Radboud University

Final Project

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

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

Contents

1 Introduction

This is where the introduction is supposed to be ....[**einstein**]

2 Description

2.1 Focus on properties

2.2 Product justification

2.3 Specifications

3 Design

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

3.1 Global design

3.2 Detailed design

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

3.2.1 Server-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 different modes: offline and online mode (singleplayer and multiplayer). In both cases we run a server. When playing in offline 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 difference 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.

Server 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 first 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 off. 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.

Client 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.

3.2.2 Parties

The server keeps track of all the clients using parties. The usage 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. This party gets assigned a random so called `party id` which can be used by the players to identify their party and join the party of their friends.

As said in section **??** every party gets updated per game tick. We also want to use this update message to determine whether the client is still connected to the server. So upon connection with the server the client gets assigned a party just for that client. The game update messages from the server send to this party every game tick can then be used to determine the connection status. Note that the connection status does not make use of this feature yet, but this is for future implementations.

A client can be in one party at the time. When the client joins a new party it first gets removed from the previous party. The party also uses the ready status of all the clients to determine if the master of a party can start a new game. This is only possible when all the clients in the party are ready (except for the master, the master is set ready automatically when it send the `StartGameMessage` to the server).

3.3 Level design

Levels are generated each time a new game is started. The levels had to fullfill a number of requirements:

1. The Level should consist of a number of different rooms, connected with roads

2. The roads connecting different rooms should not intersect (so the graph should be planar)

In order to generate a level, we start off with a small grid that represents the final level on a more global scale. Each cell on this grid is called a Block because it represents a cluster of tiles on the full map, and it takes on of the values from the enum Block (Empty, Room, Road or RoomEdge). On this globalized map of the grid, we randomly pick a number of cells and designate them rooms. Then, we increase the size of the rooms by adding cells bordering room cells to the room, so we get bigger rooms. All of these room tiles So, we have a lot of individual rooms that are not connected. In order to connect them we see each room as a node in the graph representing the grid and we add an edge between every two rooms of length equal to the manhattan distance between them (as this is a better approximation than the eculidean distance considering roads will only be added above, to the left, to the right and to the top of a room). To get a subset of edges such that none of them overlap, we use the minimum spanning tree, which we find by applying Kruskal's algorithm. The method kruskals() returns an ArrayList of Edges showing all the edges in the minimum spanning tree, and we will use these edges to connect all the rooms. We go through all the edges and do a breadth-first-search from the beginning to the end of every edge and connect the two points with the shortest path. All nodes the shortest path passes through are marked on the global grid as 1×1 tiles (as Road from the enum Block). However, as rooms contain multiple cells roads are thinner than rooms. Now we have a level that is connected and is planar. To avoid the beginning and the end of the level spawning too close to eachother, we also monitored the two nodes which are the most apart, and put the beginning in one room and the end in the other. So, the player will awlays have to walk the maximum distance. Now we are ready to enlarge the miniature grid with which we have worked until now. For that we create a new grid that contains for every cell blockheight rows and blockwidth columns, and we simply enlarge the miniature grid, copying every cell from the miniature grid into the bigger grid but blockwidth $\times$ blockheight times bigger. In this process, everything that is Empty (so not a road or a room) is converted to an empty tile, while all Block.Empty tiles are converted to walls.
Now we have an empty level and have to fill ti wth monsters. We go through all the cells, randomly assigning turning a room into a room with skeletons, with

slimes and with slimes and skeletons. If we find that a cell is the finish we spawn the boss in that cell.

3.3.1 Level Rendering

GameView (client/view/GameView.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 figuring 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 different thread compared to updateLevel and they both use the renderItems list.

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

Texture (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.

Animation (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.

Generation of RenderItems 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 stuff with it. We didn't find it worthwhile to fix this before the deadline, as we would run the server from within Android Studio for testing.

3.3.2 Game logic

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

Tile 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:

Wall A solid tile.

Empty A non-solid tile.

Void A tile outside of the bounds of the level.

Start A tile on which players are spawned at the start of a game.

Finish A tile denoting the final room of a level.

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

Player Controlled by the actual players of the game.

Skeleton Enemy of player, shoot arrows and flees from the player.

Slime Enemy of player, attacks player by jumping toward them.

Boss Enemy of player, teleports and summons slime entities.

Arrow Flies in certain direction, can either hit player or non-player entities. Deals damage to entities it hits.

Level 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).

Ability 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 **executed**. 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 **invoked** first. 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:

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

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

Melee `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.

Game logic explanation 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. **Invoke** 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. **Execute** 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. **Send updates** All changed entities are collected and sent to all party members.

3.4 Design justification

4 Project management

Introduction

This section contains a log for the process of building our teams second app for the *Research and Development* Course at Radboud University. Our process will be presented in a weekly description of the active tasks that week, project meetings, assignment of tasks, problems we encountered, etc.

Week 19

After the spring break, in which we did not yet start our new project as most would've been unable to work on it, we decided that a meeting on Monday morning would be wise. This meeting took place in Mercator I, lasted from 10:30 till 12:15, and was attended by Jelmer, Robert, Steven, Thijs and halfway through also by Thomas. The main topic of this meeting was choosing which app idea we would run with. We had three prominent ones:

- A multiplayer dungeon crawler. This idea had us very excited at first, but we realized that this would mean a very similar project structure to the Sokoban app of the previous assignment, so we dismissed the idea.
- An medical application in cooperation with medicine students, as suggested by Patrick van Bommel. This idea seemed cool, but did not motivate us as much, as we would have to deal with outside requirements instead of our own ideas about what would make a good addition to the app.
- A Mario-party-like game with multiplayer minigames. By now we had settled on the idea of building a game, as we had a lot of fun doing the previous assignment. By making a game centred around minigames, we think we'll be able to make this process fun fr ourselves. We'll have to put some effort in making a connection between two phones.

The third is the idea we had settled on about halfway through the meeting. We made some sketched of the project layout and brainstormed on some minigames (like spyfall and charades)

We planned our next meeting for Thursday, third block. By then, Steven will have set-up a new Github repository.

Thursday we had our next meeting, which was attended by everyone. In the previous days, we had all thought about the idea of something using a server, and had become sceptical about how good of an idea it was. We discussed the following risks:

- Our app will probably be tested by one person, but they will have to be able to use the app's full functionality to properly grade it. Robert has sent an e-mail to our TA with questions about this, so we'll await their response and then look at this risk again.
- It is risky that our whole app relies on one connection. If something goes wrong in the connection part, the whole app will suffer from it.

- Four weeks for an app is already a short timeframe. Will we be able to afford the time to spend on multiplayer functionality? This is something we'll have to decide upon once we have more concrete ideas about games we'd like to make.

Because of the above mentioned risks, we further discussed the possibilities of single-player games, as the idea of the app being a game remained unchanged. Thomas and Thijs were charged with the task to think about games in the rogue-like genre, in order to see if such a game would be a valuable app idea. Steven will make the structure of a client-server program. This way, we can have some extra days to decide on our idea, as we'd like the whole group to be fully behind it, hard as it is, while also getting started on some code.

Week 20

We had our first meeting of the week on Monday morning, again. Over the weekend, more doubts about the party-game idea had arisen. Everyone was present, though Robert was there only the second half. First order of business was making the final decision for our idea. We all felt this was actually long due, and were all a bit frustrated because of this. So: today we were to make a definite decision and we would stick with it.

Our main two problems with the party game were design-problems: we would have to make some smaller, individually not very impressive things, that would have a very simple structure. This did not feel like enough of a challenge, even the multiplayer aspect taken into account. The second problem was that it would be hard to make it one well-rounded app. Additionally, because it had pieces that were so disjunct from others, we would evade the whole project idea of this course. To conclude: we switched back to the idea of one game. till multiplayer. The remaining of the meeting, we discussed possibilities. Firstly, we looked at possibilities to expand on one of the discussed minigames:

- Curve Fever
- Bomberman
- Spyfall

However, for the aforementioned reason of not being complex enough, we looked further. A game like hill-climb racing was discussed, but we did not have concrete enough ideas for implementing multiplayer into that game. In the end, we landed on the idea of a multiplayer rogue-like yet again. Here is a small run-down of how we envision it:

You create a party with your friends. Every player is presented a random selection of 'abilities', of which they can choose some. They enter a level of some kind and fight enemies. On the end of a level, there is a 'boss fight', after which the players are rewarded and can go down to the next level. Some opportunities for player-versus-player were mentioned, like the strongest player being a final boss after all levels had been completed, but we decided to postpone these ideas.

When Robert joined us for the meeting, he and Steven discussed the way we would tackle the multiplayer/server aspect code-wise. They proposed different approaches:

- Steven suggested to start with a full-scale client-server connection, so that we would not have to spend much time later on in the project to change existing code to make it suitable for multiplayer and such.
- Robert suggested to first make the game in single-player form and add multiplayer later, as this could prove tricky and time-consuming.

It was agreed upon that Steven would work on his vision for the next meeting, where we would decide if we would stick with it or not.

Thursday was our next meeting. Everyone but Steven was present, because he 'is too cool to use the bus'. We started a Discord-call with him however, and he showed us his work from the past days. In the end, he used Roberts approach. Thijs made a little character animation to look at possible styles and possibility for animations. We'll definitely use pixel-art, as it is less time-consuming while still being charming.

Main item on the agenda were dividing task:

- Jelmer: Graphics rendering
- Bram: Being able to create and join parties
- Robert: Dungeon generation
- Thijs: Sprites
- Thomas: Looking at abilities

Week 21

As usual, we had our first meeting on Monday morning. Everyone was present, though Robert only attended the second half. We started with discussing what everyone had achieved over the weekend. Actually, the divided tasks were pretty big, so everyone was still busy with them. Because of Jelmer's rendering, we could see the first animations on screen.

We then discussed the game envisioned in more detail:

- We want players to be able to play the game in short bursts (like in the breaks in the middle of lectures), so we're aiming at 15-20 minutes per total game. That means that we'll probably have 3 levels per game. In the future, we can expand this with an option to choose between a short, medium and long game.
- We'll start with making just 3 abilities, of which the player can choose 2 or 3 at the beginning of the game. The idea is that this will be easy to expand on later.
- For enemies, we'll start with 3 basic ones:
  - An enemy that does damage on touch: a slime. A slime will be able to make a jump attack.
  - An enemy that attacks with range: a skeleton that will shoot at players.

  - 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: finalize the lobby activity.

Thursday was our second meeting. Everyone but Thomas was present. This is very inconvenient, as he did not finish 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 floating animations for a boss
- A tree

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

Thijs and Bram discussed the layout and style for the different 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.

Week 22

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 effectively 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 final 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.

A big part of our struggles are that most tasks seem intertwined with others: Thomas would like to work on in-game abilities, but he can't until a proper connection is made. Bram would like to work on the lobby activity, but this has little meaning when there are no abilities.

An important breakthrough this weekend was that Bram was able to join Jelmer via the app! This means the connection that *is* being made, will probably work correctly later on. Still though, we realize that working with multiplayer was probably a very ambitious idea and the risks we were afraid of in earlier weeks are staring us in the face.

Bram spent most of his weekend working on a bug that causes the player not to enter a good multiplayer mode after they have first switched to singleplayer mode – as a side note: there is a singleplayer toggle added, which just 'tricks' the app into thinking it's local device is the server, for the purpose of testing and later on for true offline game possibilities.

Thomas has not been busy yet with the UI and controls. We urge him to look at this quickly, we want (theoretical) player movement by Thursday!

Thursday was Whitsuntide, and the university was closed. We did not arrange a meeting, as most had family-obligations to attend, but we discussed some important things via Whatsapp, so that everyone would still be able to work this weekend.

There were some problems with the dungeon generation (namely that rooms are not connected) that Robert fixed immediately at Wednesday.

Thursday, Thomas added a joystick for movement, but this is not yet fully integrated in the app.

5 Evaluation