OGO 3.1 spring 2009

Design document

Computer Science, TU/e

Eindhoven, May 12, 2009

Rev: 231

Group 2

Etienne van Delden, 0618959

Edin Dudojevic, 0608206

Jeroen Habraken, 0586866

Neal van den Eertwegh, 0610024

Stef Louwers, 0590864

Leroy Bakker, 0617167

Anson van Rooij, 0596312

Contents

1 Introduction

In this phase of the project the actual program has been designed according to the specification made earlier. Modules, classes, their relations, their attributes and their methods have been designed, a protocol for the communication between program instances has been established, and an interface has been designed. The work has been divided between the group members.

This is a more technical description of the complex parts of our game. For a general description of our game, see the Specification Document.

Our game will be called Bomb3D.

1.1 The story so far

The year is 2176 and the darkest predictions about our future have come true. Humankind has grown far beyond the ability to sustain itself with earth's limited resources, and shortages are everywhere. Robots have taken over most jobs, virtual realities prevent the rich from facing the problems of the poor, and many biotechnology nightmares have come true. In this dystopian world, most of the earth's population lives in sprawling, metropolitan cities with ineffective, corrupt law enforcement, which have come under the control of all kinds of criminal organisations. The poor jobless masses have few career options, most of them illegal, and drug trafficking, muggings and violent gang wars are only a few of their problems. Because of this, the prison population has exploded, and people are desperate and unruly. The governments' solution is entertainment. Cruel entertainment.

All over the world, tournaments are held, in which prisoners are pitted against insane thrill seekers, professional gladiators, and unfortunates with powerful enemies, to serve as entertainment for the repressed masses. In these tournaments, men and women fight each other with various types of dangerous bombs and weapons provided by their patrons, plus whatever is to be found in the arena. They must use the lay-out of the battlefield and the weapons available in ever more clever ways to have a shot at freedom, fame and fortune. The object of the tournaments is simple:

it is to kill . . . or be killed.

2 Design decisions and assumptions

This section consists of an incomplete list of notable design decisions and assumptions, incomplete because we are probably unaware of certain decisions and assumptions. This is no problem as the list is not meant to be complete; creating it is merely a way of noting to our future selves that we have agreed on certain issues. For more design decisions, see the specification document, particularly the sections "gameplay", "interface" and "winning the game".

2.1 Design decisions

- The game is played in "tournaments" that consist of multiple rounds, called "games". The total score over all rounds in the tournament determines a player's ranking in the tournament, with players getting one point for winning, none for losing (not winning).
- Users can enter orders continuously. Programs act in turns, tens of times per second. When they act, they read their own user's input buffer queue, add it to the list of commands they got from peer programs, update the game's state according to this list, and send their user's input over the network to its peer programs. By the time their next turn comes around, each peer program will have sent its user's input to the program, so that it has a new list of commands to which to add its own user's commands.
- A level is rectangular and finite in size. There is a border around each level, that prevents a player from going beyond the level. Users can pick one of several different map files or make their own, to play in different levels.
- The assignment's intentionally vague food-gathering requirement will be incorporated into our program by the existence of "power-ups" in the level. Power-ups are an integral part of the game, that a player must utilise to gain the upper hand. A power-up appears whenever a "soft block" is destroyed, and confers a bonus to the player which walks over it first; either the player gets more bombs at his or her disposal, or these bombs become more powerful, or some special bonus is conferred. Which bonuses are available depends on the tournament's rules, as set by its creator.
- The winner of a round is the last player surviving. To avoid overly long matches, a tournament's creator can choose a maximum time the round will take, after this time has passed the game is decided by the number of power-ups each player has gathered over the course of the game. If that amount is also equal between multiple players, the game has multiple winners. This, " Bomberman", is the game as it works according to the standard ruleset, there is support for the creator of a tournament to change these rules.
- "Bomberman" is the ruleset we will focus on, though the tournament rules should allow for enough variation (for example: when does the round end?, how are points awarded?, which maps are used, which power-ups are available?, player stats?, players with different stats?) to allow for the playing of different kinds of games without fundamental changes.

- A consequence of the previous decision is that targeting and shooting should already be possible although it is not necessary for the "Bomberman" ruleset.
- "Chaining" is a problem that needed te be decided upon before designing the class diagrams. When a bomb explodes, it creates a blast radius. Everything in that radius; players, power-ups and soft blocks are destroyed. When a bomb belonging to a player is in that blast radius, that bomb also explodes. Chaining can be used strategically to make use of another player's bombs. We decided that whoever starts a chain "owns" the blast radius. Thus, when the bomb of player A explodes and chains with a bomb of player B, player A owns the blast radius. All players within the two blast radius' are killed and player A gets the score.

2.2 Assumptions

Our assumptions are mainly of a technological nature: we are assuming that the network will be able to handle the traffic we are trying to get across it, and allow each program to act tens of times per second. We know from experience that this is probably a safe bet (given the amount of data that modern computer games manage to pass over the internet) as long as the data is efficiently encoded.

We are not assuming anything about the operating system of the users. We will attempt to have the program work under Windows (XP), Mac OS X and Linux.

3 Program modules

3.1 The overall design

The program consists of a framework running code from the following five modules, in four different threads:

1. The Input module. During a round, it gets input from the player through the mouse and keyboard and puts that input into a queue (2) from which the engine module will read.

2. The Multicast module. It initializes games. During a round, it receives data from the network (0), puts that data in a queue (1) for the engine module, waits for the engine module to fill a different queue, and then sends the received data over the network (4).

3. The Discover module. This module connects to the network initially. It sends a signal over the network announcing the existence of the game to new, as of yet unconnected, programs or peers.

4. The Output module. During a round gets input from the engine module through a queue, then shows the new situation to the player through, among others, the 3D graphics and the minimap. This module runs in the same thread as the Engine module.

5. The Engine module. During a round, it gets input from the input module (2) and the multicast module (1) through two queues, updates the state of the game, puts the updated state data into a queue of the network module (3), and into a queue for the graphics/sounds module (4). It is the engine that handles the mathematical model of the tournament and everything needed for playing the game.

The modules go together as shown in Figure 1. In the next section, we will discuss how the separate modules are designed.

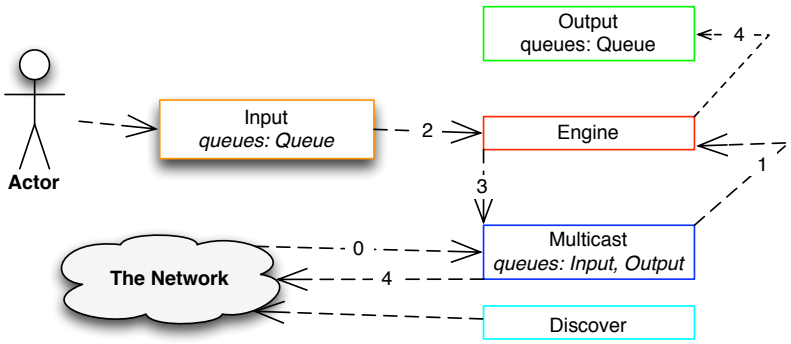


Figure 1: The 5 modules

Our game runs in a cycle that is supposed to run many times per second. This means that we assume a certain network and processing speed to be available for playing this game; enough to update every player's screen dozens of times per second and sending and receiving network data. We do not believe that this is unrealistic, given the fact that many games manage to run in this way. We will, however, have to make design decisions which keep the program efficient.

3.2 UML Class diagrams

3.2.1 The Input module

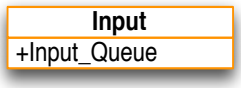


Figure 2: The class diagram for the Input module

Figure 2 shows the input module. The input module handles all keyboard and mouse signals and puts these in the queue "Input_Queue". The engine can access this queue and read all commands given by the player. PyGame will help us with reading user input, but during the writing of this section, we do not know how this should be implemented.

3.2.2 The Multicast module

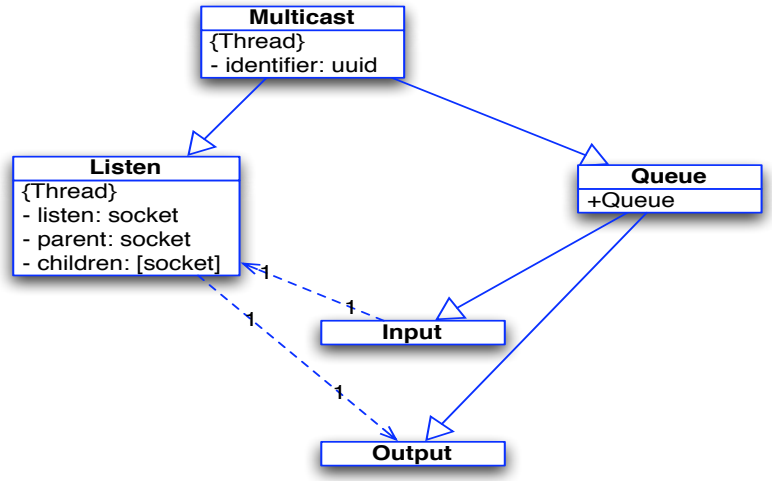


Figure 3: The class diagram for the game's Multicast module

The multicast module provides one major piece of functionality, namely communication between peers. Peer discovery is be handled by the discover module, wich is discussed in Section 3.2.3.

Communications are done over TCP by broadcasting messages across a mesh network in a multicast fashion. Each peer will be aware of the existence of every other peer, and keep them in a equally sorted list. This list will then be used for peers to take turns, providing token like communication. The messages will be queued for sending, and send when the peer's turn has come.

The only way of communicating with this modules is via the queues, no functions will be exposed.

This can be all seen in Figure 3.

3.2.3 The Discover module

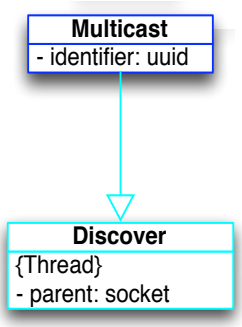


Figure 4: The class diagram for the Discover module

Peer discovery is done by broadcasting packets over UDP onto the netwerk from each peer, and then listening for others, connecting to the first peer it discovers.

The blue Network class in Figure 4 is the same class in the Multicast class diagram.

3.2.4 The Engine module

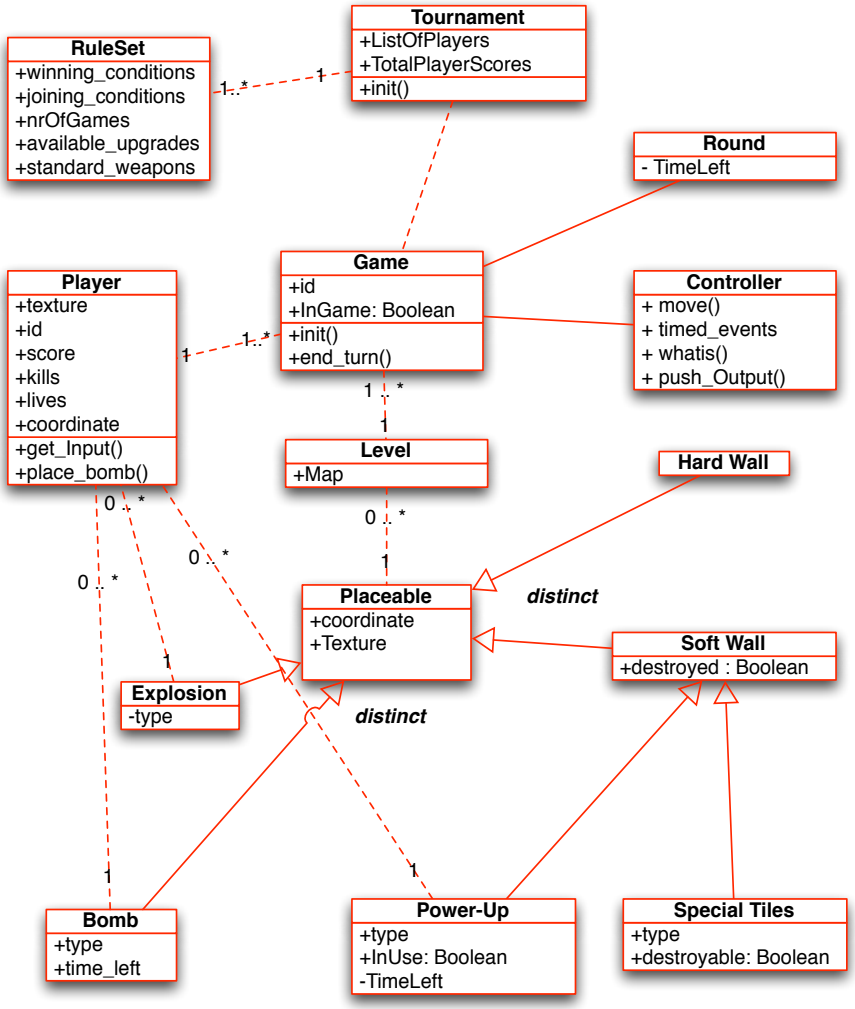


Figure 5: The class diagram for the game Engine

A player starts playing the game by creating or joining a *tournament*. A tournament consists of several *games* or rounds that need to be won before an overall winner can be chosen. For every tournament there is a specific set of rules (*RuleSet*) that states when someone wins or when anyone may join. The tournament class keeps track of all the different players and their total scores over the previous games in the tournament.

Several games will be played in a tournament consecutively. Only 1 game may be active at any given time, that game will be "InGame" (*InGame* = true). The game class handles ending the turn of a player and ending the game, making sure that all data goes to its right spot.

Every game has its own data, like the time left to play the game. This data is in the *Round* class.

The *Controller* handels moving, activating timed events (like exploding bombs, spreading the

explosion), collisions between objects and updates the data. Afterwards, it pushes the new situation to the Graphics module and the network module.

A *Player* may only be busy with 1 game, he is either waiting to join or he is playing in the active game. This class keeps tracks of the player's score, how many times he has died, where it's standing and what bombs and upgrades belong to it. It also get's the input from both the input module and the multicast module. A player can place a request to move at the controller and to place a bomb.

The *Level* class keeps track of the size of the board and the lay-out.

Several objects can be placed in a level. These objects share some properties, brought together in the *Placeable* class. This consists mainly of a location (coordinates) and a texture. A placeable object is one of four types: Hard wall, Soft wall, Explosion or Bomb.

1. Hard Wall: this is an object that a player cannot interact with. Placing these wall creates a battle field through wich the player must maneuver, or use to his advantage when placing bombs.

2. Soft wall: these are wall that can be destroyed by an explosion, but a player pass through it. This gives a player more objects to maneuver through, but can also be used to easily trap a player. When a soft wall is destroyed, it may "drop" a *Power-Up* for the player or reveal a special tile.

3. Bomb: a bomb belongs to a player and blows up when it's timer has reached zero. No player may walk through a bomb, not even it's owner. A player can use this ability to trap another player, but he can also accidentally trap himself.

4. An explosion: When a bomb explodes, it create cross-shaped explosions, extending horizontally and vertically. These explosions have a place, a short duration of time and an owner. An explosion can set off other bombs, or destroy players, soft walls and power-ups.

As has been said before, soft walls which are blown up can leave Power-Ups or reveal a special tile. Power-ups can be, but is not limited to, an increase of range, more bombs, different bombs, and a placeable soft wall ...

A special tile could be a tile that propels a player forward, a tile that changes the direction of an explosion or a tile that transports a player. Special Tiles are not part of basic gameplay and will only be implemented later in the implementation proces.

Initially, a level consists of coordinates that may have an associated object, like a wall block (indestructible) or a soft-block (destructible) or upgrade (destructible).

As been said before, soft wall can drop Power-Ups or it can reveal a special tile. A Power-up could be, but not limited to, an increase of the range of your bombs, more bombs to be placed on the field, different bombs, a placeable soft wall ...

A special tile could be a tile that propels a player forward, a tile that changes the direction of an explosion or a tile that transports a player. Special Tiles are not part of basic gameplay and will only be implemented later in the implementation proces.

3.2.5 The Output module

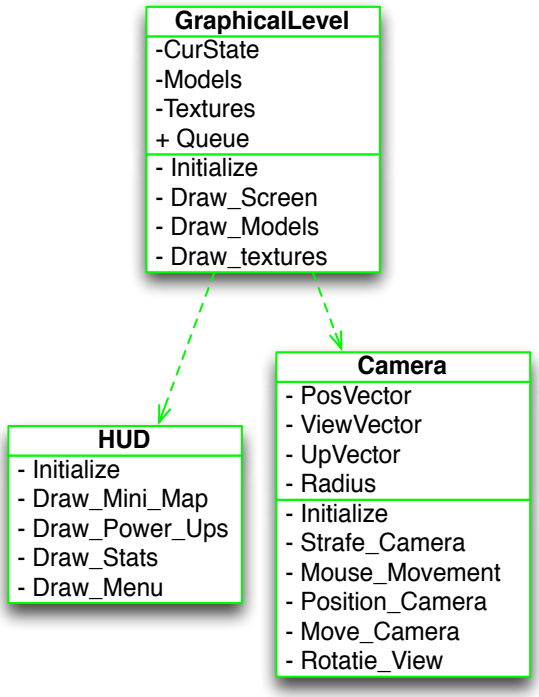


Figure 6: The class diagram for the game's Graphics

Screen
The Output module gets a game state as input from the game engine and updates the screen. This screen consists of a 3D level and 2D layer on top of the 3D level. The 3D level is drawn according to the view of the camera and consists of a 3D world with all kinds of objects. The 2D layer consists of a minimap, the player's stats, and the player's power ups, which are all drawn. The Graphics module consists of three classes, namely: the GraphicalLevel class, the HUD class, and the Camera class.

The GraphicalLevel class is the 3D level. It keeps the current game state, the models that are to be drawn and the textures that are used. After it is initialized, it can draw the 3D scene, draw the models, and apply the textures. The main use of this class is updating the 3D level based on the game state delivered by the game engine. It makes use of the two other classes to fully update the screen.

The HUD class is a 2D layer on top of the 3D level. It draws the mini map, the power ups, the stats, and possible an in-game menu when activated. This class is used to update the 2D HUD layer on top of the 3D level based on the game state delivered by the game engine.

The Camera class handles the camera movement. It can move the camera forward and backward, left and right, and rotate the camera, according to the user input delivered by the game engine. It is used as first-person view in the 3D level.

Sound
The Output module will also handle sounds when we have time left to implement it. There are some basic sounds stored, like explosions, walking etc. which will be played at certain in-game events. This means that certain keys and events will toggle a certain sound. For example when a player gets killed or kills, when a player has won, pressing an "Activate powerup" key etc.

All the sounds will be implemented in the GraphicalLevel class, since the sounds "change" the 3D level. The sounds don't change the 3D level physically, but they do change the experience in the 3D level.

4 Interpeer messages

4.1 Introduction

For all peers to be on the same page, they need to communicate. And to understand each other, we need a protocol. That is what will be defined here.

Messages will be encoded as a JSON object. We chose this because it is easy to read and write, for both humans and computers, it is compact and it can send complex datastructures. More information can be found on http://www.json.org.

Every message has a sender property, to indicate who has send the message. Also, every message has a type, to indicate what kind of message it is. And depending on the type of message, there are some more arguments, which are described below.

Some examples:

To send the chat message "Hello, world!", this message should be transmitted:

```
{"sender"="UUIDOFSENDER", "type"="chat", "message"="Hello, world!"}
```

To tell the other peers that you accepted the token, and to check the gamestate, this message should be transmitted:

```
{"sender"="UUIDOFSENDER", "type"="begin", "gamestate"="HASHOFTHEGAMESTATE"}
```

To inform a new peer about the other peers, this message should be transmitted:

```
{"sender"="UUIDOFSENDER", "type"="networkstate",
    "peers"=[{"UUID"="UUID1","IP"="10.0.0.1"}, {"UUID"="UUID2","IP"="10.0.0.2"}]}
```

4.2 Messages

- **Network related messages**
  These messages are necessary to get the network up and running, and to pass the token to everyone in the tokenring.
  - **connect** A new peer sends this message to request to join the game. This message must not be forwarded.
    Arguments:
    version: the version of the program you are running.
  - **join** Used by the first peer to tell other already connected peers that a new peer has joined the network. This message can only be send if you control the token.
    Arguments:

newPlayerUUID: the UUID of the new player.
newPlayerIP: the IP of the new player.
nickname: the nickname of the new player.

- **networkstate** Communicates the current state of the network to a new peer. This message must not be forwarded.
  Arguments:
  peers: an array of objects containing UUID and IP values of all peers in the netwerk.

- **begin** During a game, this communicates to the peers that you accepted the token, while passing a game state hash to check for possible desync faults (game state differences between peers).
  Arguments:
  gamestateHash: a hash of the game state.

- **pass** During a game, this communicates that you pass the token on to another peer.
  Arguments:
  nextPlayerUUID: the UUID of the next player.

- **part** This ends the round when a network problem has arisen (i.e. a disconnected peer).
  Arguments: None.

- **desync** This ends the round when a difference in gamestate is detected.
  Arguments: None.

- **Initialization messages**
  These messages are to initialize the game and get everything started.

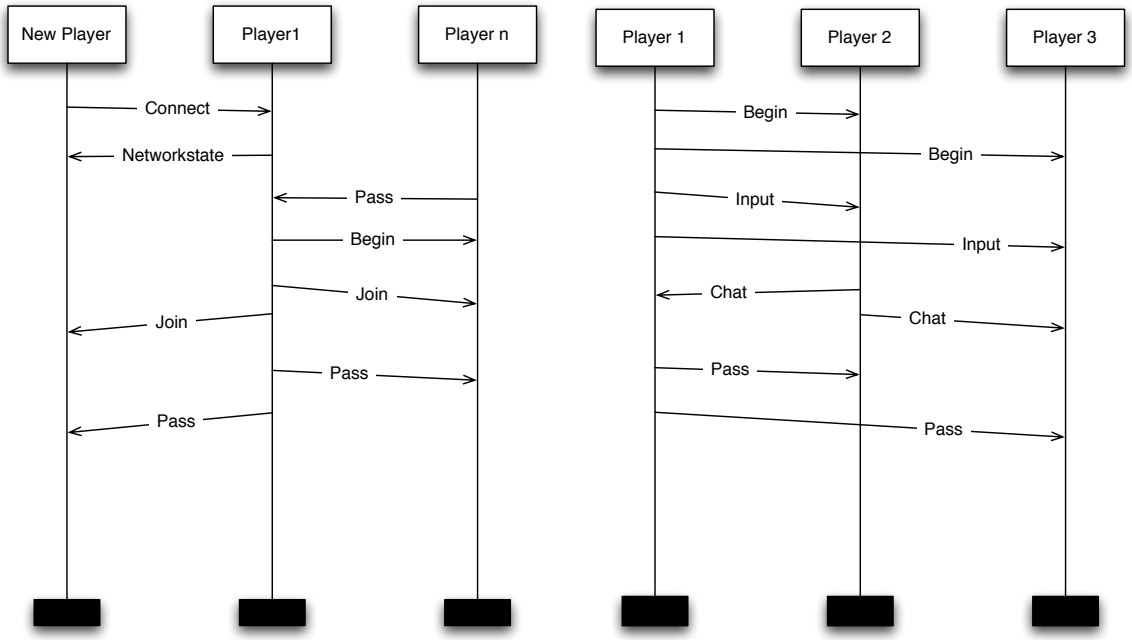
  - **level** This tells the peers which level is being used in the current game, so that they can load the level file. This message can only be send if you control the token.
    Arguments:
    levelname: the name of the level. While we don't have an option to send level files to other players, all players should have this level installed.

  - **ruleset** This tells the peers what the ruleset is for this tournament. This message can only be send if you control the token.
    Arguments:
    ruleset: an object containing key, value pairs containing the rules for the tournament (i.e. how many rounds should be played, a time limit, etc.).

- **Game related messages**
  These messages are to communicate about what happens in the game.

– **input** This tells the other peers which input has been given. They need this input to update the game state and act accordingly. This message can only be send if you control the token.
Arguments:
key: the key that was pressed. This is not the fisical key, but the logical action that that player has associated with that key.
time: the gametime the key was pressed.

– **chat** This transmits a chat message to all peers. This may be send at any time.
Arguments:
message: the chat message.



(a) A player joins the game

(b) A scenario during the game

Figure 7: MSCs of 2 typical game situation

4.3 Extension options

We thought of some options to extend the protocol. They are needed for options that are low on our priority list, so we will not design them right now, we will just give a short description.

- Join a running tournament
  To make it possible to join a tournament that is already running, some information on the current tournament should be transmitted.

- Recover from a network error (i.e. a disconnect from a player)
  When a player disconnects, the other players have to make a new network, without loops and preferably without losing more other players, so some extra communication should be introduced to make that possible.

- Level transfer
  If we have support for multiple maps, it could happen that some of the players don't have the map that is going to be played. In that case it would be nice if the map will be send to that player.

5 GUI

A number of important interface choices are described below.

5.1 Menu Structure

A list of menu types in our program:

- Main menu
- Options menu
- Tournament searching menu
- New tournament creation menu
- Lobby menu
- Tournament overview screen

When a user starts the program, it shows the *Main menu*, from which a user can enter the *Tournament searching menu* or the *Options menu*, or quit the program. In the *Options menu*, a user can change the input and output settings of the game or return to the *Main menu*. In the *Tournament searching menu*, a user can see which other instances of the program have been found on the network and which tournaments have been started on the network. He or she can text-chat with other players, join an existing tournament or return to the *Main menu*.

From the *Tournament searching menu* a user can enter the *New tournament creation menu*, where he or she can start a new game by choosing a level map file and tournament rules. After the player has clicked OK, the program shows the player a *Lobby menu*, and it will broadcast the existence of his or her tournament to the other players. In a *Lobby menu*, the player can see which other players have joined the same tournament, text-chat with other players, or start or cancel the tournament, the latter returning him or her to the *Game searching menu*.

A tournament consists of several rounds. Between rounds, and at the start and end of a tournament, players see the *Tournament overview screen*. This screen shows the current score (if any), and in it, the creator of the tournament can start the next round, remove players from the tournament or end the tournament prematurely.

5.2 Visual Output

This is the way in which our visual output to the player will be organized, as illustrated in Figure 8 :

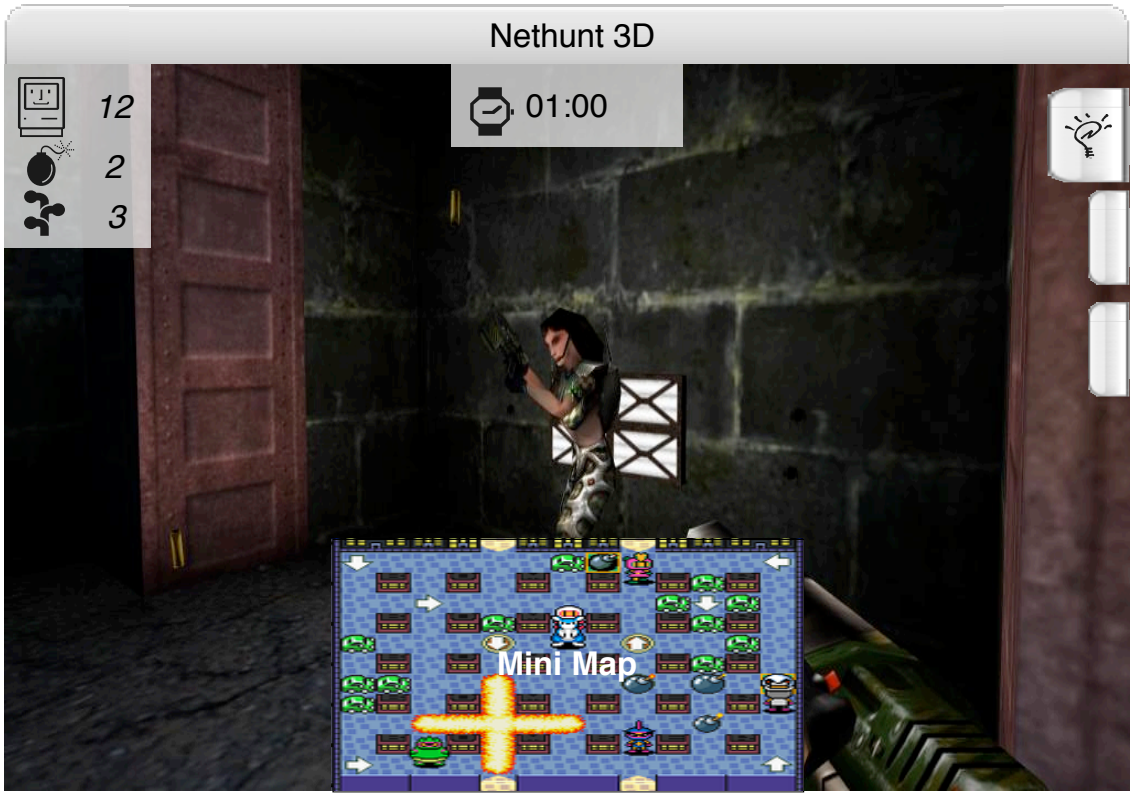


Figure 8: A mock-up of the interface

The program can be run in a window or fullscreen, with fullscreen being the default. During a round, most of the screen will be taken up a by a 3D visual depiction of the player's in-game surroundings.

In the top left corner of the screen, along the left side, will be the player's score, the maximum number of bombs they can create and the reach of their bombs' explosions. In the top right corner of the screen, along the right side, will be three slots which can be filled by the power-ups which the player has collected and has yet to use. Only three power-ups can be stored in this way at any one time.

The clock is located at the top of the screen, which displays the time left to play in the current round (if applicable).

In the lower center of the screen, at its bottom edge, will be a display of the map of the level, with moving 2D sprites giving a lot of information about the action which is taking place, facilitating strategic decision making. At the touch of a button, the minimap is enlarged to being about half as high and wide as the screen, or reduced to being about a quarter of the screen's height and length in size.

The chat function will display an input field an others' text in the bottom of the screen.

5.3 Input

While in a game, the player can use the mouse to look around. There will be a default assignment of mouse buttons and keys to actions, which can be changed by the player.

The default input configuration is as follows. The player moves around using the A, S, D and W keys. W moves the avatar forward, S moves him backward, and A/D is used to move sideways. Bombs are dropped using the left mouse button. The numeric keys are used to activate the corresponding power-up. The arrow keys will be used for basic navigation through the program's menus, which can also be controlled with the cursor.

List with input keys:

- W: Move forward.
- S: Move backward.
- A: Step sideways (left).
- D: Step sideways (right).
- 1: Activate power-up 1.
- 2: Activate power-up 2.
- 3: Activate power-up 3.
- M: Toggles the size of the minimap.
- TAB: Toggles the chat window (when implemented).
- $MouseMovement$: Look around.
- $LeftMouseButton$: Drop Bomb.

6 Division of work

- Output: Leroy, Edin, Neal
- Network, Input: Jeroen, Anson
- Framework, Engine: Stef, Etienne, Edin
- Final document, Manual: Anson
- Menus: Anyone with spare time.

The output and the engine will probably take most of the work, so both get two dedicated developers. These are Leroy and Neal for the output, because they have already done some exploratory work on the topic, and Stef and Etienne for the engine, for the same reason.

Building the networking and input modules will yet take some time, but it is not as much work as building the above two modules, so it will get just one dedicated developer. This will be Jeroen, who has already designed the network's overall structure.

Anson will work mostly on documentation, because he has already coordinated the documentation earlier in the project. He will also assist the network module's developer, particularly during the early stages, when assistance will be very helpful.

Edin will help the engine developers early on, when the design of the framework will take a lot of their time. He will move on to assisting the output developers later in the project, when they will have more work. This will have the added benefit that the output developers will have someone working with them who has some knowledge of the engine.

The menus will be created by anyone with spare time. They can be made as simple or as elaborate as we want, and their looks are not essential to the gameplay, so the amount of time spent on them can be more or less as large or as small as we see fit.