Project Hellknöw

Group 3

Henrik Sandström

Jonas Lindmark

Carl-Fredrik Sundlöf

Tim Hao Li

Table of Contents

1. Introduction

The purpose of this document is to give an overview of the project Hellknöw v0.01 alpha in detail so that a project group will be able to build the system based on this document. In other words, a project group consists of a project manager, developers, testers, and sponsors. So the intended audience of this document therefore will be all the above. An ideal document of this sort would be that each genre of audience would be able to read their own part of the document and understand exactly what they have to do and not have to read through other parts. For example, the sponsor will be able to see the potential of this project just by looking at section 2. System Overview and probably section 4. Graphical User Interface.

So to give an overview of the scope of this document, it ranges from the overview of the system under design to the slightest details. For example, we will introduce section 2. System Overview shortly which describes the underlying infrastructure of the whole system. Section 3. Design Considerations explains the basic ideas and constraints that we have to take into account during the design process. Section 4. Graphical User Interface starts going into small details on how the system should look. Section 5. Design Detail will be the most crucial read-up for developers because it gives all the class description in the project and also the hierarchy of the classes. Section 6. Functional Test Cases will be important for the testers because it should give how the tests shall be done.

The complementary readings of this document are the API(application programming interface) of Lightweight Java Game Library used for the graphics of the system at *lwjgl.org*, general API for Java at http://java.sun.com/javase/6/docs/api/ .

There are several terminology, acronyms, and abbreviation existing in the document to be clarified in the following,

1. HKP(Hellknöw Protocol)- the protocol used for communication between the client and the server.
2. LWJGL(Lightweight Java Game Library)- the library used for implementing the graphics of this system.
3. Fog of War- a term used to describe the level of ambiguity in situational awareness experienced by participants in military operations. In our implementation this will be a ray of light shooting out from player's eye representing its vision. Everything else that's not within the ray of light will be blurred out except for the contour of the map.
4. IP(Internet Protocol)- the standard protocol that is used in order to communicate with one another on a network.

5. API(Application Programming Interface)- very detailed description of a certain programming language or library that is used for programming.
6. User- the person who uses the system in order to achieve his/her goal of interest.(This is different from a Player)
7. Player- the avatar that the user controls in a game session.
8. Avatar- the representation of the Player in a game session. It is often used interchangeably with Player.
9. PM- Project manager

In short, this document depicts all the details that are needed for the project to be successfully implemented. It is divided into 6 parts including the Introduction**.** Furthermore, It would be a source for future reference if something has gone wrong after completion. In conclusion, this document will not only serve the purpose of the developing process, it should also convince the sponsors to support the project while letting the testers and the project manager understand their fundamental tasks. It will, in addition, take part in the future development of the newer version of Hellknöw.

2. System Overview

2.1 General Description

This project revolves around developing our game called Hellknöw. The game is a one on one shooter set in a 2D environment that aspires to feel like a new and exciting retro game with lots of action.

Game Idea

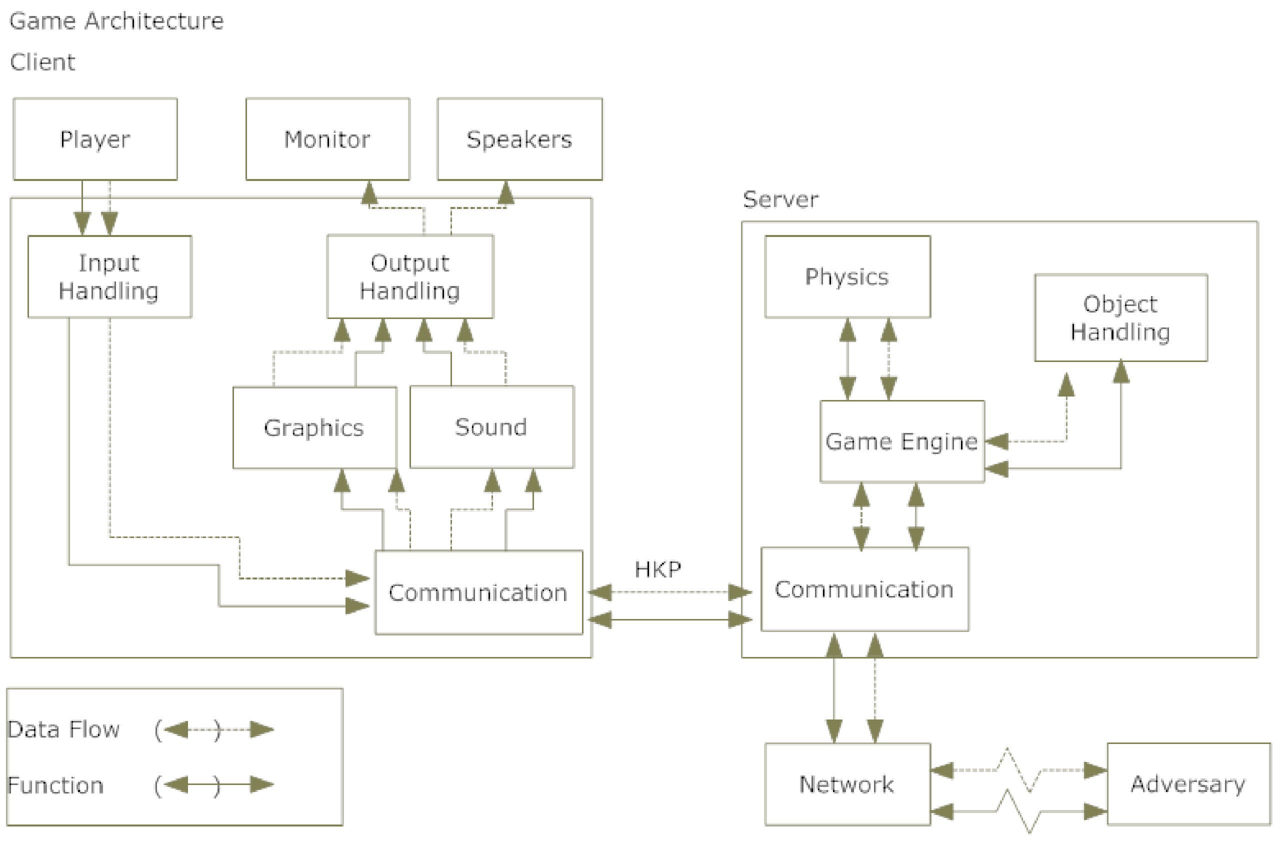
The basic idea is that the two players will move around on the playing stage trying to find each other and eventually vanquish each other using the environment and their weapons provided within the game. The game is set in a strict 2D environment with the players viewed from the side moving in the horizontal plane of the field. There will be all the regular obstacles and items on the course such as ladders and crates to hide behind. Field of view will be very important since we have a Fog of War implemented impeding view of the course forcing the players to move around and explore the field. Because of this view impediment the game will contain many opportunities for stealth and tactical positioning making the game even more interesting.

The arsenal in the game is distributed at random and will be renewed each round forcing the users to adapt to his/her tactical situation upon game start each time, there will be no way of changing the arsenal when the round has started so the players will have to utilize their weaponry as best they can. This system is designed to make the game more casual and exciting since there will be no moment of preparation before each new game session.

System design

The system will be divided into clients and servers, the server part will not be visible to the user and all control of it will go through the client. No matter what mode the player chooses behind the scenes there will always be a server and a client connected to it. The game will run over basic TSP/IP protocols for networking aspects and will require users who want to play with each other to have an established connection supporting these protocols.

2.2 Overall Architecture Description



In the basic architecture we have two subsystems interacting with each other. One is the client side which the user is going to interact directly with. The other will be the server side which handles most of the processes in mathematics and physics.

The server is not actually connected to the Internet or network in the Single Player mode. Implementing this part is actually to make the system more "homogeneous," meaning that the Single Player mode would not differ too much from the Multiplayer because we will have the Multiplayer on different machines so we will have to have servers communicating with each other. In other words, the server component will show its significance in the Multiplayer mode.

Within the client system it consists of 2 major parts that will be visible to the user. One is the Input Handling component and the other one is the Output Handling component. The Input Handling takes in raw inputs from the user and the Output Handling communicates between the Graphics, Sounds Components and the media to the user. In addition, there is the

Communication component which takes in directly from the user inputs and sends the inputs to the server side and also retrieves information from the server side then sends it to the output.

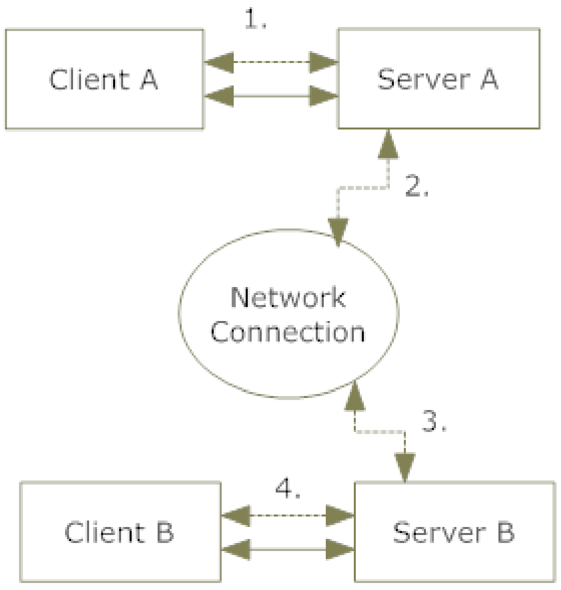
On the other hand, the server takes care of the physics and calculations in the game from the Communication retrieval. After all the calculations it sends the processed back to the Communication component.

So the key medium of the two is the Communication component on each side. It does not process the input other than breaking it down into packets using the HKP(Hellknöw Protocol). Other than that, it also puts the packets together to be interpretable by the system.

To be specific, in the Multiplayer mode we connect the server to the network. One server will be the host and the other will be the one who connects to the host.

2.3 Detailed Architecture

2.3.1 Program flow



1. Interaction between Client A and Server A

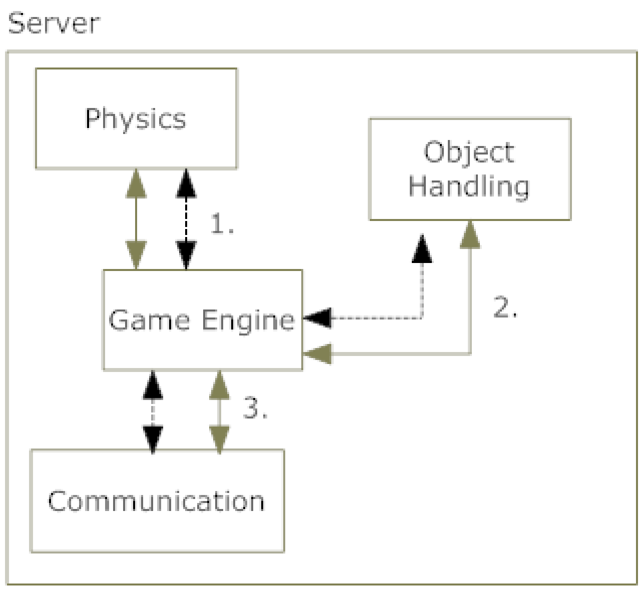
Client sends raw data information to server which the server interprets and responds to.

2. Connection between Server A and the network
Server sends update information to the network. The server receives information from the network and interprets it.

3 Connection between the network and Server B
The server receives information from the network and interprets it. Server sends update information to the network.

4 Interaction between Client B and Server B
Client sends raw data information to server which the server interprets and responds to.

2.3.2 Server flow



1. Interaction between Game Engine and Physics

The game engine uses physics to determine how to update all objects in the game.

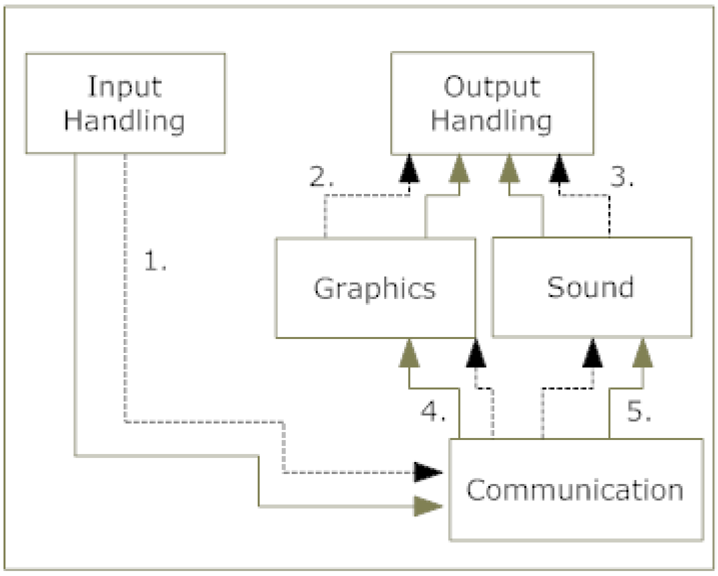
2. Interaction between Game Engine and Object Handling

The game engine uses object handling to handle all objects in the game and keep track of their position and other statuses.

3. Interaction between Game Engine and Communication.

When the Game Engine is done with all the calculations it sends the information to Communication which forwards it to the appropriate place.

2.3.4 Client Flow



1. Interaction between Input Handling and Communication

Input Handling takes information from the User and sends it to Communication for evaluation and further distribution.

2. Interaction between Graphics and Output Handling

Graphics generates the frames that are sent to Output handling so that it can reach the user.

3. Interaction between Sound and Output Handling

Graphics generates the sound effects that are sent to Output handling so that it can reach the user.

4. Interaction between Communication and Graphics

Communication sends the coordinates of all objects to graphics so that it can generate the frames.

5. Interaction between Communication and Sound

Communication tells Sound what sounds to play for the user.

3. Design Considerations

3.1 Assumptions and Dependencies

To run Hellknöw the user will be required to have a computer that runs Windows XP efficiently combined with an installation of the Java Runtime Environment version 6. This means that the computer should have at least 512 MB of ram with a processor of 1.8 GHz or more. The computer should also at least have 1 GB of free hard drive space.

To be able to play the multilayer mode, the computer will need an Ethernet card that supports transfer speeds of at least 0.5 Mbit/second. The user will of course also need and opponent user with the same setup. The users computer also needs to be able to understand the TCP protocol as all communication within the game will run on that protocol.

In the future, we are planning to expand the functionality so that more than 2 players can play the game at once. This will hopefully not change to much of our implementation, because we only need to expand the server side to allow more connections. However, it will have an impact on the game flow, both for connecting and hosting players.

3.2 General Constraints

Hardware

The hardware we can expect our users to have will be limited because we want as many users as possible to able to play the game on their system. This will limit our project mainly in graphical detail and computations behind the scenes. Which means that we will not have any more physics implemented than necessary for the game to function as we want and still not feel inadequate, also the detail of the graphics will have to be taken into account.

Software and interoperability

One great constraint to this project is that we have decided to focus on only one Operating System, Windows XP. Also it is required from the user that they have a recent version of Java installed to be able to play our game.

Memory

Our users will have limited memory capacity and therefore we will have to keep the physical size of the game as low as possible to increase playability. This will affect how big we can make the game, for example how many courses we can have.

Protocols and networking

We will have the game run over Internet using the TCP/IP protocol which naturally means we have to utilize this protocol to be able make our users clients communicate with each other.

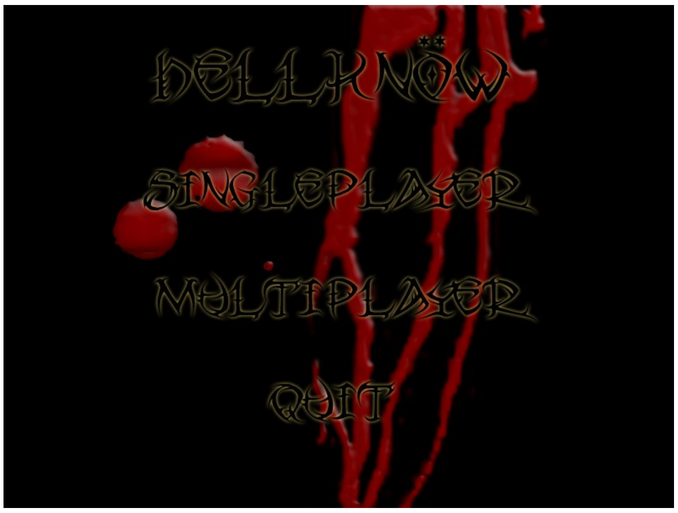
4. Graphical User Interface

4.1 Overview

In Hellknöw the user is able to play a multilpayer game against an adversary or play a single player game to practice his or her skills. To choose between the two different modes the user is presented with a main menu and various sub menus (picture 1). The users must navigate through them to make their choice. When a game has started the user will see the game screen (picture 6). On the game screen the user will see his avatar, which the user is able to move around, and various objects which will hinder the avatars movement. To overcome these obstacles the avatar is able to jump and climb ladders. The avatar is also able to shoot. When playing a multiplayer game there will be an additional avatar representing the adversary. The goal of the game is to kill the adversary's avatar. When the game screen is displayed there will be various hot keys that allows the user to make various options. When the user has finished playing the game the user is able to quit the game by pressing the corresponding hot key.

4.2 Structure of Menus

- Single player
- Multiplayer
  - Show My IP
  - Host Game
  - Join Game
  - Back
- Quit



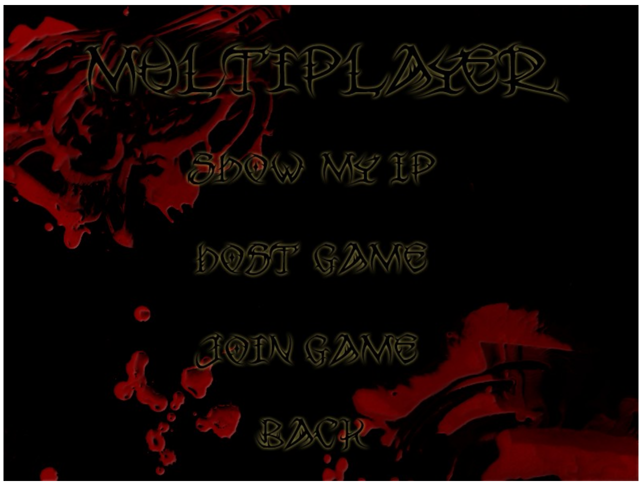
Picture 1: Main Menu

Single player

This is one of the options displayed on the first menu that appears when our application starts. Clicking this option will immediately start a single player game instance. This triggers the method to start single player, which means that a server will be created locally and a game will be initiated. The method will be called startSinglePlayer();.

Multiplayer

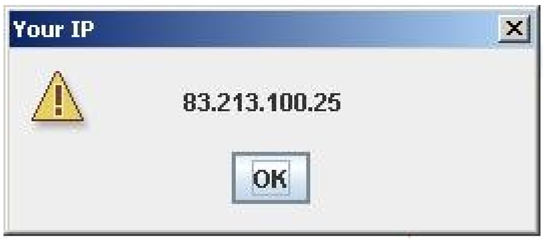
Another one of the initial menu options first displayed when the application starts. Clicking this option will change the current menu into the sub menu for multiplayer containing new options. Within that sub menu there will be a "Back" option that will return the user to the initial menu containing amongst others the multiplayer option. Clicking multiplayer will move the player on to the multiplayer menu triggering showMultiplayerMenu();.



Picture 2: Multiplayer Menu

Show My IP

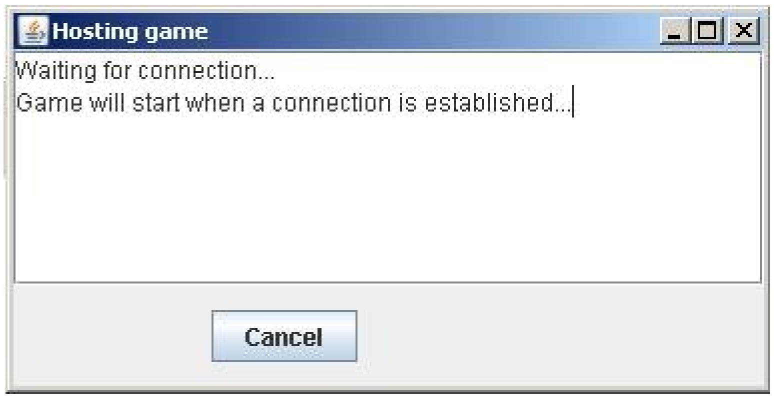
This option is located in the multiplayer sub menu. Clicking this option will trigger a box showing the users IP with an "Ok" button indicating that the user is done looking at the box and does not want to see it any more. The method will be called showIP();.



Picture 3: IP Dialog

Host Game

This option is located in the multiplayer sub menu. Clicking it will show a box indicating that the game has hosted a game and is presently awaiting connection from the adversary. There will be an option for canceling this wait process. When/if the adversary has connected the box will disappear and the actual multiplayer game instance will initiate. The method triggered will be called hostGame();.

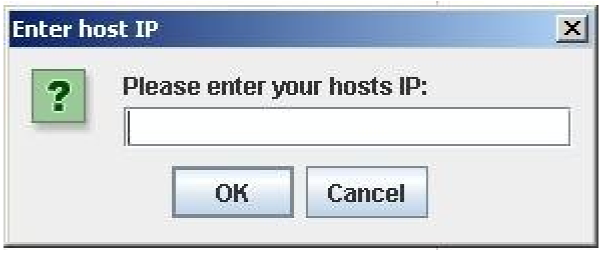


```
Waiting for connection...
Game will start when a connection is established...
```

Picture 4: Hosting Game

Join Game

This option is located in the multiplayer sub menu. Clicking this option will trigger an input box being displayed prompting the user for the hosts IP. When the IP has been inputted the game will hide the box and show a new one saying "Connecting to host", this box will have a cancel option that will return the user to the multiplayer sub menu. When/if the connection to the hos is established the multiplayer game instance will initiate. This method will be called joinGame();.


Enter host IP

Please enter your hosts IP:

Picture 5: Joining Game

Back

This option is located in the multiplayer sub menu. Clicking it will return the user to the initial main menu. This will trigger showMainMenu();.

4.3 Basic Interaction

Game play Mode

Controls and Fields

All the controls will be invisible on the screen. All the controls will be hockeys which users will be instructed to get familiar with. There will be little hints on the side of the screen indicating how to use the basic commands.

Basic commands-

1. **the *esc* key**- to quit the game. Will call System.exit(1);
2. **the *p* key**- to pause the game. Will call pauseGame();
3. **the *r* key**- to resume if in a paused session. Will call resumeGame();
4. **the *F3 key***- to turn the sound on if it's not currently on. Will call soundOn();
5. **the *F12 key***- to turn the sound off if it's on. Will call soundOff();
6. **the F11 key-** to show the hints/hitkey field. Will call showHints();

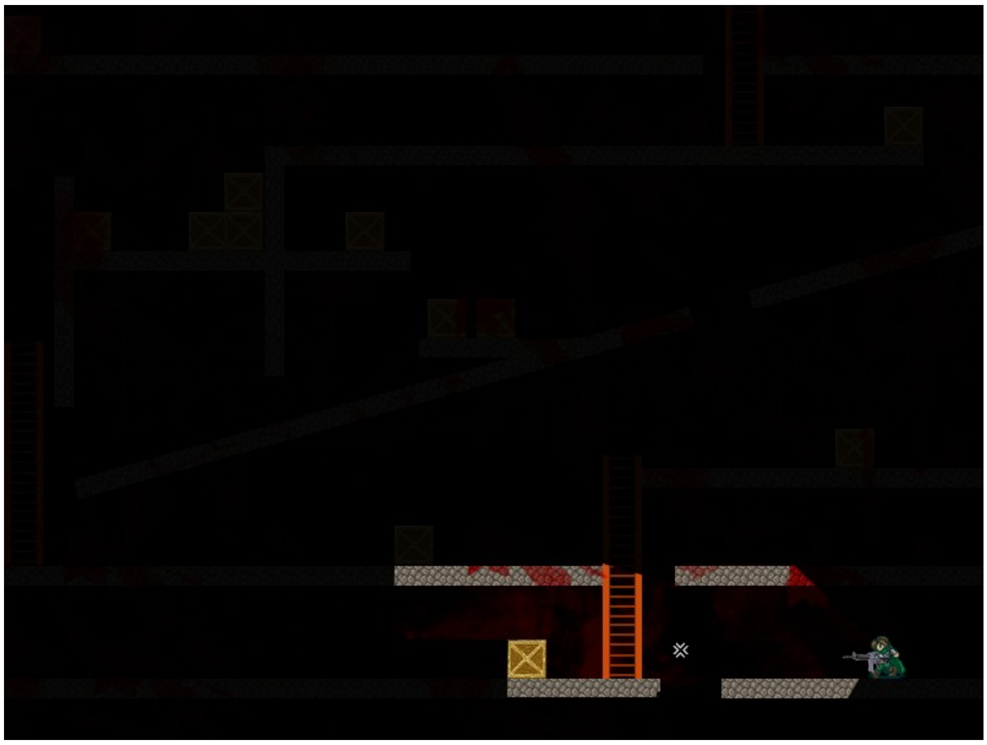
Fields on the screen-

1. **Main field**- it's the main part of the graphics display of the game where you can see the avatars and game environment.

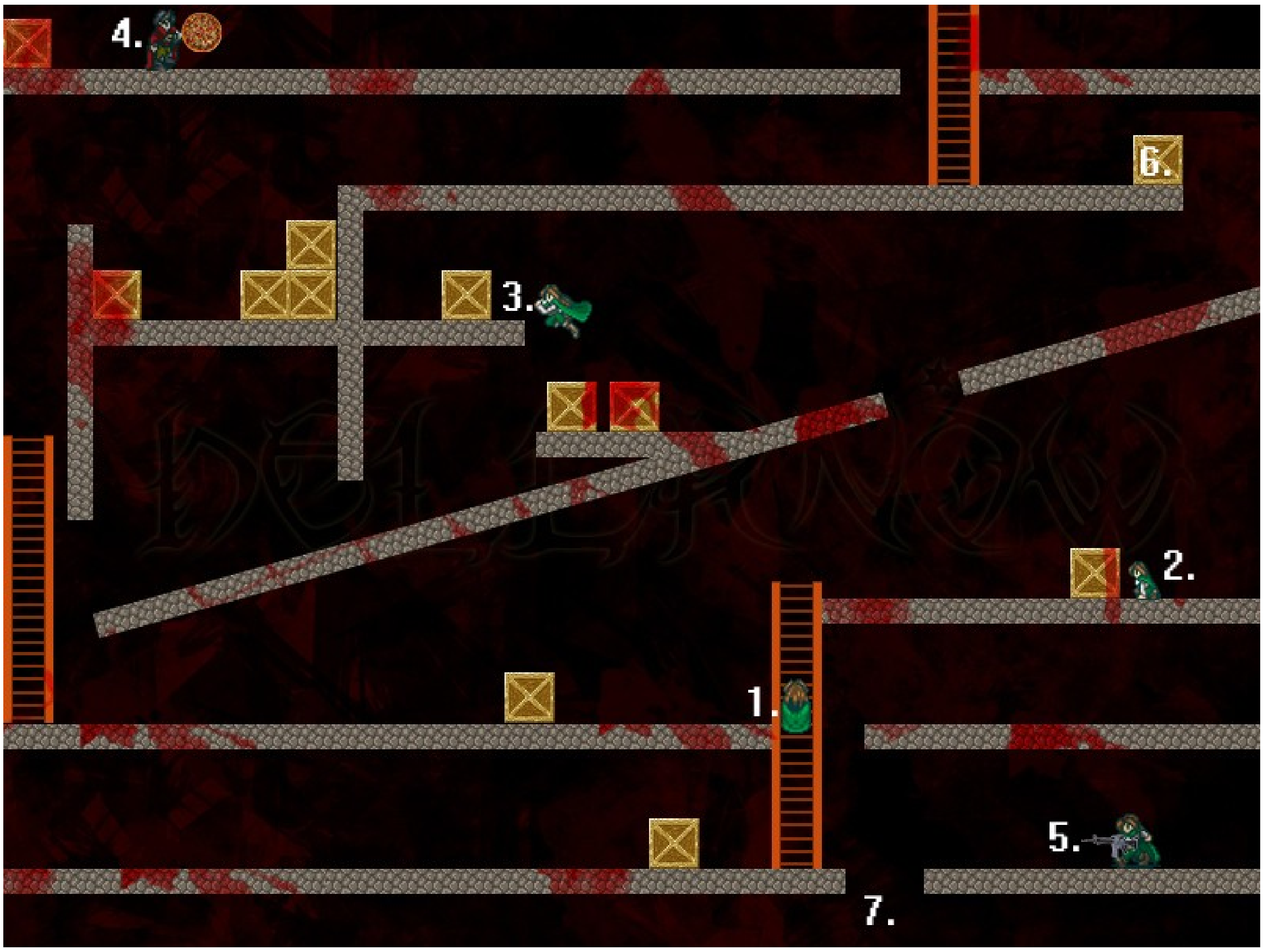
2. **Health field**- this shows the user how much more attack the player can take.
3. **Hitpoints field**- this shows the user how many hitpoints the player has obtained already in this single session. Hitpoint field will be shown when player is hit.
4. **Hints/ Hotkey field**- this shows the hotkeys of basic commands described above. Will be shown when pressing F11.

<u>Controls of the avatar</u>-

1. **the *w* key**- move the avatar up if on a ladder; if not, do nothing. Will call parseCommand("w");.
2. **the *s* key**- move the avatar down if on a ladder; if not, the avatar squats. Will call parseCommand("s");.
3. **the *d* key**- move the avatar to the right on the ground, also while falling. Will call parseCommand("d");.
4. **the *a* key**- move the avatar to the left on the ground, also while falling. Will call parseCommand("a");.
5. **navigation of the mouse**- move the centerline, which falls on the mouse cursor, of fog of war. Will be a mouse event.
6. **left button of the mouse**- shoot the weapon at hand. Will call parseCommand("mouse1");.
7. **the *q* key**- switch weapon. Will call parseCommand("q");.
8. **the space key –** jump. Will call parseCommand("space");.


Picture 6: Game Screen

4.4 References to Functional Requirements



Number in picture	*Reference to functional requirements*
1	**4.1.4.4 Climb ladders** A player climbs a ladder.
2	**4.1.4.3 Crouch** A player crouches.
3	**4.1.4.2 Jump** A player jumps.
4	**4.1.4.5 Pass through objects** A player is not able to pass through objects. Here the player is not able to pass through the ground.

Number in picture	*Reference to functional requirements*
5	**4.1.4.1 Move horizontally** The player moves horizontally.
6	**4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions.** A crate prevents the player to move.
7	**4.1.2.2 Different obstacles have different properties.** A pit is an example of a different obstacle.
8	**4.1.4.7 Use assigned weapons** The player uses his assigned weapon.
9	**4.1.4.6 Changing line of sight** The player changes his line of sight.

5. Design Details

5.1 Class Responsibility Collaborator (CRC) Cards

Class Display	
Responsibilities	**Collaborators**
Displays graphical information in a frame. Graphics	Game Engine Object Images

Class Graphics	
Responsibilities	**Collaborators**
Provides methods for displaying information on the screen.	Display

Class Images	
Responsibilities	**Collaborators**
Store images in the memory.	Display Object

Class Game Engine	
Responsibilities	**Collaborators**
Handles game flow. Contains main game loop.	Display Object Communication Input Sound Physics

Class Object	
Responsibilities	**Collaborators**
Store information about the object itself.	Game Engine

Class Stationary	
Responsibilities	**Collaborators**
Have properties that stationary objects have.	Object

Class Movable	
Responsibilities	**Collaborators**
Have properties that movable objects have.	Object

Class Physics	
Responsibilities	**Collaborators**
Handles the interaction between the objects and the world.	Game Engine

Class Communication	
Responsibilities	**Collaborators**
Communicates with the other server.	Game Engine

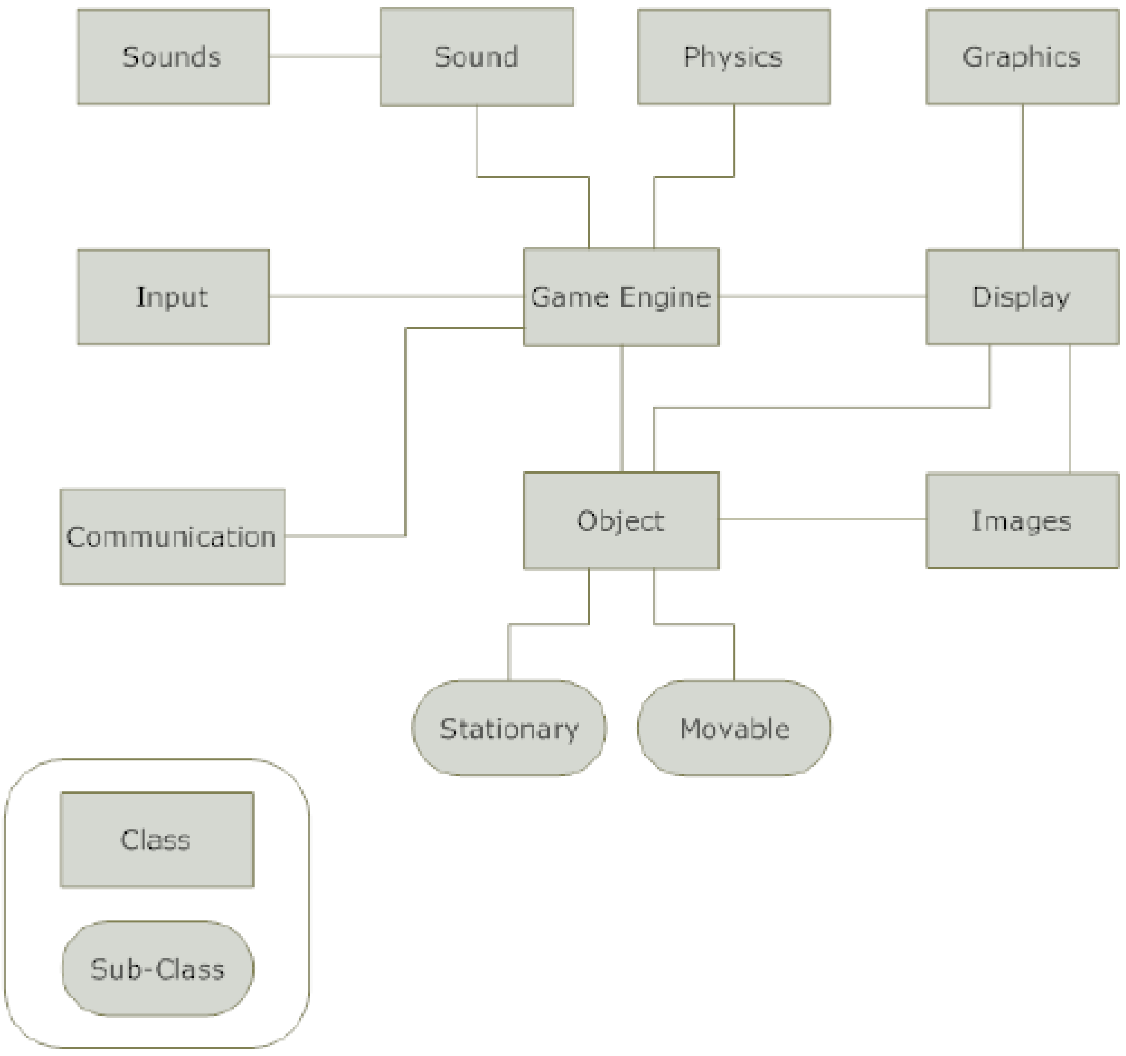
Class Input	
Responsibilities	**Collaborators**
Parse input from the user.	Game Engine

Class Sound	
Responsibilities	**Collaborators**
Provides methods for playing sound in the user's speakers.	Game Engine Sounds

Class Sounds	
Responsibilities	**Collaborators**
Store all sounds in memory.	Sound

5.2 Class Diagram

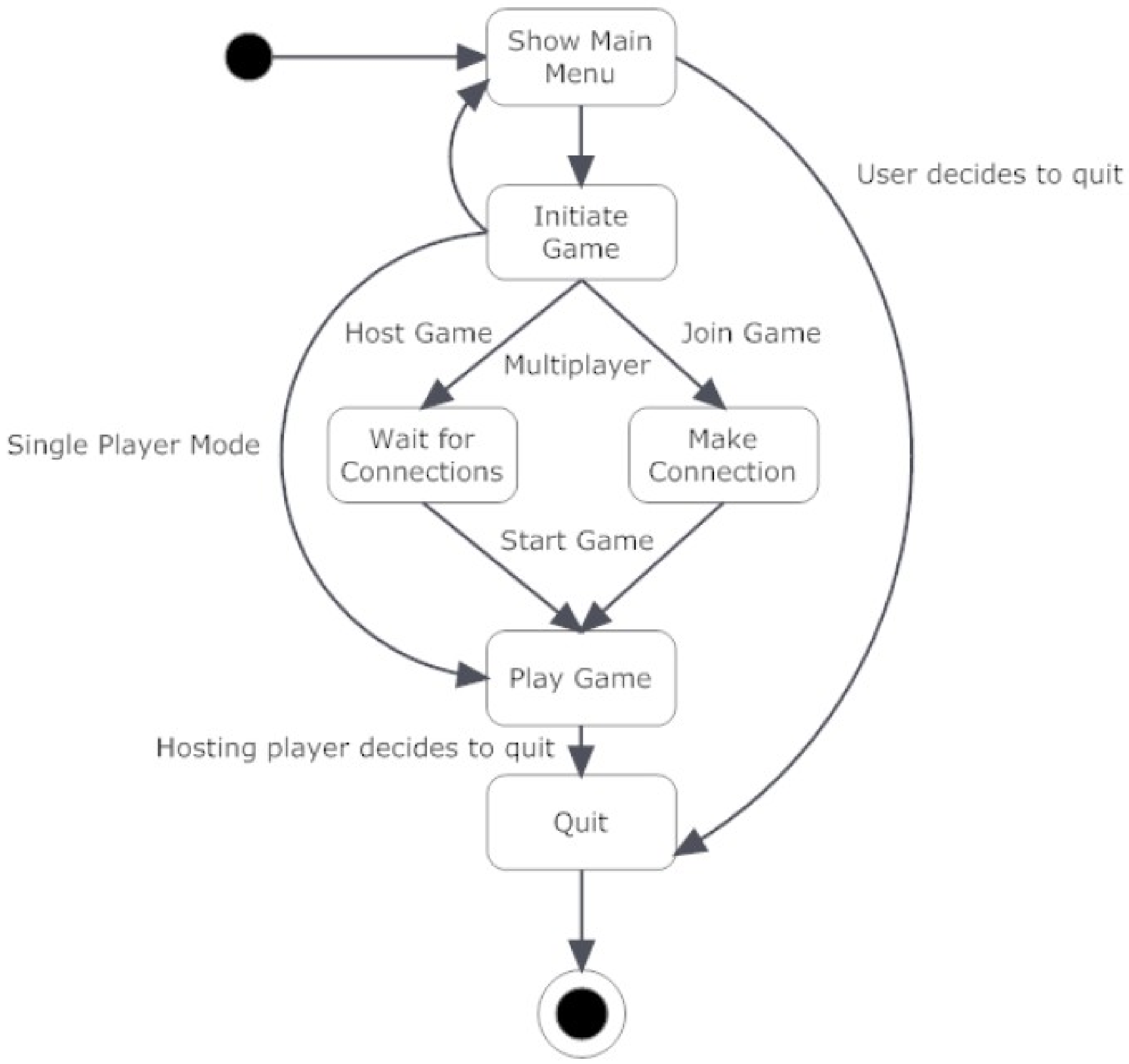
Class Structure for Hellknöw



Dependency is shown as a line between the classes/sub-classes.

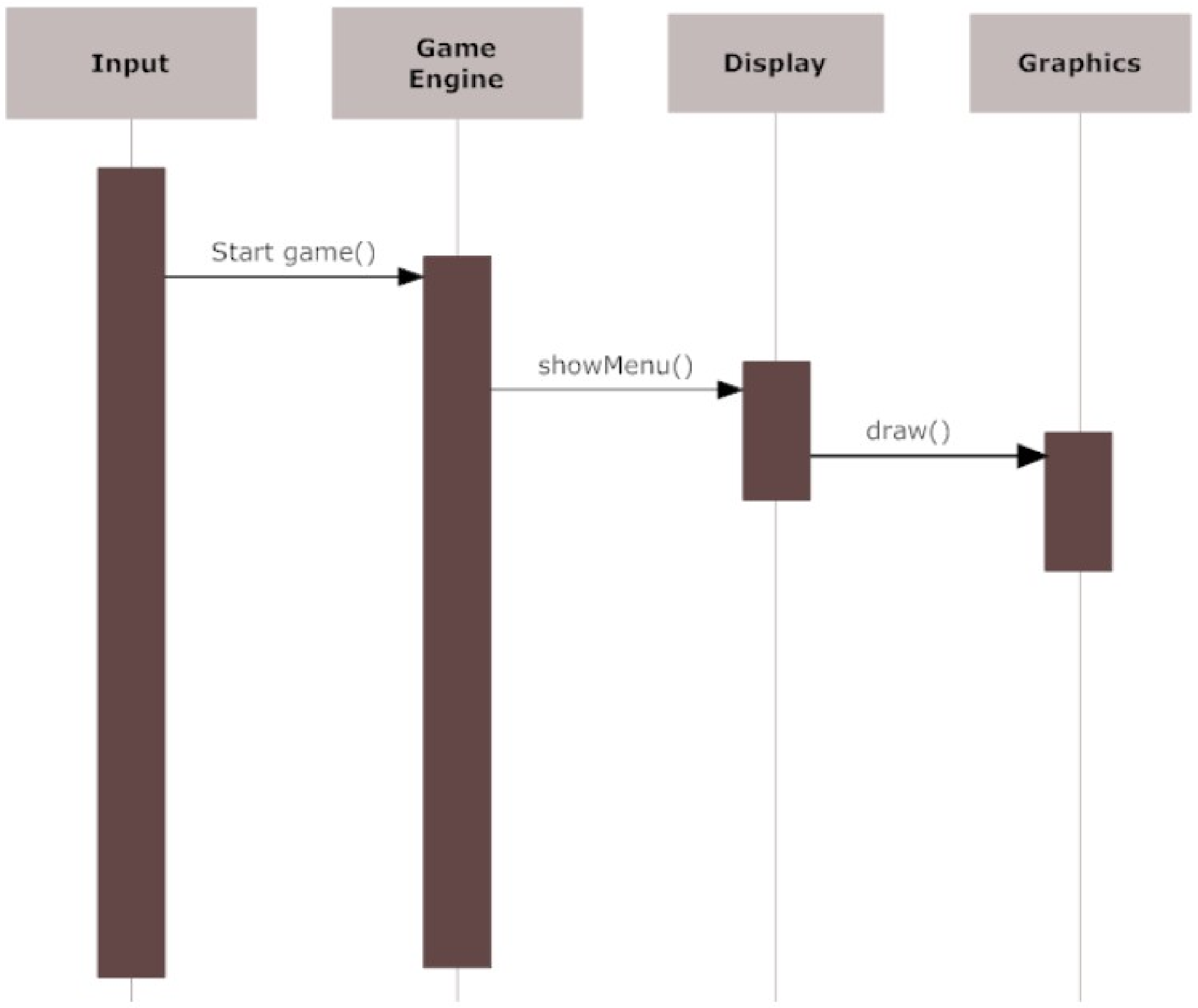
5.3 State Chart

State Chart for a game session

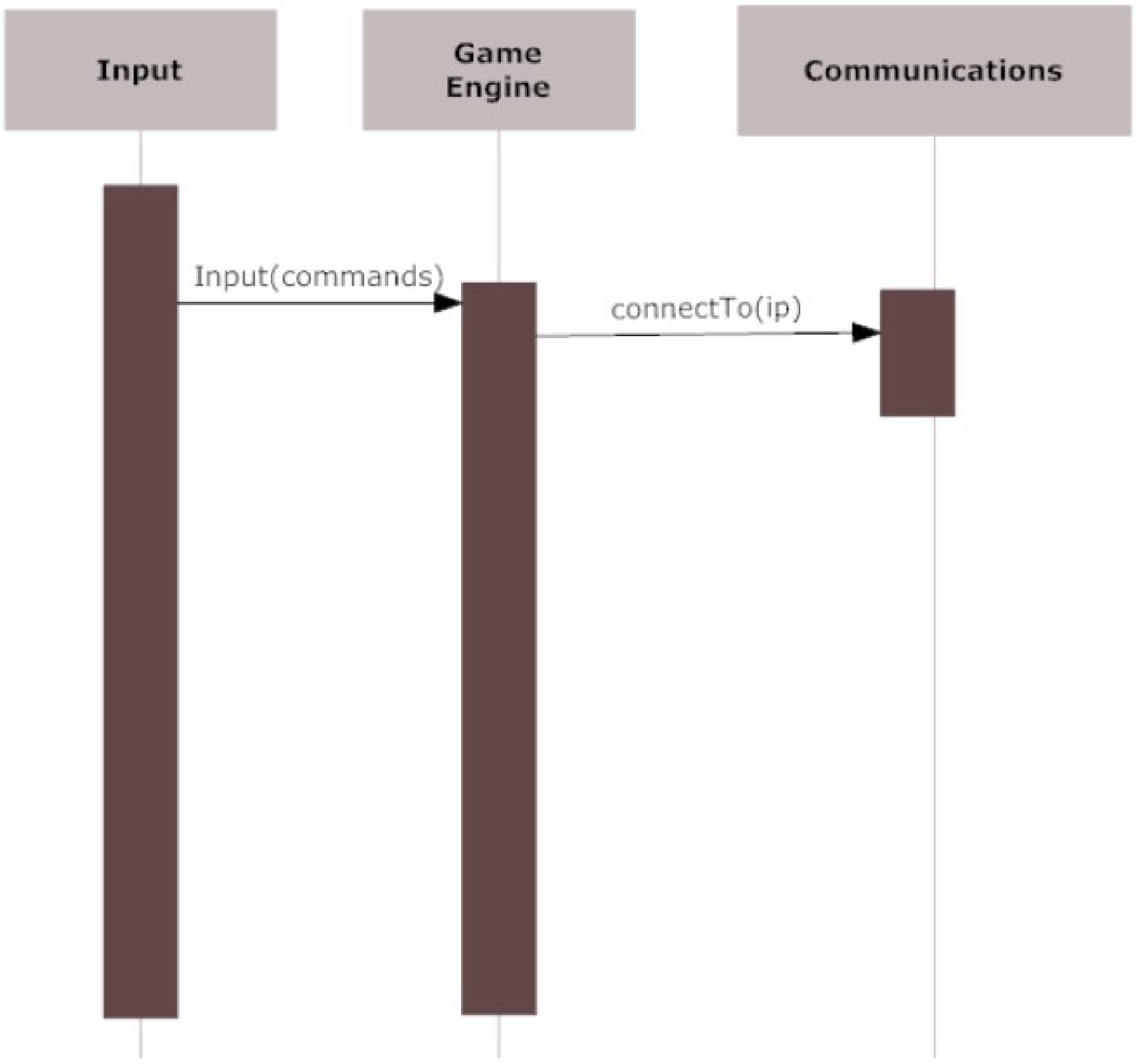


5.4 Interaction Diagrams

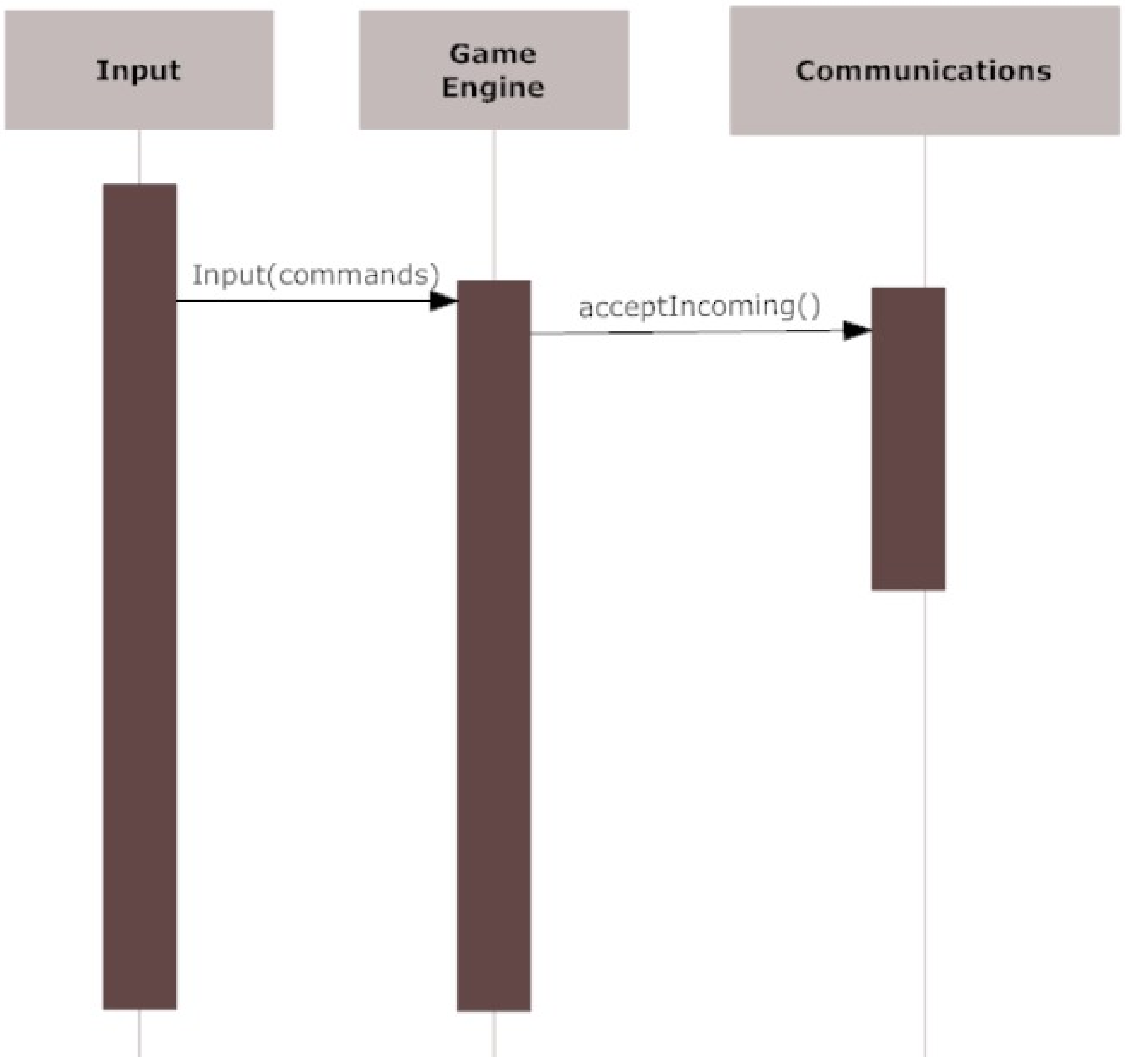
Start the game



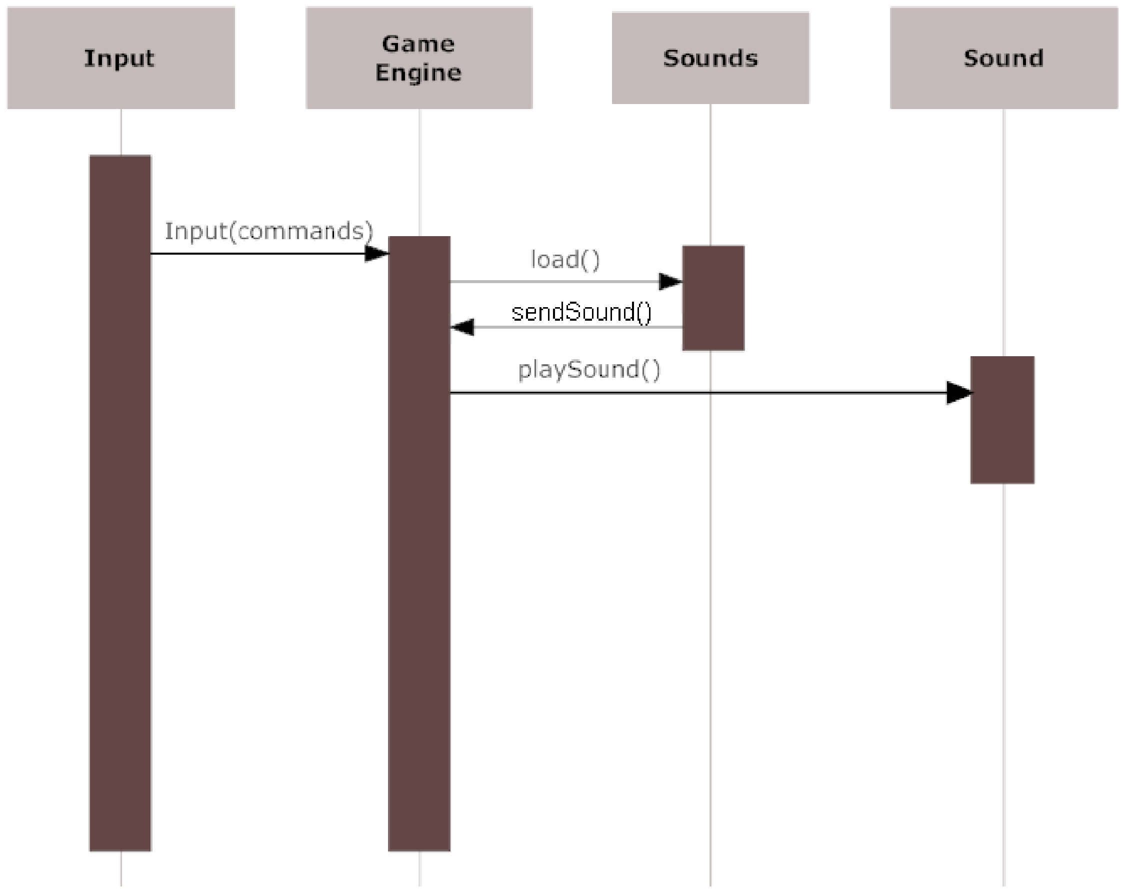
Join a game


Input
Game Engine
Communications
Input(commands)
connectTo(ip)

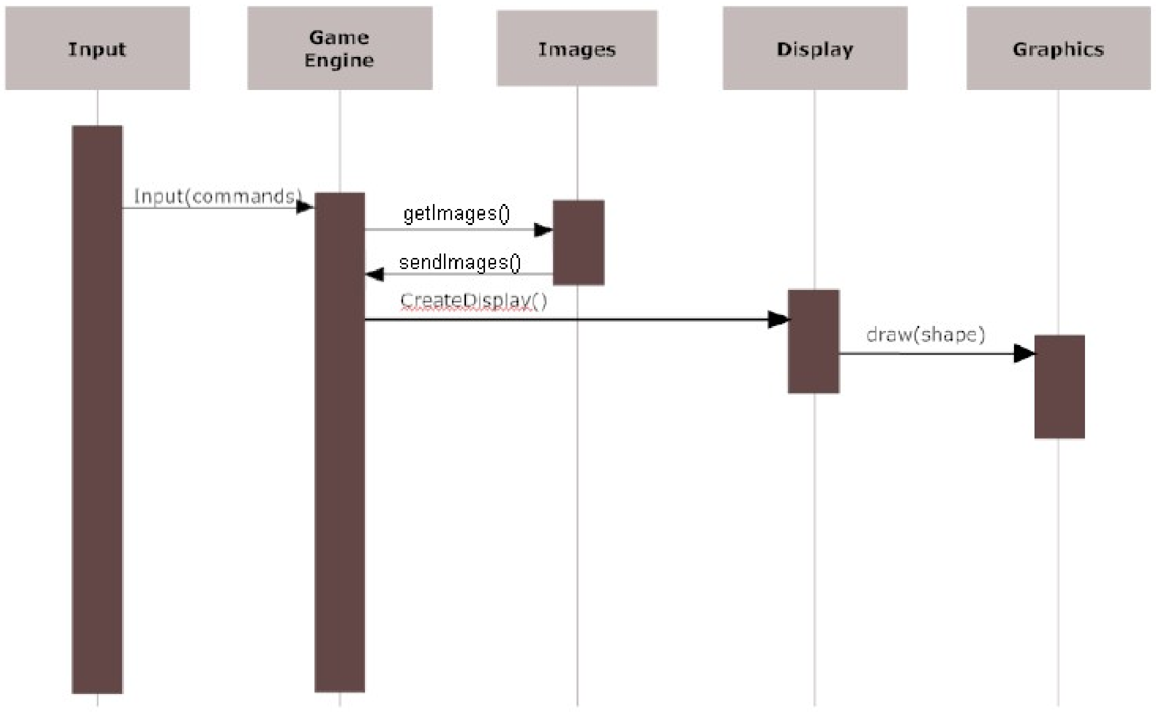
Host a game


Input
Game Engine
Communications
Input(commands)
acceptIncoming()

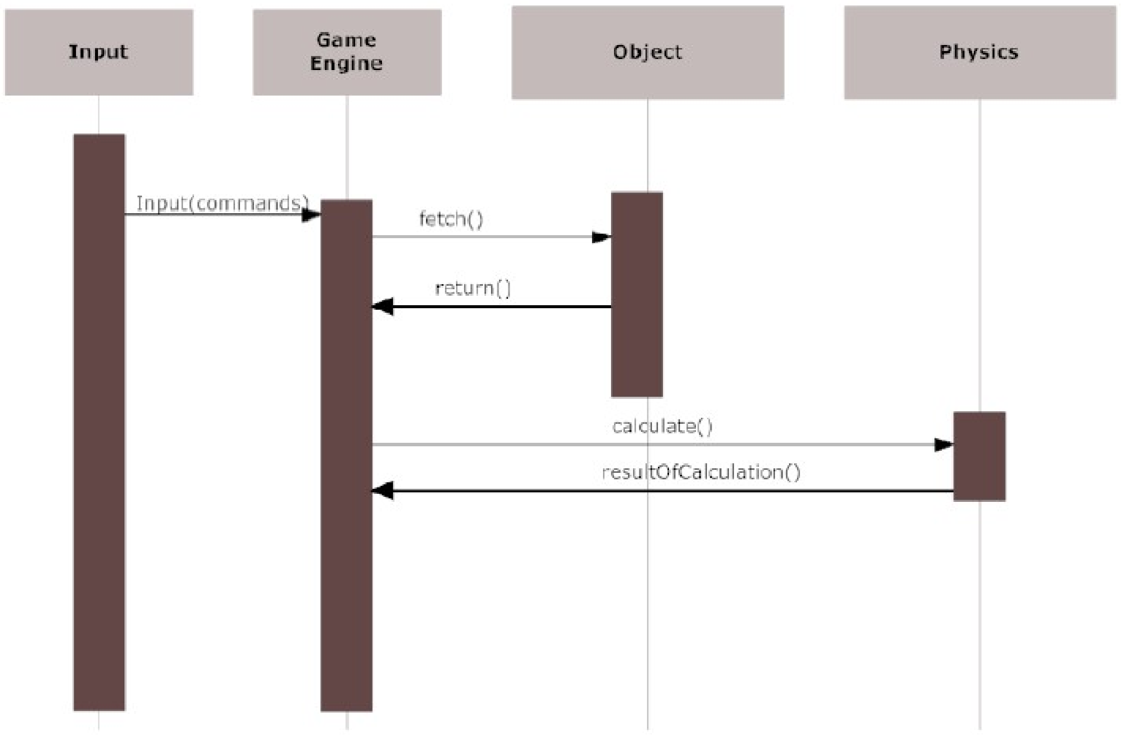
Play sound


Input
Game Engine
Sounds
Sound
Input(commands)
load()
sendSound()
playSound()

Display graphics



Update object



5.5 Detailed Design

Game Engine

Objects and Data Structures:

Object Name: physics

Type: Physics

Description

Will be an object of physics that the game engine uses to update positions and other status of all objects in the game

Funcional Requirements Reference

4.1.1 Objects, 4.1.2 Obstacles, 4.1.3 Weapons, 4.1.4 Players

Object Name: objects

Type: LinkedList<Object> (from java.util)

Description

Will be an LinkedList with all the objects in the game, both stationary and movable.

Funcional Requirements Reference

4.1.1 Objects, 4.1.2 Obstacles, 4.1.3 Weapons, 4.1.4 Players

Object Name: comm

Type: Communication

Description

Will be an object of communication that the game engine uses to establish a connection and then also us it.

Functional Requirements Reference

4.1.5 Network

Object Name: display

Type: Display

Description

Will be an object of display that the game engine uses to create graphical output.

Functional Requirements Reference

4.1.1 Objects, 4.1.2 Obstacles, 4.1.3 Weapons, 4.1.4 Players

Object Name: sound

Type: Sound

Description

Will be an object of sound that the game engine uses to create audible output.

Object Name: input

Type: Input

Description

Will be an object of input that the game engine uses to parse input arguments.

Methods:

Method Name: void start()

Takes no parameters.

Return Value: none

Description

Initiates the first screen that the game is going to display. This is going to be the Main Menu.

Data structure used: None

Pre-conditions: This functions assumes that nothing has yet been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: The main menu is displayed and run() is called.

Called by: Game Engine

Calls: run(), showMainMenu().

Method Name: void run()

Takes no parameters.

Return Value: none

Description

Contains the main game loop from which the game is handled. The game is over when this method is terminated.

Data structure used: None

Pre-conditions: Assumes that the Main Menu is currently displayed to the user.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: The game has exited and is not in the computers memory anymore.

Called by: start().

Calls: Should be able to call any method.

Method Name: void hostGame(int Players)

Players is the number of players that will play the game. Players will be either 1 or 2.

Return Value: none

Description

This method will tell communication to establish a socket for the server(s) to communicate on.

Data structure used: None

Pre-conditions: A game is not running.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: The game is either started if the connection is established or the main menu is displayed again.

Called by: run().

Calls: hostConnection(), showMainMenu().

Method Name: void joinGame(String IP)

IP is a string with the IP of the hosting server. IP will be obtained from the user through a form.

Return Value: none

Description

This method will establish a connection to the hosting user.

Data structure used: None

Pre-conditions: Assumes that the user is connected to a network.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: The players are either connected or the main menu is displayed again.

Called by: run().

Calls: establishConnection(IP), showMainMenu().

Method Name: void performAction(String action)

Action is the name of the action to perform.

Return Value: none

Description

Gets information from display and performs a certain action

Data structure used: None

Pre-conditions: None

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: The action is performed.

Called by: mouseClicked().

Calls: parse

Communication

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine

Description

A reference back to game engine.

Methods:

Method Name: socket establishConnection(String IP)

IP is a string with the IP of the hosting server. IP will be obtained from the user with a form.

Return Value: Socket which is established, or null if a connection could not be established.

Description

This method will connect the user to the hosting user.

Data structure used: None

Pre-conditions: Assumes that it will be possible to establish the connection.

Validity Checks, Errors, and other Anomalous Situations: Checks that the connection can be established.

Post-conditions: The game is either started if the connection is established or the main menu is displayed again.

Called by: joinGame().

Calls: Uses Java sockets to create the connection.

Funcional Requirements Reference

4.1.5 Network

Method Name: socket hostConnection()

Takes no parameters.

Return Value: Socket which is established, or null if a connection could not be established.

Description

This method will create the socket on which to communicate.

Data structure used: None

Pre-conditions: Assumes that it will be possible to establish the connection.

Validity Checks, Errors, and other Anomalous Situations: Checks that the connection can be established.

Post-conditions: The game is either started if the connection is established or the main menu is displayed again.

Called by: joinGame().

Calls: Uses java sockets to create the connection.

Funcional Requirements Reference

4.1.5 Network

Method Name: void send(String data)

data is the information that is to be sent to the other server.

Return Value: none.

Description

Send sends information over the socket.

Data structure used: None

Pre-conditions: Assumes that there is a established connection

Validity Checks, Errors, and other Anomalous Situations: Checks that the connection is established.

Post-conditions: The information has been sent.

Called by: Game engine.

Calls: Uses java sockets to send information.

Method Name: String receive()

Takes no parameters.

Return Value: String with the instruction received.

Description

Receive receives information over the socket.

Data structure used: None

Pre-conditions: Assumes that there is a established connection

Validity Checks, Errors, and other Anomalous Situations: Checks that the connection is established.

Post-conditions: The information has been received.

Called by: Game engine.

Calls: Uses java sockets to receive information.

Input

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine

Description

A reference back to game engine.

Methods:

Method Name: void performEvent(keyEvent input)

Input is the keyEvent to perform.

Return Value: none

Description

Preforms the action bind to the keyEvent specified in input.

Data structure used: None

Pre-conditions: A game is running.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: The game has exited and is not in the computers memory anymore.

Called by: Display.

Calls: None

Sounds

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine

Description

A reference back to game engine.

Funcional Requirements Reference

Methods:

Method Name: wav getSound(String name)

name: The name of the sound.

Return Value: wav sound

sound: The wav mapped to name, if name does not exist, return null.

Description

The function takes a name and searches it's database for a sound with a key identical to name. It then returns that Sound object.

Data structure used: A hashmap with value type Sound and key type String.

Pre-conditions: The database of Sounds is non empty.

Validity Checks, Errors, and other Anomalous Situations: If the inputted name doesn't exist in the database, returns null.

Post-conditions: None.

Called by: The Sound class will call this method when it wants to play a sound associated with an event.

Calls: None.

Method Name: void loadSounds()

Return Value: Nothing.

Description: When this method is invoked, it will attempt to load all sounds with their respective keys from our file structure into our internal hashmap.

Data structure used: A hashmap with value type Sound and key type String.

Pre-conditions: The files needed are located in the correct folder and are of non-zero size.

Validity Checks, Errors, and other Anomalous Situations: If any file that is needed cannot be found or otherwise cannot be opened the method will throw FileNotFoundException.

Post-condition: The internal database will be filled with the sounds described in the filesystem.

Called by: It's own internal constructor.

Calls: None.

Funcional Requirements Reference

Sound

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine
Description
A reference back to game engine.
Funcional Requirements Reference

Object Name: soundOn

Type: boolean
Description:
If this is false no sound will be played, if true it does not affect anything.

Methods:

Method Name: void playSound(String id)

id: The id of the sound that is to be played.
Return Value: Nothing.
Description: Makes a system call to play the sound associated with the id.
Data structure used: None.
Pre-conditions: None.
Validity Checks, Errors, and other Anomalous Situations: If there is no sound associated with the id the method will not play any sound.
Post-condition: None.
Called by: Game Engine.
Calls: getSound(id), system call to play sounds.

Funcional Requirements Reference

Method Name: void setSound(boolean on)

on: The new value of soundOn.

Return Value: Nothing.

Description: Sets whether sound should be played or not.

Data structure used: None.

Pre-conditions: None.

Validity Checks, Errors, and other Anomalous Situations: None.

Post-condition: None.

Called by: Game Engine.

Calls: None.

Images

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine

Description

A reference back to game engine.

Funcional Requirements Reference

Methods:

Method Name: Image getImage(String id)

id: The id of the image.

Return Value: Image img

img: The image mapped to id.

Description

The function will search its database for an image mapped to the key id and return it.
Data structure used: A hashmap with value type Image and key type String.
Pre-conditions: The database of images is non empty.
Validity Checks, Errors, and other Anomalous Situations: If the inputted name doesn't exist in the database, returns null.
Post-conditions: None.
Called by: Display.
Calls: None.

Method Name: void loadImages()

Return Value: Nothing.
Description: When this method is invoked, it will attempt to load all images with their respective keys from our file structure into our internal hashmap.
Data structure used: A hashmap with value type Sound and key type String.
Pre-conditions: The files needed are located in the correct folder and are of non-zero size.
Validity Checks, Errors, and other Anomalous Situations: If any file that is needed cannot be found or otherwise cannot be opened the method will throw FileNotFoundException.
Post-condition: The internal database will be filled with the sounds described in the filesystem.
Called by: It's own internal constructor.
Calls: None.

Graphics

This will include all methods described by the graphics engine we choose. All methods will be called by the Display class.

Display

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine
Description
A reference back to game engine.

Object Name: sensitiveAreas

Type: LinkedList
Description:
Will be used to store the currently active sensitive areas.

Methods:

Method Name: void showHints()

Return Value: Nothing.
Description: This method will print the hints menu on the display.
Data structure used: None.
Pre-conditions: The display has been initiated. A game instance is not active.
Validity Checks, Errors, and other Anomalous Situations: If any of the required images are not found it will throw NullPointerException.
Post-condition: The hints field is displayed.
Called by: Game Engine.
Calls: Various methods in Graphics, getImage from Images.

Method Name: void showMainMenu()

Return Value: Nothing.

Description: This will show the Main Menu and change the mouse sensitive areas in the display correctly.
Data structure used: None.
Pre-conditions: A game instance is not active.
Validity Checks, Errors, and other Anomalous Situations: If any of the required images are not found it will throw NullPointerException.
Post-condition: The Main Menu is displayed.
Called by: Game Engine.
Calls: Various methods in Graphics, getImage from Images.

Method Name: void showIP()

Return Value: Nothing.
Description: This will show a box with the user's IP.
Data structure used: None.
Pre-conditions: A game instance is not active. The user is connected to the Internet.
Validity Checks, Errors, and other Anomalous Situations: If the IP cannot be found or the user doesn not have a connection to the Internet nothing will be displayed.
Post-condition: A box with the IP is displayed.
Called by: Game Engine.
Calls: Uses the Java Net library to find the user's IP.

Method Name: void showMultiplayerMenu()

Return Value: Nothing.
Description: This will show the Multiplayer Menu and change the mouse sensitive areas in the display correctly.
Data structure used: None.
Pre-conditions: A game instance is not active.
Validity Checks, Errors, and other Anomalous Situations: If any of the required images are not found it will throw NullPointerException.
Post-condition: The Mulyiplayer Menu is displayed.
Called by: Game Engine.
Calls: Various methods in Graphics, getImage from Images.

Method Name: void mouseClicked(MouseEvent e)

e: The event generated when the mouse has been clicked within the display.
Return Value: Nothing.
Description: This method is automatically invoked when the mouse has been clicked. The purpose of this method is to identify what action to take depending on where the mouse was when it clicked, and calls that method. This is done by iterating through a list of sensitive areas and checking if the mouse was within that area, if found it will send that action to the Game Engine to be handled.
Data structure used: LinkedList.
Pre-conditions: The display has been initialized.
Validity Checks, Errors, and other Anomalous Situations: None.
Post-condition: The correct action is sent to Game Engine.
Called by: Java MouseListener
Calls: performAction in Game Engine.

Method Name: void keyPressed(KeyEvent e)

e: The event generated when a key has been pressed.
Return Value: Nothing.
Description: This method is automatically invoked when a key has been pressed. The method will send this KeyEvent to Input so that Input can interpret and handle the action.
Data structure used: None.
Pre-conditions: The display has been initialized. The Input pointer ip is not null.
Validity Checks, Errors, and other Anomalous Situations: None.
Post-condition: The KeyEvent has been sent to Input.
Called by: Java KeyListener.
Calls: performEvent in Input.

Object

Objects and Data Structures:

Object Name: gePointer

Type: Game Engine
Description
A reference back to game engine.

Object Name: x

Type: int
Description
The objects x coordinate
Functional Requirements Reference
4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.3 Frictional force
4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions
4.1.4 Players: 4.1.4.1 Move horizontally, 4.1.4.2 Jump, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Object Name: y

Type: int
Description
The objects y coordinate
Functional Requirements Reference
4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.4 Gravitational Force
4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions
4.1.4 Players: 4.1.4.2 Jump, 4.1.4.4 Climb ladders, 4.1.4.3 Crouch, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Object Name: key

Type: String

Description

The key of the sprite picture in the sprite hashmap that represent the object.

Methods:

Method name: getX ()

Return Value: int x

Returns the objects x coordinate.

Description

To know the objects x coordinate you may call the specified objects getX method to receive the x coordinate.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.3 Frictional force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.1 Move horizontally, 4.1.4.2 Jump, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: setX (int x)

The input value is the new x coordinate.

Return Value: None

Description

To set the objects x coordinate you may call the specified objects setX method to set the x coordinate.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: Changes the objects x coordinate.

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.3 Frictional force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.1 Move horizontally, 4.1.4.2 Jump, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: getY ()

Return Value: int y

Returns the objects y coordinate.

Description

To know the objects y coordinate you may call the specified objects getY method to receive the y coordinate.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.4 Gravitational Force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.2 Jump, 4.1.4.4 Climb ladders, 4.1.4.3 Crouch, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: setY (int y)

The input value is the new y coordinate.

Return Value: None

Description

To set the objects y coordinate you may call the specified objects setY method to set the y coordinate.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: Changes the objects y coordinate.

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.4 Gravitational Force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.2 Jump, 4.1.4.4 Climb ladders, 4.1.4.3 Crouch, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: getSpriteKey()

Return Value: String key

Returns the key of the sprite picture in the sprite hashmap that represent the object.

Description

When getting the sprite that represent the object you may call getSpriteKey for the specified object so that you can fetch the correct sprite from the sprite hashmap.

Data structure used: None

Pre-conditions: Assumes that the object has been created and that the sprite hashmap has beeen initialized.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: Display

Calls: None

Movable Object

Objects and Data Structures:

Object Name: vx

Type: int

Description

The objects velocity in x direction

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.3 Frictional force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.1 Move horizontally, 4.1.4.2 Jump, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Object Name: vy

Type: int

Description

The objects velocity in y direction

Functional Requirements Reference
4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.4 Gravitational Force
4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions
4.1.4 Players: 4.1.4.2 Jump, 4.1.4.4 Climb ladders, 4.1.4.3 Crouch, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Object Name: hp

Type: int
Description
The amount of health points that the object has.
Functional Requirements Reference
4.1.4 Player: 4.1.4.9 Lose health points, 4.1.4.9 Losing health points when hit by a weapon

Object Name: dead

Type: boolean
Description
True if the object is dead otherwise false
Functional Requirements Reference
4.1.4 Player: 4.1.4.9 Lose health points, 4.1.4.9 Losing health points when hit by a weapon

Methods:

Method name: getVX ()

Return Value: int vx
Returns the objects velocity in x direction
Description
To know the objects velocity in x direction you may call the specified objects getVX method to receive the objects velocity in x direction.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.3 Frictional force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.1 Move horizontally, 4.1.4.2 Jump, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: setVX (int vx)

The input value is the new velocity in x direction

Return Value: None

Description

To set the objects velocity in x direction you may call the specified objects setVX method to set the objects velocity in x direction.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: Changes the objects velocity in x direction.

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.3 Frictional force

4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions

4.1.4 Players: 4.1.4.1 Move horizontally, 4.1.4.2 Jump, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: getVY ()

Return Value: int vy
Returns the objects velocity in y direction
Description
To know the objects velocity in y direction you may call the specified objects getVY method to receive the objects velocity in y direction.
Data structure used: None
Pre-conditions: Assumes that the object has been created.
Validity Checks, Errors, and other Anomalous Situations: None
Post-conditions: None
Called by: Physics
Calls: None
Functional Requirements Reference
4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.4 Gravitational Force
4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions
4.1.3 Weapons: 4.1.3.3 Gravity affect bullets
4.1.4 Players: 4.1.4.2 Jump, 4.1.4.4 Climb ladders, 4.1.4.3 Crouch, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: setVY (int vy)

The input value is the new velocity in y direction
Return Value: None
Description
To set the objects velocity in y direction you may call the specified objects setVY method to set the objects velocity in y direction.
Data structure used: None
Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None
Post-conditions: Changes the objects velocity in y direction.
Called by: Physics
Calls: None
Functional Requirements Reference
4.1.1 Objects: 4.1.1.1 Changes in movement, 4.1.1.2 Interaction, 4.1.1.4 Gravitational Force
4.1.2 Obstacles: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions
4.13 Weapons: 4.1.3.3 Gravity affect bullets
4.1.4 Players: 4.1.4.2 Jump, 4.1.4.4 Climb ladders, 4.1.4.3 Crouch, 4.1.4.5 Pass through objects, 4.1.4.6 Changing line of sight

Method name: getHP()

Return Value: int hp
Returns the objects health points
Description
To know the objects health points you may call the specified objects getHP method to receive the objects health points.
Data structure used: None
Pre-conditions: Assumes that the object has been created.
Validity Checks, Errors, and other Anomalous Situations: None
Post-conditions: None
Called by: Physics
Calls: None
Functional Requirements Reference
4.1.4 Player: 4.1.4.9 Lose health points, 4.1.4.9 Losing health points when hit by a weapon

Method name: reciveDamage (int damage)

The input value is the damage that the object recives

Return Value: None

Description

The damage is subtracted from the objects health points. If the health points reaches below zero the boolean dead changes from false to true.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: Changes the objects velocity in y direction.

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.4 Player: 4.1.4.9 Lose health points, 4.1.4.9 Losing health points when hit by a weapon

Method name: isDead()

Return Value: boolean dead

Description

Returns the boolean dead which tells whether or not the object is alive or if it has been destroyed and therefore is dead

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: Physics

Calls: None

Functional Requirements Reference

4.1.4 Player: 4.1.4.9 Lose health points, 4.1.4.9 Losing health points when hit by a weapon

Player

Objects and Data Structures:

Object Name: weaponC

Type: Weapon

Description

The close combat weapon.

Functional Requirements Reference

4.1.3 Weapon:

Object Name: weaponR

Type: Weapon

Description

The ranged weapon.

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.2 Cool down, 4.1.3.3 Gravity affect bullets

Methods:

Method name: addWeaponC()

Return Value: None

Description

Adds the close combat weapon to the players weapon arsenal.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: GameEngine

Calls: None

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.2 Cool down, 4.1.3.3 Gravity affect bullets

Method name: addWeaponR()

Return Value: None

Description

Adds the ranged weapon to the players weapon arsenal. And sets it to the equipedWeapon.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: GameEngine

Calls: None

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.2 Cool down, 4.1.3.3 Gravity affect bullets

Method name: currentWeapon ()

Return Value: String equipedWeapon

Returns the string with the name of the currently equipped weapon.

Description

To know the weapon that the player currently has equipped you may call this function to receive the information.

Data structure used: None

Pre-conditions: Assumes that the object has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: Game Engine and the changeWeapon method

Calls: None

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.2 Cool down, 4.1.3.3 Gravity affect bullets

Method name: changeWeapon ()

Return Value: None

Description

Calls currentWeapon to see what weapon the player currently has equipped and changes the equipedWeapon to the other weapon the player has.

Data structure used: None

Pre-conditions: Assumes that the object has been created and that the player has it's two weapons initialized.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: Changes the players weapon.

Called by: GameEninge

Calls: None

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.2 Cool down, 4.1.3.3 Gravity affect bullets

Weapon

Objects and Data Structures:

Object Name: Name

Type: String

Description

The name of the weapon

Object Name: damage

Type: int

Description

The amount of damage that the weapon does

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons

Object Name: range

Type: int

Description

The range of the weapon

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.3 Gravity affect bullets

Methods:

Method name: getName()

Return Value: String name

Description

Returns the name of the weapon

Data structure used: None

Pre-conditions: Assumes that the weapon has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: GameEngine

Calls: None

Method name: getDamage()

Return Value: int damage

Description

Returns the damage of the weapon

Data structure used: None

Pre-conditions: Assumes that the weapon has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

Called by: GameEngine

Calls: None

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.3 Gravity affect bullets

Method name: getRange()

Return Value: int range

Description

Returns the range of the weapon

Data structure used: None

Pre-conditions: Assumes that the weapon has been created.

Validity Checks, Errors, and other Anomalous Situations: None

Post-conditions: None

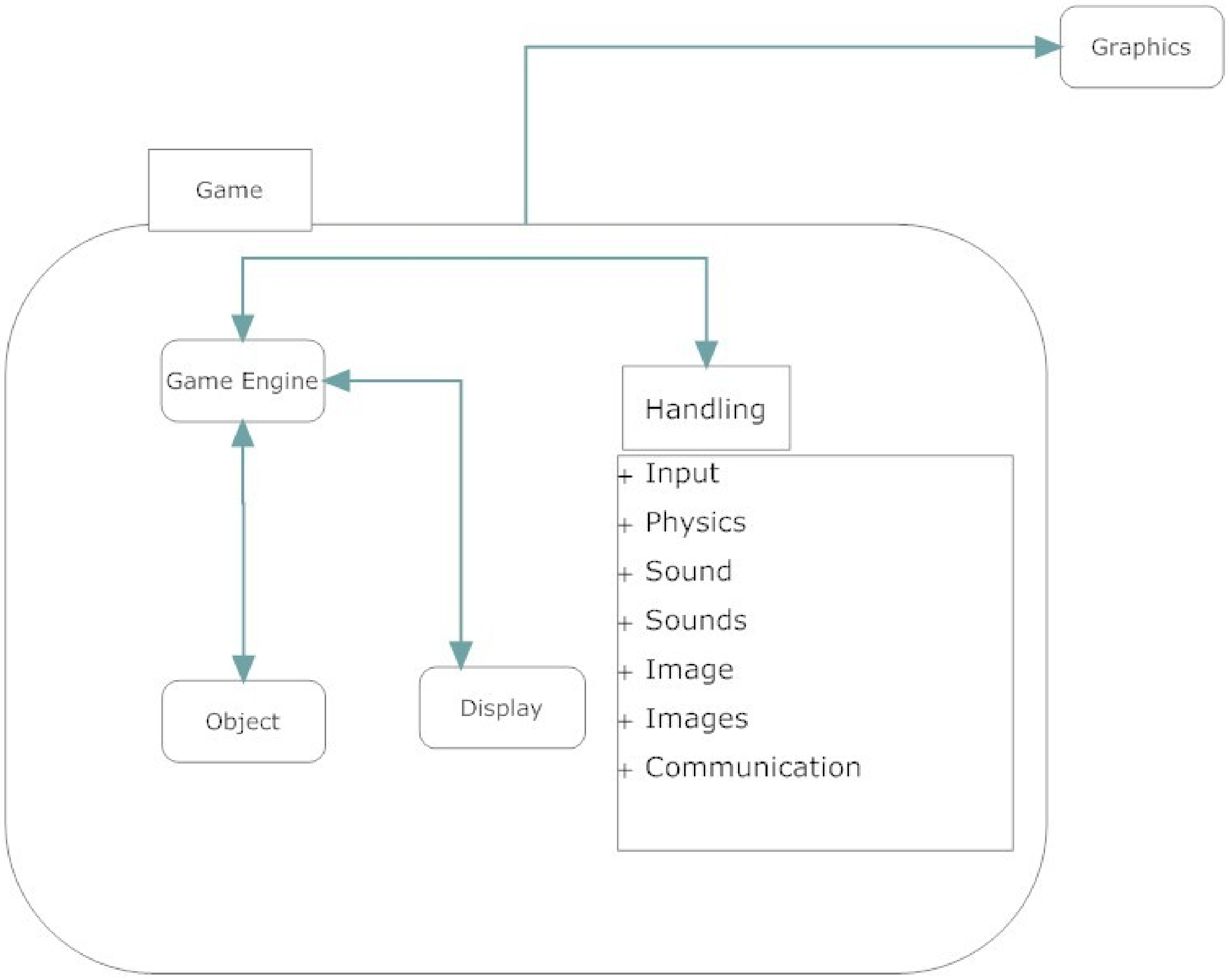
Called by: GameEngine

Calls: None

Functional Requirements Reference

4.1.3 Weapons: 4.1.3.1 Assigned weapons, 4.1.3.3 Gravity affect bullets

5.6 Package Diagram



6. Functional Test Cases

Description: Players should behave in the same way as a movable object when interacting with other movable objects or players.

Reference to the Requirements Document: 4.1.1.1 Changes in movement.

Description of the inputs: A game instance with two players and atleast one movable object. The object needs to have the same ingame weight as the players. Position and collision speed.

Description of the expected output: A change in movement speed and direction should be observed. This change should be the same whether or not two players interact or one player and a movable object.

Step by step procedure:

1. Place a movable object on a horizontal surface.
2. Have one player move and collide with this object.
3. Note by how much the movable object moved as an effect of the collision.
4. Place one player at the exact same position as the movable object.
5. Have the other player collide with the first player at the same speed as before.
6. Note the first players change in movement and compare to the change of the movable object. If they're equal the test is a success, otherwise it has failed.

Description: Physical laws should hold in regard of momentum when two movable objects or players collide.

Reference to the Requirements Document: 4.1.1.2 Interaction.

Description of the inputs: Two movable objects with known weights on a known surface. Together with initial speed.

Description of the expected output: The objects will have the same weight as before but new speed values after their collision.

Step by step procedure:

1. Calculate total momentum before the test using physics formulae P = mv.

2. Have the two objects collide and note the new speeds and directions.

3. Calculate the new momentum and compare to the initial. If equal the test is a success otherwise the test has failed.

Description: All moving objects and players shall be subject to friction.
Reference to the Requirements Document: 4.1.1.3 Frictional force
Description of the inputs: A moving object on top of a surface which is known to have friction.
Description of the expected output: After some time the moving object should have stopped.
Step by step procedure:

1. Set the object in motion on the surface.

2. Observe its change in speed until the object has stopped.

3. If the object does indeed stop within an accepted amount of time, no more than 5 minutes, and has not been subject to any other external force the test is a success otherwise it has failed.

Description: All objects with a non-zero weight must eventually hit the ground.
Reference to the Requirements Document: 4.1.1.4 Gravitational force
Description of the inputs: An object with a positive non-zero weight placed in the air some arbitrary distance above ground level.
Description of the expected output: The object should be stationary on the ground.
Step by step procedure:

1. Observe the object in the air.

2. Observe the change in speed and its direction.

3. If the object has moved downwards and is now stationary on the ground the test is a

success otherwise it has failed.

Description: A stationary object should be able to obstruct a players path.
Reference to the Requirements Document: 4.1.2.1 There will be obstacles in the game preventing the player to move in certain directions.
Description of the inputs: A non-movable object should be placed on surface and a player should be in the game instance.
Description of the expected output: The initial position of the object should not have changed. The player should not have been able to move through the object.
Step by step procedure:

1. Find a non-movable object, like a wall, on a surface and note its position.
2. Have a player try to walk through the object.
3. If the player could not and the initial position of the object has not changed the test is a success otherwise it has failed.

Description: Different obstacles will have different properties.
Reference to the Requirements Document: 4.1.2.2 Different obstacles have different properties.
Description of the inputs: A game instance with one player and one of each type of obstacle.
Description of the expected output: The defined effect of each obstacle should happen when the player interacts with them.
Step by step procedure:

1. Have the player interact with each obstacle on the course and note the effects.
2. Compare these effects with the expected effects and see if they're equal, if they are the test is a success otherwise it has failed.

Description: When the game starts the player(s) should have been assigned two weapons.
Reference to the Requirements Document: 4.1.3.1 Assigned weapons
Description of the inputs: Nothing.
Description of the expected output: A game instance where the players have in their possession two different weapons.
Step by step procedure:

1. Start up a game instance.

2. Try to change weapons using the predefined keys on the players.

3. Check wheter each player in the game instance has two weapons. If they do the test is a success otherwise it has failed.

Description: All weapons must have a cool down preventing them to be fired instantaneously repeatedly.
Reference to the Requirements Document: 4.1.3.2 Cool down.
Description of the inputs: A game instance with a player and an equipped weapon. A specified cool down for the equipped weapon.
Description of the expected output: Nothing.
Step by step procedure:

1. Have the player repeatedly try to fire his weapon.

2. Note the time between the fired shots.

3. Compare this time with the specified cool down for the equipped weapon. Cool-down times can be found in the Requirements Document. If they are equal the test is a success otherwise it has failed.

Description: Test if gravity affects bullets
Reference to the Requirements Document: 4.1.3.3 Gravity affect bullets
Description of the inputs: Pressing the left mouse button to shoot.

Description of the expected output: The bullet is not affected by gravity.

Step by step procedure:

1. Press left mouse button to start shooting.
2. The bullets leave from the weapon and travel across the screen.
3. If the bullets are affected by gravity, i.e. they are slowly decreasing towards the ground, then the test has failed. Otherwise the test is successful.

Description: Test if the avatar is able to move horizontally.

Reference to the Requirements Document: 4.1.4.1 Move horizontally

Description of the inputs: Press either the a key or the d key to move horizontally.

Description of the expected output: The avatar moves horizontal across the screen.

Step by step procedure:

1. Press either the a key or the d key.
2. The avatar reacts to the pressed key.
3. If the avatar moves horizontal in any direction with a fix rate of acceleration and can acquire a maximum speed the test is successful, otherwise the test has failed.

Description: Test if the avatar is able to jump

Reference to the Requirements Document: 4.1.4.2 Jump

Description of the inputs: Pressing the space bar to jump.

Description of the expected output: The avatar moves up into the air.

Step by step procedure:

1. The avatar is standing on flat ground.
2. Press the space bar.
3. The avatar reacts to the pressed key.
4. If the avatar moves upwards into the air the test is successful, otherwise the test has failed.

Description: Test if the avatar is able to crouch.

Reference to the Requirements Document: 4.1.4.3 Crouch

Description of the inputs: Pressing the a or the d key to move horizontally and the s key to crouch.

Description of the expected output: The avatar crouches.

Step by step procedure:

1. The avatar is standing on flat ground.
2. Move the avatar horizontally and take notice of the speed which the avatar is moving with.
3. Press the s key.
4. The avatar reacts to the pressed key.
5. If the avatar crouches the first part of the test is successful, otherwise the first part of the test has failed.
6. Move the avatar horizontally.
7. If the speed which the avatar moves with has decreased, compared with the speed the avatar moved with in step 2, the test is successful. Otherwise the test has failed.

Description: Test if the avatar is able to climb ladders.

Reference to the Requirements Document: 4.1.4.4 Climb ladders

Description of the inputs: Pressing the w key to move upwards a ladder and the s key to move downwards on the ladder.

Description of the expected output: The avatar moves vertically upwards and downwards on the ladder.

Step by step procedure:

1. Move to avatar so that it stands beneath the ladder.
2. Press the w key to move upwards on the ladder.
3. Stop moving the avatar when he reaches the top of the ladder.
4. Press the s key to move the player downward.
5. If the avatar made it to the top of the ladder and down again the test is successful otherwise it failed.

Description: Test so that the avatar is not able to pass through objects

Reference to the Requirements Document: 4.1.4.5 Pass through objects.

Description of the inputs: Pressing either the a key or the d key to move the avatar

horizontally.
Description of the expected output: The avatar is not able to pass through the object.
Step by step procedure:

1. Move the avatar so that it is standing in front of a movable object, e.g. a crate.
2. Press either the a key or the d key so that the avatar moves in the direction of the crate.
3. If the avatar is not able to pass through the object the test is successful, otherwise it has failed.

Description: To be able to explore the map and find his/her adversary, the player needs to change his line of sight and thus chaining his field of view
Reference to the Requirements Document: 4.1.4.6 Changing line of sight
Description of the inputs: mouse navigation
Description of the expected output: player successfully changes its line of sight
Step by step procedure:

1. Start a singleplayer session
2. Use the mouse to navigate the line of sight of the avatar
3. Check if the line of sight changes according to the position of the mouse cursor

Description: The player must be able to use all weapons assigned to him for the game to be functional.
Reference to the Requirements Document: 4.1.4.7 Use assigned weapons
Description of the inputs: keyboard input and mouse left button
Description of the expected output: player successfully uses each of its weapons
Step by step procedure:

1. Start a singleplayer session
2. Use the q button to switch between weapons
3. Press the left button on the mouse and see if each of them fires properly

Description: The player must be able to switch between weapons

Reference to the Requirements Document: 4.1.4.8 Switch weapons
Description of the inputs: q button on the keyboard
Description of the expected output: player successfully switches from one weapon to another
Step by step procedure:

1. start a singleplayer session
2. Use the q button to switch between weapons
3. See if it successfully loads the other weapon when q button is pressed.

Description: A player must lose health points when hit by movable objects.
Reference to the Requirements Document: 4.1.4.9 Lose health points
Description of the inputs: mouse button and control buttons on the keyboard
Description of the expected output: player successfully lose health points when hit by a moving object
Step by step procedure:

1. Start a multiplayer session
2. Let player one acquire a movable object.
3. Player one throws an moving object by using the mouse button and other keys on the keyboard.
4. Place the other player on the path of the moving object
5. See if the player loses health when hit by the object

Description: A player must lose health points when hit by weapon's effects
Reference to the Requirements Document: 4.1.4.9 Losing health points when hit by a weapon
Description of the inputs: mouse button and control buttons on the keyboard
Description of the expected output: player successfully lose health points when hit by an

effect
Step by step procedure:

1. Start a multiplayer session
2. Let player one acquire a weapon
3. Player one fires a weapon by using the mouse button and other keys on the keyboard
4. Place the other player on the path of the effects of the weapon
5. See if the player loses health when hit by the effects

Description: The players must be able to know if connectivity is the fault if their program breaks up.
Reference to the Requirements Document: 4.1.5.1 Upon loss of connection during a game players will be notified.
Description of the inputs: none
Description of the expected output: a notification stating that connections has been lost
Step by step procedure:

1. Start a multiplayer session
2. Disconnect the network (depending on what sort of connection you use. If 1) wireless, then turn off the wireless function

   2) Ethernet, pull off the wire

   3) Internet, disconnect the connection physically or by software)
3. See if a notification shows up

Description: The minimum requirements to run the game shall be a computer with Windows XP, 1.6 GHz processor, 512 MB RAM and a graphic card that supports OpenGL (Open Graphics Library)
Reference to the Requirements Document: 4.2.1.1 Minimum requirements
Description of the inputs: none

Description of the expected output: The game successfully runs

Step by step procedure:

1. Run the game on a computer with the specification described
2. See if everything runs smoothly

Description: The size of the game shall not be greater than 100 MB.

Reference to the Requirements Document: 4.2.1.2 Size of the game

Description of the inputs: none

Description of the expected output: size of the game being less than 100 MB.

Step by step procedure:

1. Fully install the game on a computer
2. Check the size of the game on the hard drive
3. The size should not exceed 100 MB.

Description: The game shall update the frames with a frequency of at least 20 frames per second

Reference to the Requirements Document: 4.2.1.3 Frame rate

Description of the inputs: none

Description of the expected output: The frame rate is right

Step by step procedure:

1. Start a game session
2. Use Frame Rate Tester implemented by our group to show the frame rate of the game
3. It shall be the number specified above.