



Eötvös Loránd University

Faculty of Informatics

Implementation of a platform game

Kitlei Róbert

Assistant lecturer

Andrey Khasanov

Software Information Technology BSc

Budapest, 2014



EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

THESIS TOPIC DECLARATION

Student:

Name: **Andrey Khasanov**

Code: **ANKTAALELTE / VXDW14**

Type: **full-time student**

Course: **Software Information Technology BSc**

Supervisor:

Name: **Róbert Kitlei**

Affiliation with address: **ELTE Faculty of Informatics**

Department of Programming Languages and Compilers

1117 Budapest, Pázmány Péter sétány 1/C.

Status and qualification: **Assistant Lecturer, MSc in Computer Science**

Title of the Thesis work:

Implementation of a platform game

Topic of the Thesis work:

Developing an implementation of the classic game "Prince of Persia" in Java using libgdx library to create executables for Android, Windows and other operating systems.

I agree to be the supervisor:



(Signature of supervisor)

I hereby request the approval of the topic of my thesis work.

Budapest, 2013/12/13



(Signature of student)

The topic of the thesis work has been approved by ELTE Faculty of Informatics.

Budapest, 2013.12.23.



(Signature of approving
Department Leader)

Contents

1	User manual	5
1.1	System requirements	5
1.1.1	Operating systems	5
1.1.2	Minimal requirements	5
1.1.3	Recommended requirements	5
1.2	Starting the game	5
1.3	Game overview	5
1.3.1	Background story	5
1.3.2	Genre	5
1.3.3	Gameplay	6
1.3.4	Screens	6
1.3.5	Controls	7
1.3.6	Cheats	7
1.3.7	Screenshots	8
1.4	Differences from the original game	13
1.4.1	Controls	13
1.4.2	Game elements	13
1.5	Errors and bugs	14
1.5.1	Error messages	14
1.5.2	Known bugs	14
2	Developer manual	15
2.1	Background	15
2.1.1	Acknowledgements	15
2.1.2	Environment	15
2.2	Projects	15
2.2.1	Accessing the source code	15
2.2.2	Structure	15
2.2.3	Editing	16
2.2.4	Executing	16
2.3	Classes and diagrams	17

2.3.1	Screens	17
2.3.2	LevelReader.....	18
2.3.3	Level.....	18
2.3.4	LevelScreen	19
2.3.5	Event	20
2.3.6	Object	21
2.3.7	GeneralBlock and blocks	22
2.3.8	Character	25
2.3.9	Kid.....	26
2.3.10	WorldRenderer	27
2.3.11	LevelRenderer	29
2.4	Game flow chart.....	31
2.5	Data file format.....	32
2.5.1	Screens	32
2.5.2	Blocks and modifiers	32
2.5.3	Room linking	35
2.5.4	Starting location.....	35
2.5.5	Events.....	35
2.5.6	Other data	36
2.6	Other game features	36
2.7	Testing.....	36
2.8	Possible improvements.....	38
2.8.1	Initial gravity	38
2.8.2	Guards and fighting.....	38
2.8.3	Palace levels	38
3	List of references.....	39

1 User manual

1.1 System requirements

1.1.1 Operating systems

Windows XP and higher (32/64bit), Linux (32/64-bit), Mac OS X (32/64bit)

1.1.2 Minimal requirements

- Java Runtime Environment 6 (jre6u29) or higher
- Intel P4/NetBurst architecture or its AMD equivalent (AMD K7). (Any CPU with MMX/SSE instructions)
- 256 MB RAM
- GeForce3 or Radeon R100 (7xxx) and up. (Any GPU with OpenGL 1.3 support)
- 13 MB for the game executable

1.1.3 Recommended requirements

- Java Runtime Environment 6 (jre6u29) or higher
- Intel Pentium D or AMD K8-Based CPUs and better.
- 1 GB RAM
- GeForce 7300 GT or ATI Radeon HD 2400 XT and higher
- 13 MB for the game executable

1.2 Starting the game

To start the game on Windows, double click the executable JAR file. If this doesn't work, run the batch file `PoP.bat` that is provided with the executable.

To start the game on Linux or Mac OS, try to double click the executable JAR file. If this doesn't work, open the terminal, change to the directory where the executable is located and run:

```
java -jar PoP.jar
```

1.3 Game overview

1.3.1 Background story

“In the Sultan's absence the Grand Vizier Jaffar rules with the iron fist of tyranny. Only one obstacle remains between Jaffar and the throne: the Sultan's beautiful young daughter...

Marry Jaffar... or die within the hour. All the Princess's hopes now rest on the brave youth she loves. Little does she know that he is already a prisoner in Jaffar's dungeons...”

1.3.2 Genre

The genre of the game is a cinematic platformer, which is generally a platformer with game physics that resemble the real world physics. An example of this would be walking a step. Since you cannot jump or perform some other action during the step duration in real world, you cannot

perform them in the game, too; however, other types of platformers would let a player jump anyways.

1.3.3 Gameplay

The player controls the prince which has to save the princess in a given time limit, while trying to survive by dodging the traps and overcoming the obstacles. The player can freely move around the level and interact with the world by using ground buttons to unlock the gates and the final door, breaking the loose tiles on the floor or drinking potions that were found during the play.


Certain actions, such as falling from a considerable height, drinking poison or getting hit by a broken board lower player's health, which can be later replenished by using the healing or life potions, which can be found around the level.

The number of lives is shown on the bottom left of the game screen as the red triangles. The triangles which are filled represent player lives and the empty triangles represent the number of lives that the player is missing and could potentially heal back. If the player drinks the healing potion while he has all triangles completely filled, there will be no result and the potion will be wasted. If the player finds and consumes the life potion, it will grant him one extra live and, thus, one extra filled triangle and the potion will also heal all the missing lives.

The number of filled triangles cannot go below 1. If the player falls onto spikes, gets hit by the iron chopper or loses his health point while only having one remaining, he will die and the level will need to be manually restarted.

The goal of the game, as mentioned in the beginning of this section, is to find the princess and free her from her captivity. If the player has successfully completed the task, he is considered a winner and the Prince of Persia.

1.3.4 Screens

When the game is started, the first thing that is shown is the title screen (Figure 1.1). This screen can be skipped by either pressing  or by clicking anywhere on the screen inside the game window.

Once the title screen is skipped, there is a loading screen which processes the level data and loads the game data, such as images and sounds, into the game.

After the loading is complete, there is a game screen where the player can control the game character using the controls listed in Section 1.3.5. When the player enters another room by moving in one of the possible directions, the game screen is updated accordingly. When the level is complete, the new loading screen appears and the next level is ready to be played. The game screen screenshots can be seen in Figures 1.2 – 1.10.

If all levels were completed and the game was won, there will be an ending screen, which explains what happened after the game events. It is not possible to get back to the intro screen from the end screen and the game has to be closed. It can be closed by either clicking anywhere on the screen, pressing **ENTER** or **ESC**, or by closing the game window. This screen can be seen on Figure 1.11.

1.3.5 Controls

← **→** – Movement

SHIFT + **←** **→** – Walking

↑ – Scaling, clipping on a block

↑ + **←** **→** – Running jump

↓ – Crouching (sitting), clipping down the block

SHIFT – Picking items from the ground

R – Restarting the level

ENTER – Skipping title screen

ESC – Going to previous screen or exiting the game

1.3.6 Cheats

Skipping a level while playing can be achieved by pressing **CONTROL** + **S** + **L** (**Skip Level**). This will skip to the next level and if there are no more levels, it will skip to the ending screen. Depending on the level, the starting level screen and starting position will vary, but this allows testing the next level right away.

It is also possible to instantly finish the game and see the ending screen by using the

CONTROL + **E** + **G** combination of keys (**End Game**).

1.3.7 Screenshots



Figure 1.1 – Titles screen

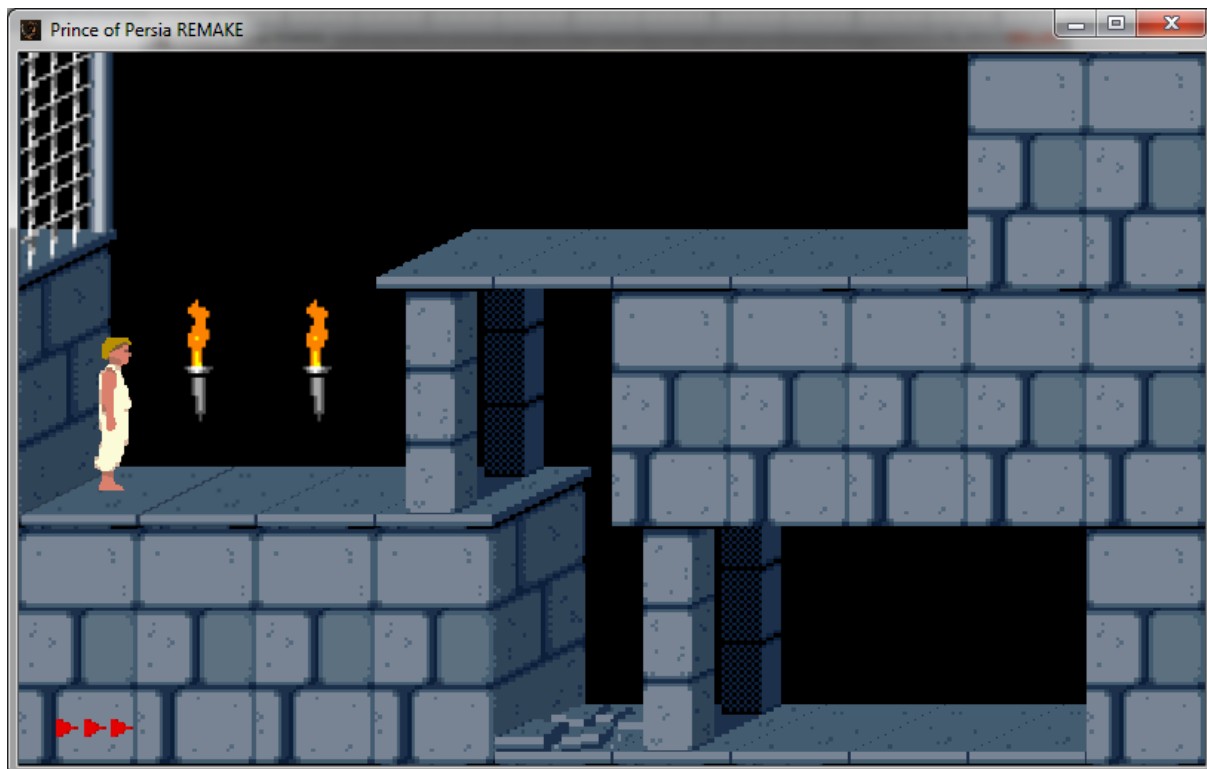


Figure 1.2 – First level



Figure 1.3 – Clipping down the block

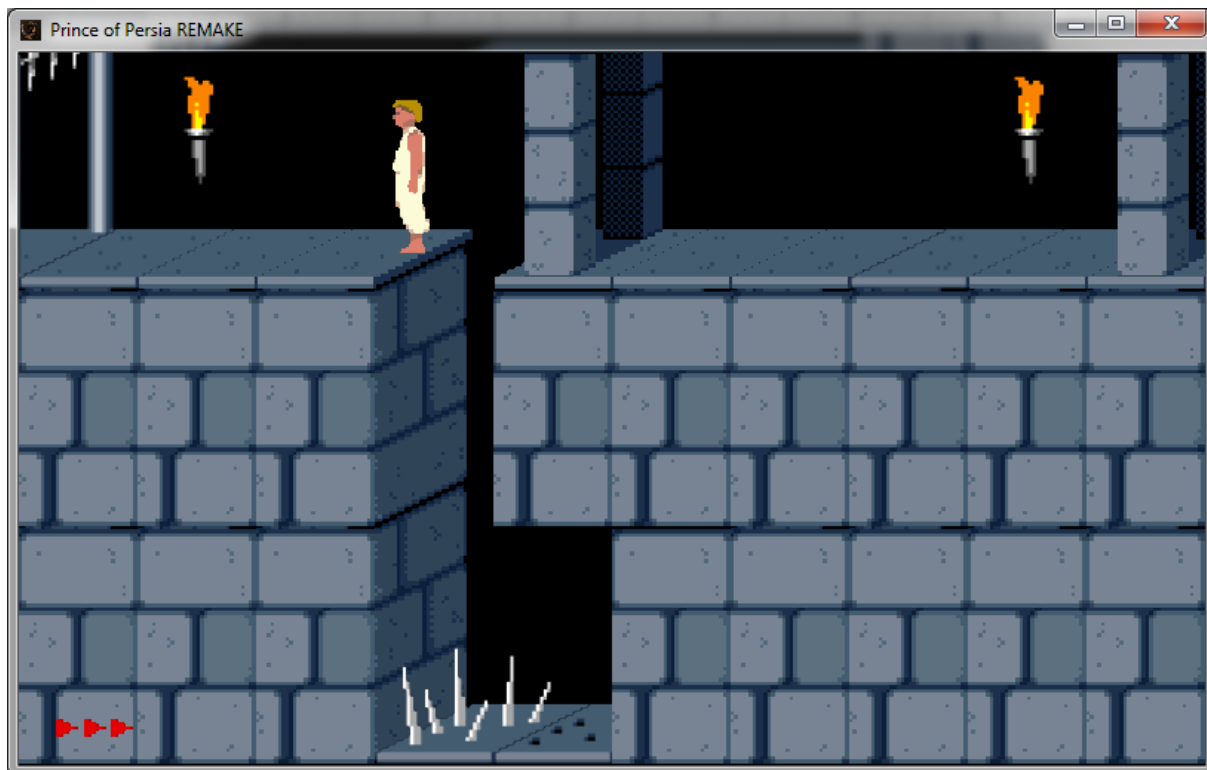


Figure 1.4 – Activating the gate and spikes

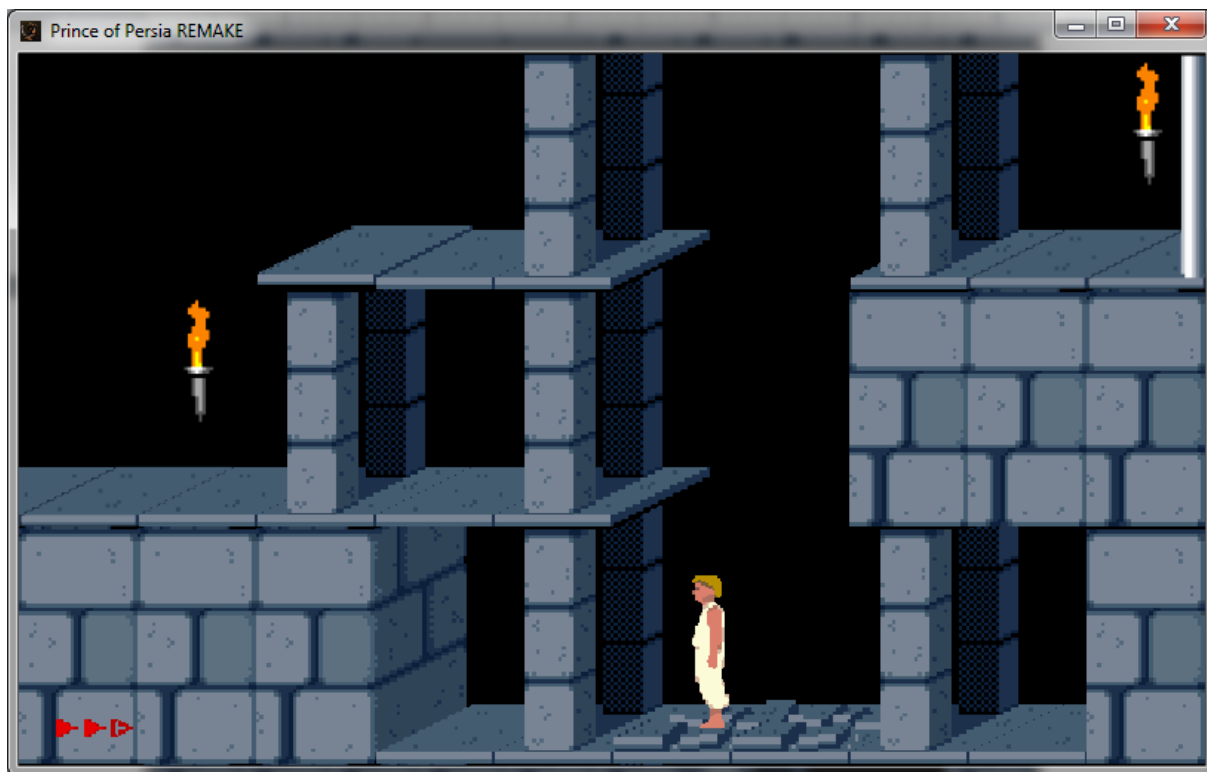


Figure 1.5 – Falling down and losing a life

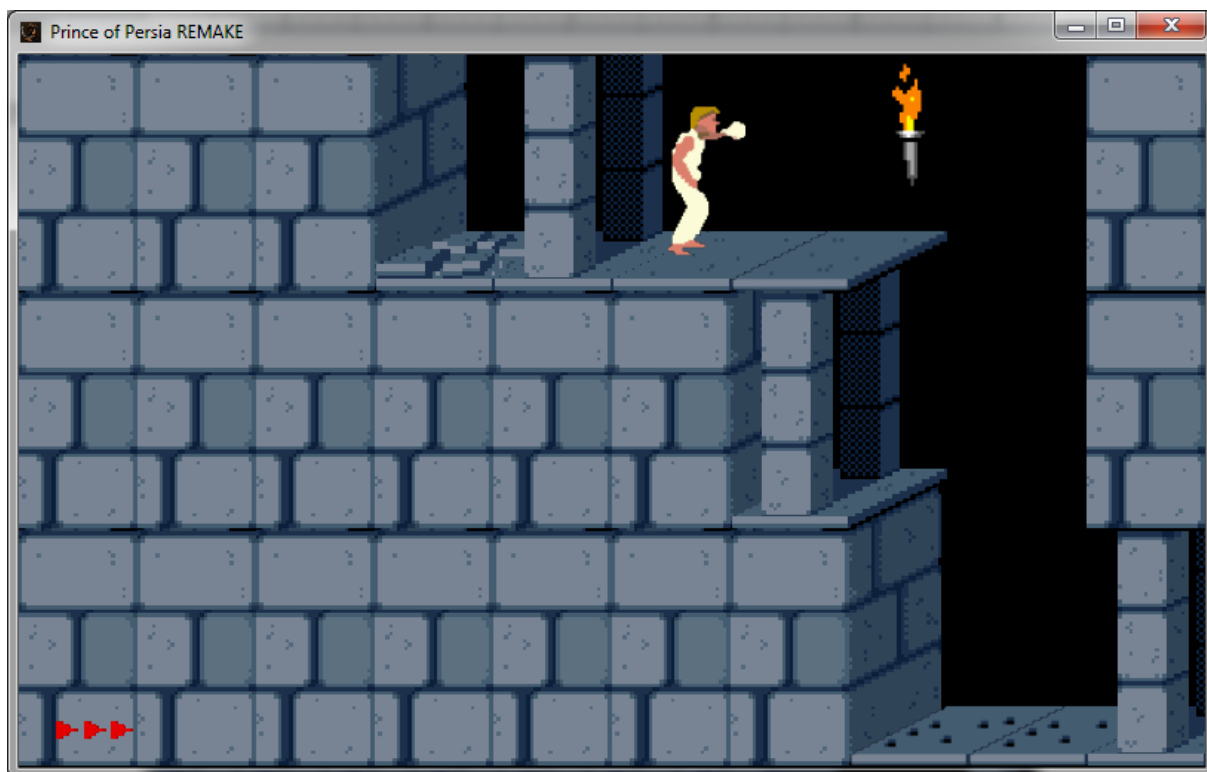


Figure 1.6 – Drinking a potion and healing back

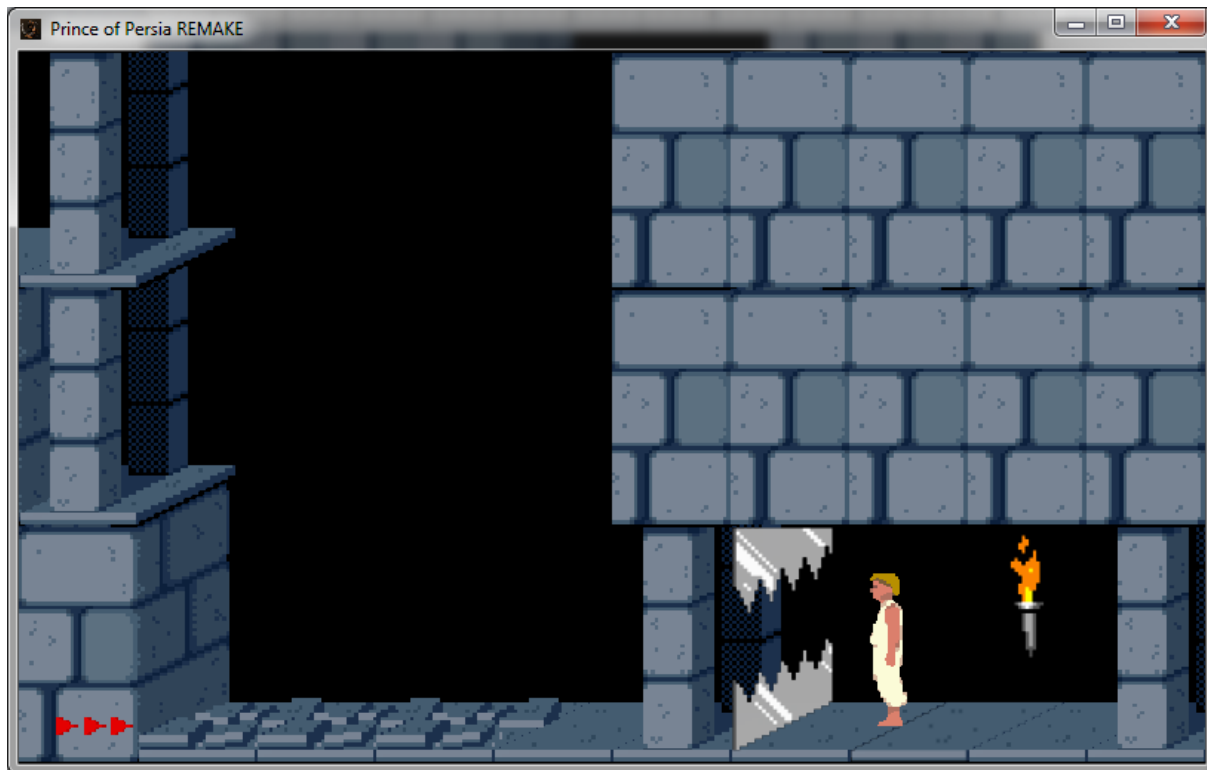


Figure 1.7 – The chopper



Figure 1.8 – Falling down with a broken tile



Figure 1.9 – The deadly fall

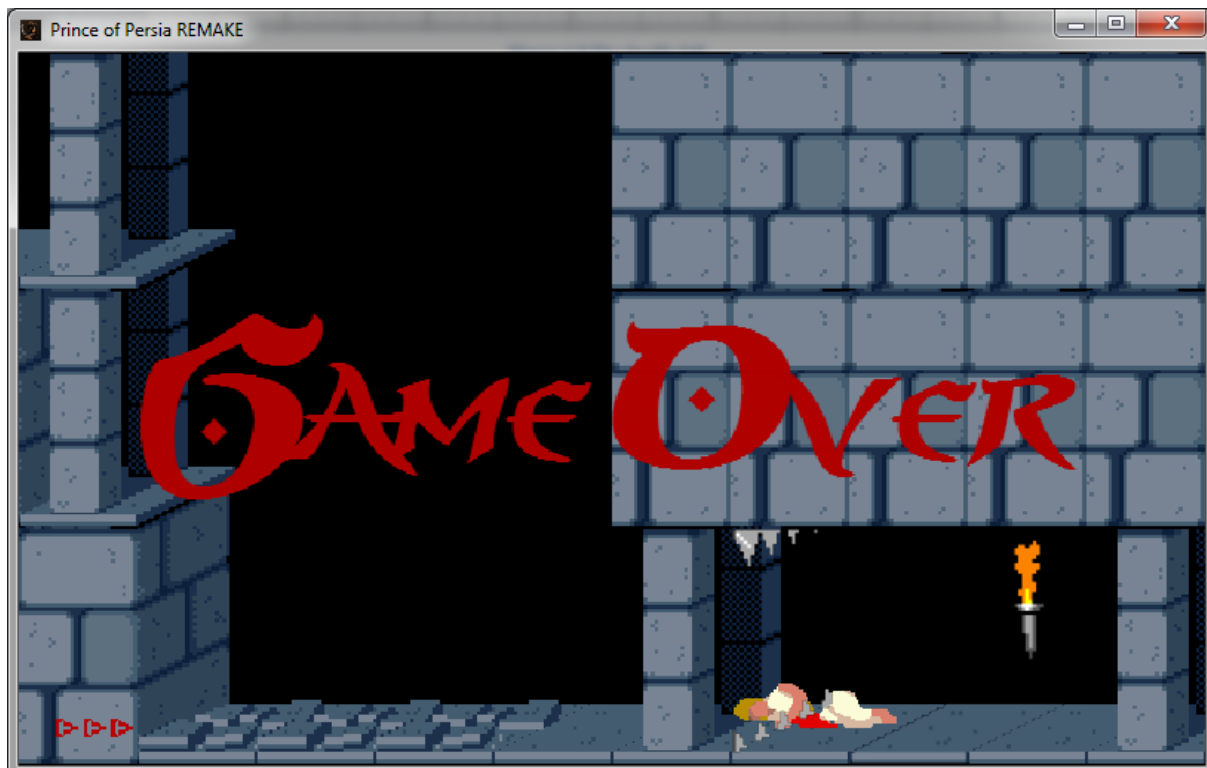


Figure 1.10 – The chopped prince

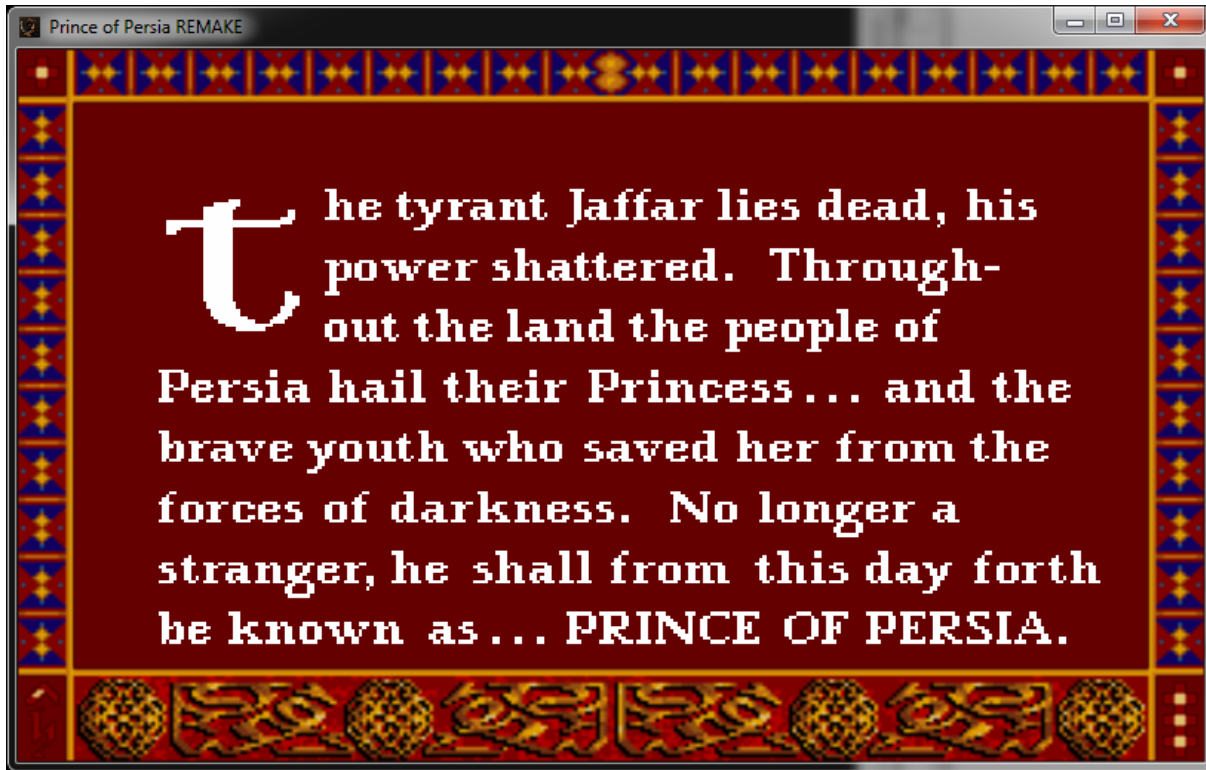


Figure 1.11 – End screen

1.4 Differences from the original game

1.4.1 Controls

There are very few differences in controls between the original game and this implementation. All of the movement controls are exactly the same with an exception that holding onto a block is not possible here. One of the levels from the original game heavily depended on this trick and if the player didn't try to hold to the block, he would fall down and die. This was the reason that the mentioned level was modified using a binary editor so that the user starts at a safer point. Also, since the battle system was not implemented, this game does not contain fighting moves and controls.

1.4.2 Game elements

Since the battle system are not implemented, there are no guards in this version of the game, as well as there are no moving skeletons or prince shadows. There is no mirror at one of the levels, as there is no point in it without the shadow of a prince.

There are a several additional graphics, such as the loading screen and the "Game Over" text overlay when the player dies during the game.

One other thing that this game is missing is the time limit. The original game has a 60 minute time limit and if the player doesn't complete the game in given amount of time, the game is over.

The reasoning behind the dropping of a time limit was to allow the player have more time to play the game out without worrying about the lack of time.

1.5 Errors and bugs

1.5.1 Error messages

There are no error messages from the game itself; however there might be an error from LWJGL (Lightweight Java Game Library) on Mac OS operating system. This is due to the fact that Mac OS version of the game should be compiled on a Mac OS operating system.

If you are a developer or an advanced user, check Section 2.2.1 of this documentation, otherwise, if you get this error, it is not possible to execute the game.

1.5.2 Known bugs

In some cases it is possible that in the beginning of the game the player falls through the floor and dies due to falling for a large amount of time. This largely depends on the computing power of the processor the game is running on. In this case try to update your video card and/or processor with a newer one.

It is also possible to perform a running jump on a closed gate and the player will land on the gate as it is shown in Figure 1.12, but the player will not be able to pass the gate, since the gate is locked. This can only be achieved by performing a running jump and not a straight jump (scaling). If the player moves a bit to the right, however, the player will fall down.



Figure 1.12 – Closed gate bug

2 Developer manual

2.1 Background

2.1.1 Acknowledgements

The source code of this project uses several level data file processing algorithms and enumeration types from FreePrince by Princed Development Team [5] written in C, and OpenPrince by Biro R. [6], written in Java. The use of such external code is documented where appropriate.

2.1.2 Environment

There are several tools needed to work with the source code of this program. First is the Android Development Tools [1]. It includes Android SDK Tools, Eclipse with ADT plugin, Android Platform-tools and Android system images. It is enough to use Eclipse only, if you do not plan to have Android version, however, since both Android and desktop versions of the program depend on the same source code from **PoP** project, it is advised to import and keep Android project in Eclipse's workspace.

Another tool used is libgdx [2], a Java game development framework, which provides APIs to deal with graphics, sounds, user input and so on. Libgdx also allows using one source code project to generate cross-platform projects. Libgdx currently supports Android, Windows, Mac OS X, Linux, HTML5 and iOS.

2.2 Projects

2.2.1 Accessing the source code

First open the terminal or the command line, change directory to the one which you will use for the projects and type:

```
git clone https://github.com/Aliamondo/pop.git
git clone https://github.com/Aliamondo/pop-android.git
git clone https://github.com/Aliamondo/pop-desktop.git
```

Import all projects to your Android Development Tools or Eclipse by using **File - Import**.

To create an executable for Windows, Linux or Mac OS X right-click the **pop-desktop** project and export it as a Runnable JAR file.

To create an executable for Android right-click the **pop-android** project, choose **Export** and select **Export Android Application**.

2.2.2 Structure

There are three different folders and, thus, three different projects in Eclipse.

PoP is the source code for all projects.

PoP-android is an instance of `AndroidApplication` with configurations described in `MainActivity.java`, which provides framework for **PoP** source code.

PoP-desktop is an instance of `LwjglApplication` with configurations described in `Main.java`, which provides the framework for **PoP** source code.

2.2.3 Editing

If one or more files of **PoP-android** have been modified from the `/assets` folder (game data, such as images, levels and sounds), then the whole project needs to be refreshed in Eclipse, as **PoP-desktop** uses its game data from **PoP-android**. To perform the project refresh it is enough to select **PoP-android** project in Package Explorer and either press F5 or right click the project and select Refresh.

2.2.4 Executing

If the project needs to be executed, right-click **PoP-android** or **PoP-desktop** projects and run them as Android Application or Java Application.

Note: If you want to run **PoP-android**, you need `adb` service to be running and an android device or android emulator to be set up. However, currently the **PoP-android** project is not performing well due to a gravity issue specified in Section 1.5.2.

2.3 Classes and diagrams

2.3.1 Screens

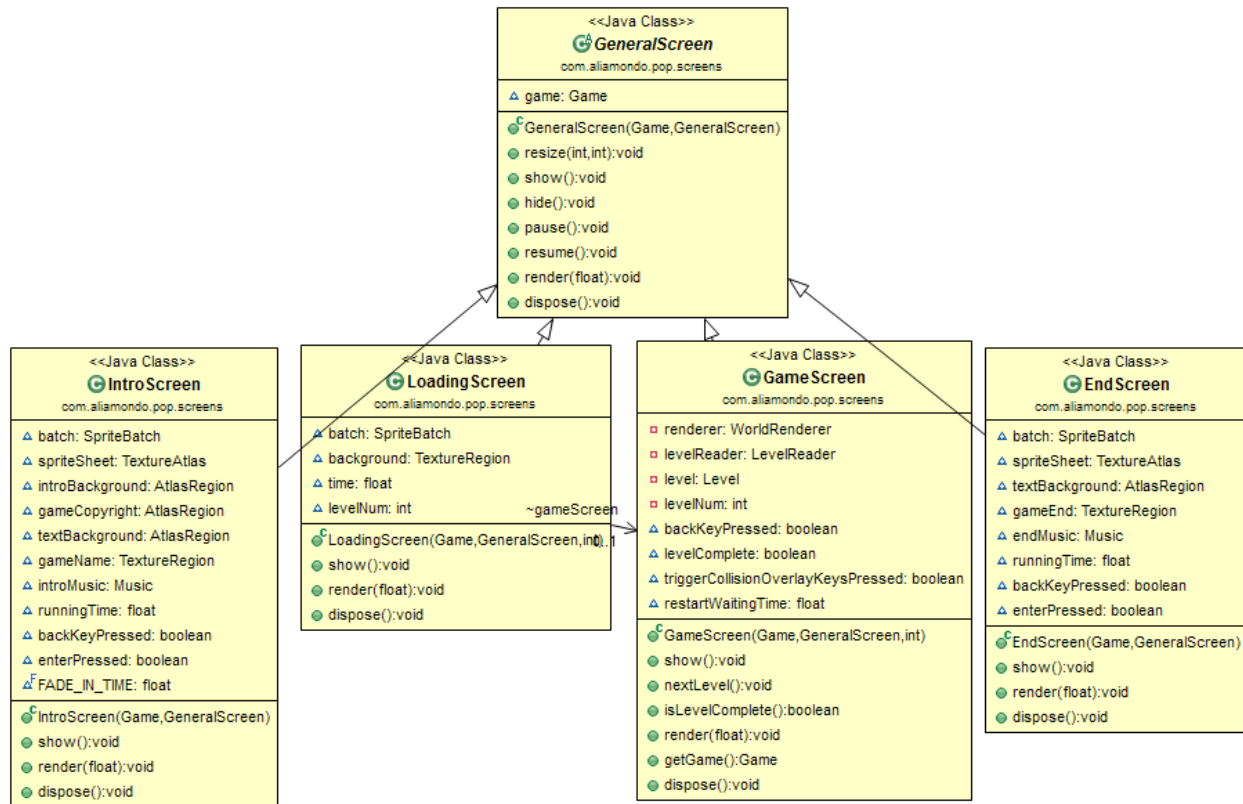


Figure 2.1 – Screens diagram

All screens extend **GeneralScreen** and they are responsible for what happens on the application screen. They all have render methods which is called at every run loop. If one screen needs to be switched for another, it calls `game.setScreen(Game game, GeneralScreen screen)`, which uses the **Game** instance to set a new screen and then disposes the screen that was previously in view. This dispose method is implemented in the abstract **GeneralScreen** class, with its child classes overriding it and calling `super.dispose()`.

GameScreen is also responsible for the user input. It looks for the predefined set of buttons that control the playing character and when those buttons are pressed, the **GameScreen** modifies the playing character's State, if needed. Some actions share the buttons, so **GameScreen** is also responsible for figuring out the type of an action. For example, getting on a block and jumping straight (scaling) is achieved via the same UP button and depending on if the character can actually get on a block (there is a free block next to it), the needed state is chosen.

2.3.2 LevelReader

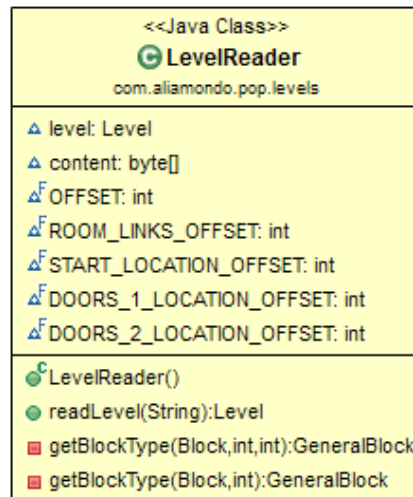


Figure 2.2 – LevelReader diagram

LevelReader is responsible for data file processing of a specified level. It reads the file into `content` as an array of bytes and using predefined constant offsets it finds the level data needed to construct the level, such as blocks, modifications, room connections, initial location, etc. To find out more about the structure of the data file see Section 2.5.

2.3.3 Level

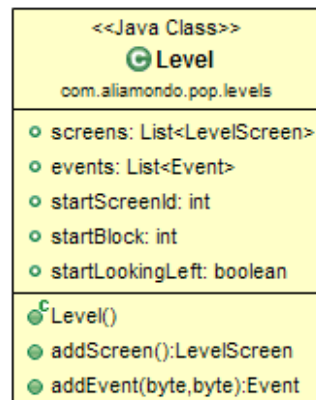


Figure 2.3 – Level diagram

Level class contains the data of a current level. It has `screens` and `events` lists, which will be explained in Sections 2.3.4 and 2.3.5, respectively. All the data, such as block positions and modifications of the blocks are stored in respective position of `screens`.

2.3.4 LevelScreen

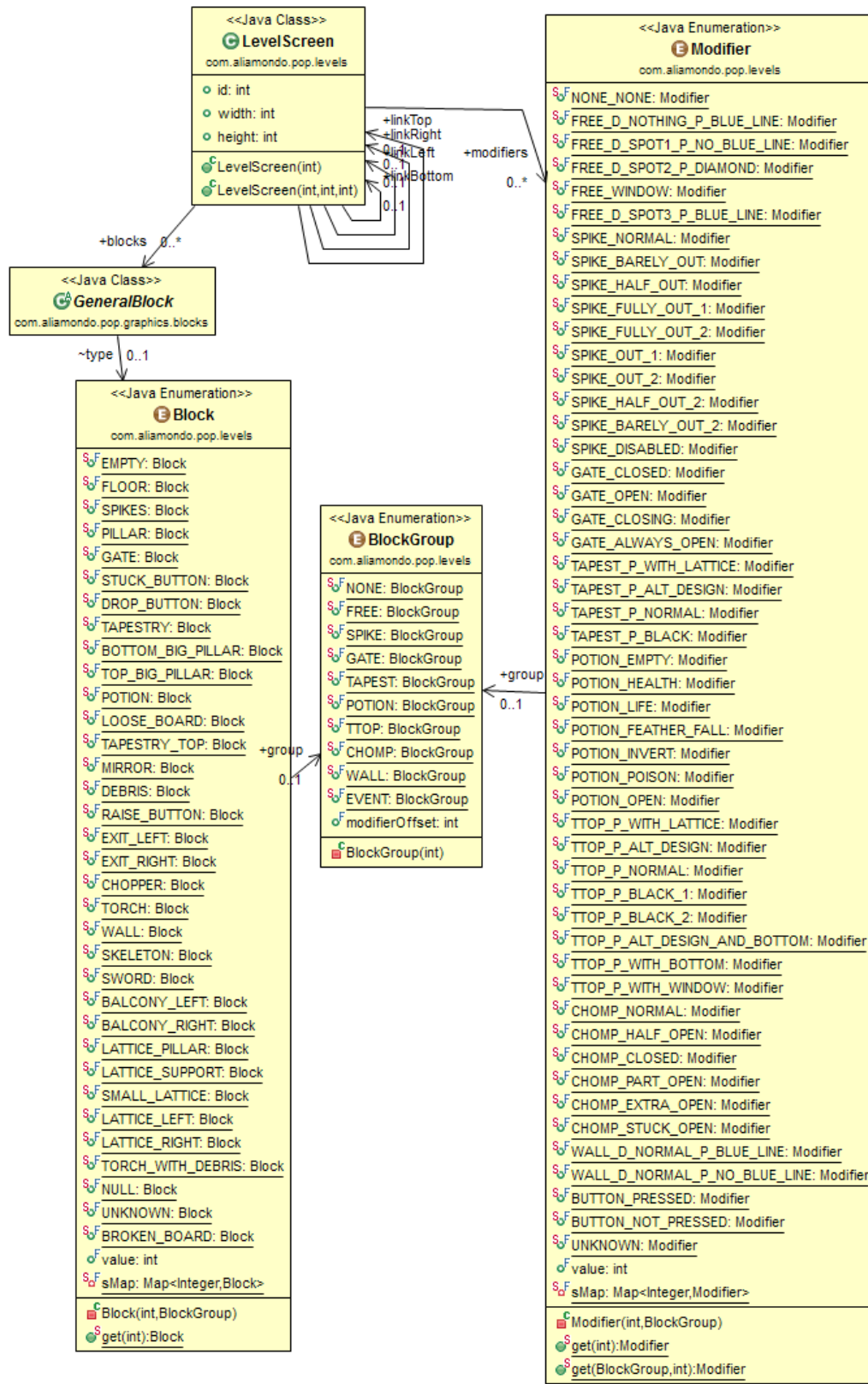


Figure 2.4 – LevelScreen diagram

LevelScreen object is essentially one of the rooms of the level. There are 24 different screens for each level and each LevelScreen object has an id value which is used in calculating the connections between the rooms. LevelScreen also contains blocks and modifiers lists, which can be seen as connecting arrows on Figure 2.4. The blocks list has size of 30 and it contains the block types for current LevelScreen. In the game these blocks are put on screen by their index number. There are 3 rows and 10 columns of each LevelScreen, so to calculate the actual position of the block with index x , it is enough to perform an integer division $x / 10$ to calculate the row of the block and $x \% 10$ to calculate the column of the block.

Every LevelScreen also contains the modifier list, which has similar structure to blocks, but it contains modifications for the blocks, such as if the gate is locked or open or the buttons in the game are pressed by the prince. This list is dynamic, because it will need to be changed constantly in case of spikes, chopper, gates or doors, because they always change their state (opening/closing, etc.)

LevelScreen object also contains 4 pointers to other LevelScreen objects, which can either be a proper LevelScreen or a null. These are connections in between the screens which are used to move the prince accordingly or to render the left and bottom screens, because they need to be seen in the game.

The blocks and modifiers types can be seen in the Figure 2.4 and they are interconnected by BlockGroup enumerator, which specifies the group type of the block and its possible modifications, as, for example, BUTTON_PRESSED modification cannot be applied to GATE block type.

2.3.5 Event



Figure 2.5 – Event diagram

Event object contains data such as the screen id and the block position, which needs to be opened or closed. Usually the block is a gate or a door and the action will open or close it, but if the block is of any other type, the event will not happen.

If triggersNextEvent value is true, then all of the events in the events list of Level (Section 2.3.3) with an index larger than current event will be executed until one of them will have

`triggersNextEvent` set to `false`. This is helpful if one buttons needs to open two or more doors at once, but it can also be used for other purposes.

Event object doesn't store the type of event, so to figure out if the gate or door need to be opened or closed, it is enough to check the type of the block user is standing on. As it can be seen in Figure 2.4, there are two different types of blocks: `DROP_BUTTON` and `RAISE_BUTTON`, which handle closing and opening of the doors, respectively. Both of these types of blocks are later registered as a part of `EVENT` type of `BlockGroup`, however, it cannot be seen from the diagram itself.

2.3.6 Object

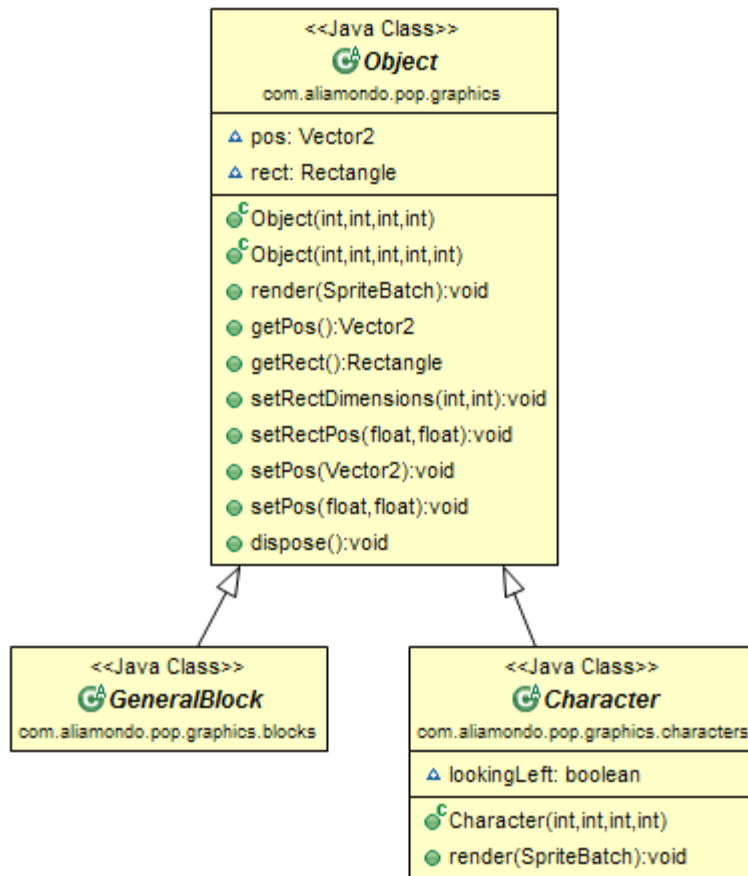


Figure 2.6 - Object diagram

`Object` is a main abstract class which has the main function implemented. All the other game objects, such as blocks and characters are extending it. `Object` contains the position vector and the rectangle around the object, which is used in calculating the collisions during the game.

2.3.7 GeneralBlock and blocks

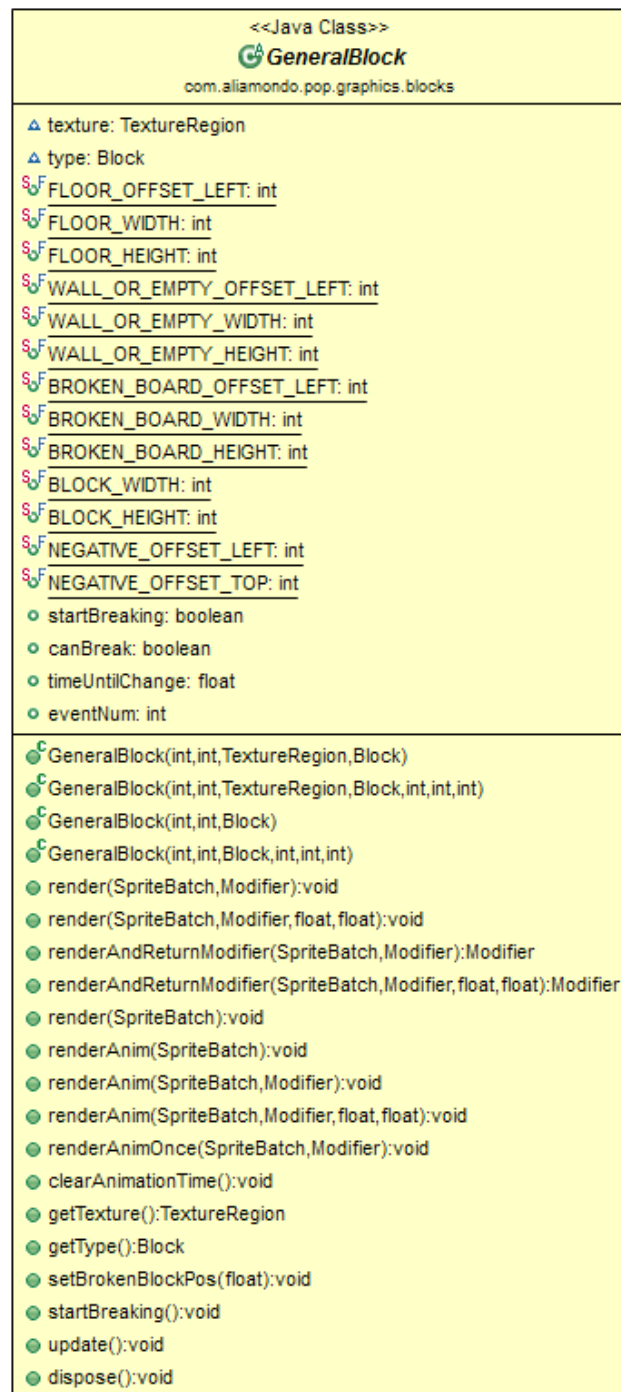


Figure 2.7 – GeneralBlock diagram

GeneralBlock is a parent abstract class for other block classes, which you can see in Figure 2.8. It also contains a texture, which can be specified at the initialize time and then the rendering of that texture is done automatically, or if the texture is null, the rendering needs to be done manually. This is needed for torches, gates and other animated or dynamic types of blocks.

There are several different `render` methods which make it easier to display the block on the screen. The regular `render()` is the default method and depending on the arguments it got, it can either render the block by default, render the block including the `Modifier` that this block has, which is helpful for some blocks like `Potion` or `Torch`, and render the block at a specific position, which overrides the position vector in the block, which is especially helpful to render the rooms on the bottom and on the left.

Other two methods are `renderAnim()` and `renderAnimOnce()`, which are very similar to `render()` described above, with the only difference between them is that `renderAnimOnce()` is meant for blocks which need to be animated for a short period of time, but are otherwise non-animated static blocks, such as `LooseBoard`, when it breaks and becomes `BrokenBoard`. `renderAnim()`, on the other hand, is meant for the blocks that are animated all the time, such as a `Torch` or a `LooseBoard` during the shaking because of the player movements.

The last rendering method is `renderAndReturnModifier()`, which, as it can be expected from the name, renders the block and then based on the previous modifier and the previous state of the block, this method returns a new `Modifier` for the block. The assignment of the returned `Modifier` to the `Modifier` of the block should be done manually during the rendering. This method is absolutely needed for blocks with dynamic modifiers, such as `Gate`, `Spikes` and `Chopper`, which can restrict the player movement or kill the player during one or several of their `Modifier` states.

There are two more methods which are meant only for `LooseBoard`, such as `startBreaking()`, which changes the state of the block to broken, if the block can actually be broken, and `setBrokenBlockPos()`, which sets the Y-coordinate of the falling `BrokenBoard`. This is needed because there is an offset for the collision rectangle of every single block to make the collisions look realistic. This means that the block rendering X-coordinate is not the same as the block's collision rectangle X-coordinate and the usual `setPos()` specified in the `Object` will move the coordinates of the collision rectangle to the rendering coordinates of the block.

The variables that `GeneralBlock` has are mostly constants of the offsets of different types of blocks; however, it also has `canBreak` boolean, which is true if the block is a `LooseBoard`. `startBreaking` and `timeUntilChange` are needed for an animation of the `LooseBoard` becoming a `BrokenBoard` and `eventNum` simply stores the event number of the event that the block links to, which is needed for `DropButton` and `RaiseButton` blocks.

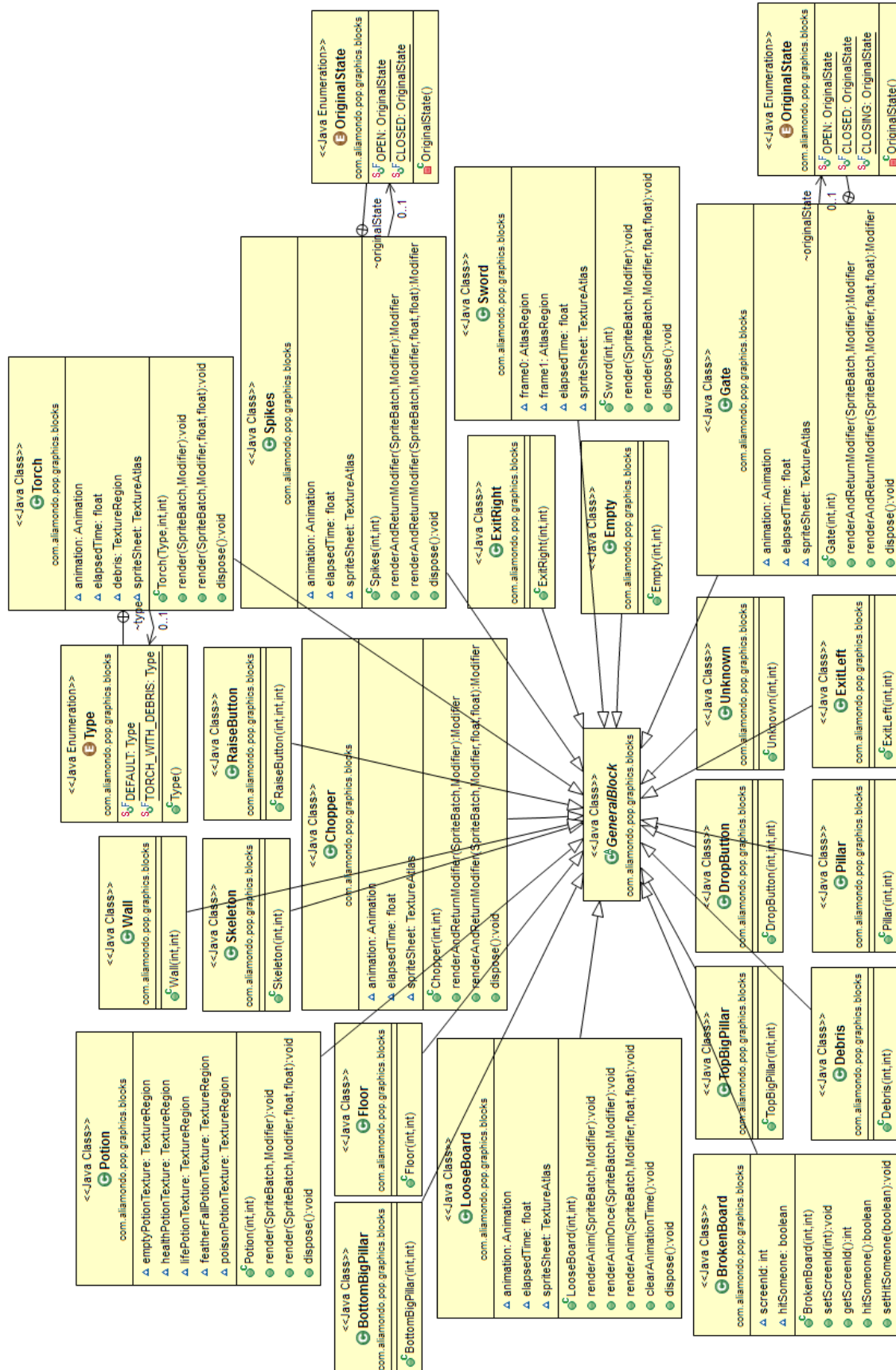


Figure 2.8 – GeneralBlock children diagram

2.3.8 Character

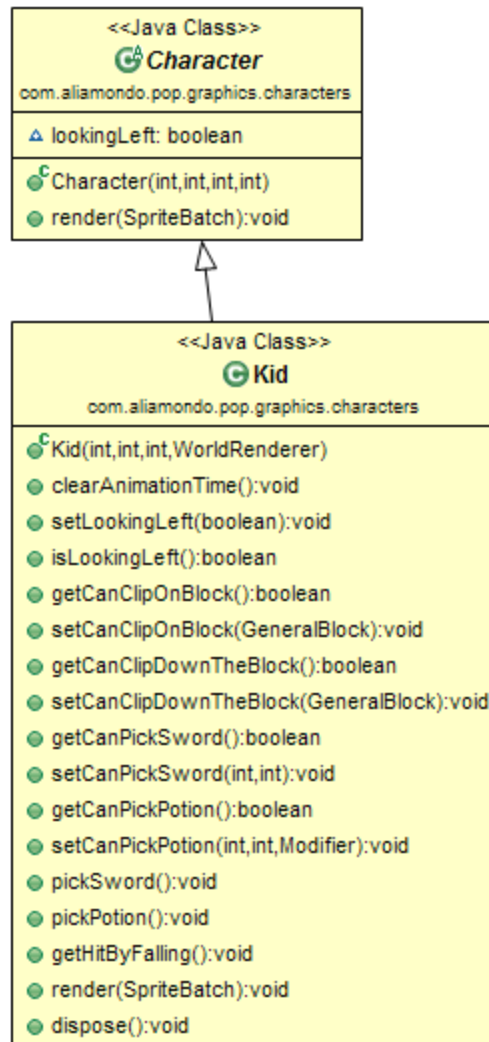


Figure 2.9 – Character diagram

Character is also an abstract class, which is responsible for all the characters in the game, such as the kid or guards. Its structure can be seen in Figure 2.9. Currently it does not have any special methods and has only one boolean value of `lookingLeft`, which is responsible for the direction the character is facing.

Character is also supposed to be a superclass for **Guard** classes; however, the **Guard** classes are not implemented due to the complexity of their implementation. The specific reasons and issues are described in Section 2.8.2.

2.3.9 Kid

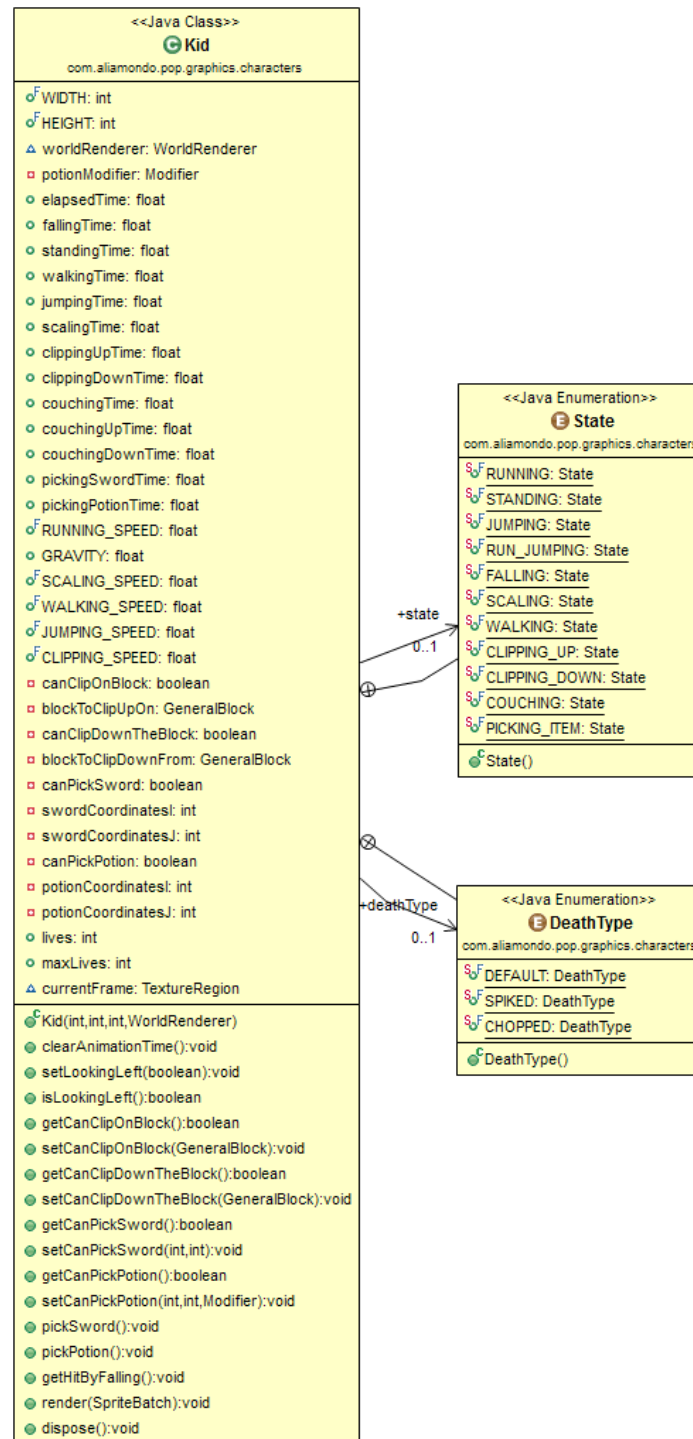


Figure 2.10 – Kid diagram

Kid is the controllable object which is used to play the game out. It has animation and texture variables omitted from Figure 2.10. There should only be one Kid object at a given time, however, it is not a must.

This object contains the speed constants to calculate how fast the kid should be falling or running, depending on the `State` of the kid. All such calculations are performed in the `Kid` object itself during `render()` function.

`Kid` has time variables which are needed to finish animations before starting new ones, because we cannot jump while we are walking a step or run when we are trying to get on a block. These time variables are updated according to the current state of kid, so the walking time variable won't be updated during falling or crouching.

The falling time variable is also responsible for the damage that kid gets once he has landed. If this variable is larger or equal than a predefined minimum value and smaller than the maximum allowed value, then the kid will get hurt and lose one health point. If the falling time was larger than the possible maximum, the kid dies on impact and the level needs to be restarted. In case of falling time being smaller than a needed minimum, there is no damage taken by kid on impact and the game continues without major changes.

`Kid` object also has methods such as `pickSword()`, `pickPotion()` or `getCanClipOnBlock()`, which are updated as kid steps on or near the specific block types. After picking sword or potion, the block becomes of a `Floor` type and kid cannot perform any other action on it after that. `getCanClipOnBlock()` is only true when the kid did a straight jump (scaling) and there is a block on the side of kid's looking direction; the block has to be next to the kid. There should also be no blocks which could obstruct the getting on the block action, such as a non-empty block over kid's head or non-empty block over the block that kid wants to get on.

2.3.10 WorldRenderer

`WorldRenderer` is responsible for all the visual updates in the `GameScreen`, as well as playing background music or death sounds for the game. It stores the instances of `Kid` and `LevelRenderer` to make the rendering of the world easier.

`WorldRenderer` is also responsible for checking the collisions of the world with kid and it also checks if the kid is in the air or if he is standing on the ground. If `WorldRenderer` finds out that the kid is in the air, it does **not** change kid position, but rather sets its state to `FALLING` and the `Kid` object does the rest itself. While kid's gravity is checked, it is also checked which block kid is currently standing on as it could be a button, sword or a spike and the movement in that area could lead to the opening of the door or to the death of the kid.

The collisions that `WorldRenderer` checks for are of different types, too. It can either be a usual collision, when kid is near the impassable terrain such as wall or closed gate, or a screen collision, when kid collides with the borders of the screen. In the latter case, kid is moved to a new room, if there exists one at a needed position.

`WorldRenderer` class is also doing the same collision and gravity checks between a kid and the last 3 vertical blocks of left room connection and top 10 horizontal blocks of bottom room

connection. If the connections happen to be null, then, if there is a collision with the left room, the left room is treated as the room constructed of walls only. If there is a collision with the bottom room that is null, which shouldn't happen, the kid falls into the hole and dies.

The structure of the WorldRenderer object can be seen in Figure 2.11, which is located below.

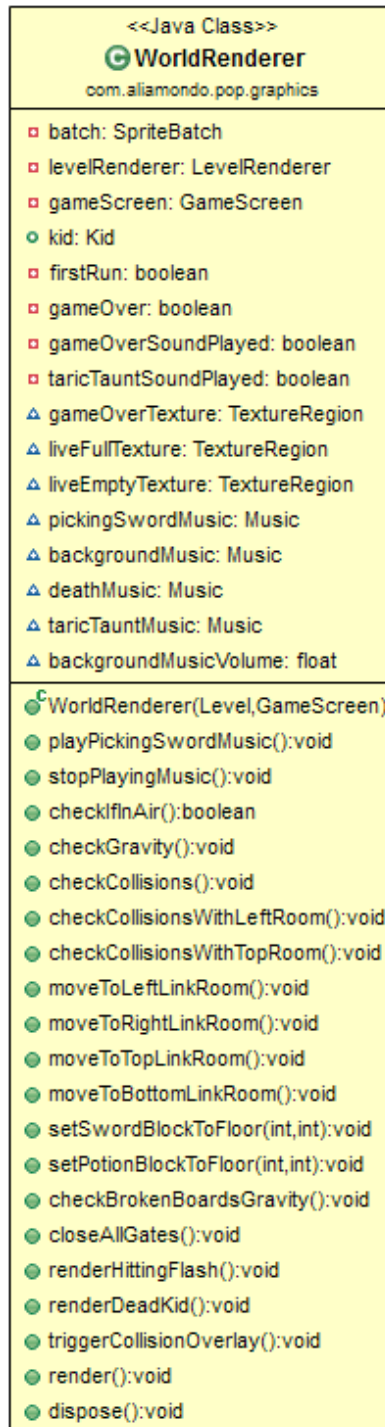


Figure 2.11 – WorldRenderer diagram

2.3.11 LevelRenderer

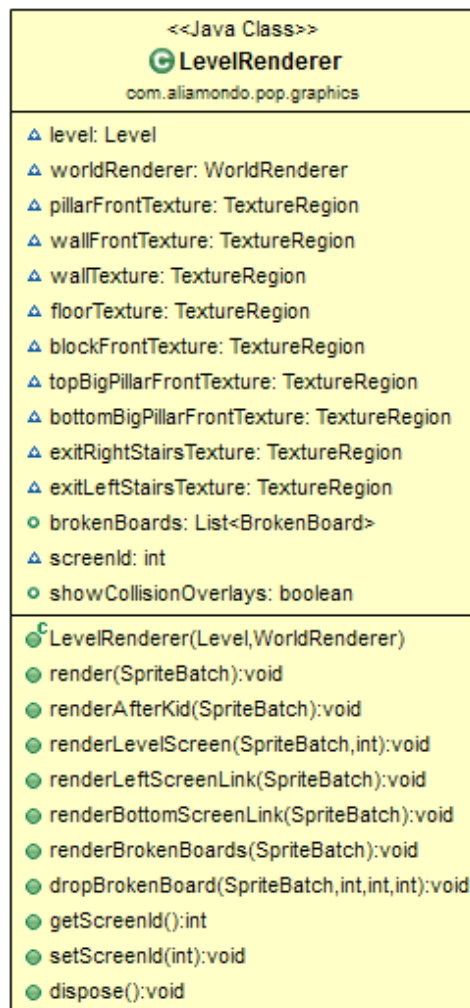


Figure 2.12 – LevelRenderer diagram

LevelRenderer is essentially a helper for a WorldRenderer that renders the current level screen (room). It contains the Level instance to access the screen that is to be rendered and the screenId integer as well, which is equal to the index of the current screen in screens list of level.

While LevelRenderer renders the current level screen, it also renders the last 3 vertical blocks of current screen's left connection and first 10 horizontal blocks of the bottom connection in a similar way to WorldRenderer.

Since we want kid to appear partially behind specific blocks, such as walls or pillars in front of the kid, LevelRenderer contains its own front textures to render them after the Kid.render() method is called by WorldRenderer. The method is renderAfterKid() and it renders those front textures when kid collides with the specific blocks to make it look like kid is passing behind the block, which looks more natural. This can be seen on Figures 2.13 and 2.14.



Figure 2.13 – Kid behind a pillar



Figure 2.14 – Kid behind the wall

2.4 Game flow chart

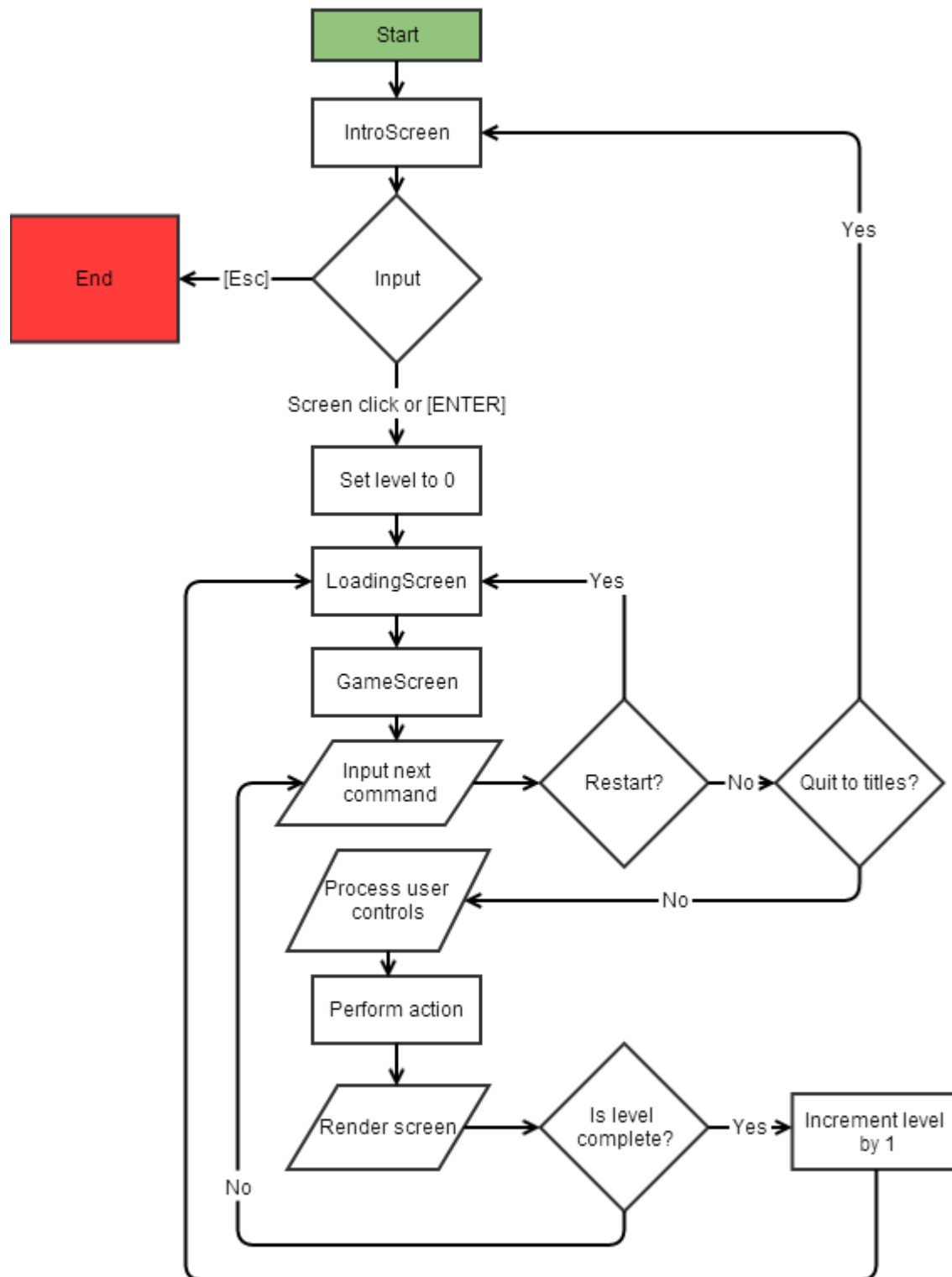


Figure 2.15 – Game flow chart

2.5 Data file format

The data file is of a binary file format; however, there are predefined offsets where specific data is located, as well as length of those file sectors. The offsets and sectors are specified in the Table 2.1, which is defined in the specifications of the Prince of Persia file formats [3]:

Length	Offset (after initial offset)	Block(sector) Name
720	0	pop1 foretable (blocks)
720	720	pop1 backtable (modifiers)
256	1440	doors I
256	1696	doors II
96	1952	links (connections)
64	2048	unknown I
3	2112	start position
3	2115	unknown II
1	2116	unknown III
24	2119	guard location
24	2143	guard direction
24	2167	unknown IV (a)
24	2191	unknown IV (b)
24	2215	guard skill
24	2239	unknown IV (c)
24	2263	guard color
16	2287	unknown IV (d)
2	2303	0F 09 (2319)

Table 2.1 – Level data file structure

However, in the beginning of the file there is an initial offset, which is 19 and at that position of the file there is a file start symbol. The fact that there are 19 bytes before the start of the actual level is explained by the fact that PLV file format used in the project is unpacked from the originally packed level sets and it contains POP_LVL string at the beginning of the file as well as other data to specify that this is a level data file.

2.5.1 Screens

Each level consists of 24 screens and each screen consists of 30 blocks that are sorted into 3 rows of 10 columns each. The screens are numbered from 1 to 24, due to 0 being reserved for special cases. The number of a screen will be used later in room linking (Section 2.5.3).

2.5.2 Blocks and modifiers

The blocks are numbered from 0 to 29. There are $24 \times 30 = 720$ blocks in all 24 screens of the level, thus every 30 blocks make a new screen. The types of blocks can be seen in Figure 2.4.

Modifiers are also numbered from 0 to 29. The types of modifiers can be seen in Figure 2.4. It is a good idea to assign certain modifiers to certain block groups, such as spikes modifiers for spikes group; however, it is not a must.

When the blocks bytes are read, the hex values are matched against the Table 2.2, and when the modifiers are read, the hex values are matched against the Table 2.3.

It is possible that some blocks have values of, for example, 0x34 instead of 0x14. That's why it is not a bad idea to calculate the hex value of the block modulo 0x20, especially because the value of 0x20 isn't used by the file format anyways. This, however, can potentially lead to losing some data from the hex value, but it seems there is no problem with that as the modifiers that affect the blocks are stored in another part of the file.

Note: the modifiers for blocks from the **event** group are the indexes of the events that need to be triggered. More on events is in Section 2.5.5.

Hex	Binary	Group	Description
0x00	00000	free	Empty
0x01	00001	free	Floor
0x02	00010	spike	Spikes
0x03	00011	none	Pillar
0x04	00100	gate	Gate
0x05	00101	none	Stuck Button
0x06	00110	event	Drop Button
0x07	00111	tapest	Tapestry
0x08	01000	none	Bottom Big-pillar
0x09	01001	none	Top Big-pillar
0x0A	01010	potion	Potion
0x0B	01011	none	Loose Board
0x0C	01100	ttop	Tapestry Top
0x0D	01101	none	Mirror
0x0E	01110	none	Debris
0x0F	01111	event	Raise Button
0x10	10000	none	Exit Left
0x11	10001	none	Exit Right
0x12	10010	chomp	Chopper
0x13	10011	none	Torch
0x14	10100	wall	Wall
0x15	10101	none	Skeleton
0x16	10110	none	Sword
0x17	10111	none	Balcony Left
0x18	11000	none	Balcony Right
0x19	11001	none	Lattice Pillar
0x1A	11010	none	Lattice Support
0x1B	11011	none	Small Lattice
0x1C	11100	none	Lattice Left
0x1D	11101	none	Lattice Right
0x1E	11110	none	Torch with Debris
0x1F	11111	none	Null

Table 2.2 – Block binary values

Group	Code	Description
none	0x00	This value is always used for this group
free	0x00	Nothing, Blue line
free	0x01	Spot1, No blue line
free	0x02	Spot2, Diamond
free	0x03	Window
free	0xFF	Spot3, Blue line?
spike	0x00	Normal (allows animation)
spike	0x01	Barely Out
spike	0x02	Half Out
spike	0x03	Fully Out
spike	0x04	Fully Out
spike	0x05	Out?
spike	0x06	Out?
spike	0x07	Half Out?
spike	0x08	Barely Out?
spike	0x09	Disabled
gate	0x00	Closed
gate	0x01	Open
tapest	0x00	With Lattice
tapest	0x01	Alternative Design
tapest	0x02	Normal
tapest	0x03	Black
potion	0x00	Empty
potion	0x01	Health point
potion	0x02	Life
potion	0x03	Feather Fall
potion	0x04	Invert
potion	0x05	Poison
potion	0x06	Open
ttop	0x00	With lattice
ttop	0x01	Alternative design
ttop	0x02	Normal
ttop	0x03	Black
ttop	0x04	Black
ttop	0x05	With alternative design and bottom
ttop	0x06	With bottom
ttop	0x07	With window
chomp	0x00	Normal
chomp	0x01	Half Open
chomp	0x02	Closed
chomp	0x03	Partially Open
chomp	0x04	Extra Open
chomp	0x05	Stuck Open
wall	0x00	Normal, Blue line
wall	0x01	Normal, No Blue line

Table 2.3 – Modifier binary values

2.5.3 Room linking

There are $24 \times 4 = 96$ bytes of room linking file part and every 4 bytes specify links for every room in an order of left, right, top and bottom. If the link to a certain room is 0, that means there is no room in that direction.

It is possible for a room to link to itself in one or more directions and it is also possible for a room to link to another room, which doesn't link back. For example, the room on the right of room 4 could be room 5, but the room on the left of room 5 could be room 3. In general, however, rooms have mutual links, unless the level was designed with a different thought.

2.5.4 Starting location

The starting position of the kid is of 3 bytes and it has a very simple data structure. First byte is responsible for the room number of spawning (ranged from 1 to 24), second byte specifies the block index (ranged from 0 to 30) and the last byte specifies the direction which the kid will be facing. If the value of direction is 0x00, kid will be facing right, but if the value is 0xFF, kid will be facing left.

2.5.5 Events

The event reading technique is by far the most complicated in this data file type. There are 256 events which are numbered from 0 to 255. There are few things to keep in mind; the events may trigger the events after themselves and so on in a loop, if the initial event and next events are all able to trigger the next event. The last 256th event cannot trigger the next event and thus it has to be terminal.

The events don't store the information such as if the doors need to be opened or closed, this can be found out only when the game is played out and one of the buttons is pressed and the event stored in that button is raised. If the button was a RAISE_BUTTON, the door will be opened and if the button was a DROP_BUTTON, the door will be closed.

The events usually link to the doors, but if they link to the walls, floors or other block types, they do not need to be completely processed and only the ability to trigger the next event matters.

There are Doors I and Doors II file sectors which need to be processed together. Let Doors I byte for a specific event be byte I and Doors II byte for the same event be byte II. These bytes then will have the following structure of bits:

$$\text{byte I} = t_1 s_4 s_5 l_1 l_2 l_3 l_4 l_5$$

$$\text{byte II} = s_1 s_2 s_3 00000,$$

where t_1 can be 0 or 1 and it stands for the triggering of the next event, $s_1 s_2 s_3 s_4 s_5$ are 5 bits of the screen index where the event takes place and $l_1 l_2 l_3 l_4 l_5$ are 5 bits of the block location in the specified screen. It is possible to use bit shifting technique to extract those bits from these two bytes and then use bitwise AND operator to get the unsigned value from the 5 bits of screen or

location value. In this particular case it's needed to do bitwise AND with 0x1F, as it is simply 2^5 in hexadecimal form.

2.5.6 Other data

There is other data stored in the files; however, it is not implemented in this project, such as guard positions and guard difficulty, therefore it will not be covered in this documentation, but it is possible to find this data online and the source will be listed in the references at the end of this documentation.

2.6 Other game features

This game has several sounds, such as background music or death sound, as well as couple Easter egg sounds. One of them happens when the sword is found and picked and the other is triggered when the kid gets killed after falling down for a fatal amount of time or when the broken block falls on his head and kills him.

2.7 Testing

Testing is done manually, without the use of any unit testing frameworks, such as JUnit or JTest. If some value needs to be checked, a usual `System.out.println()` function is used to print it out and then it is manually decided if the code is working properly.

This way the code can be tested for anything that requires testing. Usually, however, the program is tested for graphical changes, such as if the LooseBoard is going to shake, when the kid is near it, as it can be seen in Figure 2.16.



Figure 2.16 – LooseBoard shake check

However, to make the testing easier, the `LevelRenderer` class has a method `renderAfterKid()`, which can also render the green overlay over the blocks that are colliding with the player. To turn this feature on or off, use the `CONTROL` + `C` + `O` key combination (**C**ollision **O**verlay). An example of this feature can be seen in Figure 2.17.

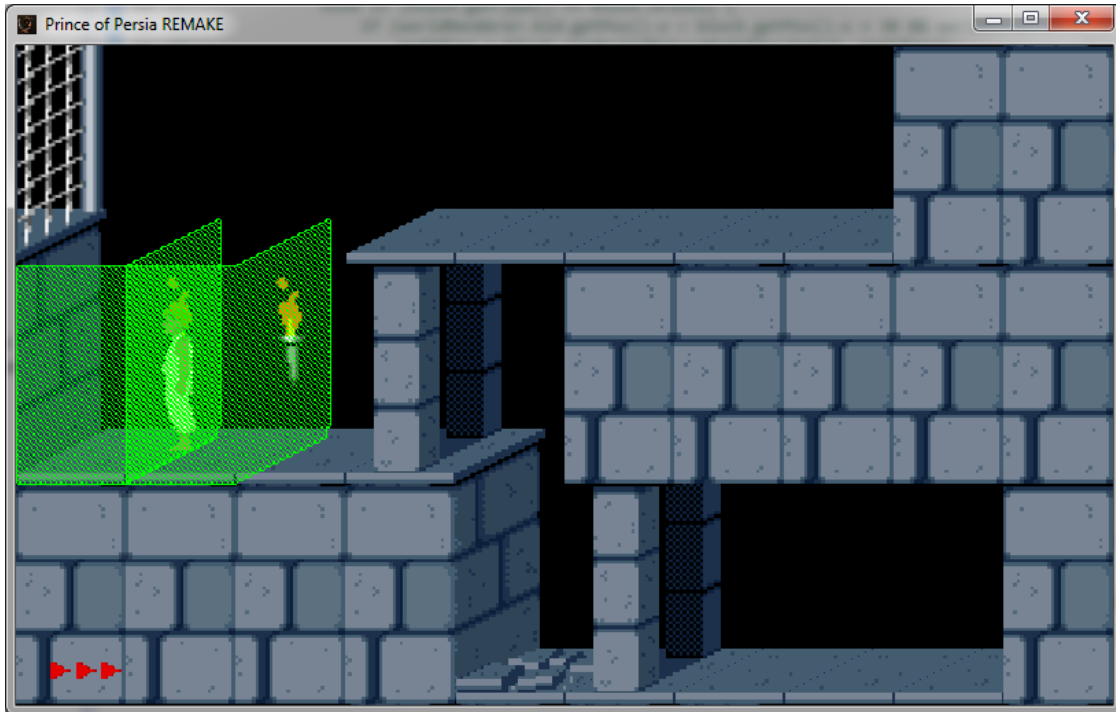


Figure 2.17 – Collision testing

As it is described in Section 1.3.6, skipping a level while playing can be achieved by pressing `CONTROL` + `S` + `L` (**S**kip **L**evel). This will skip to the next level and if there are no more levels, it will skip to the ending screen. Depending on the level, the starting level screen and starting position will vary, but this allows testing the next level right away.

It is also possible to instantly finish the game and see the ending screen by using the `CONTROL` + `E` + `G` combination of keys (**E**nd **G**ame).

To play a specific level or to start from a specific screen, it is needed to modify the value of `screenId` in `LevelRenderer` instance manually for a specific screen or modify the value of `levelNum` in `GameScreen`. In this case, however, if the level is complete, the next level to load will be the same as the finished level, so this should only be used for debugging purpose.

2.8 Possible improvements

2.8.1 Initial gravity

The first run gravity issue described in Section 1.5.2 should be resolved by either optimizing the `LevelReader` or by making `WorldRenderer` check collisions and gravity only after the world was rendered on the `GameScreen` instance. However, this was tried in numerous ways and the only way to solve this problem was to turn off the gravity check when the player is in air while standing. This adds another problem with, for example, loose blocks, which break when the player is standing on them, as they will break, but the player will not fall. Also, there is one of the original levels which starts with falling from a large height, so turning off the gravity check when `State=STANDING` is not a solution.

2.8.2 Guards and fighting

Another improvement would be to add the guards and a battle system into the implementation of the game, however, the battle system is complex and there are no animations of the kid fighting with the sword in his hands. The way to properly animate the kid fighting would be to specify the sword position on the screen manually depending on kid's frame of current attack or defense animation and then move the sword texture depending on the next move. This would also need to be done with the guards, as they don't have swords in their fighting animations as well.

Battle system in itself is also very realistic and complex and both kid and a guard can block the hit, attack the enemy, move during battle with the sword out or switch the sides in the battle. And to add to all of the above, either fighter can fall down after being attacked or if he stepped off the block and either fighter can die from a neighboring chopper or spikes.

2.8.3 Palace levels

Currently only dungeon level type is implemented, however, in original game there was also a palace level type. This is not in the current implementation of the game due to the fact that it was not possible to locate a place in a data file which is responsible for the level type. This level type is also not mentioned in "Specifications of File Formats" by Princed Development Team and it currently seems the only way to figure out if the level is set in the dungeons or palace is by manually adding a byte in the level file which will be responsible for that.

3 List of references

- [1] Android Development Team. (2014). *Android Development Tools* (Version 2014-03-21) [Computer program]. Available at: <http://developer.android.com/sdk/index.html> [Accessed 25 March. 2014].
- [2] Zechner, M. (2014). *libgdx*. (Version 0.9.8) [Computer program]. Available at: <http://libgdx.badlogicgames.com/> [Accessed 5 September. 2013].
- [3] Princed Development Team, (2008). *Prince of Persia. Specifications of File Formats*. 1st ed. [ebook] pp.10 - 17. Available at: http://www.popot.org/documentation/documents/FormatSpecifications_05Jan2008.pdf [Accessed 3 May. 2014].
- [4] Mechner, J. (1989). *Prince of Persia. Technical Information*. 1st ed. [ebook] Available at: <http://jordanmechner.com/wp-content/uploads/1989/10/popsourc009.pdf> [Accessed 3 May. 2014].
- [5] Princed Development Team. (2011). *FreePrince* (Version 2011-12-04) [Source code]. Available at: <http://www.princed.org/downloads/#FreePrince> [Accessed 11 May. 2014].
- [6] Biro, R. (2012). *OpenPrince* (Version 1.0) [Source code]. Available at: <https://github.com/DarthJDG/OpenPrince> [Accessed 3 May. 2014].