The *ClientWrapper* in the *ContextApp* module has the exact same design.

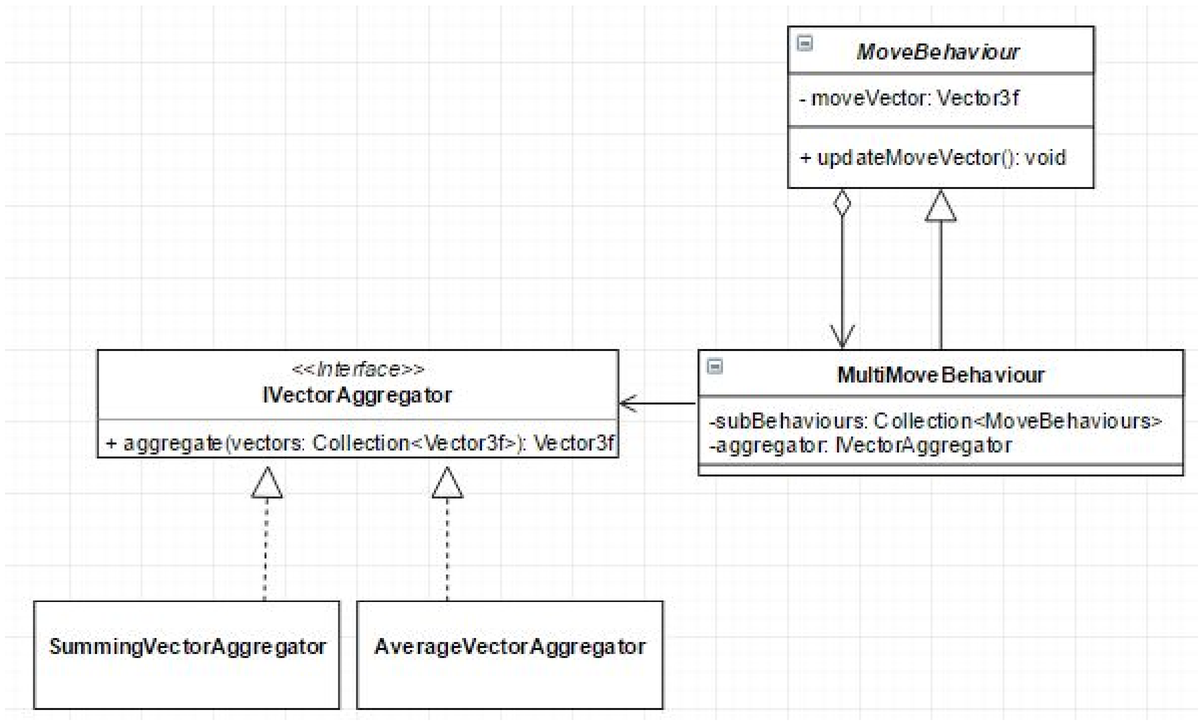
MoveBehaviour
- moveVector: Vector3f
+ updateMoveVector(): void
<<Interface>>
IVectorAggregator
+ aggregate(vectors: Collection<Vector3f>): Vector3f
MultiMoveBehaviour
-subBehaviours: Collection<MoveBehaviours>
-aggregator: IVectorAggregator
SummingVectorAggregator
AverageVectorAggregator

Diagram 3. The MultiMoveBehaviour and the composite + strategy pattern
This diagram shows the basics of the *MultiMoveBehaviour* class. It’s a *MoveBehaviour* that is composed of multiple other behaviours, which allows for more intresting and complex behaviours for certain entities. This relates to the composite design pattern, because the *MoveBehaviour* can be seen as the component, the *MultiMoveBehaviour* as the composite and all other behaviours as the leaves.

The *MultiMoveBehaviour* combines all the other behaviours using an aggregator, which is there to ‘summarize’ a collection of vectors into a single vector. Examples of such aggregators are the *SummingVectorAggregator*, which takes the sum of all vectors and the *AverageVectorAggregator*, which takes the average. Thes *IVectorAggregator* can be seen as a strategy, giving the strategy pattern, mixed with the composite pattern.

5. Glossary

Scrum: A software development framework that uses agile designing and rapid prototyping. It requires a working version of the software to be delivered at the end of each 1-week sprint.

Sprint: An iteration in the scrum process. Every sprint has the same length, with a maximum of time one month. This project uses sprints of 1 week each.