NTNU
Kunnskap for en bedre verden

TDT4240 - Software Architecture

”The Nearly Impossible Game” Implementation

Group 12:
Jakob E. Eikeland,
Teo Chen Ning,
Tan Chuan Xin,
Gaute Brandser,
Ruben Rokkones,
Sigve Sjøvold

COTS: Android, LibGDX, Firebase

Primary QA: Modifiability
Secondary QA: Usability Secondary QA: Availability
February, 2022

Contents

1 Introduction

The goal of this project is to create a functioning multiplayer game or app for mobile devices. This document provides a detailed description of the implementation and testing process. This is the phase where the actual game was created. In addition it is tested in relation to the previously defined quality and functional requirements.

1.1 Description of the game concept

Our group had several ideas about what type of game we wanted to create. after some discussion we decided to try to create a version of "The Impossible Game", a mobile game some of us had played when we were younger. It is a game where you play as a square going through a course made up of various obstacles appearing in its path with increased frequency and difficulty. The goal of the game is simply to reach the end of the level without dying. The basic flow of the game is that the game starts, the square (the player) starts gliding forward on the ground, and spikes start appearing. If hit from any direction by the spike, you will die, and restart the game. The player controls the square in each direction, and can jump.

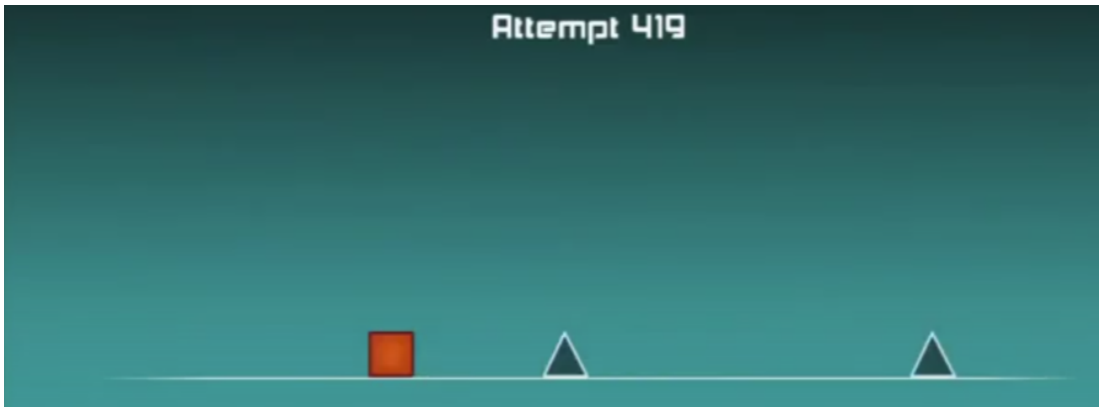


Figure 1: Original game

Our version of this game will have both single player and multiplayer modes. In single player mode, it is similar to the original Impossible Game. In multiplayer mode, two players will alternate in turns playing the game. We will also implement an online high score list. We considered various different ways of having multiplayer functionality, such as two mirrored courses on each side of the screen, or maybe two players next to each other on the same course, but decided to have a two players taking turns playing the same level. When the first player dies or wins, the game switches to the other player, who tries playing until he dies or wins. When both players have beaten the level, or died, they can type in their names and submit their scores to the database. They also have the option to skip this step, if they feel their scores are too low.

This version of this game ends up looking very different. We wanted to make our own game, and not copy the amazing original game. However we kept the core components of blocks and spikes. In our game you play as a jelly, and your goal is to get to the end of the map.

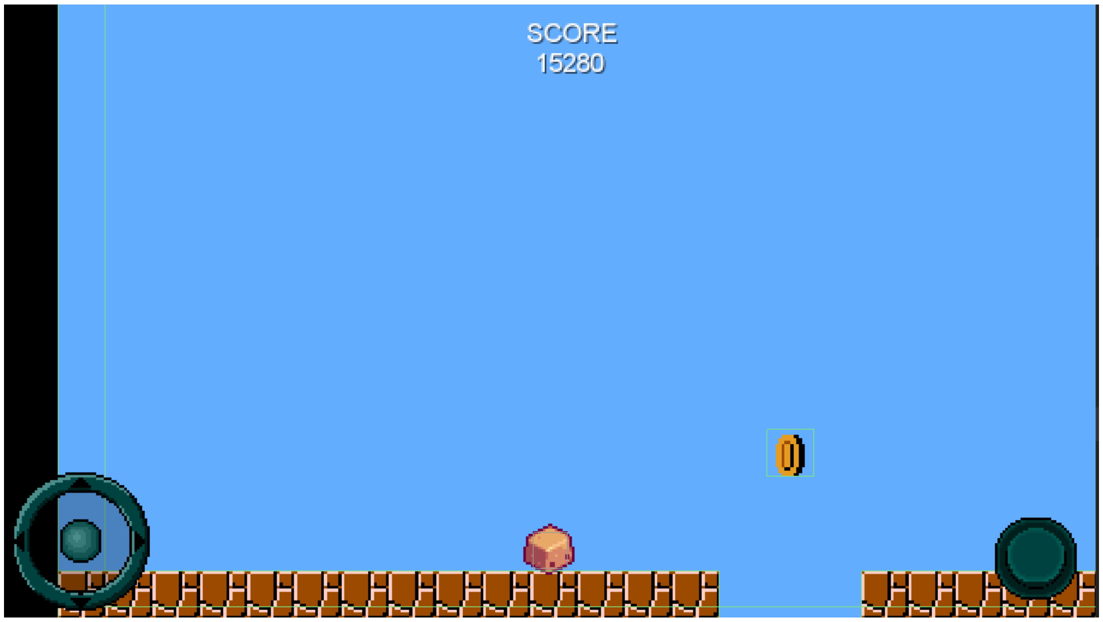


Figure 2: Beginning of game

The player has a jump button, and a joystick that controls lateral movement. On the way to the goal, the objective is to get the highest score as possible. This is achieved by picking up as many coins as you can, and going as fast as you can. The player only has one life, because this is "The Nearly Impossible Game". Another element we added to the game is fireballs. When you're gliding through the map with ease, a fireball can suddenly appear and mess up your game.

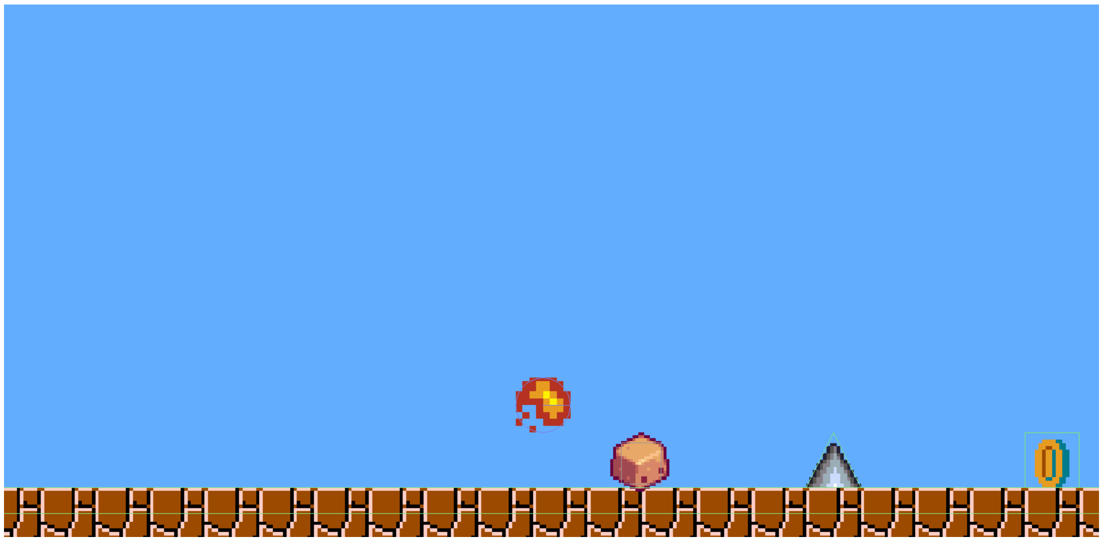


Figure 3: Fireballs and coins

Another way of messing the player's game up comes from the other player in when playing multiplayer. The player not currently playing can press a button in the top right hand corner, shoving the active player in a random direction in order to try to stop him from getting a good score.

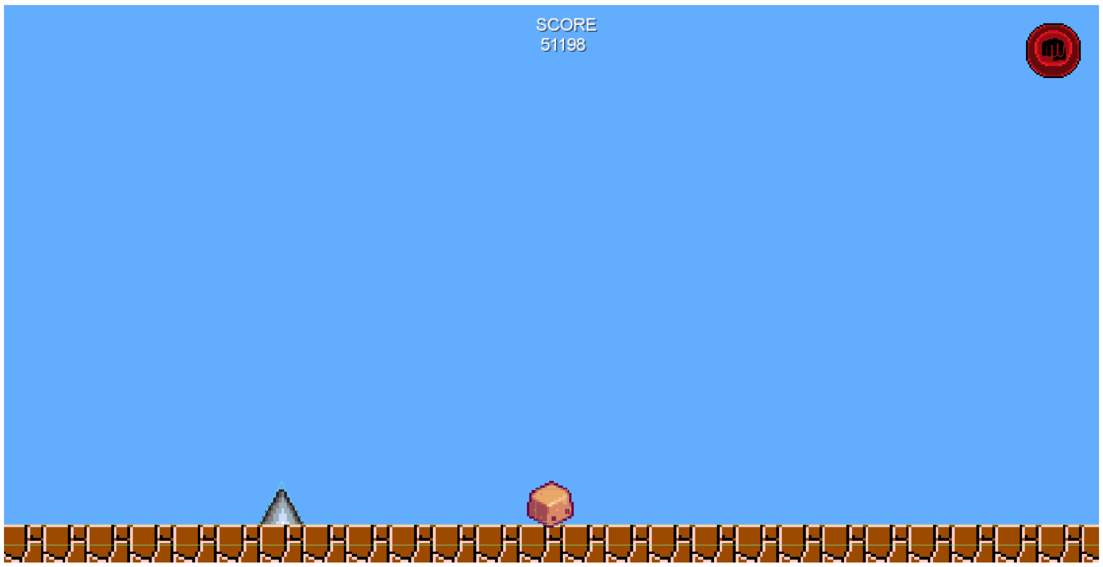


Figure 4: Button to mess with opponent

1.2 Structure of the document

This document gives an overview of the architecture we have outlined for our game. Section 2 describes how the project was designed and implemented in great detail, while section 3 is a guide on how to download, build, run and play the project. Section 4 is a detailed report of the tests we have performed surrounding quality and functional requirements, and 5 is about the relationship between the implementation and the architecture. Section 6 is about patterns relevant to the project, 7 shows the actual architectural design for the project, with illustrations. Section 8 is about the consistency of these views, and 9 discusses architectural rationale. Finally, we show the individual contribution of each member, changes in the project from beginning to end, and references

2 Design and Implementation

There is a significant difference between our initial planned architecture and the final product, such as the various classes and interfaces we would implement. As we went along, we created abstractions as we saw fit in order to encourage decoupling and allow us greater modifiability in the future.

While the design decisions made produced a lot more classes, interfaces and packages than we previously thought, we feel that they are necessary for the game architecture to remain flexible and easily modifiable. They also allow us to depend on abstractions, not implementations, allowing us to easily add features without major modifications and refactoring.

A good example of this are the obstacles in our game. At this time, we have a Spike class that extends the AbstractModel class, which implements three different interfaces: GameObject, Model and ContactObject (figure 5). These interfaces provide a contract for us to decouple implementation from generics that the controllers can function off on. For example, our physics Engine class works off the ContactObject class to detect collisions, rather than on the Spike implementation itself. The ObstacleFactory class also works off AbstractModel to spawn various kinds of obstacles, relying on the interface contract, while the specifics are left to each implementation.

To that effect, adding another obstacle such as a moving Fireball would just require us to implement the required interfaces and it would be easily pluggable into our game as there is an agreed upon interface to make things work.

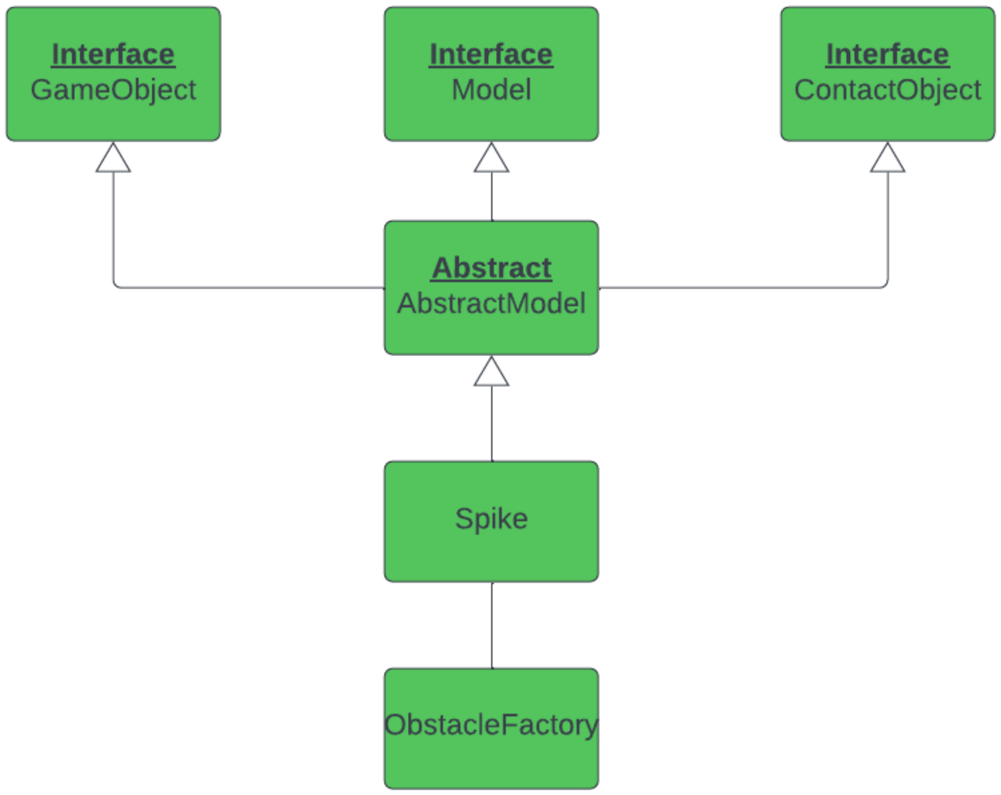


Figure 5: Spike class diagram

2.1 Design patterns

In our Architectural Design Document, we postulated that we would adopt a Model-View-Controller architectural design (MVC). We have largely stuck to that implementation, trying to separate out the concerns:

- Models: Mainly concerned with holding data and state
- Controllers: Holds business logic and dependencies
- Views: Displays and rendering based on state

In some cases, a pure MVC approach may not be feasible. For example, a physics Body may hold state about itself such as its orientation, shape, position and velocity. However, it also implements specific business logic on what happens when a collision occurs. In some cases, co-locating code based on function rather than separating them based on design makes more sense.

We have also made extensive use of various other design patterns to augment the MVC architecture, in order to encourage further decoupling and encourage modifiability.

2.1.1 Dependency injection

In our Architectural Design Document, we mentioned the use of Singletons for various controller classes. For example, there can be one AssetLoader that is in charge of loading in various assets, implemented as a Singleton such that it is accessible from everywhere.

In practice, we decided to opt for dependency injection over Singletons. In essence, every object declares upfront (in its constructor or otherwise) the dependencies and types it requires to function, which are then injected into it at runtime.

This results in more testable and easy to reason about since all dependencies are easily seen from within the object, and mocks or stubs of the dependencies can be created for testing without having to recreate the fully functional Singleton so that the object or class can work.

2.1.2 Factory

We have made extensive use of the Factory pattern in our code due to our use of abstract interfaces. For example, we can have an AbstractModel that contains base attributes like position, rotation and so on, and have a Factory that is able to instantiate objects based on this interface.

Specific details such as business logic for each individual object and obstacle can then be implemented within the implementations of the AbstractModel interface, while the generic Factory can simply call upon more specific Factories to spawn the required objects while relying on the abstraction.

2.1.3 Strategy

Similarly, we also made use of the Strategy pattern to select specific code to execute at runtime. For example, our various Factories execute different instantiation code based on the type of object it is asked to spawn. The ModelFactory delegates the spawning of objects to other more specific factories based on the entity required to spawn, allowing for a more specific set of algorithms and code to be run for the spawning.

2.1.4 Template

We also used Templates, such as a generic AbstractModel that serves as a base for implementations (e.g. Spike, Fireball). Other classes use references to the generic interface for functionality, while the specific implementation is injected at runtime.

2.1.5 Publisher-subscriber

We initially wanted to use the Observer pattern so that interconnected classes can call upon others easily. For example, a Player dying would signify that the game has ended and thus the screen should change to GameOver and the GameMap should stop rendering. However, this quickly became unwieldy to maintain since there are lot of interconnected dependencies in a game.

Instead, we choose to adopt a pubsub pattern via the use of an EventManager. Publishers publish events to the EventManager, which then sends out messages to Subscribers that express interest in that event (push messaging). This allows us to decouple classes by simply publishing an event, whereby all other dependencies will be notified. Since the publisher does not know of the subscribers, the implementation is decoupled and each listener can handle the event data specifically themselves.

2.1.6 Client server

We made use of Firebase to implement a database, whereby players can publish their scores to after the game has ended. Clients cannot modify the database beyond publishing their scores which will then be stored on the server (Firebase), ensuring data integrity and high availability of the leaderboards.

The websocket connection between client and server also allows for near instantaneous updates to the leaderboard, allowing multiple clients globally to share a unified view of the latest leaderboards.

2.2 Description of the implemented classes

This section gives a description of the most important classes in the project, how they relate to each other and the packages they reside in.

2.2.1 Model

Interfaces The main interfaces for the models are Model, ContactObject, and GameObject. These interfaces have specific purposes and we've tried to design them such that any further extensions is unecessary. The Model interface is used to achieve total abstraction between the model, view and controller. The controller and the views only know about the Model interface and not the concrete classes, resulting in loose coupling between the modules. The ContactObject interface for collisions in the physics engine, and communication between the different models which are colliding with each other. The GameObject interface is specifically used by the BodyBuilder class to obtain information about a game object so that it can create a Box2D body for all the model classes.

AbstractModel The AbstractModel class implements the interfaces from above and is one of the most important classes in the Model package. All of the gameobjects like blocks, coins and obstacles inherits from this class and use the interfaces to communicate with eachother and the other modules in the project. The AbstractModel class contains everything of which the models share between eachother on an abstract level. This makes sure that creating new subclasses will be easy and done with few lines of code.

GameWorld This class encapsulates the Box2D World which implements the Engine interface in the rest of the project. The point of this class and the interface is to increase modifiability by avoiding direct references to the Box2D world such that the physics engine can be switched out later without too much hassle if necessary.

2.2.2 Controller

Game Initializer The GameInitializer interface is implemented by the NormalGame class and is used to initialize all the object controllers in the game based on the map it receives as a parameter, and then stores them for the GameManager to use. There might not be a need for different types of game initializers in the future, but there will be a possibility for it through the interface.

Managers We use three types of managers in our project: GameManager, EventManager, and ScreenManager. The GameManager creates new games using the initializer, updates the controllers, tracks the state of the game, and changes the state based on events from the EventManager. This class is as mentioned in section 2.1.5 vital for the implementation of the Publish-Subscribe pattern, which we've used extensively in our project to create an event-driven architecture. For example the GameManager subscribes to the event that a player dies, and therefore knows when to change the state of the game. Lastly, the ScreenManager contains all the logic for changing the screens.

Object Controllers The controllers are divided into static controllers, and animated controllers. The only difference is that the animated controllers contain a view including to the model. The purpose of these classes is to connect the view with the model in some way and to update them both through the controller. These two classes use high abstraction such that they can be used for all the objects in the game.

2.2.3 View

Screens We've created different classes for all the screens like main menu and the game screen, which are shown using the ScreenManager. All of the screens extends the AbstractScreen class which contains all the common fields and features of the screens.

Model Views The Model views are used for rendering and animating the dynamic game objects like the player and fireballs. The AbstractView class implements the View interface and stores a Model interface object to get the necessary information to render the model. The model however has no reference to the view or the controller.

2.3 Class diagrams

Following are snapshots of the class diagram of the complete project. For readability, we have cut the diagram into the Model, View and Controller sections. the full diagram can be viewed at the end of the document in Figure 15

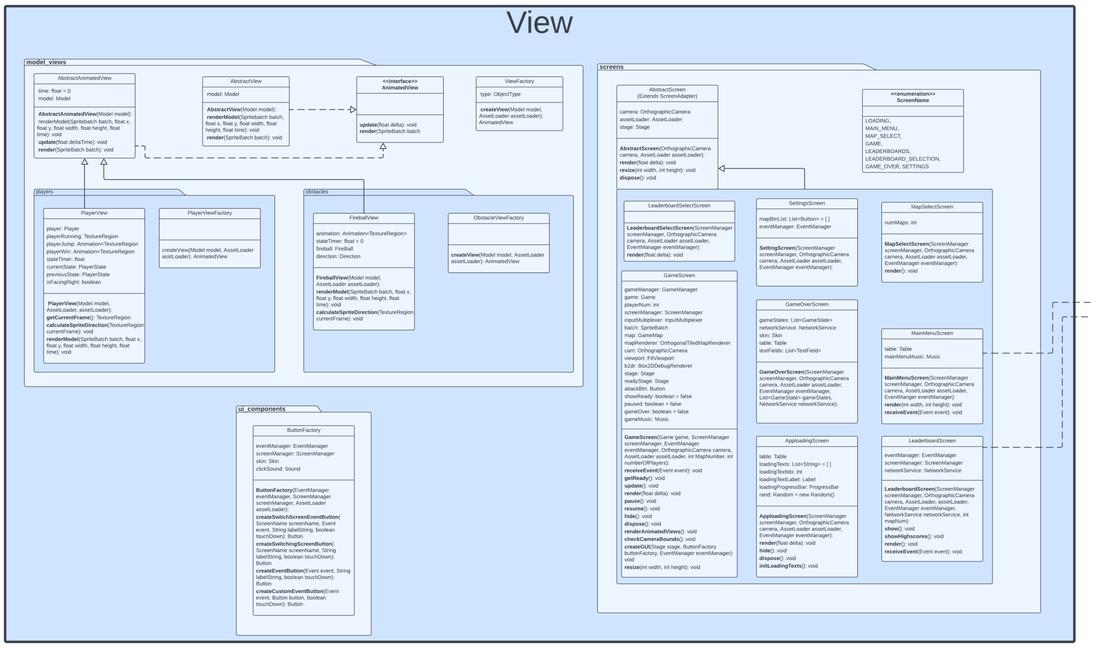

View
model_views
AbstractAnimatedView
time: float = 0
model: Model
AbstractAnimatedView(Model model):
renderModel(Spritebatch batch, float x, float y, float width, float height, float time): void
update(float deltaTime): void
render(SpriteBatch batch): void
AbstractView
model: Model
AbstractView(Model model):
renderModel(Spritebatch batch, float x, float y, float width, float height, float time): void
render(SpriteBatch batch): void
<<interface>> AnimatedView
update(float delta): void
render(SpriteBatch batch
ViewFactory
type: ObjectType
createView(Model model, AssetLoader assetLoader): AnimatedView
players
PlayerView
player: Player
playerRunning: TextureRegion
playerJump: Animation<TextureRegion
playerWin: Animation<TextureRegion
stateTimer: float
currentState: PlayerState
previousState: PlayerState
isFacingRight: boolean
PlayerView(Model model, AssetLoader, assetLoader):
getCurrentFrame(): TextureRegion
calculateSpriteDirection(TextureRegion currentFrame): void
renderModel(SpriteBatch batch, float x, float y, float width, float height, float time): void
PlayerViewFactory
createView(Model model, AssetLoader asstLoader): AnimatedView
obstacles
FireballView
animation: Animation<TextureRegion>
stateTimer: float = 0
fireball: FireBall
direction: Direction
FireballView(Model model, AssetLoader assetLoader):
renderModel(SpriteBatch batch, float x, float y, float width, float height, float time): void
calculateSpriteDirection(TextureRegion currentFrame): void
ObstacleViewFactory
createView(Model model, AssetLoader assetLoader): AnimatedView
ui_components
ButtonFactory
eventManager: EventManager
screenManager: ScreenManager
skin: Skin
clickSound: Sound
ButtonFactory(EventManager eventManager, ScreenManager screenManager, AssetLoader assetLoader):
createSwitchScreenEventButton(ScreenName screenName, Event event, String labelString, boolean touchDown): Button
createSwitchingScreenButton(ScreenName screenName, String labelString, boolean touchDown): Button
createEventButton(Event event, String labelString, boolean touchDown): Button
createCustomEventButton(Event event, Button button, boolean touchDown): Button
screens
AbstractScreen (Extends ScreenAdapter)
camera: OrthographicCamera
assetLoader: AssetLoader
stage: Stage
AbstractScreen(OrthographicCamera camera, AssetLoader assetLoader):
render(float delta): void
resize(int width, int height): void
dispose(): void
<<enumeration>> ScreenName
LOADING, MAIN_MENU, MAP_SELECT, GAME, LEADERBOARDS, LEADERBOARD_SELECTION, GAME_OVER, SETTINGS
LeaderboardSelectScreen
LeaderboardSelectScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader assetLoader, EventManager eventManager):
render(float delta): void
SettingsScreen
mapBtnList: List<Button> = [ ]
eventManager: EventManager
SettingScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader assetLoader, EventManager eventManager):
MapSelectScreen
numMaps: int
MapSelectScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader assetLoader, EventManager eventManager):
render(): void
GameScreen
gameManager: GameManager
game: Game
playerNum: int
screenManager: ScreenManager
inputMultiplexer: InputMultiplexer
batch: SpriteBatch
map: GameMap
mapRenderer: OrthogonalTiledMapRenderer
cam: OrthographicCamera
viewport: FitViewport
b2dr: Box2DDebugRenderer
stage: Stage
readyStage: Stage
attackBtn: Button
showReady: boolean = false
paused: boolean = false
gameOver: boolean = false
gameMusic: Music
GameScreen(Game game, ScreenManager screenManager, EventManager eventManager, OrthographicCamera camera, AssetLoader assetLoader, int MapNumber, int numberOfPlayers):
receiveEvent(Event event): void
getReady(): void
update(): void
render(float delta): void
pause(): void
resume(): void
hide(): void
dispose(): void
renderAnimatedViews(): void
checkCameraBounds(): void
createGUI(Stage stage, ButtonFactory buttonFactory, EventManager eventManager): void
resize(int width, int height): void
GameOverScreen
gameStates: List<GameState>
networkService: NetworkService
skin: Skin
table: Table
textFields: List<TextField>
GameOverScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader assetLoader, EventManager eventManager, List<GameState> gameStates, NetworkService networkService):
MainMenuScreen
table: Table
mainMenuMusic: Music
MainMenuScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader assetLoader, EventManger eventManager):
render(int width, int height): void
receiveEvent(Event event): void
ApploadingScreen
table: Table
loadingTexts: List<String> = [ ]
loadingTextIdx: int
loadingTextLabel: Label
loadingProgressBar: ProgressBar
rand: Random = new Random()
ApploadingScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader assetLoader, EventManager eventManager):
render(float delta): void
hide(): void
dispose(): void
initLoadingTexts(): void
LeaderboardScreen
eventManager: EventManager
screenManager: ScreenManager
networkService: NetworkService
LeaderboardScreen(ScreenManager screenManager, OrthographicCamera camera, AssetLoader, assetLoader, EventManager eventManager, NetworkService networkService, int mapNum):
show(): void
showHighscores(): void
render(): void
receiveEvent(Event event): void

Figure 6: Class diagram for the View package

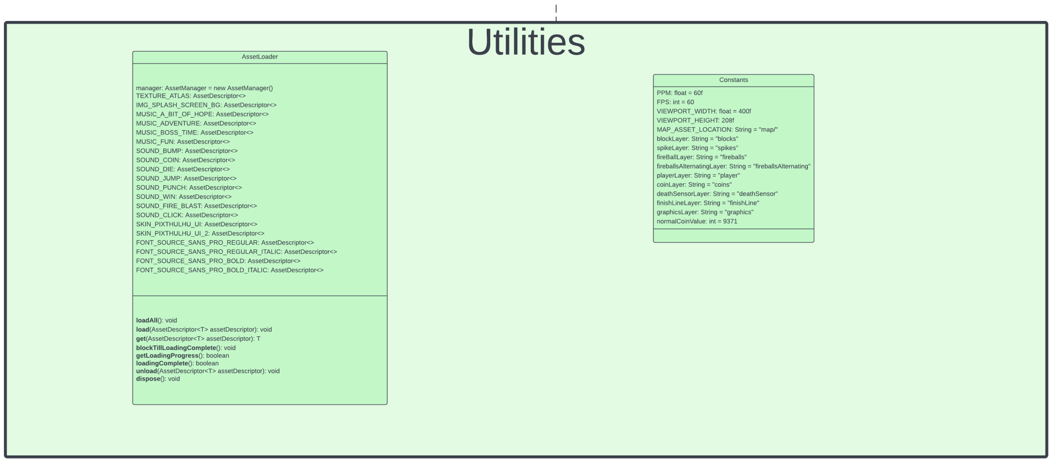

Utilities
AssetLoader
manager: AssetManager = new AssetManager()
TEXTURE_ATLAS: AssetDescriptor<>
IMG_SPLASH_SCREEN_BG: AssetDescriptor<>
MUSIC_A_BIT_OF_HOPE: AssetDescriptor<>
MUSIC_ADVENTURE: AssetDescriptor<>
MUSIC_BOSS_TIME: AssetDescriptor<>
MUSIC_FUN: AssetDescriptor<>
SOUND_BUMP: AssetDescriptor<>
SOUND_COIN: AssetDescriptor<>
SOUND_DIE: AssetDescriptor<>
SOUND_JUMP: AssetDescriptor<>
SOUND_PUNCH: AssetDescriptor<>
SOUND_WIN: AssetDescriptor<>
SOUND_FIRE_BLAST: AssetDescriptor<>
SOUND_CLICK: AssetDescriptor<>
SKIN_PIXTHULHU_UI: AssetDescriptor<>
SKIN_PIXTHULHU_UI_2: AssetDescriptor<>
FONT_SOURCE_SANS_PRO_REGULAR: AssetDescriptor<>
FONT_SOURCE_SANS_PRO_REGULAR_ITALIC: AssetDescriptor<>
FONT_SOURCE_SANS_PRO_BOLD: AssetDescriptor<>
FONT_SOURCE_SANS_PRO_BOLD_ITALIC: AssetDescriptor<>
loadAll(): void
load(AssetDescriptor<T> assetDescriptor): void
get(AssetDescriptor<T> assetDescriptor): T
blockTillLoadingComplete(): void
getLoadingProgress(): boolean
loadingComplete(): boolean
unload(AssetDescriptor<T> assetDescriptor): void
dispose(): void
Constants
PPM: float = 60f
FPS: int = 60
VIEWPORT_WIDTH: float = 400f
VIEWPORT_HEIGHT: 208f
MAP_ASSET_LOCATION: String = "map/"
blockLayer: String = "blocks"
spikeLayer: String = "spikes"
fireBallLayer: String = "fireballs"
fireballsAlternatingLayer: String = "fireballsAlternating"
playerLayer: String = "player"
coinLayer: String = "coins"
deathSensorLayer: String = "deathSensor"
finishLineLayer: String = "finishLine"
graphicsLayer: String = "graphics"
normalCoinValue: int = 9371

Figure 7: Class diagram for the Utilities package

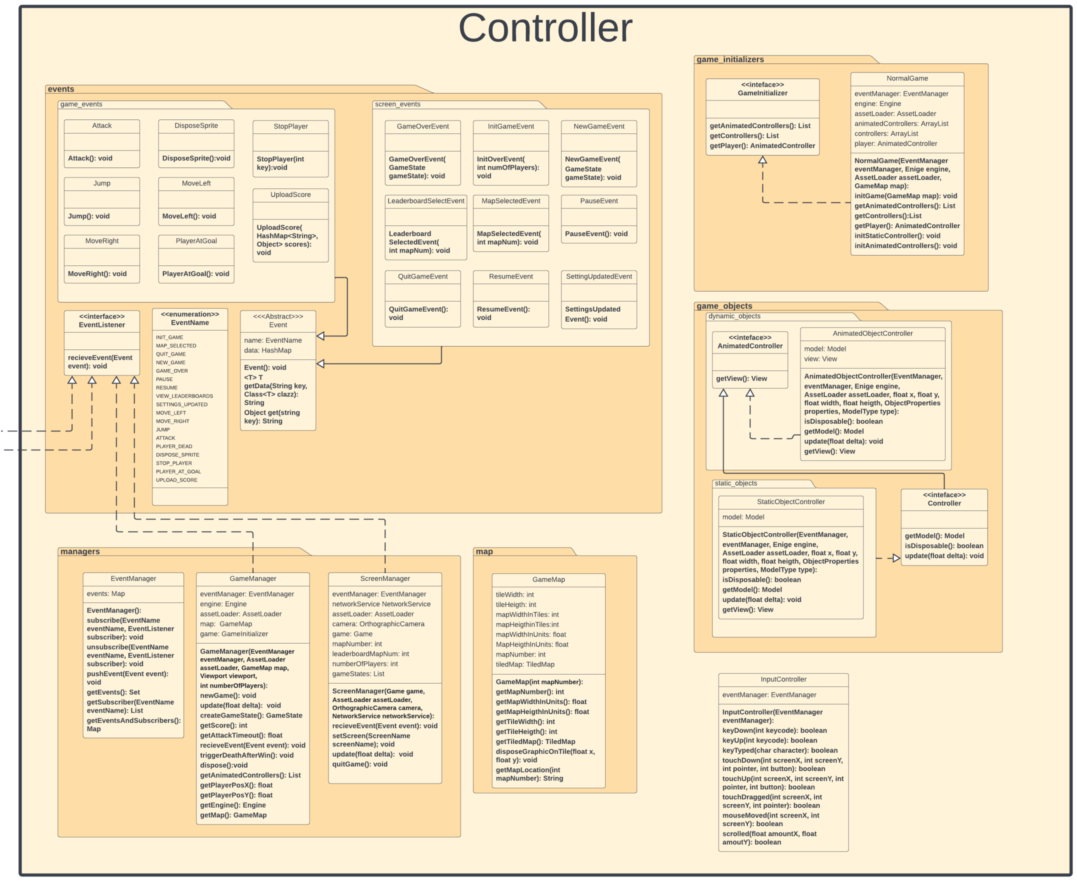

Controller
events
game_events
Attack
Attack(): void
DisposeSprite
DisposeSprite():void
StopPlayer
StopPlayer(int key):void
Jump
Jump(): void
MoveLeft
MoveLeft(): void
UploadScore
UploadScore(HashMap<String>, Object> scores): void
MoveRight
MoveRight(): void
PlayerAtGoal
PlayerAtGoal(): void
screen_events
GameOverEvent
GameOverEvent(GameState gameState): void
InitGameEvent
InitOverEvent(int numOfPlayers): void
NewGameEvent
NewGameEvent(GameState gameState): void
LeaderboardSelectEvent
Leaderboard SelectedEvent(int mapNum): void
MapSelectedEvent
MapSelectedEvent(int mapNum): void
PauseEvent
PauseEvent(): void
QuitGameEvent
QuitGameEvent(): void
ResumeEvent
ResumeEvent(): void
SettingUpdatedEvent
SettingsUpdated Event(): void
<<interface>> EventListener
recieveEvent(Event event): void
<<enumeration>> EventName
INIT_GAME
MAP_SELECTED
QUIT_GAME
NEW_GAME
GAME_OVER
PAUSE
RESUME
VIEW_LEADERBOARDS
SETTINGS_UPDATED
MOVE_LEFT
MOVE_RIGHT
JUMP
ATTACK
PLAYER_DEAD
DISPOSE_SPRITE
STOP_PLAYER
PLAYER_AT_GOAL
UPLOAD_SCORE
<<<Abstract>>> Event
name: EventName
data: HashMap
Event(): void
<T> T getData(String key, Class<T> clazz): String
Object get(string key): String
game_initializers
<<inteface>> GameInitializer
getAnimatedControllers(): List
getControllers(): List
getPlayer(): AnimatedController
NormalGame
eventManager: EventManager
engine: Engine
assetLoader: AssetLoader
animatedControllers: ArrayList
controllers: ArrayList
player: AnimatedController
NormalGame(EventManager eventManager, Enige engine, AssetLoader assetLoader, GameMap map):
initGame(GameMap map): void
getAnimatedControllers(): List
getControllers():List
getPlayer(): AnimatedController
initStaticController(): void
initAnimatedControllers(): void
game_objects
dynamic_objects
<<inteface>> AnimatedController
getView(): View
AnimatedObjectController
model: Model
view: View
AnimatedObjectController(EventManager, eventManager, Enige engine, AssetLoader assetLoader, float x, float y, float width, float heigth, ObjectProperties properties, ModelType type):
isDisposable(): boolean
getModel(): Model
update(float delta): void
getView(): View
static_objects
StaticObjectController
model: Model
StaticObjectController(EventManager, eventManager, Enige engine, AssetLoader assetLoader, float x, float y, float width, float heigth, ObjectProperties properties, ModelType type):
isDisposable(): boolean
getModel(): Model
update(float delta): void
getView(): View
<<inteface>> Controller
getModel(): Model
isDisposable(): boolean
update(float delta): void
managers
EventManager
events: Map
EventManager():
subscribe(EventName eventName, EventListener subscriber): void
unsubscribe(EventName eventName, EventListener subscriber): void
pushEvent(Event event): void
getEvents(): Set
getSubscriber(EventName eventName): List
getEventsAndSubscribers(): Map
GameManager
eventManager: EventManager
engine: Engine
assetLoader: AssetLoader
map: GameMap
game: GameInitializer
GameManager(EventManager eventManager, AssetLoader assetLoader, GameMap map, Viewport viewport, int numberOfPlayers):
newGame(): void
update(float delta): void
createGameState(): GameState
getScore(): int
getAttackTimeout(): float
recieveEvent(Event event): void
triggerDeathAfterWin(): void
dispose():void
getAnimatedControllers(): List
getPlayerPosX(): float
getPlayerPosY(): float
getEngine(): Engine
getMap(): GameMap
ScreenManager
eventManager: EventManager
networkService NetworkService
assetLoader: AssetLoader
camera: OrthographicCamera
game: Game
mapNumber: int
leaderboardMapNum: int
numberOfPlayers: int
gameStates: List
ScreenManager(Game game, AssetLoader assetLoader, OrthographicCamera camera, NetworkService networkService):
recieveEvent(Event event): void
setScreen(ScreenName screenName); void
update(float delta): void
quitGame(): void
map
GameMap
tileWidth: int
tileHeigth: int
mapWidthInTiles: int
mapHeigthinTiles:int
mapWidthInUnits: float
MapHeigthInUnits: float
mapNumber: int
tiledMap: TiledMap
GameMap(int mapNumber):
getMapNumber(): int
getMapWidthInUnits(): float
getMapHeigthInUnits(): float
getTileWidth(): int
getTileHeigth(): int
getTiledMap(): TiledMap
disposeGraphicOnTile(float x, float y): void
getMapLocation(int mapNumber): String
InputController
eventManager: EventManager
InputController(EventManager eventManager):
keyDown(int keycode): boolean
keyUp(int keycode): boolean
keyTyped(char character): boolean
touchDown(int screenX, int screenY, int pointer, int button): boolean
touchUp(int screenX, int screenY, int pointer, int button): boolean
touchDragged(int screenX, int screenY, int pointer): boolean
mouseMoved(int screenX, int screenY): boolean
scrolled(float amountX, float amoutY): boolean

Figure 8: Class diagram for the Controller package

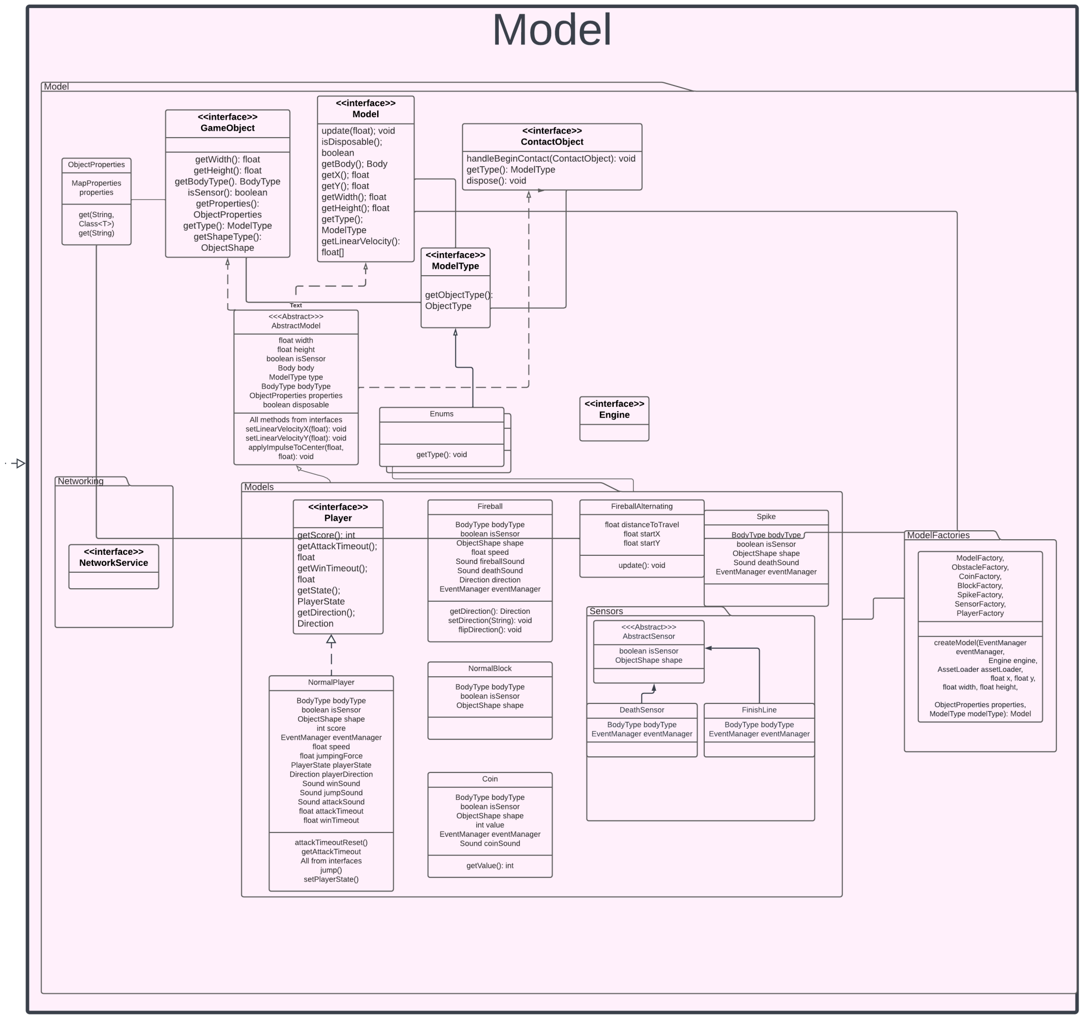


Figure 9: Class diagram for the Model package

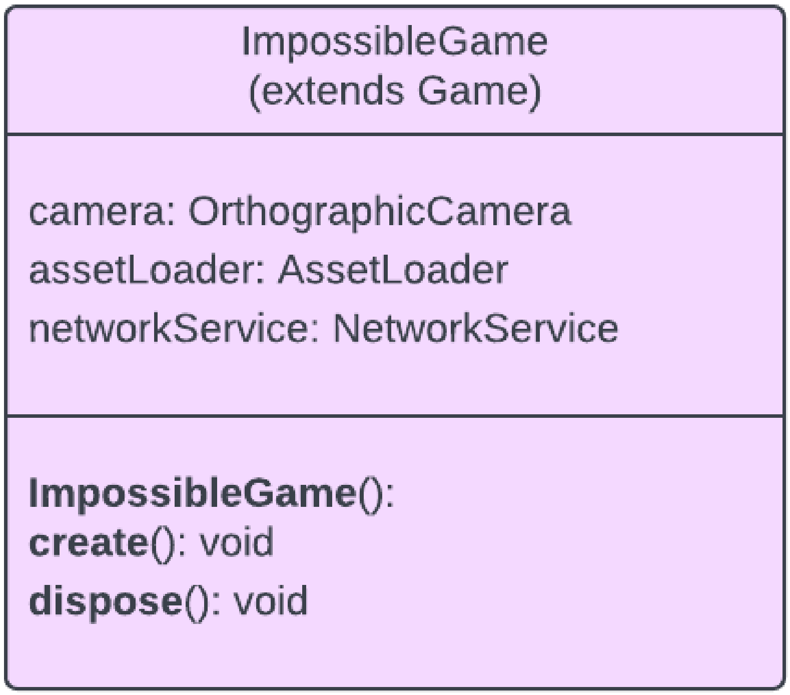


Figure 10: ImpossibleGame class diagram

3 User Manual

The User Manual gives a complete guide on how to install and play our game as it was intended.

3.1 Requirements

To run this game you will need an Android device with API level 30 or later, or Android Studios. In order to save scores and see the high score list, you need to be connected to wi-fi and use either Android emulator or an Android device. The app can be run on desktop, but the online multiplayer component(leaderboards, score submission) will not work.

3.2 Installing the game

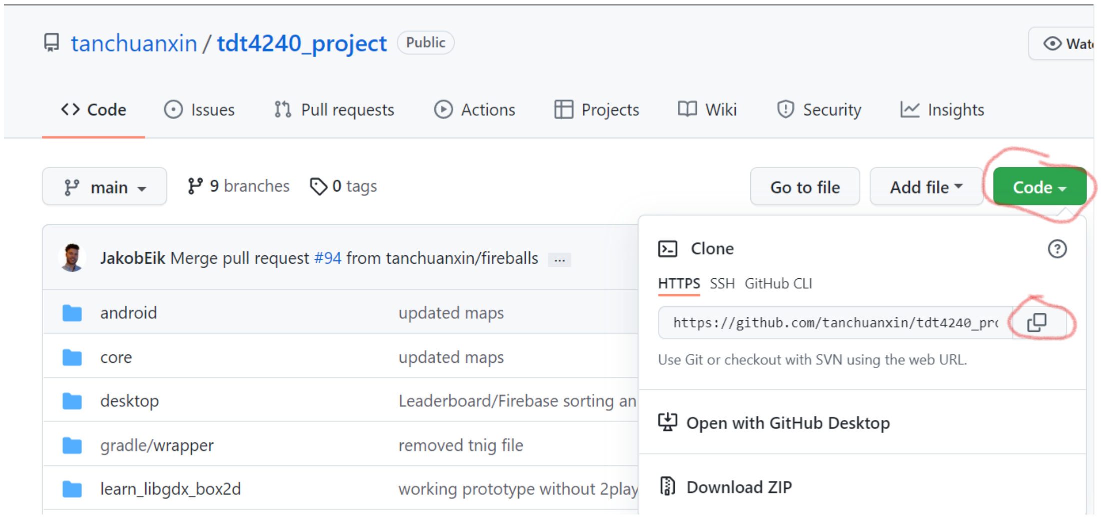
There are two ways of playing our game. You can either download the game for your android device, or you can clone the codebase and run the game through the Android Studio IDE(if you don't have it, just google it and download).

3.2.1 Play on android device

1. Make sure your device has an Android API version 32 or more. Less than 32 can work, but no guarantees.
2. Make sure that the browser you will be using for the download of the APK-file from has enabled downloads from unknown sources (Settings — Security — Install from unknown sources — Enable browser).
3. Click this link
4. Install the APK file on your Android device
5. Open the app you have installlled to play the game

3.2.2 Clone project - Android Studio

The first thing you need to do is to go to our GitHub repository, and clone the project. this is done by clicking the "Code" button in the project, and copying the link that pops up.



JakobEik Merge pull request #94 from tanchuanxin/fireballs ...

android	updated maps
core	updated maps
desktop	Leaderboard/Firebase sorting an
gradle/wrapper	removed tnig file
learn_libgdx_box2d	working prototype without 2play

Figure 11: Copy the link for cloning like this

Once you have the link, you need to clone the project into Android Studio. There are several ways to do this, the easiest is to open a terminal in Android Studio, navigate to the folder you want the project, and typing "git clone *link*", with link as the copied link. When that is done, you need to open the project, and select an emulator to launch the game.

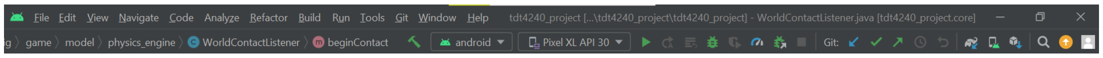

File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help
tdt4240_project [...\tdt4240_project\tdt4240_project] - WorldContactListener.java [tdt4240_project.core]
game model physics_engine WorldContactListener beginContact
android
Pixel XL API 30
Git:

Figure 12: select emulator by making it look like this

When this is done all you need to do is press run.

3.3 Playing the game

3.3.1 Menus

When the game launches, you will be shown the menu in figure 5. Here you can select game mode(1 or 2 players), or go check the leaderboards, or exit the game. To start a game against a friend, click 2 Player.

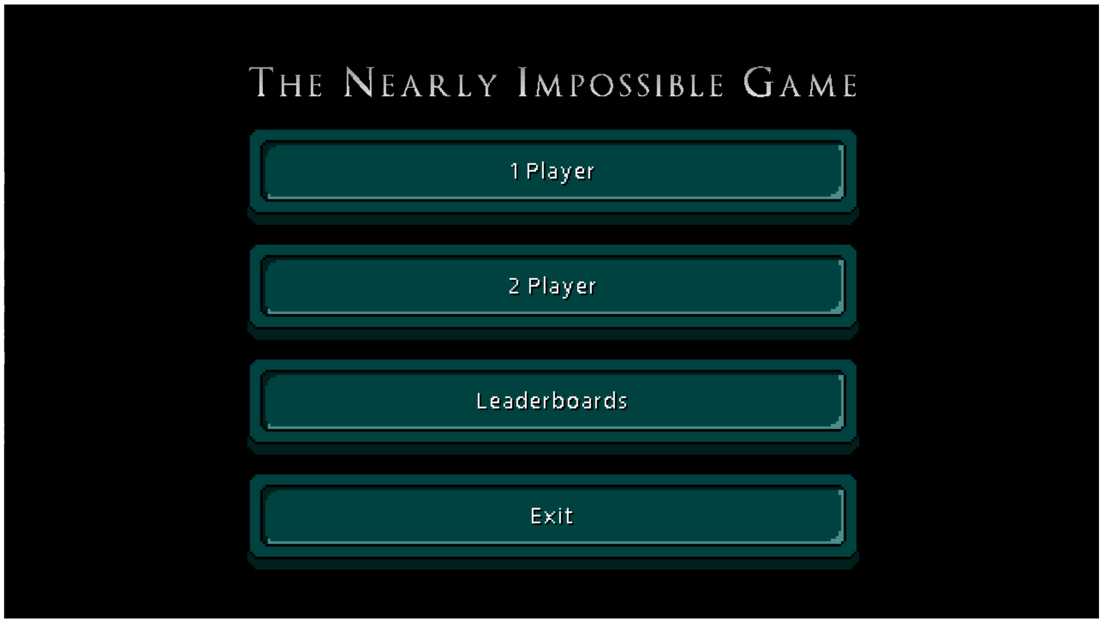


Figure 13: Initial menu screen

Then you will be directed to a map select menu(figure 5). To begin the game, click the map you wish to play.

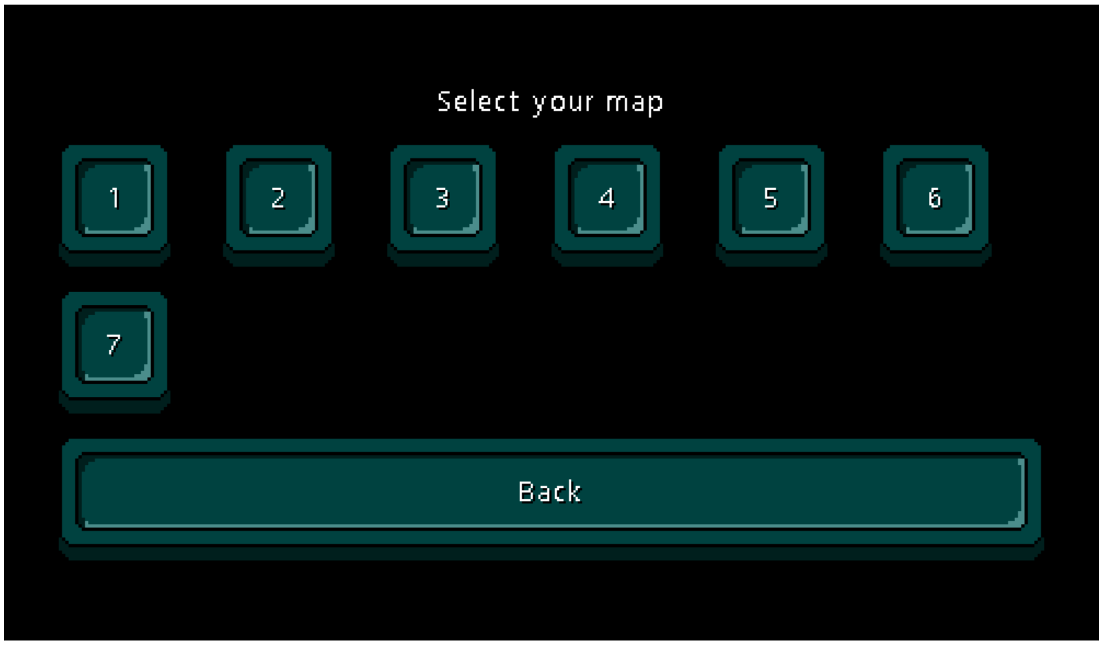


Figure 14: Map select menu

3.3.2 Game

When you then get into the game, you will see something like this, depending on which map you have chosen.

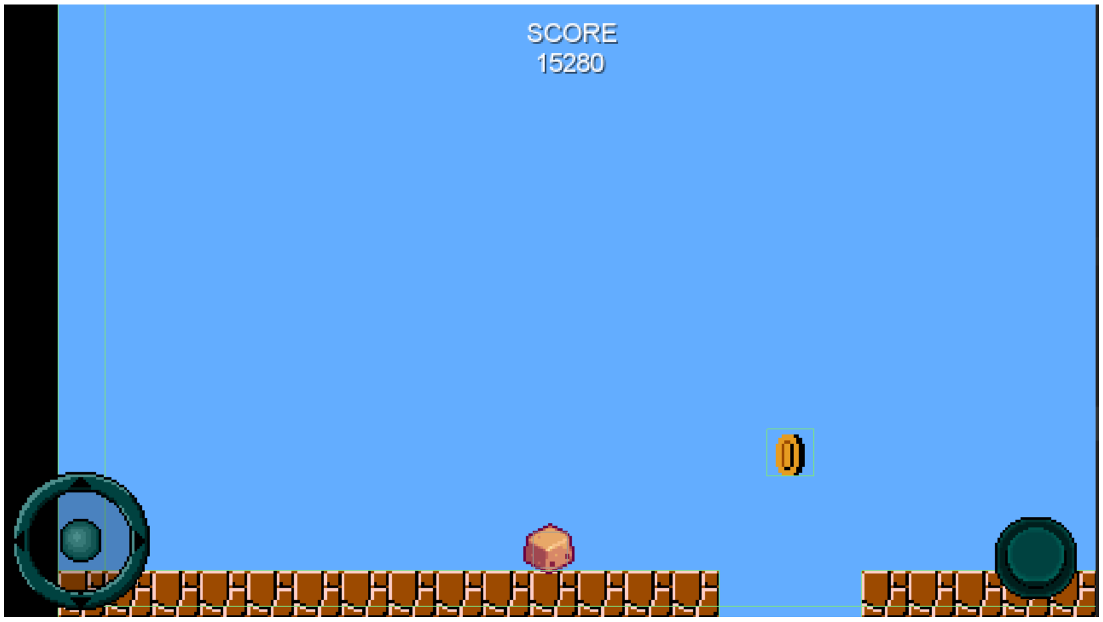
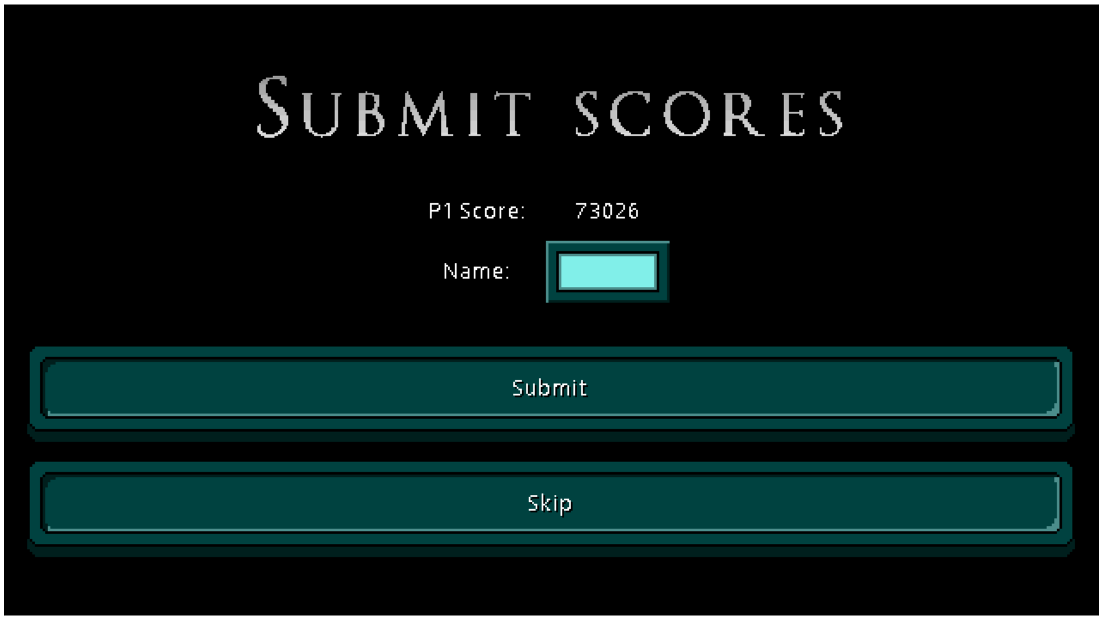


Figure 15: Game state

Here you are the jelly in the middle. For the mobile version, the controls are on each side of the screen. In the bottom left, there is the forwards/backwards movement controls, and in the bottom right, you have the jump button. In the top right is the attack button, used by the other player to interfere with you in multiplayer mode.

The desktop version has no jump button and touchpad. Instead, conventional arrow keys or WASD combinations are used to control the movement.

At the top of the screen is your score. In front of you there can spawn coins, obstacles and fireballs shooting through the map. Pick up the coins you can, avoid all the obstacles and fireballs, get to the end of the course to complete the map. When you die(or win), your friend gets their turn to play. When both players have died, you will see a ”game over” - screen, where you can type in your name(s), and the scores are sent to the leaderboards, and then go to the main menu.



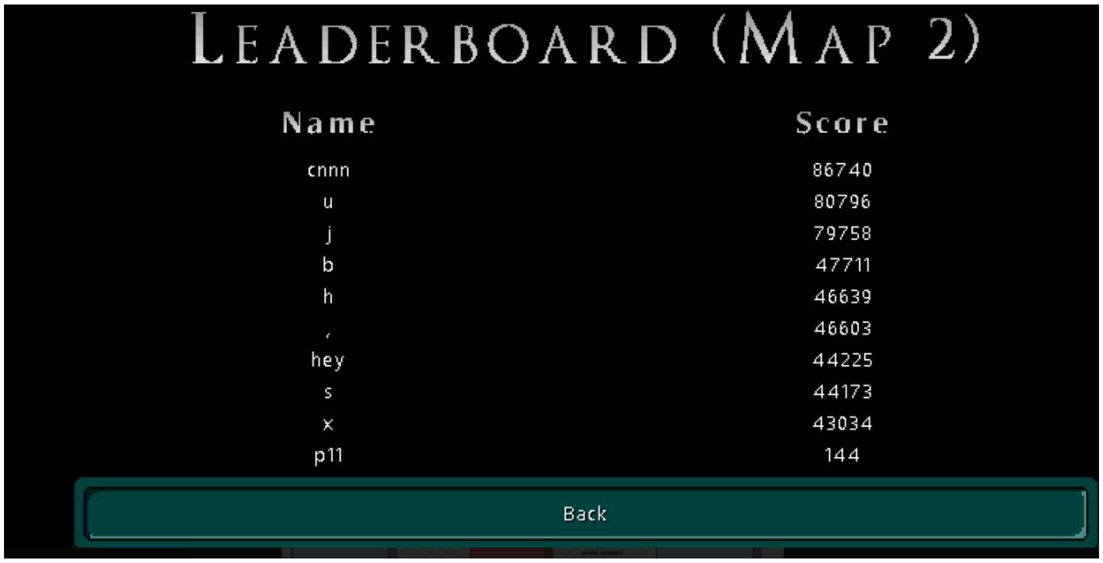
SUBMIT SCORES

P1 Score: 73026

Name:

Figure 16: Score submission screen

From there you can go to the leaderboards and see where you rank among your friends. This is the basic runthrough of how to play our game.



LEADERBOARD (MAP 2)

Name	Score
cnnn	86740
u	80796
j	79758
b	47711
h	46639
,	46603
hey	44225
s	44173
x	43034
p11	144

Figure 17: Leaderboards screen

4 Test Report

FR1.1, FR1.2	
Description	The game should allow and respond to touches from the user
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success
Comment:	Very simple, launched the game, tested clicking the jump button, which works as intended

FR1.3	
Description	The user should be able to make their character jump higher by holding down the jump button
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed
Comment:	FR discarded due to being deemed unnecessary, game has wall jumping instead

FR1.4	
Description	The user should be able to move forwards and backwards by pressing the left and right buttons
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success
Comment:	Movement controls work, but is more of a joystick of sorts, not multiple buttons

FR2.1	
Description Executor:	The game should produce obstacles like spikes and boxes Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success
Comment:	Game creates boxes to jump on and spikes to crash into. Also fireballs

FR2.2	
Description Executor:	The game should produce obstacles that increase in difficulty, creating a more complex environment Gaute Brandser
Date:	23.04.2022
Time used:	4 minutes
Evaluation:	Success
Comment:	Our maps are shorter than originally planned, so there is not a huge amount of variation inside of a single map, but they get harder throughout, and there are eight of them, so there is an increasing complexity of environments

FR3	
Description Executor:	The game should cause the character to die if it hits a damaging obstacle (spikes) Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success
Comment:	Player dies when they hit a spike. Also when they hit a fireball

FR4	
Description Executor:	The game should have support for two players alternating to run, allowing for offline turn by turn multiplayer Gaute Brandser and Ruben Rokkones
Date:	23.04.2022
Time used:	2 minutes
Evaluation:	Success
Comment:	Friends can play 1v1 as intended

FR5	
Description	The second player should be able to create obstacles with a touch
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed(sort of)
Comment:	FR discarded/changed. Player who is not currently playing can push a button to shove the active player in a random direction, likely killing them.

FR6	
Description	The game should have the music inspired by the original game
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success(sort of)
Comment:	There is some music in the game, but not the same as the original game. Our music is much better

FR7	
Description	The game should give the player a score based on the amount of deaths and time taken to complete a course
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	2 minutes
Evaluation:	Success(sort of)
Comment:	Conditions for how and when the player gets their score was altered. Final version gives score based on coins collected, how far the player gets, and time. Also and player only has one life

FR8	
Description	The user should have a unique character model as compared to the other player
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed
Comment:	Both players have the same character model, wasn't implemented due to time constraints

FR9	
Description	The game should have gravity, such that after a jump, the user falls downward
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success
Comment:	Player falls downward after jump

FR10	
Description	The game should have different courses/maps, like the original game
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Success
Comment:	I go from main menu into 1 player, and see multiple different maps to choose from

FR11	
Description	The game should have an option for creating savepoints in the course, giving up high score
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed
Comment:	FR always planned as a luxury functionality, not something important. Discarded, not actually implemented

FR12	
Description Executor:	The game's background color should change a different points throughout the course Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed
Comment:	Luxury functionality, not implemented due to lack of time.

FR13	
Description Executor:	The game should have support for an online high score list Jakob Eikeland, Teo Chen Ning and Tan Chuan Xin
Date:	24.04.2022
Time used:	2 minutes
Evaluation:	Success
Comment:	Game has leaderboards and player high score is posted

FR14	
Description Executor:	The game should have support for a co-op mode Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed
Comment:	Discarded FR, focused more on the regular game mode of playing against each other

FR15	
Description Executor:	The game should spawn power-ups (e.g. revive on the spot) Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed(sort of)
Comment:	We went away from the idea of power ups due to lack of time, but instead implemented coins to pick up for additional score, and fireballs for additional challenge

FR16	
Description	The players will get to pick their characters at the start of each run
Executor:	Gaute Brandser
Date:	23.04.2022
Time used:	less than 1 minute
Evaluation:	Failed
Comment:	Players use same character model

M1	
Description:	Developer should be able to add another level within 120 minutes.
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Design time
Stimuli:	Wants to create another level for the players.
Response:	The player should be able to play the new level once they update the game, and the level should be correctly integrated into the architecture and tested.
Expected response measure:	Done within 120 minutes without adjusting other modules.
Observed response measure:	Done within 45 minutes without adjusting other modules.
Evaluation:	Success
Comment:	Well within expected response measure.

M2	
Description:	Developer should be able to add a new character design in 30 minutes.
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Design time
Stimuli:	Add new character design to the list of available characters
Response:	Player should be able to play with the new character after they update the game, and character should be correctly integrated in the architecture and tested
Expected response measure:	Done within 30 minutes given that the design already exists
Observed response measure:	Not able to be done within 30 minutes
Evaluation:	Failed
Comment:	Functionality for players choosing characters is not currently implemented, so this scenario needs work to be able to be completed in 30 minutes

M3	
Description:	Developer should be able to create new obstacles in 90 minutes
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Design time
Stimuli:	Create a new type of obstacle which can be implemented into current and new levels
Response:	Player should experience levels with a new type of obstacles and new obstacle should be correctly integrated in the architecture and tested
Expected response measure:	Done within 90 minutes.
Observed response measure:	Done within 60 minutes.
Evaluation:	Success
Comment:	performed this test by adding fireballs well within expected response measure.

M4	
Description:	Developer should be able to add new game mode within 240 min.
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Design time
Stimuli:	Create a new game mode which allows for a different gaming experience within the limitations of the existing game
Response:	Player should have the opportunity to select preferred game mode, and play both of them after update. New game mode should be correctly integrated in the architecture and tested
Expected response measure:	New game mode should be functional after 240 minutes
Observed response measure:	New game mode is functional after 240 minutes.
Evaluation:	Success
Comment:	We began with a multiplayer mode, and then performed this test by adding the singleplayer mode within the expected response measure

M5	
Description:	Developer should be able to add new music within 20 minutes
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Design time
Stimuli:	Implement new music to an existing level.
Response:	After update the level should have the new music implemented during play time. New music should be correctly integrated in the architecture and tested
Expected response measure:	Done within 20 minutes
Observed response measure:	Done within 10 minutes
Evaluation:	Success
Comment:	We tried changing music both in menu and in the game, and it worked within expected response measure.

M6	
Description:	Developer should be able to add power-ups and environmental changes within 150 minutes
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Design time
Stimuli:	Create a new element for the second player to interact with while the first player is playing the game
Response:	After update the level should have the new power-ups and environmental changes correctly integrated in the architecture and tested
Expected response measure:	Done within 150 minutes
Observed response measure:	Not done within 120 minutes.
Evaluation:	Failed
Comment:	We were not able to add to add power-ups due to time constraints. Environmental changes by the opposite player was changed to the ”shove”-mechanic where you can push a button to shove the other player

U1	
Description:	The end user should be able to understand the mechanics of the game by playing it the first time.
Executor:	Torbjørn Brandser
Date:	24.04.2022
Environment:	During first play
Stimuli:	Understand the game by playing it.
Response:	The end user should understand the game mechanics just by playing the game
Expected response measure:	Learn the game mechanincs by playing for 2 minutes.
Observed response measure:	Learned the game mechanics in 2 minutes.
Evaluation:	Success
Comment:	Performed this test on an outside party with no prior knowledge, and it was a success within the expected response measure

U2	
Description:	User configuring settings
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Runtime, but before play state.
Stimuli:	Wants to change settings, most likely volume.
Response:	User finds settings button, and changes desired setting
Expected response measure:	User changes a given setting within 10 seconds
Observed response measure:	Not able to change setting.
Evaluation:	Failed
Comment:	Settings screen not implemented due to time constraints. If user doesn't want sound, they can mute their phone.

U3	
Description:	User starts a game with a friend
Executor:	Gaute Brandser and Ruben Rokkones
Date:	24.04.2022
Environment:	Runtime, but before playstate.
Stimuli:	Success
Response:	User intuitively finds buttons for play, player select(1 or 2), map select and character select to open a new game
Expected response measure:	User goes from opening app to starting a game within 2 minutes.
Observed response measure:	User started a game within 2 minutes.
Evaluation:	Success
Comment:	Within expected response measure

A1	
Description:	Unexpected game crash should be handled
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Runtime
Stimuli:	Handle game crash
Response:	If game crashes/freezes/collapses for unexpected reason, the app should return to main menu without a problem
Expected response measure:	Game goes from unexpected crash happening to main menu ready to go again in less than 40 seconds.
Observed response measure:	Could not get game to crash...
Evaluation:	Uncertain
Comment:	It is good that I could not get it to crash, but it also brings uncertainty as to how a crash is handled. There is insufficient error-checking and error-handling code in the codebase, but no user behaviour generates exceptions.

A2	
Description:	Game should respond correctly to player input
Executor:	Ruben Rokkones
Date:	24.04.2022
Environment:	Runtime
Stimuli:	Player action
Response:	Game should respond to player input with intended action
Expected response measure:	The users input is handled correctly at least 99 percent of the time.
Observed response measure:	The user input is handled correctly hundred percent of the time during this test.
Evaluation:	Success
Comment:	This test could not find any input that the game did not react to.

5 Relationship with Architecture

There aren't many inconsistencies between the final implementation and our planned architecture, but in the following segment we will discuss a few of them.

5.1 Singletons

As mentioned in Section 2, we phased out the use of Singletons in favor of dependency injection in order to get better visibility into the dependencies of our classes at compile time and to aid testing by opening up the possibility of injecting mocks rather than having to fully create the class in the case of a Singleton.

5.2 Dependency Injection

Extensive use of dependency injection was present in our implementation of the game. Several of our classes like Spike, Fireball, Player etc. were all concrete implementations of AbstractModel which represents an object in our game map, and required by our game controllers to function. Various factory classes were used as the injector, which constructs the service classes (Spike, Fireball etc.) and provides them to the client class (StaticObjectController, AnimatedObjectController) for use. In this way, the client classes are removed from the details of the service, and can simply deal with the abstractions and handle object instantiations at runtime.

5.3 Controllers

Our initial plan was to have well separated controllers for each class. For example, we could have a generic ObstacleController and fan out using the Strategy pattern to specific controllers like SpikeController and such.

However, we quickly realized that this would result in too much duplicated code without much benefits, since it became difficult to maintain if a change had to be made. Rather, we had each Obstacle (e.g. Spike) implement its own business logic, combining the model and controller into one object.

While this breaks away from traditional MVC, it made more sense in our game architecture to co-locate this functionality, making it easier to modify and reason about our code. It is our belief that an architecture evolves as the project grows, and this architecture has worked well for us so far at our scale.

5.4 Pubsub

We were initially planning on leveraging direct dependencies for classes to call each other on certain events (e.g. player death). However, this quickly proved unwieldy and we did a quick implementation of a messaging queue through an EventManager, which allowed us to decouple our classes and make our code a lot cleaner since publishers of events did not have to concern themselves with subscribers, encouraging decoupling and allowing us to modify effects based purely off event data.

5.5 Multiplayer

Our original plan was to NOT implement real-time multiplayer as that would be too complex (syncing of various clients), rather opting for a turn based game. Our leaderboard was supposed to be real-time, allowing clients to see real-time updates. However, after internal discussion, we decided to implement it on an on call basis, whereby it only updates when the screen is refreshed. This saves on network bandwidth and performance, and in playtesting does not detract much from the user experience, making it a good trade-off.

5.6 Error handling

For usability, we planned to have appropriate error handling so users don't face unexpected exceptions with no resolution. During development, we encountered various errors and crashes, which required the application to be restarted. While we have fixed most major bugs, due to limited time, our error handling leaves much to be desired.

Future work should include better error messages, handling of exceptions instead of propagating it (and crashing the app), and try to keep the game running in the face of errors (resilience).

6 Problems, issues and lessons learned

6.1 Cooperation, communication, planning and workload

Our group had some issues during the implementation phase of the project. All the group members felt confident after the first phase, where we had outlined the requirements and architecture, and were ready to go. However, there were some problems when we were going to create the actual game. One problem was that several group members didn't feel confident in using LibGDX and creating games in general, and had little practice outside the first two exercises in the course. Thus we decided to take some time to learn and prepare. This was neccessary given the situation, but took up time that could have been used towards the actual project under different circumstances. When we began work on the project there were some problems in communication and cooperation within the group. We assigned tasks and got to work, but we found ourselves not making anywhere near the progress that we wanted. We weren't communicating problems they had trying to do the tasks, and we didn't get things done as quickly as planned, and ended up postponing implementation of different parts of the project. We ended up having to do way too much work towards the end of the project, because we had postponed and neglected a lot of things. However, during the final phase of the project, with a lot to do, everyone stepped up in a big way, and we managed to complete a product we are pretty happy with. It could have been worse, and in retrospect, the biggest lesson learned for us was to be clear in communication about problems, tasks, deadlines, and progress, and to actually try to make progress earlier on in the development in order to not be overwhelmed with work towards the end. This may be a very simple lesson, but hey.

6.2 Choices surrounding architecture, tactics

We had some issues planning the architecture early on. None of the group members had any significant experience working with software architecture, and so making decisions on what type of architecture we would need/want was not always easy. Also, deciding on different patterns and tactics to use when coding, before anyone has actually started coding was a completely new experience for most of us, and it was challenging. This also goes for the different views we were tasked with creating. Planning so far ahead that you can make diagrams describing in detail a game you haven't begun creating, for example, was difficult.

6.3 Detailed architecture/code structure early

One thing we think would have helped us immensely is to have created a structure/architecture in detail, and all become familiar with how it connects and what needs to be done. In retrospect we realise that is sort of the point of the course, to be able to plan the architecture thouroughly to help with work. We, however, as many other groups surely, didn't fully understand this at the time, and didn't put as much effort into the first architecture delivery as we probably should have.

6.4 Summary

Despite the issues, we all had fun working together at points. In spite of the time crunch towards the end, we had a lot of fun playtesting the game together and figuring out features, bugs, listening to the game music and more. Also, we definetly learned a lot about software architecture working on this. To some of us this seemed like a pretty simple game, but creating a solid architecture around it was very challenging.

7 Individual contribution

Everybody contributed equally.

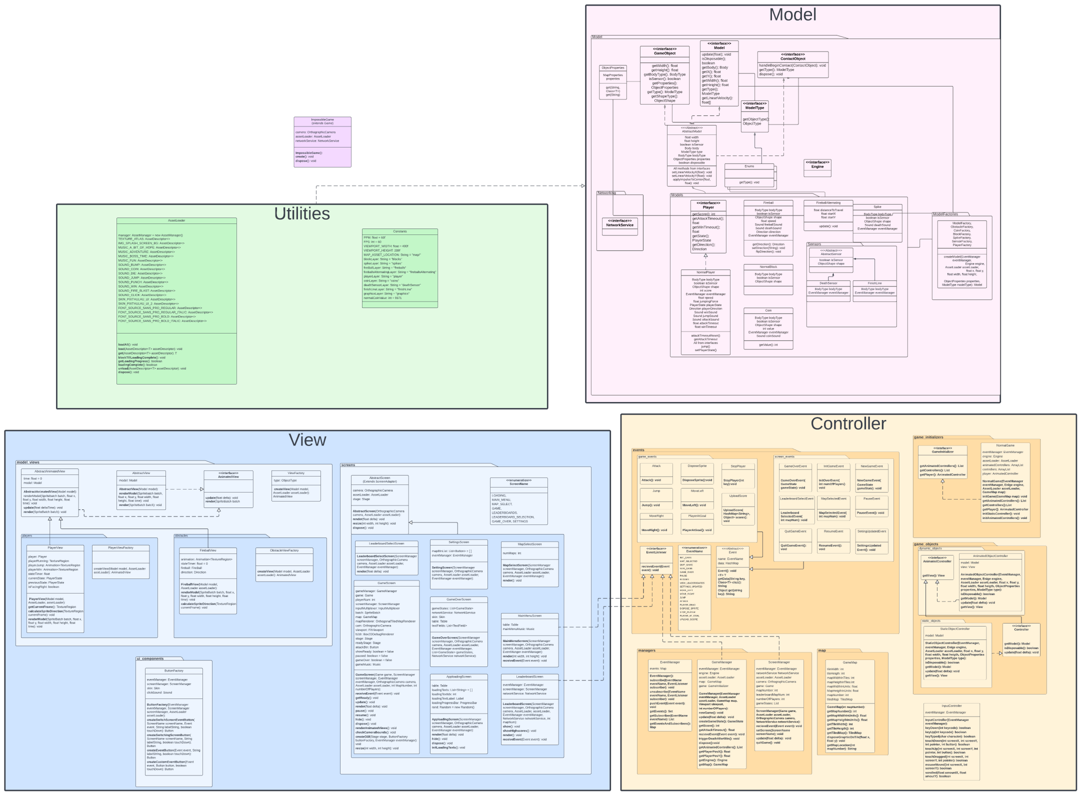


Figure 18: Full class diagram

References