OGO 3.1 (2IO45) spring 2009

Project Report

Technische informatica, TU/e

Eindhoven, January 5, 2010

Group 2
Leroy Bakker, 0617167
Etienne van Delden, 0618959
Edin Dudojevic, 0608206
Neal van den Eertwegh, 0610024
Jeroen Habraken, 0586866
Stef Louwers, 0590864
Anson van Rooij, 0596312

Contents

List of Tables

List of Figures

Chapter 0

Summary

This document combines all our previous delivered documents. It shows from start to end how we designed Bomb3D, as well as the division of work between team members. You can also find our specification, a list of design decisions and alternatives to the implementation. Some sections have been added which descibe our reflections on the old documents, with an evaluation section added at the very end of this new document.

Chapter 1

Introduction

During this OGO we had to make a network distributed 3D game, with a synchronization demand. After a brainstorm session, we have chosen to implement our own version of Bomberman (called Bomb3D), a multiplayer game by Hudson Soft. In Bomb3D, players must destroy each other, by first destroying soft wall that can reveal power-ups. These power-ups must be synchronized, when player A picks it up, no other player can pick it up again. With this decision, we differentiated from the original food gathering demand, but kept the synchronization challenge.

Chapter 2

Plan of Work

Because we cannot predict how much work goes into each part of the development of Bomb3D, we set an internal deadline to finish products. This deadline is one week before the official deadline. This also gives us the opportunity to receive feedback on documents. What we would like to do, is hand in products two weeks in advance, but time will tell if we can reach those goals.

2.1 Roles within the project

We have divided different roles to each

- Anson will be our chairman. He will also be responsible for all documentation
- Edin will be our time manager, making sure everyone fills in his logbook.
- Etienne is our secretary and will handle external contacts.
- Neal is our Quality Assurance Manager for documents.
- Stef is our Quality Assurance Manager for code.
- Jeroen is our network and Python specialist.
- Leroy is our OpenGL specialist.

2.2 Deciding about the type of game

To help determine the type of game we will make, we held a brainstorm session. This session was split in two parts, the first to chose a genre, the second to determine a play-style or game we would replicate.

2.3 Reflection

During the entire project, the exact roles of everyone changed a lot. This was due to changing priorities and people helping out with other parts of code. Documentation is the prime example, everyone has worked on their own part of documentation and sometimes on the whole document, or someone else took over documentation so that more people could code the game.

Chapter 3

Specification

3.1 Introduction

In this phase we made the specification of the game. The description in the assignment has been kept quite vague by the OGO-organisation and in this phase we have made it much more detailed. First we had a brainstorm session, generating a number of possible variations. Then we chose one of these variations and selected features, organising those features by feasibility and attractiveness, using the MOSCoW method, as mentioned in the orientation document ("werkdocument"). In this part, we describe these choices.

3.2 The assignment

This is the assignment, as described in the "projectwijzer":

'Gevraagd wordt om een interactief, gedistribueerd 3D spel te ontwikkelen. Gedistribueerd houdt hier in dat iedere speler zijn eigen beeldscherm gebruikt en dat alle schermen een weergave van dezelfde spelsituatie tonen. Op het speelveld bevinden zich twee tot zes spelers. Verder is verspreid over het speelveld "voedsel" aanwezig. Doel van het spel is zo veel mogelijk voedsel te verzamelen. Wat zich verder op het speelveld bevindt wordt aan de ontwikkelaars overgelaten. Belangrijk is dat het spel leuk is om te spelen, er attractief uitziet en natuurlijk nooit faalt.'

3.3 Definitions

Table 5.1 has some definitions of terms used in this document.

Term	Explanation
Bomb3D	The name of the game. We usually mean the application itself.
Game	An instance of Bomb3D, currently running in a match.
Avatar	A 3D model, representing the player in the game (we will use word player mostly
Continuous Movement	More or less fluid movement; the player does not stick to a grid or take discrete steps
First Person	The player views the game from the eyes of his avatar
Level	A model of a maze-like 3d-environment, described by a map
Mini map	A small map of the level on the edge of the screen
Power-up	An item, represented by a model in-game, that can be picked up by an avatar and give him or her some bonus or extra power, hence the name
Spawn	Being placed in the level
Soft wall	A soft wall can be destroyed by a bomb
Range	The range of a bomb is the size of its explosion. A range of 2 means that the explosion of a bomb continues to 2 coordinates to the left/right/up/down of the bomb, hence a cross shape

Table 3.1: Overview of the document definitions

3.4 Our brainstorming session

We started this project with a brainstorming session. We wrote all our ideas down on a whiteboard and selected parts that we did and did not like. We eventually ended up with a Bomberman-themed game with strategic elements (exploring a maze, controlling an area) and shooter elements (action-orientation, fierce competition).

We decided on this formula because the basics are familiar, making for relatively straightforward implementation and because it has a popular and fast kind of gameplay with a low barrier to entry. Also, bombing and power-ups (see also: Table 5.1) allow us to give the game a unique look and feel.

3.5 The features

3.5.1 Must have

After our brainstorm session, we made a list of features that we want to implement, to make it a playable and fun game. These features are given in a MOSCoW list, giving features a higher to lower priority.

- Collision detection
- A map to play on
- Network functionality

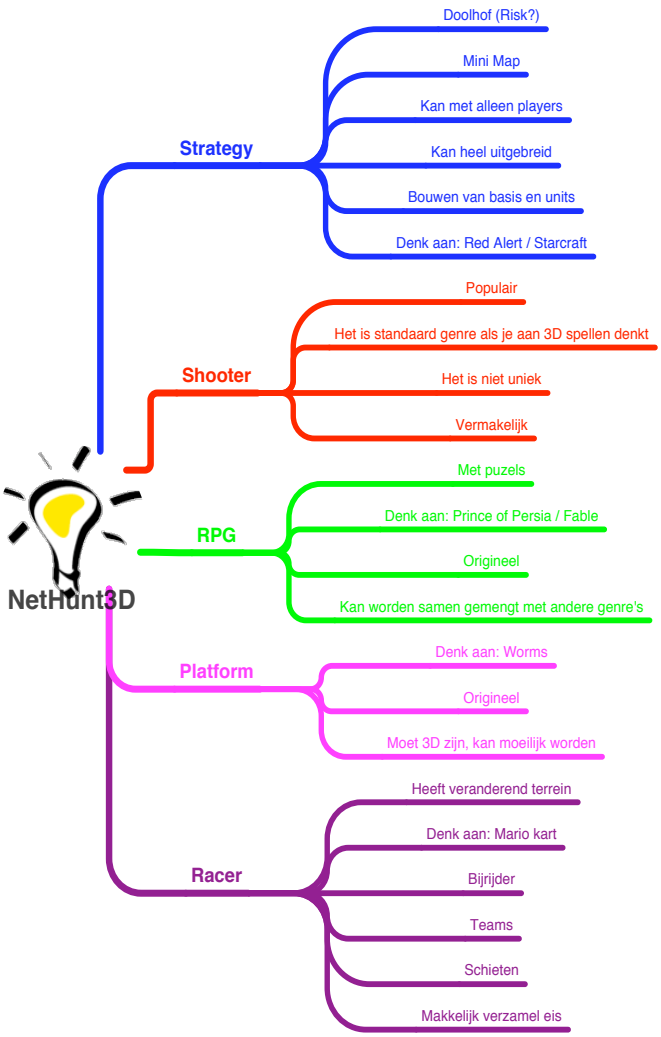

Figure 3.1: The mind-map we came up with when choosing the genre

- An interface
- Power-ups

Reflection

These features are all included in the game.

3.5.2 Ought to have

The game ought to have these features to conform to reasonable expectations.

Figure 3.2: The different styles of games are given in this mind-map. We eventually choose Bomberman

- Weapons
- Audio

Reflection

We have not been very clear on what we meant by 'weapons'. Originally the idea behind this point was to have multiple kinds of weaponry, as many shooter games do. As the project progressed and our ideas about the desired gameplay solidified, it turned out that this would not contribute much to the gameplay, so it was scrapped. The program does have audio: there are sound effects and there is background music.

3.5.3 Should have

The game should have these features, they are feasible and improve it greatly.

- Chatting
- Multiple maps
- Multiple avatar textures

Reflection

There is support for chatting in the game, but we do not currently show chat messages on the screen. Chat has been lowered in priority because the players are usually in the same room when playing our game anyway. We do have multiple maps support available for players (currently hard-coded). We do not have multiple avatar textures, which is the first serious point to be scrapped; there simply wasn't time to make or find them.

3.5.4 Could have

We might include these features, so we should design the game so that we can add them later.

- A map editor
- Game rules (for mods)
- Different game modes
- Intro/victory/scenario movies

Reflection

The game does not have a map editing function, although making maps takes little work once one knows the basics. The game does have an event engine, and therefore it supports the easy addition of new rules or the creation of new game modes in a way similar to that in which many shooter games do; people with some rudimentary programming knowledge can create new scripts, and put a reference to these rules in a map file. Any non-techsavvy player could then choose a map file when creating a game, and in this way have a completely customized game experience. The game does lack movies; it might have been better for us to put these in the 'want to have'-section.

3.5.5 Want to have

If we have a lot of extra time, these are the features we want to add to the game.

- Dynamics
- Physics
- No copyright material / Own copyright material

Reflection

The textures we use are not copyright free. The sounds can be used for non-commercial use. We have researched the Open Dynamics library, but that would mean we would not have to make collision detection our selves, so that as scrapped. Integrating a better physics and dynamics engine would good for the gameplay of the game itself.

3.6 Gameplay

This description of the gameplay is based on the "should have" version of the game. Bomb3D is similar to the original Bomberman. Players, represented by avatars (see also: Table 5.1), spawn (see also: Table 5.1) in a level (see also: Table 5.1). They have a starting score, number of bombs and bomb range. They can move continuously (see also: Table 5.1) through a maze-like environment. Of course, players can not move through the walls of the maze, nor can they move through bombs, but they can move trough other avatars. There are two types of walls, walls which can be blown up by bombs, called "Soft Walls"(see also: Table 5.1) and walls that cannot, the "Hard Walls" (see also: Table 5.1). The number of bombs denotes the amount of bombs which can be at most be set on the map at any given time. When a bomb of a player explodes, the player can place a new bomb.

Player interaction occurs mainly through "bombing"; players can have a number of bombs at any one time, which they can leave anywhere in the maze. These bombs explode after a while, hitting anyone who is in range. The range of a bomb is the total range of the player. Players can gain extra bombs and bomb range by finding power-ups. These are located inside the soft walls of the games. When a certain wall is blown up, the power-up appears.[1]

If players are hit, they are removed from the level and must wait for the next round. When only one player is left, that player has won the game. Each game has a time limit in which players can try to survive. When the clock has reached zero, the player with the most power-ups wins. The players' scores are kept throughout multiple games, showing their ranking in the ongoing competition.

Reflection

This has all been implemented as stated, with two exceptions: first, games are currently not time-limited, and second, scores are not kept throughout a 'tournament' consisting of multiple games.

[1] Not all walls contain power-ups, they are scattered randomly. Some may convey another bonus, if we can find the time to implement that.

3.7 Interface

The players have a first person view (see also: Table 5.1) of the level, which takes up most of the screen. The other players' avatars and the level are shown in 3D, but the level is essentially 2D, that is, it can be described by a 2D map.

The players' number of bombs, their power and any special power-ups are all displayed in the game window, which also contains a minimap (see also: Table 5.1) showing the player's environment.

The graphical theme will be similar to the original Bomberman: cartoony and futuristic. This theme will not influence the game's implementation until much later in the project, however and the specifics are not set in stone. We expect our ideas about the theme to evolve as we continue working on the project.

Players use a combination of the mouse and keyboard to control their movements: they move the mouse to turn, click mouse buttons to perform basic actions (place bombs), use the arrow keys to move and sidestep and the other keyboard buttons for any special moves and actions which might be implemented.

Reflection

This has all been implemented as stated.

3.8 GUI

A number of important interface choices are described below.

3.8.1 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 player starts the program, it shows the *Main menu*, from which a player can enter the *Tournament searching menu* or the *Options menu*, or quit Bomb3D. In the *Options menu*, a player can change the input and output settings of the game or return to the *Main menu*. In the *Tournament searching menu*, a player 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 player 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 games. Between games, 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.

Reflection

The graphical player interface supports most of these functions, though the menu structure is slightly different, and there is no way to chat with your peers over the network (though you can, of course, chat with them in real life, as you are probably in the same room)

3.8.2 Visual Output

This is the way in which our visual output to the player will be organized, as illustrated in Figure 3.3.

The program can be run in a window or fullscreen, with fullscreen being the default. During a game, 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 game (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,

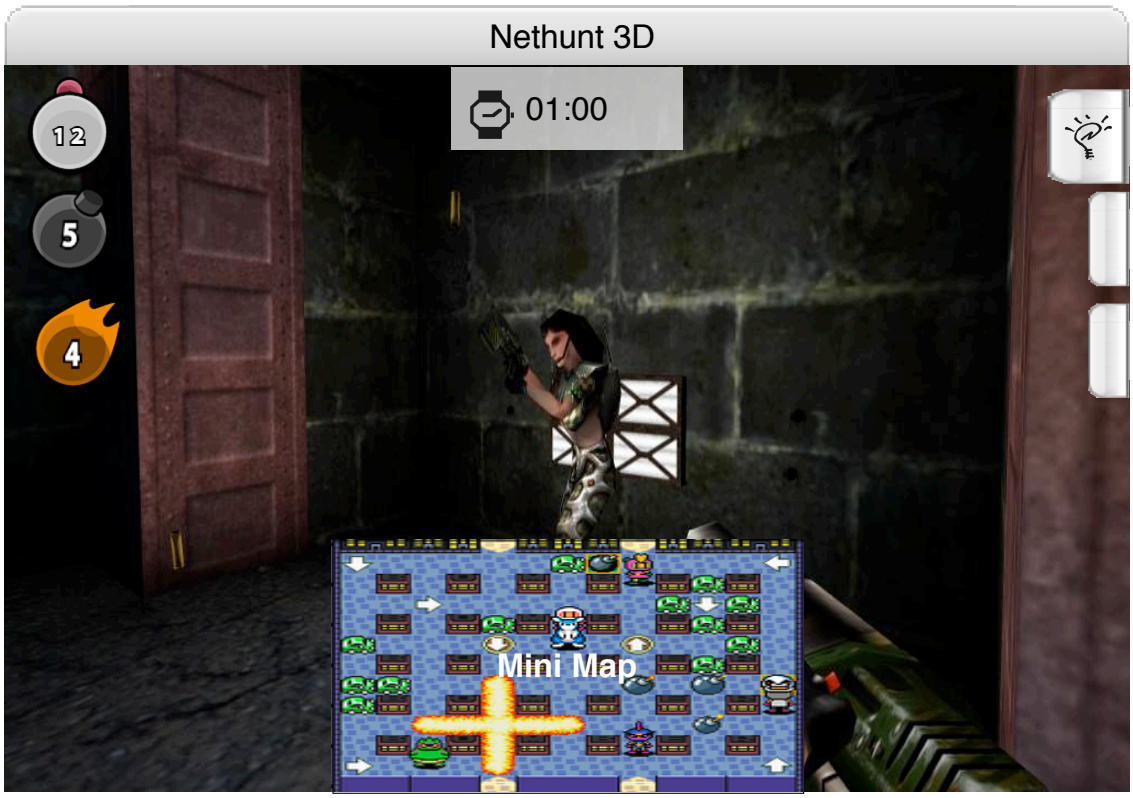

Figure 3.3: A mock-up of the interface

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.

Reflection

There is no way to change the display mode from windowed to fullscreen in-game. For testing purposes, we changed the default to windowed mode. There is also no clock, as games are not timed. The minimap cannot be enlarged; it is large enough to see everything which is going on in the level, and enlarging to would block a lot of the player's vision. There is no chat functionality.

The interface is a part of the game which could have been improved a lot with little extra time; most of the things it lacks are simple to implement. Unfortunately, we had other things to finish first.

3.8.3 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 W, A, S and D 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.

Reflection

There is an ini-file containing the key mappings, though there is currently no way to change it.

3.9 Winning the game

The main goal of the game (varying slightly between game modes) is being the last man standing in the most games.

We might include the option to play the game in different modes, in which the objectives vary, if we have time. Inspiration on that matter can easily be drawn from so called "shooter" games and these modes may include:

- Deathmatch, in which the objective is doing as much bombing of others as possible within limited time or within a limited number of respawns.
- Capture the flag, in which the players are divided into teams and the objective is to steal the other team's "flag"-power-up from their base and run back with it to your own base.
- Greed, in which the player who gathers most power-ups wins.

Reflection

This works as stated. No extra game modes have been implemented, though can easily be scripted.

3.9.1 Technological

We will use a star-network for communication between the players. For the graphics we will use Python and OpenGL with the PyOpenGL package for binding them. We will also use the PyGame library for some auxiliary functions, though we will still have to do the main graphics programming ourselves, as PyGame only helps with 2D graphics.

The program will consist of several components: a main component which handles the graphics and the game state, one for the network and one to handle player input.

Reflection

Very shortly before delivering this document for the first time, we changed the network type from "token-ring" to "star", to reflect our plans at that time. Unfortunately, these plans turned out to be too ambitious. We turned to using a token-ring network anyway when the implementation phase rolled around. The PyGame library turned out to be useful, though quirky. It turned out, for example, that input reading under PyGame could not be executed in another thread than screen updating.

As explained in the design document, our program is more complex, and has more components, than stated here. Significantly, graphics and game state handling (engine) have been separated. This specification document gives only a vague, preliminary sketch, so that was to be expected.

Chapter 4

Design

4.1 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.

4.2 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".

4.2.1 Design decisions

- The game is played in "tournaments" that consist of multiple games, called "games". The total score over all games 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 player'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 player's input over the network to its peer programs. By the time their next turn comes around, each peer program will have sent its player's input to Bomb3D, so that it has a new list of commands to which to add its own player'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 game is the last player surviving. To avoid overly long matches, a tournament's creator can choose a maximum time the game 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 game 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 we solved 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.

4.2.2 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 players. We will attempt to have Bomb3D work under Windows (XP), Mac OS X and Linux.

4.2.3 Reflection

Unfortunately we were not able to implement some of the design decisions. The game doesn't consist of a tournament with several games. In fact you only play one game. Also players cannot make their own levels that easy. Time was lacking to do so. All the other design decisions are correctly implemented. The network is able to handle all the traffic, but the speed is slow. We assumed that each program acts tens of times per second, but in practise the token went game 5 times per second. Because of this the multiplayer game is really slow. The rulesets as we designed weren't implemented, in favour of an event system on wich scripts can register.

4.3 Program modules

4.3.1 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 game, 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 network module

3. The Output module. During a game 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.

4. The Engine module. During a game, 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 5.1. In the next section, we will discuss how the separate modules are designed.

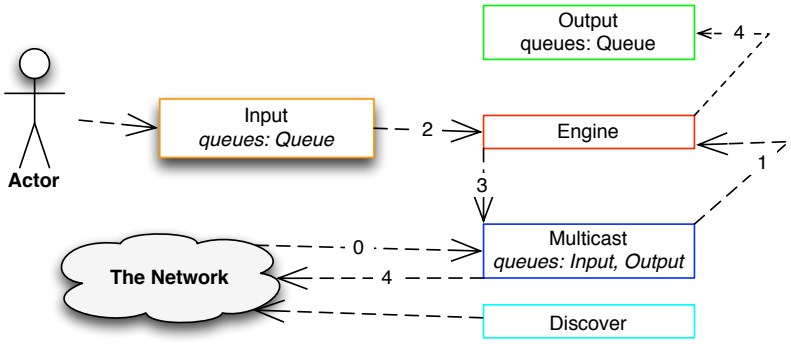

Figure 4.1: The 5 modules

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 Bomb3D efficient.

4.3.2 UML Class diagrams

The Input module

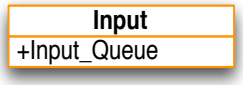

Figure 4.2: The class diagram for the Input module

Figure 4.2 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 player input, but during the writing of this section, we do not know how this should be implemented.

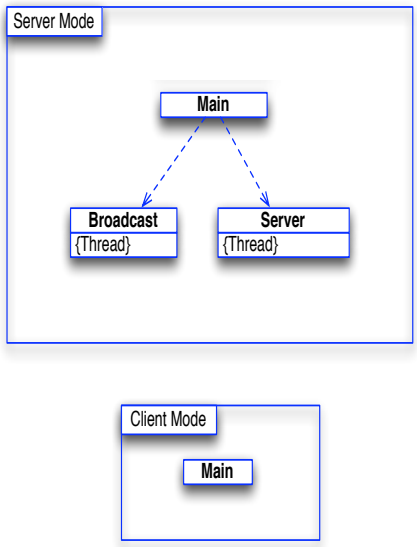

Figure 4.3: The class diagram for the network module

Starting the game and Discovery

We use a client/server model for starting a game. One client creates a server, that sends a broadcast message to the entire network. All clients that are waiting to join a game, send a message back to the server. When the server times out, it tells every other client to whom he needs to connect. This makes a token-ring no longer requiring a server.

The blue Network class in Figure 4.3 is the same class in the network module diagram.

The Engine module

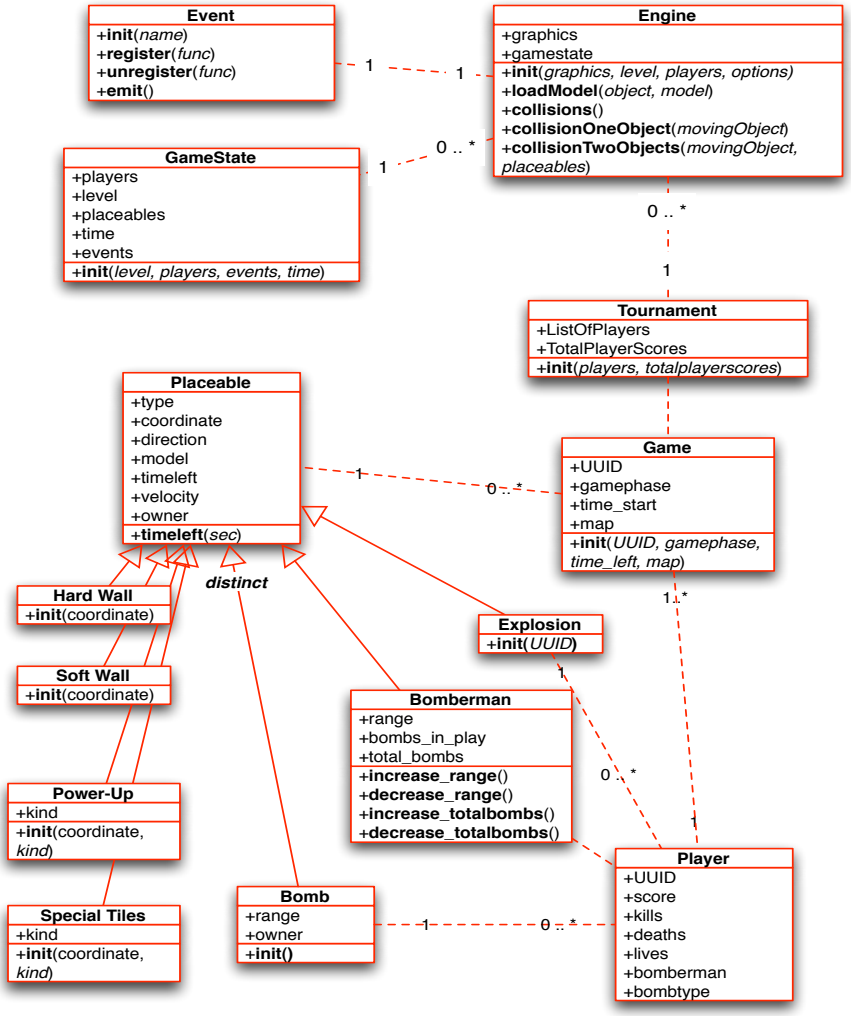

Figure 4.4: The class diagram for the game Engine

A player starts playing the game by creating or joining a *tournament*. A tournament consists of several *games* or games that need to be won before an overall winner can be chosen. A game can have one or more players, though, of course, a game with one player in it is not very exciting. For every tournament there is a specific set of rules 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". 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.

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.

Events

To connect the engine to the game logic, and to keep that connection flexible and extensible, we are going to use events. Events are things that can happen in a game, for example a player dies, a collision occurs, even a player who presses a key.

Events often require something in the gamestate to change. For example when a player dies, he has to be removed from the game and points have to be given to the killing player. And when a player walks against a wall, a collision occurs and that player has to be stopped. So to do that, we'll introduce scripts that handle those things.

A script is a piece of code that is interested in one of more events. So when that event happens, that piece of code gets called and makes changes to the gamestate accordingly.

The big advantage of using events this way is that the engine stays very clean, and the game logic can easily be adjusted. We could easily make a shooter or a tetris game in this engine!

The Output module

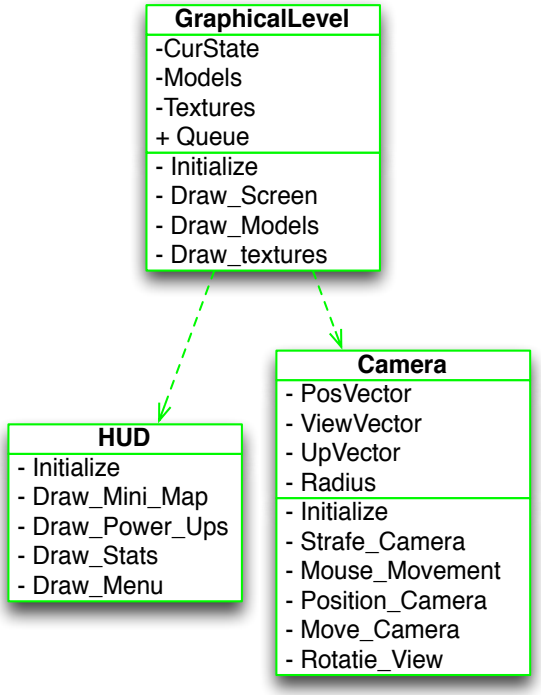

Figure 4.5: The class diagram for the game's Graphics

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 player 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.

4.3.3 Reflection

About the output module: We do not have 3D models for the objects displayed in the game; we use simple geometric shapes instead. This does not detract from the enjoyment of the game. In the HUD, the player cannot resize the minimap but otherwise things work smoothly. The sound also works.

About the engine module: The engine works, and customizing the scripts is not as easy as we imagined it could be. The engine code is stable, though it does not contain all features which we wanted to include, and so things which should have been customizable are hard-coded instead. The engine has been connected to the rest of the game relative late due to time pressure, so debugging it took a lot of time.

About the input module: this works fine.

About the network module: This works well as long as the player is in-game. The module has unfortunately turned out to be too simple in its design to have much use in the framework menus, which has caused us to drop many options (such as an easy way to choose a map and ruleset, and chatting before the game) which were originally planned. This module also cost us a lot of debug time.

Overall: Bomb3D works more or less as it should. There were some synchronisation issues during early testing, but as far as we know these are now resolved through standardisation of the framerate. Bomb3D does run too slow, though, which impedes proper testing. The modules communicate very well, there's no problems there.

4.4 Interpeer messages

4.4.1 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"}]}
```

4.4.2 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 game when a network problem has arisen (i.e. a disconnected peer).
  Arguments: None.

- **desync** This ends the game 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 games 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.
    order: the place of the input message in relation to other input messages sent at the same game time.
  - **chat** This transmits a chat message to all peers. This may be send at any time.
    Arguments:
    message: the chat message.

4.4.3 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.

4.4.4 Reflection

The interpeer messages work as they are described here. There are no problems with the messages, and the network works fine. Only the the token goes game five times per second, which is really slow. We don't know what exactly causes the token to travel at such a low rate. The extension options are not implemented simply because we didn't have time to do so, and there were other priorities during the implementation phase.

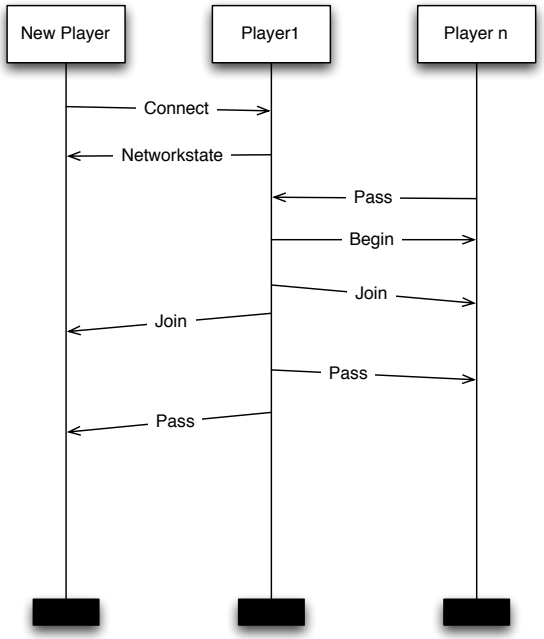

Figure 4.6: A player joins the game

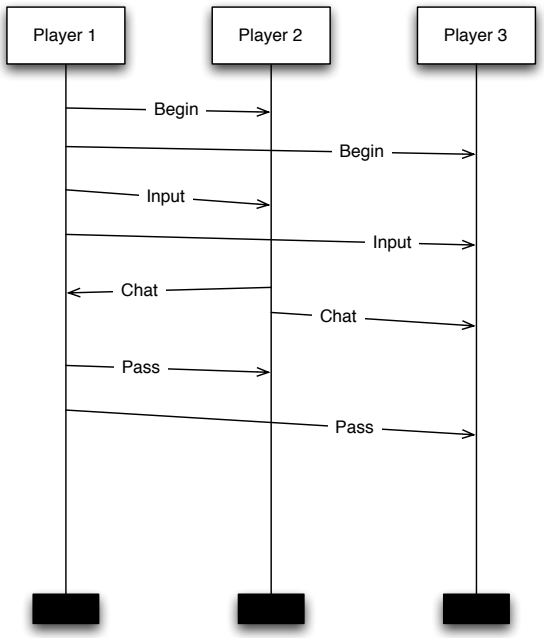

Figure 4.7: A scenario during the game

Figure 4.8: MSCs of 2 typical game situation

4.5 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.

4.5.1 Reflection

The division of work as written above, actually worked pretty well. Everyone had there own tasks and everyone could work independently when needed. Also most of the group worked on different tasks than they were scheduled when this was needed. This was the case at the end of the project when everyone has fixing bugs and typing the documentation. Although the game didn't turn out exactly the way we wanted, we think that the division of work wasn't the problem.

Chapter 5

Evaluation

We enjoyed making the game. Creating the game was frustrating, because it took really long to get a working version. When we finally had a working game, we enjoyed playing it, despite the horrible frame rate.

5.1 Problems with the current set-up

This was the first time OGO 3.1 was given in 1 semester. We have found a few problems with the way the project was organized:

- Too much time was planned for the design document. We wasted a lot of time making small changes to the design, changes which, during implementation, often turned out not to matter, not to be as good as the original, or not to be implemented at all due to time constraints.
- There was too little time for actually making the game. A lot of errors in the game were found after the presentation. If we had 2 more weeks, we would have finished the game.

te veel tijd voor design document te weinig tijd voor code

5.2 Peer reviews

At the last meeting with our tutor, Stef van den Elzen, we made anonymous peer reviews Below is

Person	Comments
Leroy Bakker	Goed inzet getoont mbt openGL
Etienne van Delden	Goed aanwezig en houdt het overzicht
Edin Dudojevic	Minder discussieren, meer accepteren en coden
	Af en toe lang discussieren over dingen die al besloten zijn
	Minder praten, meer coden
	Erg veel commentaar vaak
	Ging af en toe te diep op dingen in
Neal van den Eerthwegh	Duidelijker aangeven wat je doet/wilt
Jeroen Habraken	Minder op IRC zitten
Stef Louwers	Goed ingezet
Anson van Rooij	Goed de groep er bij gehouden

Table 5.1: Outcomes of Peer reviews

Groepslid	oordeel groep	Afw.	Cor.	Opmerkingen
Leroy Bakker	3,71	0,73	0,37	
Etienne Delden	3,14	0,16	0,08	
Edin Dudojevic	2,71	-0,27	-0,13	
Neal van den Eertw	2,29	-0,69	-0,35	
Jeroen Habraken	2,57	-0,41	-0,20	
Stef Louwers	2,86	-0,12	-0,06	
Anson van Rooij	3,57	0,59	0,30	

Figure 5.1: The group evaluation

Bibliography

[1] Guido van Rossum, *A Python Tutorial*. release 2.6.2, 2009.