Radboud University

ina ro ect

Authors:
Steven Bronsveld
Jelmer Firet
Thijs van Loenhout
Thomas Berghuis
Robert Koprinkov
Bram Pulles

Teacher:
P. van Bommel
Student assistant:
Jan van Bommel

ontents

$ntroduction$

This is where the introduction is supposed to be : : :.[1]

esi n

In this section we give a global and detailed description of the design of Arcemii. Furthermore we give a justi cation for our design choices.

o a desi n

etai ed desi n

In this section we give a detailed description of the design in terms of classes, methods and attributes.

erver client relation

Arcemii makes use of a server and clients to enable the possibility for playing the game in multiplayer mode. In this section we will describe the most important details of this server client relation.

Arcemii can be played in two di erent modes: o ine and online mode (singleplayer and multiplayer). In both cases we run a server. When playing in o ine mode the server is ran on the background of the mobile phone. When playing in online mode the server is ran on a dedicated server which can be connected to through the internet. Apart from this the only real di erence between singleplayer and multiplayer mode is to which ip the mobile phone will try to connect. In singleplayer mode this is the so called `loop back address`, also know as `localhost` or `127.0.0.1`, whereas the multiplayer mode tries to connect to the ip of the dedicated server.

erver Let's start with a detailed description of the server. The server-side has four classes: `ArcemiiServer`, `Server`, `ServerGameHandler` and `Console`.

Since the server needs to be able to run on its own we have a `main` method in the `ArcemiiServer` class. This method creates a new object of all the other classes to start the server. This class also contains a `stop` method which stops the `Server` and `ServerGameHandler` classes from running. When the server is ran on a dedicated machine we can just run the program separately from the rest of the application. When the server is ran on the background of the mobile phone we just call the `main` method which simulates the exact same behavior, but then locally.

The rst class we instantiate when we run the server is the `Console` class. This class creates a terminal interface for interaction with the server. This is very useful when the server is ran on a dedicated machine to enable some control over the program. The console has three commands at the moment: `help`, `stop` and `log`. The `help` command gives a list of all the available commands. The `stop` command terminates the program. And the `log` command toggles the logging on and o . The logging is very useful for debugging the server. All the classes in the server call the method `log` in the console to send debug information.

The second class we instantiate is the `ServerGameHandler` class. This class, as the name already says, handles all the game logic on the server-side. When this class starts it creates a new thread which sends an update message to every party on the server every tick (which is at the moment of writing this section 20 ticks per second). Apart from this there are two very important methods in

this class, namely: addPlayer and handlePlayerInput. When the first method is called a new thread is created which listens for messages coming from that specific client. Every time a message arrives the second method is called to handle the message from the client. This method calls the appropriate method for all the possible messages. The messages are all a subclass of the abstract class Message. Some examples of messages are: CreatePartMessage to indicate to the server that the client wants to create a party, JoinPartyMessage to indicate to the server that the client wants to join a specific party and the ActionMessage to indicate which actions the player wants to execute while playing the game.

The last class we instantiate is the Server class. This class continuously listens for new clients, creates a connection with these clients and adds the client to the ServerGameHandler. In order to prevent overloading of the server we also check if there is already a client on the server with the same ip-address. If this is the case the old client will be removed since this one is now replaced by the new client. This reduces the load on the server, because now the server does not have to listen for messages coming from the old client anymore. Note that we have an exception for the loop back address to make it possible to connect with multiple emulators to the server for testing purposes.

lient Here we will give a detailed description of the client. The client-side has two classes: Connection and ClientGameHandler.

The first class which will be instantiated already from the MainActivity is the singleton ClientGameHandler class. The first thing this class does is create a new connection in either offline or online mode, more about that in the next sub-paragraph. When this connection is created the client starts listening for messages from the server. This works exactly the same as on the server. We listen for messages and handle the messages with the handleInput method. Next up the ClientGameHandler starts a listener for a change is server mode (offline/online), since it is possible to change this setting in the SettingsActivity. Whenever a change a server mode is detected the ClientGameHandler stops the old connection and starts a new one. The last thing the ClientGameHandler does when instantiated is start the gameLoop method. This method draws the game on the screen and gets the actions of the player and sends these to the server every tick.

The second and last class which will be instantiated is the Connection class. This class either starts a connection with the online server or starts a server on the background and connects to this server. The ip-address of the server is set according to the server mode. This class contains the very important method sendMessage which is called whenever the client wants to send a message to the server. This class also contains a stop method which is very important to prevent memory leaks (due to not stopped threads) and to be able to start a new connection with a server.

arties

The server keeps track of all the clients using parties. The use of parties enables the clients to play with their friends. Whenever a client is connected to the server the client can create or join a new party.

Whenever a player enters the server the player gets a party assigned. At the start this party is still empty.

evel Renderin

ame ie client view Game iew.java The rendering of the level with all it's entities is done in the GameView class. GameView extends the standard Android View, therefore it has to be assigned to a layout. This is done within the GameActivity class (client/activities/GameActivity.java: onCreate). After it is assigned to a layout the init() function is called to initialize all objects used in rendering of the GameView, using the dimensions of the layout. Then two functions are used to render levels: updateLevel and onDraw. `updateLevel(Level)` prepares a level to be drawn by converting it's tiles and entities to RenderItem's, and guring out in which order to draw them. `onDraw(Canvas)` then takes these RenderItem's and draws them to the canvas, in such a way that the player is centered. Locks are used between these functions because onDraw is executed on a di erent thread compared to updateLevel and they both use the renderItems list.

Render tem client view RenderItem.java All aspects of rendering a single texture are handled in the RenderItem class. A RenderItem contains the following attributes: `texture` de nes which bitmap to draw; `x,y` de nes the position where the texture should be drawn; `refX,refY` de nes the position within the bitmap used for alignment; `layer` de nes the layer on which this object is drawn; `animationOffset` the number of frames this object's animation is ahead of the default animation; `rotation` de nes the number of degrees to turn this image counter-clockwise; `flip` de nes whether this image should be ipped horizontally.
The function `compareTo(RenderItem)` determines which of two RenderItems should be drawn rst. The function `renderTo(Canvas)` renders this RenderItem to the canvas, applying all transformations speci ed.

e t re client view Texture.java This class handles the loading of textures, to prevent a Bitmap to be loaded every time it needs to be drawn. It has a HashMap that stores all textures using a String as a key. The key corresponds to the path of the texture within the assets/sprites folder. `getTexture` is a factory method that loads a Bitmap to the HashMap if necessary and returns the corresponding Texture. `getBitmap` returns the Bitmap associated with a Texture object.

nimation client view Animation.java Animation is a decoration of the Texture class, in order to be able to handle animations. It's `getBitmap` function takes the current time into account to return a certain frame of the animation.

eneration o Render tems RenderItems are created in the classes of the objects they visualize (shared/entities/...; shared/tiles/...). This was done to prevent shadow classes for each entity that only generates RenderItems. Unfortunately the objects are shared between the server and client, which means that a server can't be run without compiling all client stu with it. We didn't nd it worthwhile to x this before the deadline, as we would run the server from within Android Studio for testing.

$ame\ lo\ ic$

Before we can explain how the game-logic works a few de nitions have to be given:

ile A Tile is a building-block of the world. It can either be solid, or non solid, meaning entities can either move through it or not. A few of the tile-types in Arcemii are:

- all A solid tile.
- mpt A non-solid tile.
- oid A tile outside of the bounds of the level.
- $tart$ A tile on which players are spawned at the start of a game.
- $inish$ A tile denoting the nal room of a level.

$ntit$ An entity is a non-tile game-object with a speci c position within a level. An entity's position is not limited to whole numbers (where the a tile's position is). Every entity is identi ed by an unique UUID. A few of the entity-types in Arcemii are:

- $la\ er$ Controlled by the actual players of the game.
- $eleton$ Enemy of player, shoot arrows and ees from the player.
- $lime$ Enemy of player, attacks player by jumping toward them.
- oss Enemy of player, teleports and summons slime entities.
- rro Flies in certain direction, can either hit player or non-player entities. Deals damage to entities it hits.

$evel$ A level contains a grid of tiles (instances of the `Tile` class). These tiles represent the world. Furthermore the level contains a list of entities (instances of the `Entity` class).

$ilit$ The game Arcemii is based upon the notion of abilities. Player can choose the abilities they want to use in-game. Every entity has abilities. An ability can be $e\ ec\ ted$. When a ability is executed, its action is performed on the level or the entity that performed the ability. A ability can be executed directly, or in some cases it can be $invo\ ed$ rst. If an ability is invoked, certain parameters needed to execute the abilities are set. For example: If an entity wants to execute the `Move` ability, it needs to specify in which direction it wants to move, so the entity has to invoke the ability with a `direction` parameter. How invocation a execution works in practice is explained in more detail later. What follows is a list of some of the abilities present in Arcemii:

- o `When executed:` Create a new arrow entity.
  `To invoke:` The direction in which to shoot and whether to hit player is needed.

eal When executed: Heal the entity that executed the ability
To invoke: The amount to heal is needed

elee When executed: Deals damage to entities in range of the entity that executed the ability.
To invoke: Whether to damage players and the amount of damage is needed.

ame lo ic e planation Because Arcemii is a multiplayer game, it is split into the two main sections: the server and the client. In relation to the actual game, the client acts only as the view, and a little bit as the controller part. The actual game-logic happens on the server side. The current state of the game is saved as a property of Party, because each party has their own game-state. This game-state is saved in the shared object Level. The level class contains the current world-state of the game. The server-gameloop updates all parties every TICKSPEED amount of time. When a party is in-game, and update is called the following happens:

1. *nvo e* The invokeAll method of all entities except for players is called. This method invokes all abilities that the entities want to execute (based upon their A.I.). All executed abilities are saved in an actions list. The actions players want to take are added to their respective actions when they submit new actions to the server.

2. *ec te* All entities now have a list of the actions they want to execute. The execute method of these actions (abilities) are called and the changes are recorded.

3. *end pdates* All changed entities are collected and sent to all party members.

esi n usti cation

- A boss that is bigger and does more damage.

We discussed abilities being on cooldown versus using mana to use them. The majority is pro cooldown, but we'll come back to this later when the implementation becomes relevant.

We set our next meeting for Thursday and gave everyone new tasks (well, most of them are continuations of the last ones). New ones are:

Steven: make it able to test multiplayer.

Thomas: work on ability selection

Bram: nalize the lobby activity.

Thursday was our second meeting. Everyone but Thomas was present. This is very inconvenient, as he did not nish his task.

Jelmer had continued with his rendering and the only thing left is rendering outside of a level.

Thijs was assigned the task to work on some sprites, and he made these animations:

Idle and walking animations for 4 player characters

Idle and jumping animations for a slime

Idle, walking and shooting animations for a skeleton

A oating animations for a boss

A tree

Robert is almost done with the dungeon generator, and he'll nish this for next Monday. We made some more dungeon speci cations for him to generate: enemy locations and start en goal positions.

Thijs and Bram discussed the layout and style for the di erent activities. Mainly, the lobby activity, where the party waits for the game to start and players choose their abilities, has to change so that abilities will actually be able to be chosen. This is a task given to Bram for next Monday.

Steven aims to make all game-logic that is not directly dependent on tasks given to others.

Thomas is given the task to start on the in-game UI.

e e

At Steven's request, because he had an appointment, the meeting started later than usual. This meant a shorter meeting, as Thomas had to leave us in the break. Sadly, not everyone was on time, so we e ectively had only 45 minutes for our meeting. This is very sub-optimal as we start to realize that there is still a lot of work to be done before we have an presentable application.

Robert has completed the dungeon generator, but will make some nal adjustments before he will merge it with the master branch on GitHub.

Steven did some work on passing objects between the server and the client. Still a lot has to be done in this part before we can properly test everything.