Bilkent University

Department of Computer Engineering

CS 319 - Object Oriented Software Engineering Project

Bullet Drop

Design Report

Group 2F

Ulaş İş, Alperen Erkek, Ömer Faruk Ergün, Abdullah Seçkin Özdil

Course Instructor : Bora Güngören

Table of Contents

1.Introduction

1.1 Purpose of the system

Bullet Drop is a 2D shooting game. The main difference of Bullet Drop from the other arcade shooting games is that the target is invisible in the beginning. Therefore, the user doesn't know where to shoot on the first attempt, so they need to track the route the bullet follows and observe the outside effects to be successful on the next attempts. This way, we aimed to make the game more challenging and enjoyable for the player. Moreover, Bullet Drop will be graphically appealing to the player. We will have realistic pictures of weapons, bullets and maps that will be created through various programs and integrated into the game so that the game will include realistic models of weapons, bullets and maps(background) with different effects.

1.2 Design Goals

1.2.1 Adaptability

The game will be implemented in Java because one of our most important goals is that the game should work in all platforms and since Java provides cross-platform portability, it will allow the game to work in all JRE installed platforms. For that reason, we preferred Java to develop the game over other alternatives.

1.2.2 Efficiency

It is really important that the game runs fluently and the movement of the objects should be very smooth for the satisfaction and entertainment of the player. Response time is also very crucial. When player adjusts the position of the weapon or he/she fires the bullet, there shouldn't be any delay or lag. This would affect both gameplay and the enjoyment of the player. The requests of the user should be responded instantly. Since the game will have high-quality graphics and animations, it is really important that it doesn't affect the performance and smoothness of the game.

Therefore, we want to minimize the input lag as much as possible and the game should run smoothly.

1.2.3 Reliability

One of the most common reasons for the bugs in games are invalid inputs. Therefore, we need to create the system such that it will ignore any irrelevant or unexpected input so that the game will not give any errors during run-time and it won't crash. To achieve this, the tests will be done throughout the implementation of the game. We will try to consider each situation that can happen in order not to miss anything that can cause crashes in the game. Briefly, the game will be bug-free and reliable for the user.

1.2.4 Usability

It is really crucial for us that our game will be user-friendly. To achieve this, we will have a very simple yet effective user-interface. The player will easily learn how to play the game and not get lost in a complex user-interface. There won't be much options on the setting screen and one of the options will be how to play which will teach the player about the game, which is really easy to play anyway. Therefore, the player won't have difficult time trying to figure out how to play the game and how things work. The only controllers in the game are arrow keys and space key in the keyboard. Again, these features will be explained to the player in the game. As one of our most important goals is to make the game entertaining for the player, the user-interface should be simple enough, learning how to play should be easy, but this doesn't mean that the gameplay will be easy either. Figuring out how the game works will be easy, however, the game itself should be difficult enough to keep the player in the game and prevent them from getting bored. Of course, the game shouldn't be very difficult either, that would also decrease the enjoyment of the player.

1.2.5 Extensibility

We desire our game to be open ended, meaning it should be open to new features and can be developed further since the system is object-oriented. There can

be more weapons, weapons or outside forces in the game, or the current versions of these features can be modified easily. By adding new features, we aim to keep the player interested in the game. Moreover, one important goal of the project is that adding new features or modifying them should be simple and easy, so there won't be any problems and conflicts.

1.2.6 Tradeoffs

1.2.6.1 Efficiency - Reusability

One of our most important design goals is efficiency and in order to achieve this, we need to sacrifice reusability. Since none of our classes in our project will be used for different projects, we will design them so that they will only be functional for our game, which will help us write more simple codes which will improve efficiency of the system.

1.2.6.2 Usability - Functionality

In our system, it is really important to make the game easy to use for the player. The player should be able to play it with a simple and easy user-interface. Therefore, to achieve this goal, we prioritized usability over functionality. In other words, our system won't have complex user-interface or complex features that may be difficult for the user to learn and use. We want our system to be simple and easy to use but also effective. Therefore, we won't sacrifice any basics of the game, but remove some details and insignificant features for our game. For instance, we won't have any difficulty level for the game. We also won't have too many options and settings in the game that can confuse the players.

1.2.6.3 Performance - Memory

As discussed before, performance and efficiency is one of the most important design goal for our project. We want the movements of the objects be really smooth so performance is a really important goal for our system. To get and maintain high performance, the memory space required may increase. This will also increase the speed of our game and prevent and input lags or other problems related to efficiency and performance. Increasing memory will also reduce the workload of "Bullet Drop"

class. Hence, we will sacrifice memory in order to achieve our performance goal. However, it will increase the speed of the game and the game will run more smoothly, and we will provide the player with a better game experience.

1.3 Definitons, acronyms and abbreviations

- JRE : Java Runtime Environment. It's a software package that contains what is required to run a Java program.[1] Therefore, JRE needs to be installed on your computer in order to run Java applications.[2]

- FPS : Frames Per Second. FPS is the frequency at which frames are displayed in an animated display. [3]

- Bullet Drop : The core class for our system. It will use different components and will work pretty much like an engine.

1.4 References

[1] https://en.wikipedia.org/wiki/Java_virtual_machine. Retrieved December 2, 2017

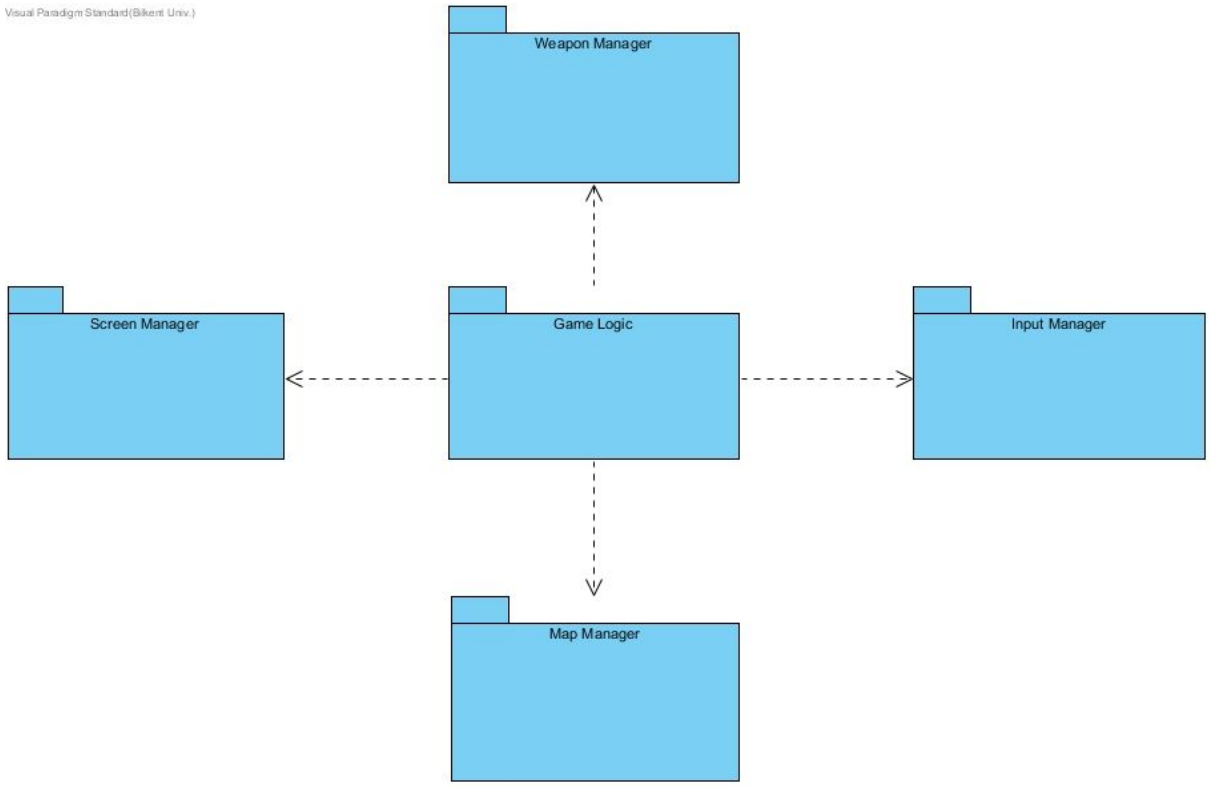
[2] https://techterms.com/definition/jre. Retrieved December 2, 2017

[3] https://en.wikipedia.org/wiki/Frame_rate. Retrieved December 2, 2017

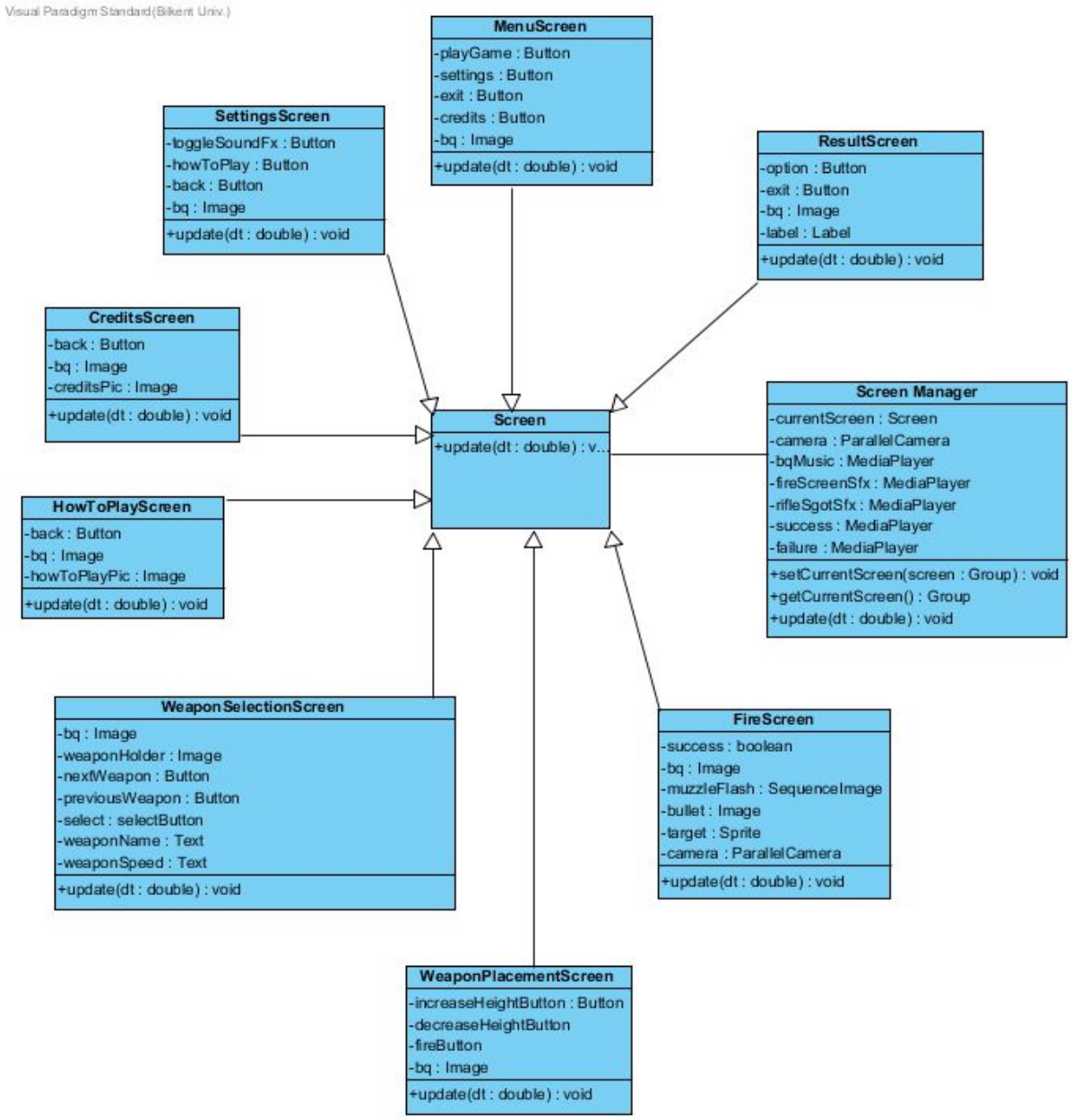
2. Software Architecture

2.1 Subsystem Decomposition

We have decided to create our system based on 4 major components and 1 game logic to control them all.



Above figure shows these components and their controller. “Weapon Manager, “Input Manager”, “Screen Manager” and “Map Manager” will be main components.The “Game Logic” component will use these components and control them.



Screen Manager will be responsible for switching between screens and screens itselves. There will be 8 different screens. These will be: “MainMenuScreen” as the name suggest, will be showing main menu and corresponding buttons. “SettingsScreen” will be showing settings such as “SFX Toggle”, “How to play” and similar settings like that. “CreditsScreen” will be showing contributors. “WeaponSelectionScreen” will be showing different weapons and first map that user will be showing. So that user will be looking at current map select its weapon based

on that map. “WeaponPlacementScreen” will be showing current position of the selected weapon proper hints to how to place it in an efficient way. “FireScreen” will be showing current movement of the bullet and how it behaves under forces. “HowToPlayScreen” will show the player how to play the game. Finally “ResultScreen” will be showing whether user hit the target or not. All of these screens will be used by Screen Manager and this component will have proper methods to chose current screen.

Similarly “WeaponManager” component will be responsible for 4 different weapon types and current selected weapon. It will be holding anything related to weapon’s properties.

“Input Manager” as the name suggest, will be responsible for the inputs that user give. When “Bullet Drop” ask the “Input Manager” whether a key-button has been pressed, it will do the calculations and tell the answer.

“Map Manager” will be responsible for the map properties such as their forces. There will be 4 different maps and each map will have different force value and different force type. “Map Manager” can tell “Bullet Drop” the current map, current forces or “Bullet Drop” can tell “Map Manager” to create a random map or update the level.

2.2 Hardware/Software Mapping

Since we’ll be developing this game on Java, game will require JRE to be executed. As hardware requirements, the game will be require a keyboard and a mouse. So that user can interact with the game. Since we’ll be using sound effects and graphical visual, user will also be needing a sound device to hear the sounds and proper GPU to see visuals. Java’s graphics does not require a complicated GPU so anything would work in that case. Only requirement here is, GPU should have GPU Acceleration feature.

2.3 Persistent Data Management

We’ll be saving our resources in the same file where game will be located. They won't be encrypted. Any modders who would like to “mod” the game in their

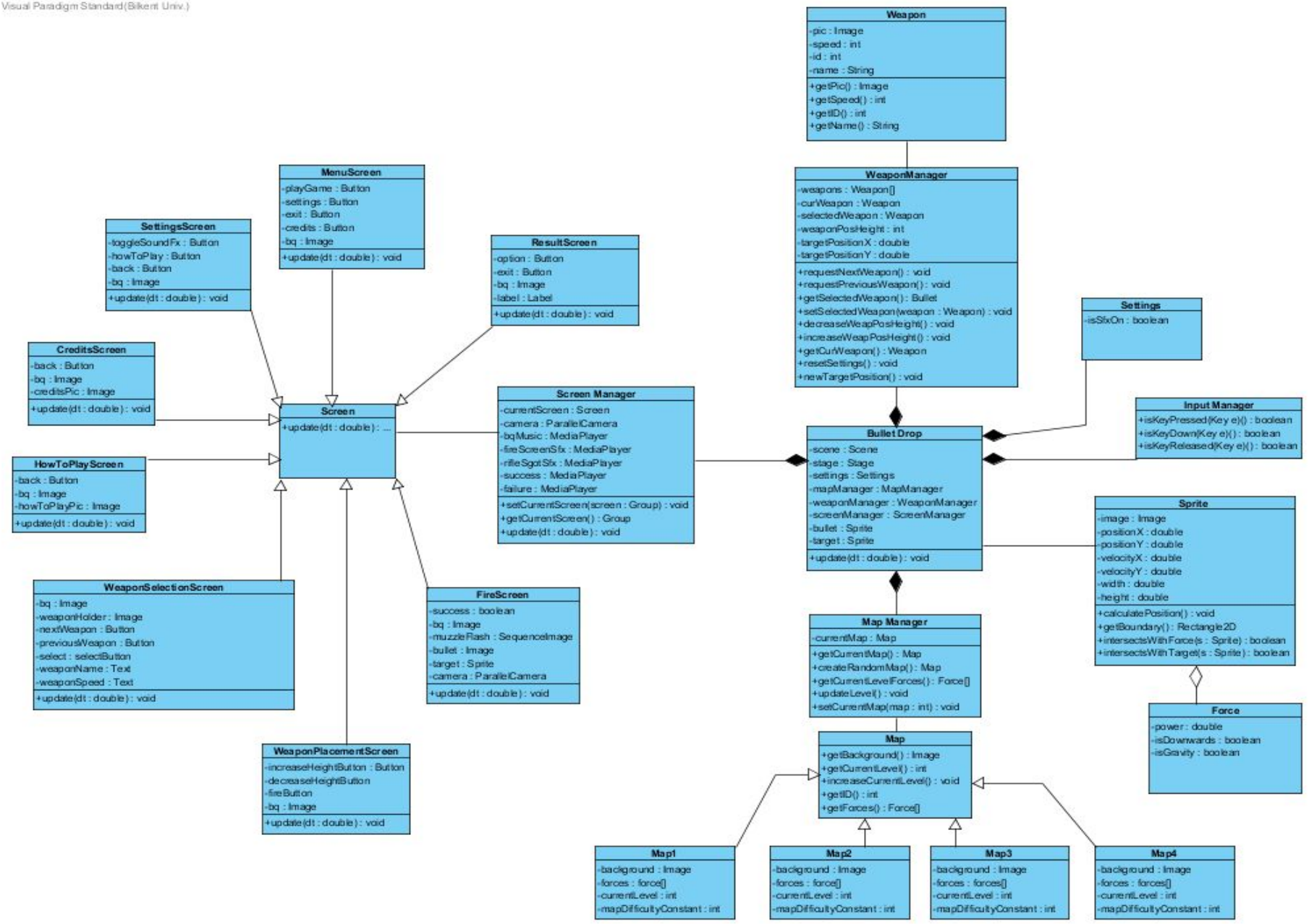
desire will be able to change it. That's being said, we also will not be require complicated data management system. We will simply "load" the resources that will be required in that screen. Then once we used that screen and wanted to switch next one, we'll load new ones. Since we'll use minimum amount of pictures and sound effects, loading them in a efficient way won't matter in our case.

2.4 Access Control and Security

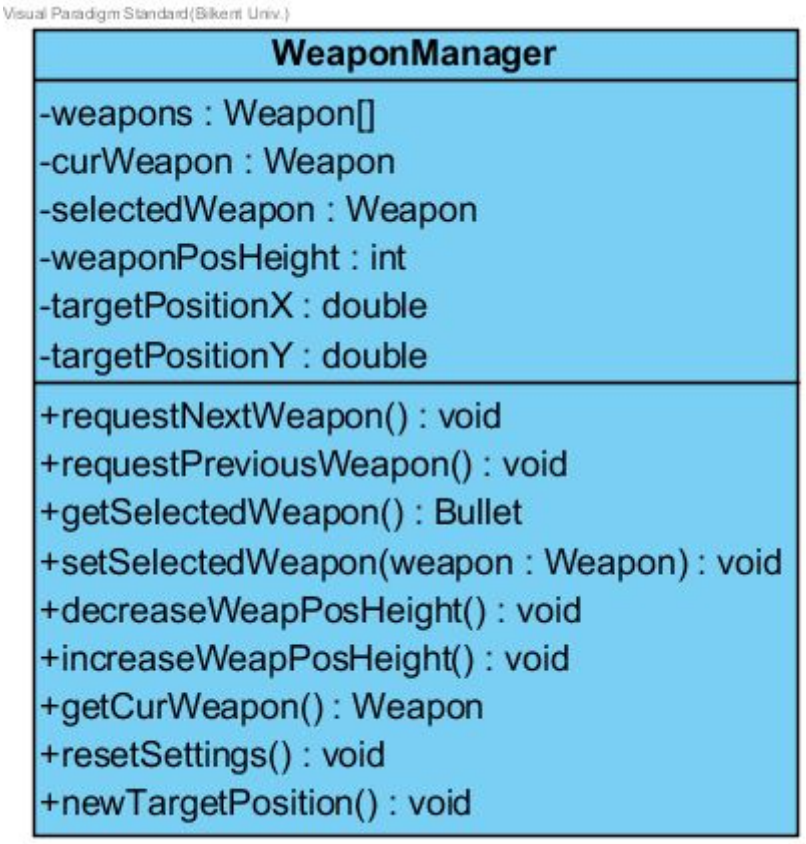
There won't be any security level other than input controls. We want our user to not reach our core components. Therefore game logic should only use the functions of these components. The variables inside the components should be private and should be written only in their class. Our game logic will only read it. The part of the reason why we separate controller, model and view is this. We want components to responsible only themselves.

3. Subsystem Services

The detailed class diagram is given below. It features basic functions and features that each component does and should give basic understanding of how each component works. "Bullet Drop" will have the main class.



WeaponManager Class



Bullet class determines and rules the main options about different weapon types and their bullet reactions under the effect of external forces. It also handles target's position that is changing every level and map.

Attributes

-weapons: Weapon[]

This arrays keeps the weapons that can be used in game. It's being filled in constructor by different weapon properties. Everytime we go through weapons in weaponSelectionScreen, we do it with this array.

-curWeapon: Weapon

Our game has different weapons for players. As we mentioned before players need to be select weapon before starting the game so that this method keeps this

weapon until it change. In other words, this method keeps the track of the weapon that user is seeing in weaponSelectionScreen.

-selectedWeapon: Weapon

This variable is for keeping the selected weapon from the user at that time. Once user selects its favorite weapon from GUI, this value is being set. At that moment, in other words, it marks curWeapon as selected.

-weaponPosHeight: int

This is the position of the weapon, that is, its height from the ground.Initially it is assigned to be in the middle of the map in constructor. Then in weaponPlacementScreen, user changes this value through methods called increase and decrease weapon height.

-targetPositionX: double

This variable is the position of the target in x dimension, that is, the distance of the target from the weapon. It's assigned to be at the end of the map in the constructor.

-targetPositionY: double

This is the distance of target from the ground, means its height. It's being assigned randomly between the values 0 and map's height-that is 720 in the constructor.

Methods

requestNextWeapon() : void

This method changes the curWeapon to the next one.If it's the last weapon, then simply shows the first weapon. It's being called in weaponSelectionScreen through GUI.

requestPreviousWeapon() : void

This method is similar to the method above. It simply does the job in reverse way. It's also being called in weaponSelectionScreen.

getSelectedWeapon() : Weapon

It returns current selected weapon as a Weapon object.

setSelectedWeapon(weapon : Weapon) : void

After choosing process is done this method sets the as selected. This method takes a weapon object as parameter and sets selected weapon if that weapon is not null.

decreaseWepPosHeight() : void

This method decreases weapon height by changing weapPosHeight variable.It checks first if current height is in the map's height-that is it's not out of the screen then it decreases the value. It's being called in the weaponPlacementScreen.

increaseWepPosHeight() : void

This method is reverse version of above method. Only difference is it checks if current height is in the map so that we won't go above map height.

getCurWeapon() : Weapon

Returns the curWeapon:Weapon variable.

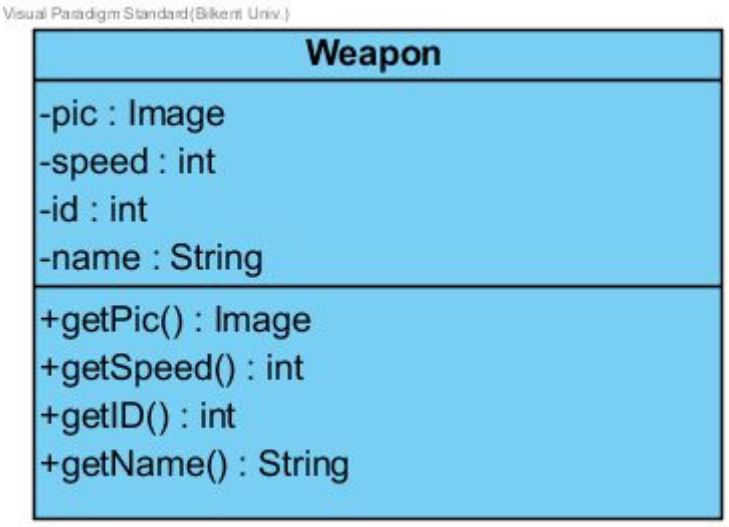
resetSettings() : Void

This method first checks whether our weapon array is empty or not. If it's empty, it gives error. Then it sets selectedWeapon variable as null and sets the curWeapon variable to the first item on the weapon array. Then it sets target's x position to the end of the map and y position to the random number between proper map heights. This method being called every time we finish a map.

newTargetPosition() : Void

It resets the target position described as above.

Weapon Class



This is our main weapon object classs. In our WeaponManager class, everytime we want to add a new weapon in weapons array, we create new Weapon object.

Attributes

-pic: Image

This variable holds the picture of the weapon. It's being set in constructor.

-speed: int

This variables effects how fast bullet goes once its fired. It's being set while creating object and defined by implementer.

-ID: int

This variable holds the ID of the weapon. It's i'th weapon we created. It also helps us to load picture of the weapon from resources. It's being set while creating object and defined by implementer.

-name:String

This variable hold the name of the weapon. Similarly, It's being set while creating object and defined by implementer.

Methods

-getPic(): Image

Returns picture of the weapon as Image.

-getSpeed(): int

Returns speed variable of the bullet as int

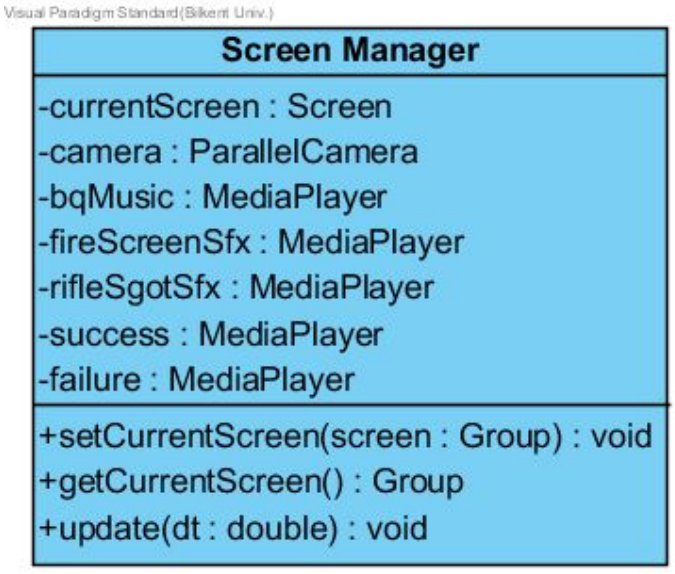
-getID(): int

Returns id of the weapon as int.

-getName(): String

Returns name of the weapon as String.

ScreenManager Class



This class rules which screen will appear on game according to users choices. It also handles minor sound effects. We did not create an additional sound manager because we only had 3 sound effects which was enough to cover whole game.

Attributes

currentScreen: Screen

Current screen is the one which is the main screen of the game that is seeing by user. It has different possibilities such as main menu, weapon selection or settings screen.

camera:ParallelCamera

This variable is useful to us in fireScreen where we move the whole scene to the right while following the bullet.

bqMusic,fireScreenSfx,rifleShotSfx,success,failure: MediaPlayer

These are the variables that holds the sounds in them. We start or stop playing them once a scene is changed. For example we start bqMusic almost every screen except fire and result screen. In fireScreen, we play rifleShot and fireScreenSfx(tension sound) and in result screen we play success or failure depends on the result. These variables are handled in Update method.

Methods

setCurrentScreen(screen Group) : void

Sets the new screen to change the screen that seen by user. It takes a variable called screen, which is a Group.

getCurrentScreen() : Group

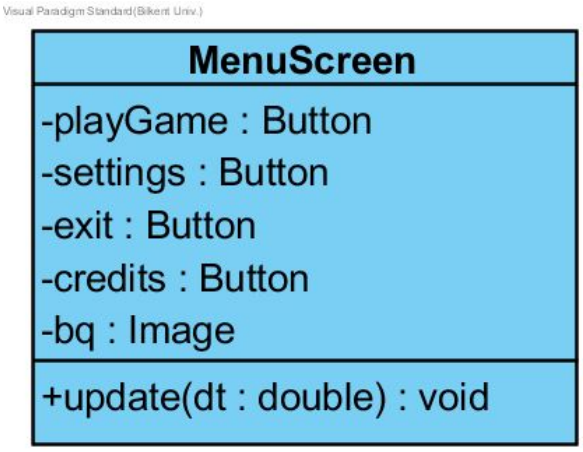
Returns the current screen panel.

update(dt: double): void

This is the method where the screen is updated in the game. According to the current screen chosen by user, it has different if statements in which curScreen

is compared to each screen and curScreen is instanceof that screen, that means there are several possibilities which screen the user chose. For example, if the user clicked “back” on credits screen, that means the we need to show mainScreen. Therefore, we create a new main menu screen and set it to the current screen through setCurrenScreen. This method takes a double variable called dt. We also handle our sound effects here. While changing screens, we play or stop sound effects depending on our screen. This method also calls the update method of current screen.

MenuScreen Class



It is the class of the first screen that user see when he/she open the game. It will consist of buttons which will have all listeners for their actions. Every button signals screen manager to change the current screen.

Attributes

playGame : Button

This button is to start game. Actually, after running the game you can go to the weapon change screen by using play game.

settings : Button

By using ‘Settings’ button you can go to option menu to change volume of game and see how to play screen.

credits: Button

With this button you can go into creditsScreen.

exit: Button

Exits the game.

bq: Image

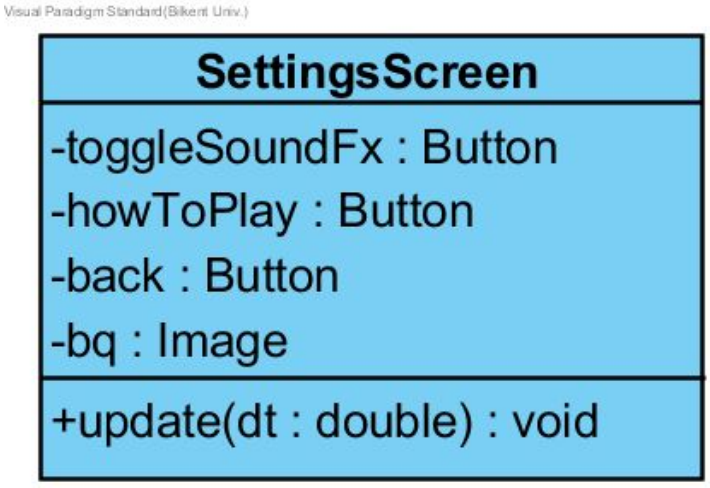
It holds the background image of the main menu. It's ben set in constructor.

Methods

update(dt: double): void

Draws the background image once it gets called.

SettingsScreen Class



Keeps the basic choices/settings that user can change.

Attributes

toggleSoundFx: Button

User can toggle on/off the sound effects of the game by clicking this button. It changes the isSfxOn value in Settings class.

howToPlay: Button

User can have information about how to play the game by clicking this button. A new screen is opened when clicked this button.

back: Button

Signals screen manager to change current screen to main menu.

bq: Image

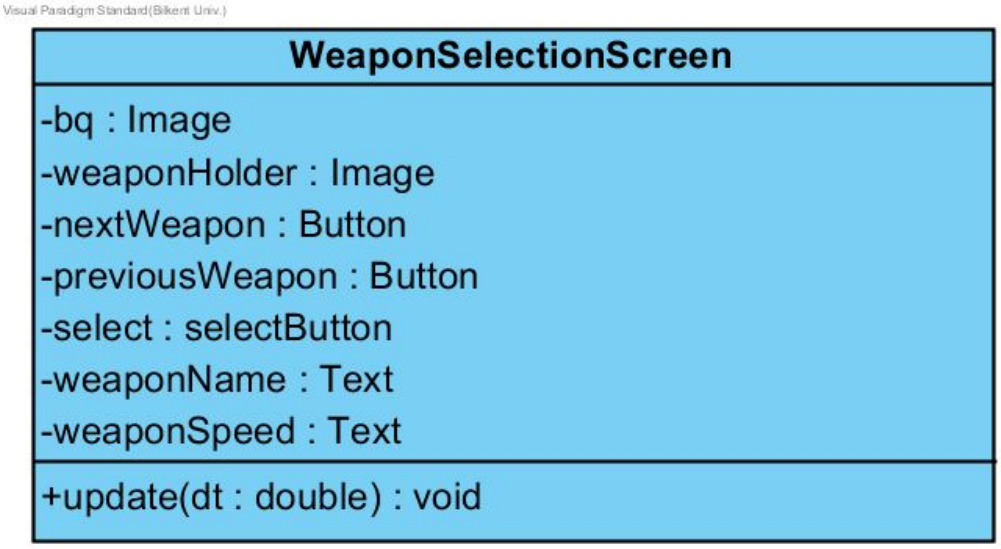
It holds the background image of the main menu. It's ben set in constructor.

Methods

update(dt: double): void

Draws the background image once it gets called.

WeaponSelectionScreen Class



It is the screen that user can select/change his/her current weapon.

Attributes

bq: Image

This is the background image of the current map. It's being set in constructor.

weaponHolder: Image

An image of a wooden holder that will be the background of weapons.

nextWeapon: Button

The button for requesting the next weapon. It calls the weaponManager's method called requestNextWeapon

previousWeapon: Button

The button for requesting the previous weapon. It calls the weaponManager's method called requestPreviousWeapon

select: Button

The button for selecting the current weapon. It calls the weaponManager's setSelectedWeapon and then signals screen manager to change current screen to weaponPlacementScreen.

weaponName: Text

It holds the name of the current weapon's name and updated on Update methods.

weaponSpeed: Text

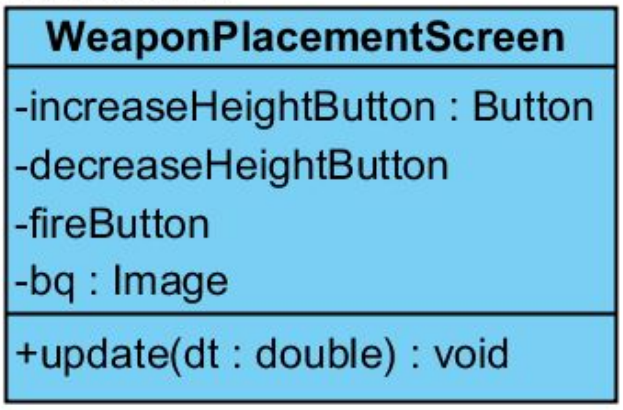
It holds the speed of the current weapon's name and updated on Update methods.

Methods

update(dt: double): void

Draws the background, weaponHolder and currentWeapon's image once it gets called and updates the weaponName and weaponSpeed.

WeaponPlacementScreen Class



After selection of weapon this screen will be shown by ScreenManager. It is the screen that user can first determine the position of weapon to fire.

Attributes

increaseHeight : Polygon

When clicked that button, it calls weaponManager's method called increaseWeaponHeight. The reason is it's Polygon type is shape of the button is triangle.

decreaseHeight: Polygon

When clicked that button, it calls weaponManager's method called decreaseWeaponHeight. The reason is it's Polygon type is shape of the button is triangle.

fireButton: Button

When clicked that button, the bullet is fired from the current position of the weapon and signals screenManager to change current screen to fireScreen.

bq: Image

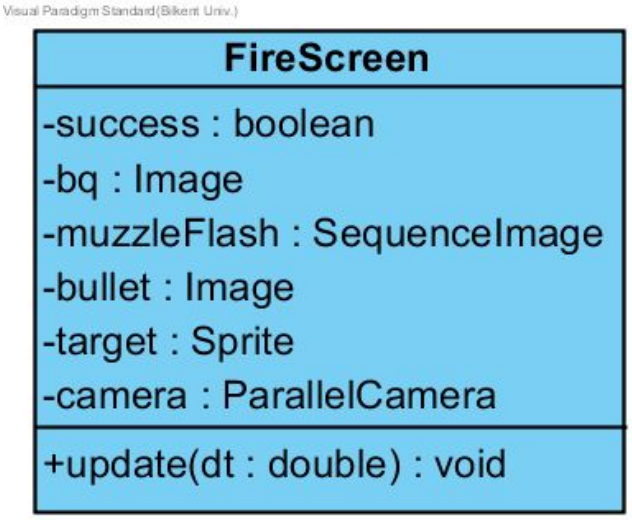
This is the background image of the current map. It's being set in constructor.

Methods

update(dt: double): void

Draws the background and updates the current weapon's image based on its position.

FireScreen Class



The screen that fire process occur after the placement of weapon is done.

Attributes

success : Boolean

Boolean variable that is determined true if user can hit the target, otherwise it is false. It is determined by the final positions of the bullet and the target and checked by collusion method in Sprite class.

bq: Image

This is the background image. It's being set in constructor and depends on the map.

muzzleFlash: SequenceImage

This is a animation that starts at the beginning of the fireScreen. It shows muzzle flash effect. It's properties has been set in constructor.

targetImage: Image

The image of the target. Target's position is being given by the weaponManager.

bulletImage: Image

The image of the bullet. It's position will be updated in Update method.

camera: ParallelCamera

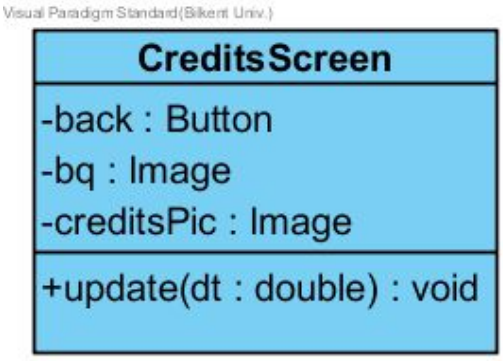
This variable able us to move the scene to the right while following bullet. It's translation is being set in Update method.

Methods

update(dt: double): void

Draws the scene and updates the camera based on bullets position. It will tell screenManager to show resultScreen once bullet hits target or misses the target.

CreditsScreen Class



Basically it is credits screen and will have text and return back button.

Attributes

back : Button

Returns user back to main menu.

bq: Image

The background image of the credits screen.

creditsPic: Image

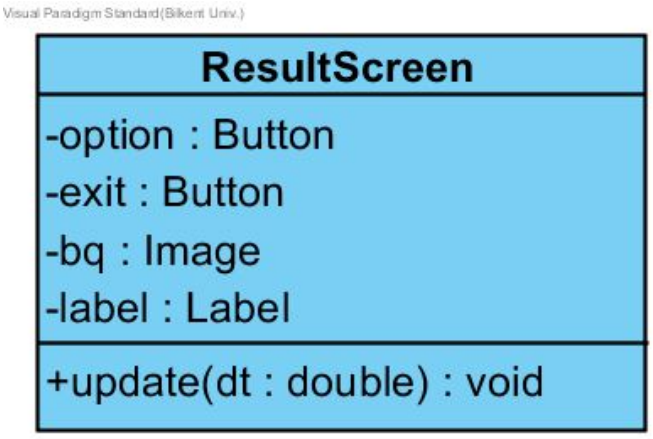
The image that wil show the credits.

Methods

update(dt: double): void

Draws the background and creditsPic.

ResultScreen Class



After fire is done, result screen appears, shows the result and if he/she cannot hit gives retry option as well as exit option. If player hits the target, it will show next button.

Attributes

option: Button

This button's behaviour depends on the result of the player's shot. If player hits the button, this button will show "next" text on it and once clicked, it will tell screenManager to switch to weaponSelectionScreen or weaponPlacementScreen (depends on in which level player succeeds). If player misses target, then this button will show "try again" text on it and similarly it will tell screenManager to set curScreen to weaponPlacementScreen if it gets clicked.

exit: Button

Once clicked, it will signal screenManager to set current screen to MainMenu.

bq: Image

The backgorund image of result screen class. Depends on current map.

label: Label

Shows “Success” or “No hit” based on the result.

Methods

update(dt: double): void

Draws the background image.

HowToPlayScreen Class

The class for how to play screen. User can learn how to play the game in this screen.

Attributes

back : Button

Once clicked, it will signal screenManager to set current screen to mainMenu

bq: Image

The background image of the credits screen.

howToPlayPic: Image

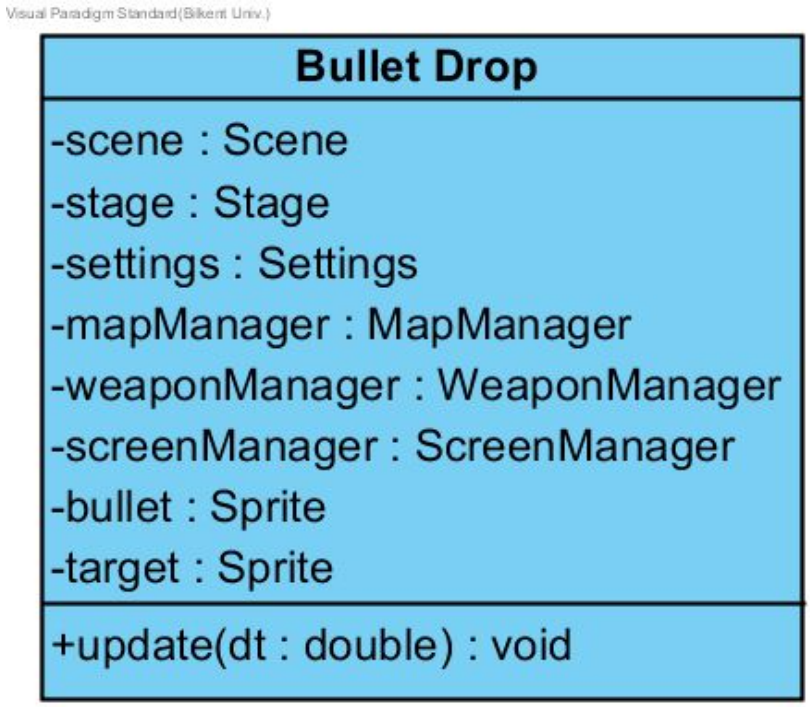
The image that wil show how to play.

Methods

update(dt: double): void

Draws the background image as well as howToPlay picture.

BulletDrop Class



It is the main class of the game which will have InputManager, ScreenManager, weaponManager and MapManager components and will combine them all to play the game. It's logic of our game.

Attributes

scene: Scene

The scene for our game. It will have different screens inside it and it is inside of the stage.We set our scene by screenManager's currentScreen.

stage: Stage

This is the primary stage of our game. It is the main component of the game. We're upgrading our stage in update method.

settings:Settings

This is the instance of our settings class. It's first initialized in here.

mapManager: MapManager

This is the instance of our Map Manager class. It's first initialized in here and will be used in our calculations as well as drawing background images of the maps.

weaponManager: WeaponManager

This is the instance of our Weapon Manager class. It's first initialized in here and will be used in our bullet calculations as well as drawing pictures of the weapons and target.

screenManager: ScreenManager

This is the instance of our Screen Manager class. It's first initialized in here.

bullet: Sprite

This the bullet object in order to calculate it's position based on the affecting objects.

target: Sprite

We use it to calculate whether our bullet hit our target. It's position is being taken by the Weapon Manager

Methods

update() : void

As it is the main ruler class, it calculates elapsed time between each update and triggers screenManager's update method and passses this elapsed value to it.

It's also responsible for main logic in fireScreen. It works like this, once the bullet has been fired, it applies horizantal force to the bullet sprite object based on the value it takes from weaponManager's selectedWeapon's power. Then it takes currentLevel's forces from mapManager as well as their constant about how much it should effect the bullet. Then it checks whether our bullet is in collusion with a force. If it does, it applies vertical force(properties of this force is depending on map). Additonally it also checks whether our bullet hits our target or not. At the end of the this method, it calculates position of the bullet (calculation is actually being done in Sprite class). Then finally, it gives these values to Screen Manager so that it can draw.

InputManager Class

Input Manager class arrange the inputs which are pressed by users.

isKeyPressed(Key e): boolean

This operation checks arrow or space key's position. It means if user press one of them it returns false or true (depends on the implementation).

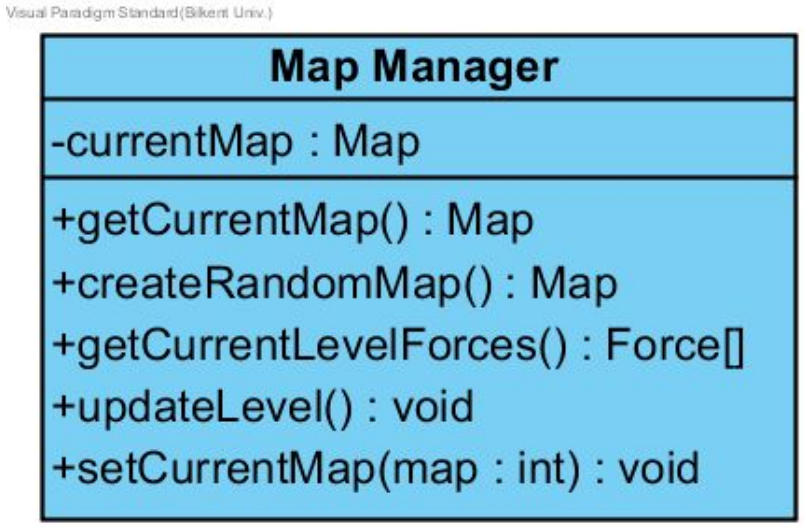
isKeyDown(Key e): boolean

This operation takes the duration of pressing and return a boolean.

isKeyReleased(Key e): boolean

When user stop pressing button, this method invokes the other screen or proccessing managers.

MapManager Class



This class is responsible for managing current map as well as responsible for returning forces in it.

Attributes

currentMap: Map

The variable for the current map object. It holds the current map from 4 different maps.

Methods

getCurrentMap(): Map

It returns the current map object.

createRandomMap(): Map

It calculates a random integer from 1 to number of maps and then sets the current map to that random map.

setCurrentMap(int map): void

It checks ID's of every map and sets current map to that map. If given value does not match any map's ID, it will give error.

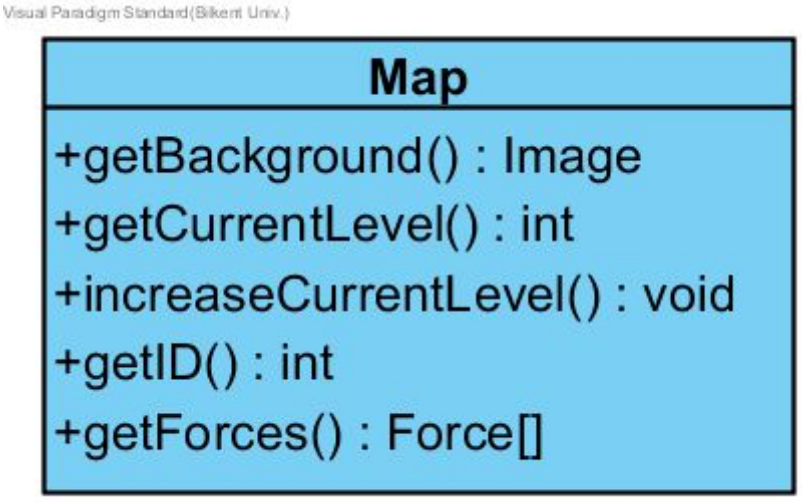
getCurrentLevelForces(): Force[]

It gets the sets of the forces(gravity, wind) according to the level of the current map and then returns that set of forces so that they will be active in the game. It gets all forces from the map object and the puts them into an array

updateLevel(): void

It increases current level(by calling current map's increaseCurrentLevel method) and if it's last level of the current map, it changes current map to next one.

Map Classes(1-2-3-4)



The reason why we have 4 different classes but not have a map array is, in every map we manually set forces' position as well as their individual powers. Aditionally, we may wanted change weight of the map as well. If we would use array to hold all the maps, creating these maps in Map Manager would result in as much as code that would creating it in different class.

getBackground: Image

Returns the bq variable, which is the background image of a specific map.

getCurrentLevel(): int

Returns currentLevel variable for a map that is initialized in the constructor.

increaseCurrentLevel(): void

This method increments the variable currentLevel by one.

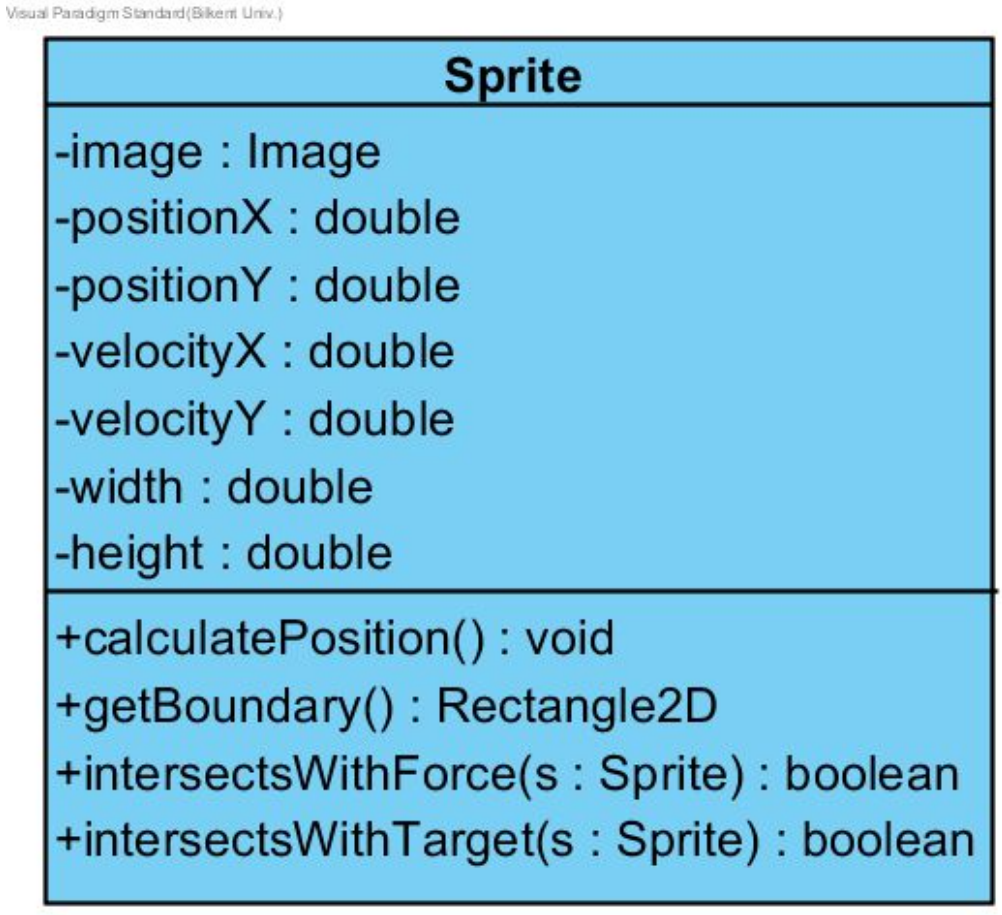
getID(): int

Returns the id variable of that map which is initialized in the constructor.

getForces(): Force[]

This method returns the array of forces for that specific map, the force array defined in the constructor and each map has different set of forces.

Sprite Class



Attributes

image: Image

The image of the object.

positionX: double

The X position of the object.

positionY: double

The Y position of the object.

velocityX: double

The velocity of the object in the horizantal direction.

velocityY: double

The velocity of the object in the vertical direction.

width: double

Width of the object.

height: double

Height of the object.

Methods

calculatePosition(time: double): void

Calculates the position of the the object in a certain elapsed time. It does it by takes the last position of the object and adds results of the multiplication of the velocity and the elapsed time. It does it by for both position X and Y.

getBoundary(): Rectangle2D

It returns a Rectangle2D object based on the position and the height-weight of the objects.

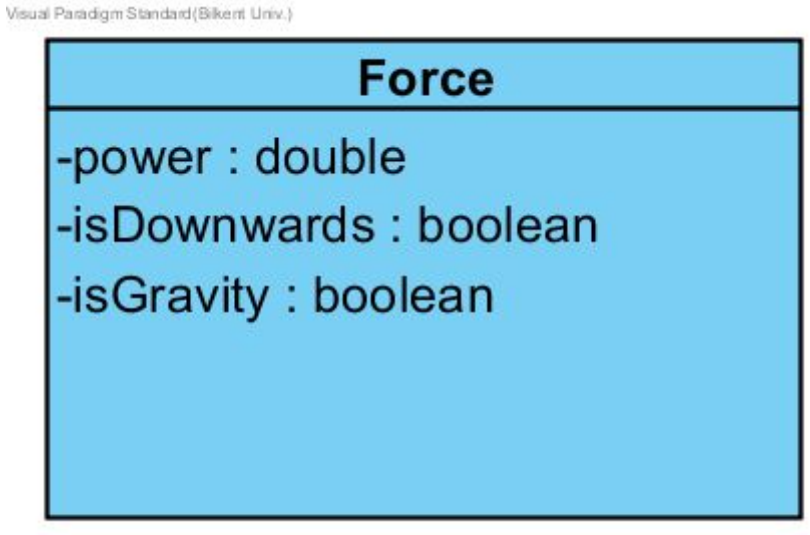
intersectsWithForce(s : Sprite) : boolean

It compares bullet's and force's rectangles and then returns true if minimum x value of force is smaller than the minimum x values of the bullet and the maximum x value of the force is greater than the maximum x value of the bullet, false otherwise.

intersectsWithTarget(s : Sprite) : boolean

This time, it compares target's and bullet's rectangles and returns true if the bullet intersects with target using intersects() method of the Rectangle2D class. The difference between this method and above one is, in this method it also includes height of the object whereas including it in above method results in wrong answer.

Force Class



This class extends Sprite class.

Attributes

power: double

The power of the force, i.e. how will the force affect the bullet, how much it will change the position of the bullet. It is double.

isDownwards: boolean

The boolean variable that tells whether the force is downwards or not.

isGravity: boolean

The boolean variable that tells whether the force is gravity or not.