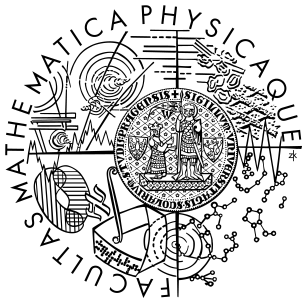
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



David Slabý

Ditchers

Katerda aplikované matematiky

Vedoucí bakalářské práce: RNDr. Bernard Lidický
Studijní program: Informatika, obecná informatika

2010

Děkuji RNDr. Bernardovi Lidickému za vedení této bakalářské práce a za mnohé užitečné rady. Dále bych rád poděkoval Janu Jeronýmovi Zvánovci za tvorbu balíčků pro Debian, své manželce Alžbětě za tvorbu hezké herní grafiky a také všem přátelům, kteří byli ochotni projekt testovat a měli k němu cenné připomínky.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 23. května 2010 David Slabý

Contents

Název práce: Ditchers
Autor: David Slabý
Katedra: Katedra aplikované matematiky
Vedoucí bakalářské práce: RNDr. Bernard Lidický
E-mail vedoucího: bernard@kam.mff.cuni.cz

Abstrakt: Cílem této práce je skloubit hratelnost a myšlenku legendární hry Tunneler s možností programování umělé inteligence pro počítačové hráče. Psaní skriptů a jejich použití ve hře je oddělené od hry samotné, aby autorovi skriptu stačilo pro úspěšné tvoření inteligentních robotů znát kromě skriptovacího jazyka pouze několik funkcí rozhraní. Zároveň je hra dostatečně atraktivní i pro běžné hráče, je snadno ovladatelná a oproti Tunnelerovi obohacena o další možnosti, například výběr zbraní, mapy i typu robota. Důležitou možností je také hraní na lokální síti.

Klíčová slova: Tunneler, hra, programovatelná AI

Title: Ditchers
Author: David Slabý
Department: Department of applied mathematics
Supervisor: RNDr. Bernard Lidický
Supervisor's e-mail address: bernard@kam.mff.cuni.cz

Abstract: The aim of this work is to combine the playability and the idea of the legendary game Tunneler with the option of programming artificial intelligence for computer players. Writing scripts and their use in the game is separated from the game itself, so the author of a script only has to know the scripting language and a few interface functions to successfully create an intelligent robot. At the same time the game is sufficiently attractive to casual players, it is easy to operate and has some additional features, such as multiple weapons, maps and robot types. Another important feature is the option of playing over LAN.

Keywords: Tunneler, game, programmable AI

Chapter 1

Introduction

In 1991, Geoffrey Silverton had written a split-screen game for two players, where two underground tanks were put randomly on an island and their only quest was to find and destroy the opponent by digging tunnels in the soil and shooting with some kind of a machine gun. This game, named Tunneler[9], did soon become a famous one and for several following years many players considered it to be one of the best freeware multiplayer games. In nowadays, it was replaced by other, more sophisticated games, but the glory of Tunneler will never be forgotten.

The main goal of Ditchers is to revive the idea of Tunneler. Majority of current games is famous for its stunning graphics and cool effects while the basic principles are quite boring. In Tunneler, we find a few unique features that new games lack. First, the terrain in the game is modifiable; modern games are often based on vector graphics with immutable objects that remain still the same (even if hit with a nuclear bomb). Second, a player has only a limited information about others and it is what makes the game interesting. Third, the controls and the screen of a player are as simple as possible.

Ditchers honors these three features, but to attract today's player, more has to be added. Many additions are simply using up capabilities of new processors and graphic cards, for instance hardware accelerated graphics enables playing on large maps in high resolution at formidable speed. The new option of toroid maps greatly extends strategic options. New weapons, some with quite intriguing features, were added. To make it possible for more than two players to enjoy the game at the same time the option of network game was implemented.

Last but not least, computer players may be present in the game. Actually, it is even possible to only watch two computer players how they fight each other. The unique feature of the possibility of artificial players is the option of writing own scripts of artificial intelligence for these players. Therefore the Ditchers project may be interesting not only for common players who used to enjoy Tunneler, but also for advanced computer users capable of writing intelligence scripts, who might compare abilities of their created intelligent robots with others.

To sum it up, Ditchers has three basic goals. First, to remind the glory of Tunneler by creating an up-to-date remake. Second, to amuse common players of two-dimensional action games. Third, to provide a comfortable interface for writing scripts of artificial intelligence.

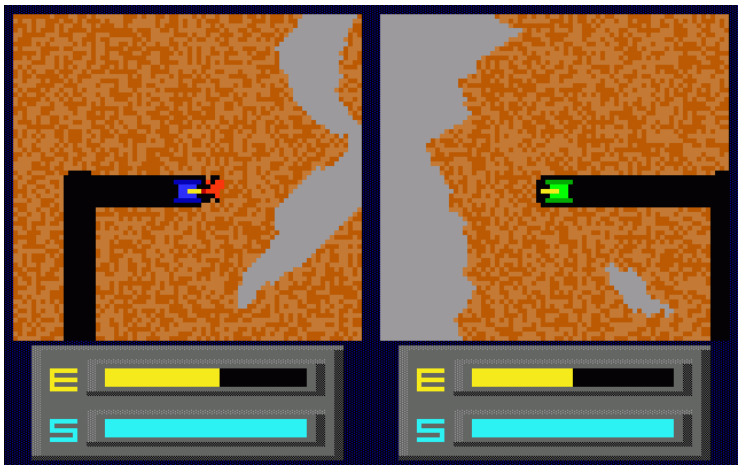


Figure 1.1: Screen-shot of old Tunneler from 1991

Chapter 2

Game

The whole project consists of two applications: the first application, `ditcher`, manages everything that is required for local game and also works as the client part in a network game. The second application, `ditchs` (which is shortening of Ditchers server), is a very lightweight program that works as the server for network games and to start the server means simply to launch this application, with optional parameters `-p/--port portnumber` to set server port (default 8421) and `-m/--mute`, `-v/--verbose` to set amount of console output. This chapter describes `ditcher`.

2.1 Graphic User Interface

To make game settings as comfortable as possible, it was necessary to wrap the game into some kind of Graphic User Interface (GUI). The intention was to keep this GUI simple, lightweight and intuitive.

2.1.1 Splash screen

As in many other games, the first graphics that appears after starting the game is a splash screen. When loading of game settings and data took longer than a blink of an eye, the need for something to appear immediately after starting the application emerged; it is definitely more pleasant to watch an image depicting the game than a blank screen while waiting for the game to load. The splash screen is always visible at least for one second.

2.1.2 GUI windows

The whole GUI consists of several windows and the user can switch between them using buttons. When the game is loaded, the main window is shown. At each moment exactly one window is visible. Diagram of switching between windows is shown in Figure 2.1.

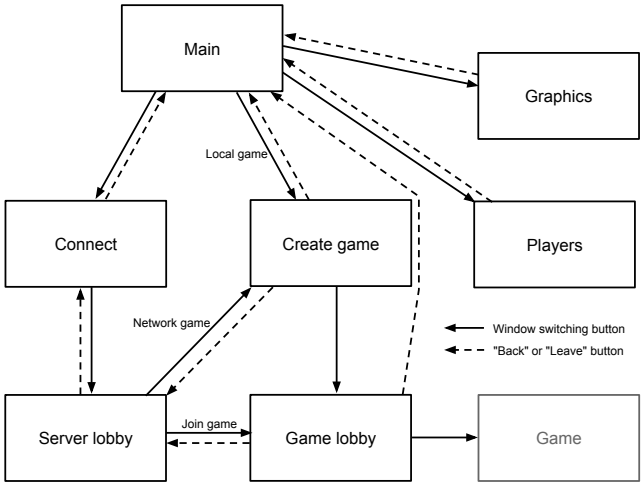


Figure 2.1: Diagram showing switching between windows

2.1.3 Main window

Main window consists of six buttons. First button, "Local game", switches to the window where a local game is created. Second button, "Network game", switches to the window where user can connect to a remote server. Third button, "Graphics", leads to graphics settings. Fourth button, "Players", shows window where local players may be managed. Fifth button, "Credits", shows information about the whole game and its creators and sixth button, "Quit", exits the game immediately.

2.1.4 Graphics settings window

In this window it is possible to set graphics resolution and whether the game should run in full-screen or in windowed mode. The change is performed immediately. If the resolution is not supported by user's graphic adapter, it will switch to windowed mode.

2.1.5 Players management window

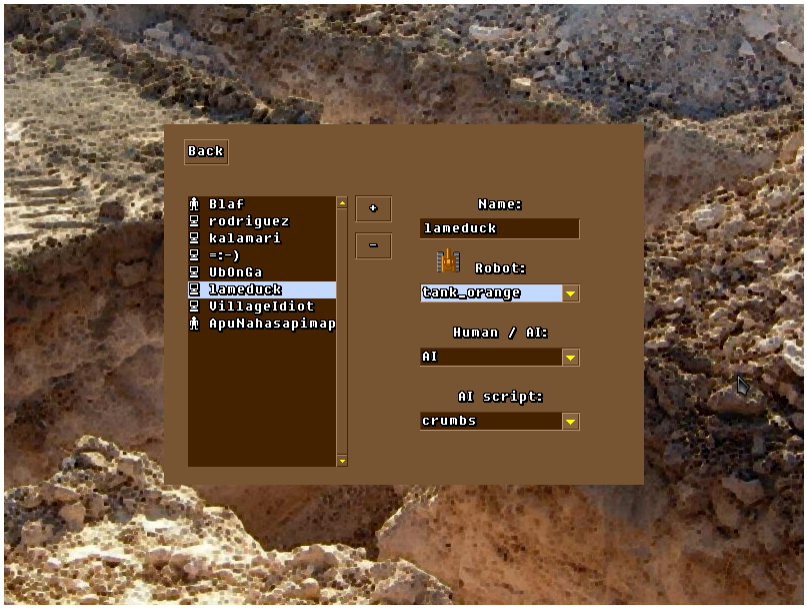


Figure 2.2: Players management window

In this window the user can add, remove and edit local players. These players will be then available in the game.

On the left side of the window there is a list of all local players. The "+" button adds a new player with initial settings to the end of the list. After clicking on a player, it is possible to delete the player from the list by clicking the "-" button or to modify his attributes.

The first attribute is player's name by which he will be recognized in games. Any white-spaces typed are discarded.

The second attribute is player's robot. Robots do not differ in anything but their picture.

The third attribute decides whether the player is artificial or human. When set to artificial, fourth attribute is uncovered.

The fourth attribute is a script of artificial intelligence. This script defines how clever and skilled the artificial player will be.

A few examples of available robots: 

2.1.6 Game creation window

This window enables user to create a game. A game has only two attributes, its title and the map which it will be played on. When selecting this map, a small preview may be displayed under the drop-down list. If the map was never played before, no preview will appear as it is generated when starting the game.

When the game is set up, pressing the "Create" button switches to game lobby window where players may be added. This window is described in 2.1.7.

2.1.7 Game lobby window

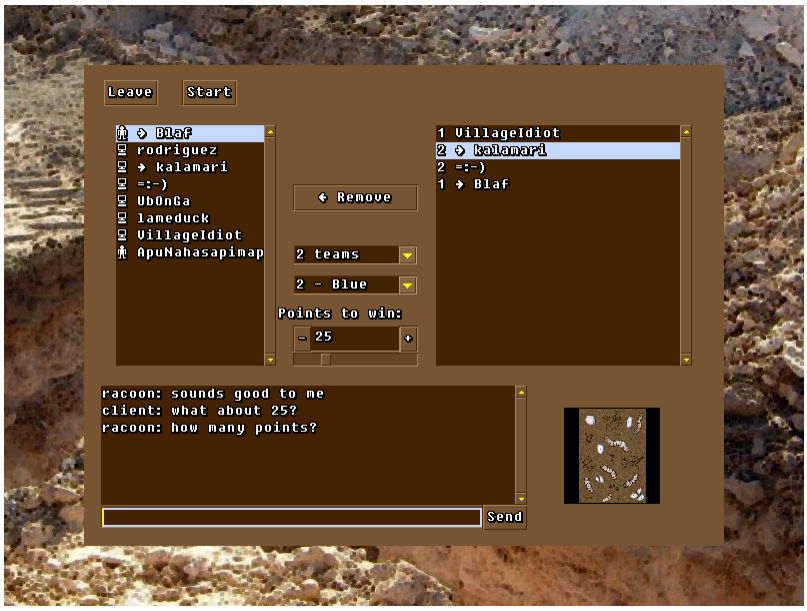


Figure 2.3: Game lobby window

In this window players are chosen to play the game and some additional settings are done. When the game is ready, it can be launched by the "Start" button. It is possible to start the game only if at least two players of different team (or no team) are in the game.

There is a list of local players on the left side of the window. Next to players' names there is a little icon informing whether the player is human or artificial. On the right side there is a list of players currently in the game.

Players in the game are highlighted by arrow in the left list. In network game, local players are highlighted in the right list as well. User can modify or remove only local players.

A player is added to the game by selecting and pressing the "Add" button. If the map has specified number of players and their homes' positions, an empty slot has to be selected when adding a player (except the first player who will be added to the first empty slot). If the map has unlimited number of players with random homes position, they are always added to the end of the list.

It is not possible to add more than two human players. If two are added, they will play in the split-screen mode. These modes are described in 2.2.2. If no human player is in the game, it will run in the spectator mode (described in 2.2.9).

Removing a player from the game is done oppositely to adding, by selecting him in the right list and pressing the "Remove" button.

There is a drop-down list between these lists where it is possible to select up to eight teams. A team is then assigned to a player by clicking and then selecting his team from the other drop-down list. Numbers next to players in the game indicate their team membership.

The last game modifier is the slider below drop-down lists which sets how many points a team or player needs to win. User (every user, in network game) can modify this value by dragging the slider or by clicking "-" and "+" buttons next to it.

In the network game, there is also a chat box underneath. Users connected to the game can communicate through it by writing messages and pressing Enter or clicking the "Send" button. The message then appears prefixed by the client's name in the text area.

2.1.8 Network connection window

The purpose of this window is to enable connecting to a server. Client name is the identifier by which other clients will recognize this user. Address and port are parameters of the server. Button "Connect" is used to establish the connection. If an error occurs, a message window is displayed and then the user is returned to this window. If the connection is performed successfully, game switches to the server lobby window described in 2.1.9.

2.1.9 Server lobby window

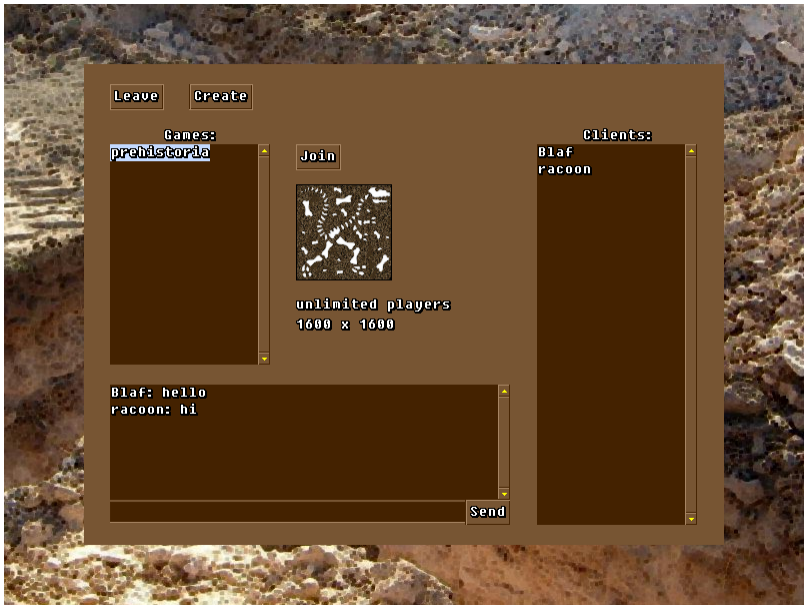


Figure 2.4: Server lobby window

In this window it is possible to communicate with other clients connected to the server, to connect to a game or to create one.

The server can be left using the "Leave" button. Next to it, user can establish a network game with the "Create" button. It leads to the window that was described in 2.1.6.

The list of games is on the left. Only games that had not started yet are visible. After clicking on a game, a preview of map is displayed (under same conditions as in 2.1.6) and if the game map is equal to a map stored at the local computer, the "Join" button appears. Equality of maps is tested by md5 hash of the map files, it is a precaution to avoid loose of synchronization. Joining the game using the "Join" button switches to the game lobby window that was described in 2.1.7.

On the right side there is list of all connected clients. Underneath there is a chat box. Clients in the server lobby can communicate through it by writing messages and pressing Enter or clicking the "Send" button. The message then appears in the text area prefixed by client's name.

2.1.10 Credits window

This window contains just a brief information about game version and names of its creators.

2.2 Gameplay

Figure 2.5: The look of the game with one human player

In this section the game itself is described, its rules, options, features and controls. When the user clicks the "Start" button, map is loaded, players and their robots are created and initialized and the game begins.

2.2.1 Maps

The chosen map determines the character of the game. Small maps that are easy to survey usually promise a dynamic and action game while large and maze-like maps are preferred by players who enjoy strategic thinking.

Every map consists of two layers. The first layer is a soil and in this layer the ditching takes place, it is the layer where tunnels are created by robots' movement and shots explosions. The second layer, called rock, cannot be destroyed by any kind of weapon and is impassable for both robots and shots.

There are two topological types of maps: planar maps and toroidal maps. Planar maps are rectangle-shaped and map boundaries cannot be trespassed as if everywhere around was rock. Toroidal maps have no boundaries, leaving the map at one side means appearing on the other side.

Players' homes may be placed on the map randomly or at specified points. Maps with random homes' placement have unlimited slots for players while maps with some number of defined homes positions have exactly the same amount of slots for players.

The last option of a map is the type of players' homes. There are three types of homes: circular with four exits, circular with two exits and no home (meaning no walls, it still refills health and energy). Homes are placed into the map before the game starts and all rock and soil is removed from inside the home.

2.2.2 Game screen

During a game, screen is divided into sections. If two human players are present in the game, the game runs in so-called split-screen, meaning that information for each human player is on one half of screen. Each player's view is divided into two parts: map view and status view. In the square map view all the doing on the visible part of the map is shown while in the status view there are indicators of health and energy (described in 2.2.4) and weapons and reloading (described in 2.2.6).

2.2.3 Map view

The map and the game activity are shown in map view and only the activity performed in visible part of map can be watched. In the center of the map view there is the controlled robot. Above are tiny health and energy indicators and below is written the robot's name in team color. These are displayed next to any robot which has its center in the map view.

If a network game is being played the size of the map view is set to the smallest size among all players. This is necessary because having bigger line

Figure 2.6: The look of the game in split-screen mode

of sight would mean a great advantage and therefore unfair conditions.

2.2.4 Health and energy

Health (armor, shield, life), with blue indicator and shield icon, expresses how much damage can the robot take before it is destroyed. It can be decreased only by a weapon and restored only in own or teammate's home.

Energy (power, fuel), with yellow indicator and lightning icon, expresses how much activity the robot can perform before it dies of energy failure. It is decreased when robot moves, shoots and slightly even if no action is performed. It can be even drained by a special kind of weapon. Energy is quickly refilled in own or teammate's home. In opponent's home, it is refilled as well, but at a lower rate.

2.2.5 Robot movement

The old Tunneler has very basic rules for movement. Robots are square-shaped and movement is possible only in eight basic directions according to the combination of pressed movement keys. Furthermore, the speed in oblique directions was higher than in axis directions, because the movement vector was a simple sum of axis vectors. This can be mathematically described as using metrics of absolute value instead of Euclidean metrics.

All these deficiencies are removed in Ditchers. The used metrics is Euclidean, meaning that speed is the same in all directions (in equivalent terrain). Robots can move in 36 different directions because movement keys do not directly determine movement direction, but Left and Right keys rotate the robot counter-clockwise and clockwise by 10 degrees and Up and Down keys move it forward and backward.

Movement speed is determined by the amount of soil and rock in the way. 'In the way' means that robot would cover it if he moved in current direction at sufficient speed. Every robot is approximated by a circle of a constant radius. Speed is counted in pixels per loop (ppl), but it does not have to be an integer. If no soil or rock is in the way, the robot moves at 5ppl forward and 3ppl backward. If a rock is in the way, the robot must stop before it would cover it, so the speed is decreased adequately. If some soil is in the way, it linearly decreases the speed by up to 4ppl, meaning that if fully surrounded by soil, robot moves forward at 1ppl and cannot move backward at all.

If a robot cannot move in the given direction because of obstructing rock, it tries to "slide" along it by checking whether it is possible to move in similar directions. Robot's orientation remains unchanged after sliding and sliding speed is decreased adequately according to the difference of the original and the sliding direction. This feature greatly improves playability because the player does not have to bother with insignificant obstacles. Sliding is depicted in Figure 2.7.

After robot's movement, all soil contained in the circle that approximates the robot is removed.

2.2.6 Weapons

Weapons may be used for both ditching and destroying enemies. They are rotated by a special key and there are also keys for selecting a specific kind

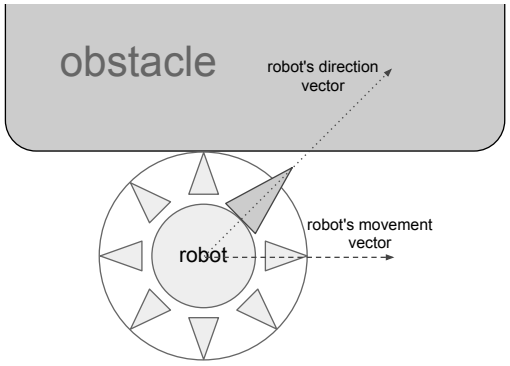


Figure 2.7: Image explaining robot sliding principle

of weapon (more in 2.2.10). Selected weapon is displayed on the bigger icon in the status view. Shots can be released with another key.

When a shot is released by a robot, it decreases robot's energy and starts moving in the direction of the robot at its speed. It moves until it existed for a predefined time or hit any obstacle – soil, rock or a robot. At the moment it explodes, ditches a hole into the soil and deals damage to nearby robots. The robot cannot release a shot if it does not have enough energy.

Weapons differ in several basic attributes and some have special features. Basic attributes are:

- Energy cost: how much energy is required to use this weapon
- Cadence: how quickly another shot may be fired
- Speed: how fast the shot is
- Time to live: how long does the shot exist before it blows automatically
- Ditch: how big hole the shot creates in the soil after explosion
- Splash: how far the damage spreads
- Damage: how much health it drains from the hit robot

Weapons attributes and features are listed in the table 2.1.

Image	Weapon	Energy	Cadency	Speed	TTL
	Machine gun	Low	Medium	Medium	High
	Rear rocket	Medium	Slow	Low	High
	Sniper rifle	High	Very slow	Very fast	Very high
	Flamethrower	Low	Highest	Medium	Very low
	Grenade	Medium	Slow	Low	Low
	Land mine	High	Very slow	None	Very high
	Incrush	Very high	Very slow	Low	Low
	EMP shockwave	High	Medium	Fast	Medium
	Recharger	Low	Highest	Fast	High

Image	Ditch	Splash	Damage	Special feature
	Small	None	Medium	
	Medium	Medium	High	Moves backward
	Tiny	None	Extreme	
	None	Small	Low	
	Medium	Medium	High	Bounces at least three times
	Large	Very large	Very high	Does not move
	Inverse	None	None	Creates soil incrush
	None	Medium	Low	Drains energy from robot
	None	None	None	Refills energy to robot

Table 2.1: Weapons' attributes and features

2.2.7 Score

Every robot has its count of points (frags) and deaths. When a robot A is destroyed by a shot released by a robot B, the robot A increments its deaths count and the robot B increments its points count. If a robot is destroyed due to energy failure or with its own shot, its deaths count is incremented but his points count is decremented. This way robots may reach negative number of points. It is a precaution to discourage players from suicide missions.

When a human player's robot is destroyed, a table containing these counts of points and deaths is shown in the middle of the map view.

Game ends when a robot (if no teams are in the game) or a team (in the team game) reaches the predefined number of points. Team's points is simply sum of points of all its members' robots. Deaths counts have no importance.

2.2.8 Chatting and logging

Players have possibility of chatting during the game. There are two types of chatting, global and team-only, and these types are invoked by different keys. In split-screen, both players share the global chatting but each can communicate with his own team separately. Chat messages appear in lines from the top of the screen.

When a robot is killed, a log message specifying killer, victim and deadly weapon appears on the screen the same way chat messages do.

2.2.9 Spectator mode

If no human player is added by the user, there is no robot controlled by the user. It is a special mode of the game called spectator mode which is used to observe artificial players and/or network players.

In the spectator mode, it is possible to switch between watched robots by pressing a key and it is also possible to watch two robots simultaneously in split-screen by pressing another key. These keys are specified in 2.2.10. Naturally, the spectator may have smaller line of sight than the observed robot.

Spectator can send global chat messages but he cannot send team-only messages (as he is not part of any team and knows strategic information about all robots).

2.2.10 Controls

First group of keys used in the game are movement keys that rotate and move robot both forward and backward in the sense of 2.2.5.

Second group are keys managing weapons and shooting. There is a key for releasing a shot, switching to next weapon and there are also auxiliary keys that switch to a specific one of the nine available weapons. They work as macros – are implemented as pressing the weapon switching key until the desired weapon is selected.

Third group are keys that manage chatting, one key for global chatting (G) and one key for team-only chatting (T). After pressing such key it is not possible to control the robot until the chat message is written and sent by pressing the Enter button.

The F key switches the game between full-screen and windowed mode and Esc key exits the game.

List of keys used by a player if he is the only human player:

Movement	Fire	Next weapon	Weapons	Chat G/T
Arrows	Enter	Backspace	123456789	M/N

List of keys used by players in splitscreen mode:

Player	Movement	Fire	Next weapon	Weapons	Chat G/T
Right	Arrows	Enter	Backspace	Special keys*	M/N
Left	WSAD	Tab	"`" key	123456789	M/C

* Special keys: RightShift, Insert, Delete, Home, PageUp, PageDown, End, "[", "]"

List of keys used by a spectator:

	Main / right view	Left view	Split-screen	Chat
Spectator	Enter	Tab	Backspace	M

Chapter 3

Implementation

The game is implemented in C++[1] using SDL graphics library[8]. I decided to use SDL because it has a friendly API and supports hardware accelerated surfaces blitting. There is a number of free libraries for GUI and I decided to use Guichan[4], which seemed to be exactly the kind of library I needed – a lightweight portable C++ GUI library designed for games using SDL.

3.1 Settings and data

Game settings and data are stored in XML files and images in directory hierarchy under `/usr/share/games/ditchers` directory in case of installed version or `./data` directory in case of compiled-only version. Both versions also use directory `~/.ditchers`. Subdirectories prefixed by the dot or the underscore are skipped. All settings are loaded into memory during application start.

User should feel encouraged to add new robot models, maps and AI scripts, so the intention is to make it as easy as possible by storing information in well-structured XML files. Pictures of weapons are not supposed to be modified by a casual user, so I'll just mention that their pictures are stored in `weapons`.

3.1.1 Global

In configuration file `settings.xml` there is information about last used graphics settings (resolution and fullscreen) and about last used network settings (client name and server address and port).

These settings are written into the file when leaving the GUI window where they may be changed, the graphics settings window and the connection window.

Example of `settings.xml` is in Figure 3.1.

```
<?xml version="1.0" ?>
<settings>
    <network client="racoon" host="localhost" port="8421" />
    <graphics width="800" height="600" fullscreen="false" />
</settings>
```

Figure 3.1: settings.xml

3.1.2 Players

Information about local players is stored in the file `players.xml`. Each player has name, robot type according to 3.1.4 and control, which defines player's artificiality and has only two allowed values – AI and human. If control is AI, fourth attribute, AI script set according to 3.1.5, is required.

These settings are written into the file when leaving the Players window in GUI.

Example of `players.xml` is in figure 3.2.

```
<?xml version="1.0" ?>
<players>
    <player name="Racoon" robot="mole" control="human" />
    <player name="Squirell" robot="tank_yellow" control="human" />
    <player name="Joe" robot="spider" control="AI" script="default" />
    <player name="Jimbo" robot="ufo" control="AI" script="stupid" />
</players>
```

Figure 3.2: players.xml

3.1.3 Maps

Maps are stored under directory `maps`. Each map has its own subdirectory containing all map data. Each map consists of up to three images of the

same size and optionally an XML file. Those three images with file names `base.png`, `soil.png`, `rock.png` define three layers of the map.

Image `base.png` is underlying and has no but aesthetical meaning while `soil.png` and `rock.png` define the colors and shape of soil and rock layer. The part of picture that is supposed to be transparent must be painted with the magic pink: RGB(255, 0, 255).

The XML file, if present, must have name `map.xml`. It contains Information about map name, identification, author, size and topology. Name and identification are any strings without spaces, topology is either torus or plane. Attribute basetype may have one of three values: 4-way, 2-way and none and defines the shape of players' homes. Optionally, if attribute blob is set to false and limit is set to a number, the file may contain also list of players' homes (bases). Number of these must equal the limit and this limit determines number of players slots in a game that uses this map.

In map creation window the name attribute is used to identify the map. If no such attribute exists, map's directory name is used.

Example of `map.xml` is in Figure 3.3.

```
<?xml version="1.0" ?>
<map name="Zoo" unique="racoon-2-v1.3" topology="torus">
    <size width="1200" height="1200" />
    <players limit="4" blob="false" basetype="4-way">
        <base index="0" x="200"  y="200"  />
        <base index="1" x="200"  y="1000" />
        <base index="2" x="1000" y="1000" />
        <base index="3" x="1000" y="200"  />
    </players>
</map>
```

Figure 3.3: map.xml

3.1.4 Robots

Pictures of robots are stored under the directory `robots` and each robot has its own subdirectory with one file `robot.png`. This image is supposed to have size 23×23. Robot's directory is used to identify it and every player must have its robot correctly set.

3.1.5 Scripts

AI scripts are stored under the directory `scripts` and each script has its own subdirectory with file `main.lua`. Scripts directories are used to identify it and every artificial player must have its script correctly set.

Scripts of artificial intelligence are described thoroughly in section 4.1.

3.2 Architecture

In this section I will mention some main classes, relations between them and basic principles how the application works.

3.2.1 Main classes

There are five important classes that wrap global variables and methods: `Gfx`, `Settings`, `UserFace`, `GamePlay` and `Network`. These classes are instantiated during application start-up and there is exactly one instance of each. Besides these, there are only a few very simple (e.g. arithmetic) global functions.

Names of these classes are quite self-explaining. `Gfx` contains attributes related to graphics settings (e.g. screen resolution) and methods working with images (e.g. for retrieving the color of a specified pixel). Class `Settings` provides methods for loading game data and settings and contains data structures to store these. `UserFace` is shortening for User Interface and is dedicated to everything related to GUI. Class `GamePlay` only cares for playing the game and `Network` contains only attributes and methods related to networking.

Relations between main classes are drafted in Figure 3.4.

3.2.2 Application progression

After basic initialization of SDL the splash screen is loaded and shown. While the user is watching it, main classes initialize and settings are acquired from files. When it is done, the GUI loop begins.

In this loop there are events handling and GUI displaying performed. If some settings are changed, they are saved to files immediately. For actions like connecting to a server or changing game settings according to the network data the `Network` class is used. `UserFace` class contains information

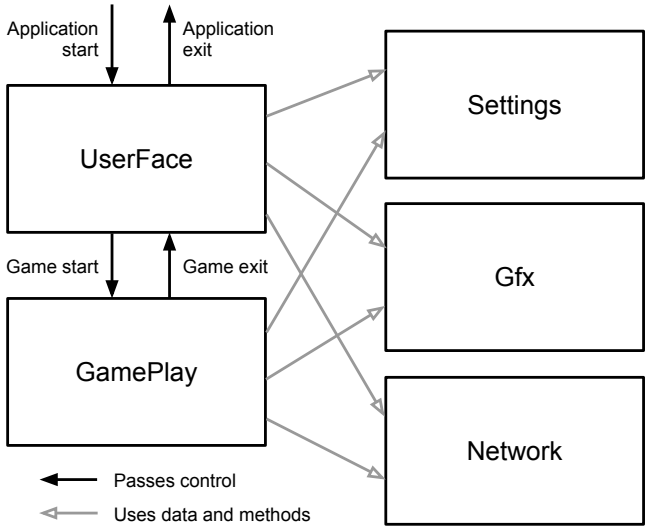


Figure 3.4: Main classes and relations between them

about the window currently visible, has a method for displaying a message box and a method for launching the game.

Once the game is launched, `GamePlay` class is in charge of the application. After the game is initialized the game loop begins. This loop has three basic duties: receiving user or network input, changing game state and displaying graphics. It also has to keep correct FPS. These tasks are performed in this order (network tasks are skipped in local game):

- Keyboard input, AI thinking
- Sending information to the server – players' actions
- Displaying graphics
- Delay to keep 25 FPS
- Input from the server – actions of network players
- Compute next game state

This order was selected to minimize the time spent waiting for network information to come as there are both graphics displaying and FPS delay

between sending and receiving network data. The gameplay and the class `GamePlay` are also described in Figure 3.5.

When a user decides to end the game, it returns control to the `UserFace` class and GUI loop. When the user clicks the "Quit" button in the main GUI window, the loop is left, memory cleaning performed and the application exited.

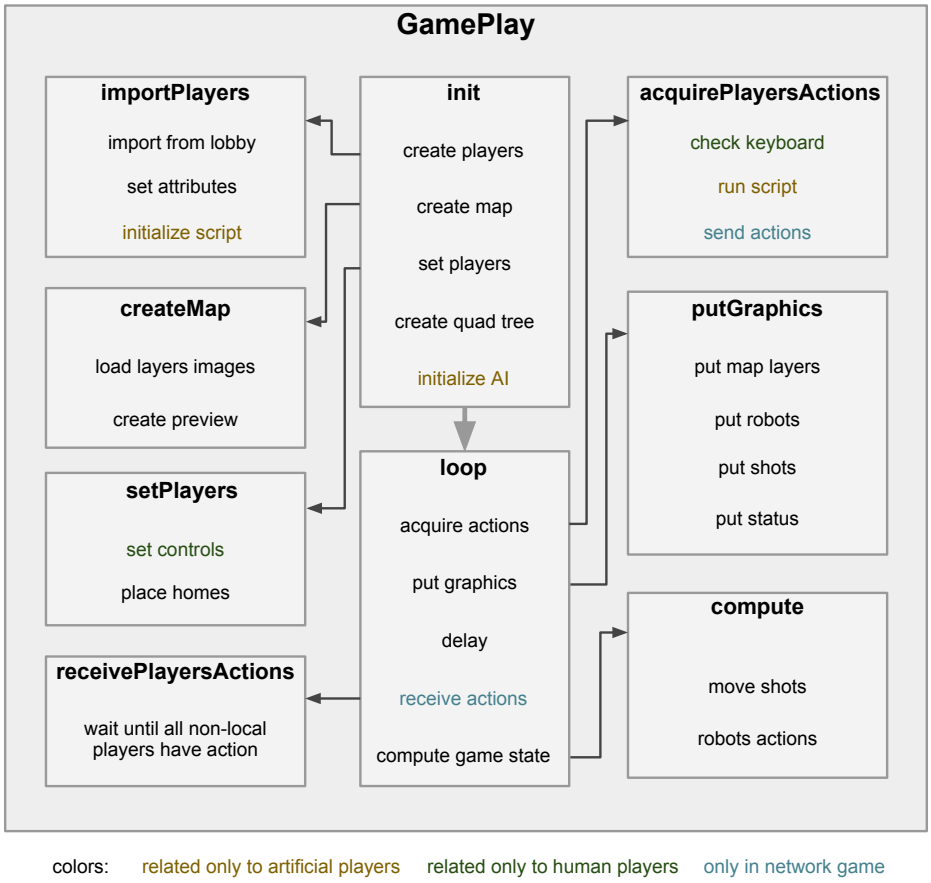


Figure 3.5: Progression of the gameplay

3.3 Network communication

The main decision of the designer of a client-server application concerns allocation of work between the client part and the server part.

3.3.1 Paradigms

There are two basic paradigms: the first is keeping clients very simple only to acquire input, to send it to a server, to receive results and to display output, while server computes a new application state. The second is having a simple server that keeps only the essential data and for the majority of time only redirects messages while clients do the computation of new state. There are also many approaches in between these paradigms that lean towards the former or the latter.

I have chosen to follow strictly the latter paradigm. The server cannot compute states of all its games – it would put unbearable demands to both CPU and memory as any number of games may be played simultaneously. There are also disadvantages I have to cope with.

The main disadvantage (or group of disadvantages) is synchronization. While clients have all the game data and do not share them with the server, the synchronization might break down if not all clients have exactly the same. Synchronization is discussed thoroughly in 3.3.2.

Another disadvantage is cheating – the client application has all the data about the game and therefore if someone decided to alter its code he would get a tactical advantage, e.g. knowing location of other robots. I decided to ignore this with the hope that all players of this game are fair. (To make it clear, more serious cheating like enhancing weapons or robot's abilities is not possible as it would yield synchronization problem immediately.)

3.3.2 Synchronization

To avoid loose of synchronization, md5 hash of maps are compared when joining a network game – different versions of maps would most probably yield the OOS (Out Of Synchronization).

Sometimes a random number generator (RNG) may be needed in the game, e.g. to decide whether the grenade should bounce or explode. While this decision is made at client's side, the RNG must be synchronized as well. This is solved by using own implementation of RNG which yields a perfectly same progression of numbers according to a seed that is randomly generated by the server and sent to clients when the game begins. (In a local game, the seed is generated locally). This seed is required to avoid same progressions of numbers in every game. AI scripts might need randomization as well, but the generator is not shared with the game itself.

If the network player's robot type is not found, it is replaced by some local robot type. As difference of robot types does not affect the gameplay, it is a sufficient solution. AI script of a player is only required with the client whose player it is, so there is no concern of synchronization. The only other synchronization problem could be yielded by a difference in application versions. It cannot be recognized from the received data because the only information the server receives about the game state are bit masks of six main action keys. Therefore, to recognize the OOS, the server receives control information – coordinates of all players from all clients – every 25 game loops (one second). If any sync problem occurs, players' coordinates soon does not match and in this case the server cancels the game.

3.3.3 Communication protocol

The data between clients and the server are transferred using the TCP protocol to avoid concerns for session managing and to ensure all the data is received in the same order as they were sent. Messages are composed using a special text protocol. No maximal length of message is defined except limitations of TCP protocol and C++ string, because a robust system of escape characters and network buffers concatenation is used to retrieve the whole message.

Any communication between the client and the server is initiated by the client. Message types differ in the way the server responds to messages of the type. Server may not respond at all (e.g. to synchronization data), may respond to all clients connected to the game (e.g. to adding of a player), or to all clients connected to it (e.g. to connection of a client).

A message consists of two command letters and a data part. Command letters define the type of the message. The first letter puts the message into one of three domains: server lobby, game lobby or gameplay. The second command letter specifies the message type in the given domain.

The list of message types by command letters (CL), contents of their data parts and ways the server responds follows. The first command letter is `s` for server lobby, `l` for game lobby and `g` for gameplay. Table rows are grouped by client messages, "gamecast" means game broadcast.

	CL	Meaning	Data part
Message	`sn`	Connected to server	name and screen resolution
Broadcast	`sw`	List of clients	clients' names and ID
Broadcast	`sg`	List of free games	games' names and map attributes

Broadcast	`sn`	New client	new client's name and ID
Reply	`si`	Your ID	client's ID
Message	`sc`	Creating a game	name and map attributes
Broadcast	`sa`	New game	game name, ID and map attributes
Message	`sj`	Joining a game	game's ID
Reply	`sj`	Game joined	game's ID
Reply	`ll`	Points limit	game's points limit
Reply	`lt`	Teams count	game's teams count
Reply	`lp`	Players in game	list of players and their attributes
Message	`sl`	Game left	–
Broadcast	`sr`	(Empty game removed)	game's ID
Message	`sm`	Server lobby chat	chat message
Broadcast	`sm`	Server lobby chat	chat message
Message	–	Disconnected from server	–
Broadcast	`sq`	Client left	client's ID
Message	`ll`	Set points limit	new limit
Gamecast	`ll`	Points limit set	new limit
Message	`lt`	Set teams count	new teams count
Gamecast	`lt`	Teams count set	new teams count
Message	`lc`	Change player's team	player's ID and team ID
Gamecast	`lc`	Player's team changed	player's ID and team ID
Message	`l+`	Add player	player's name, robot and position
Gamecast	`l+`	Player added	player's attributes, client's ID
Message	`l-`	Remove player	player's ID
Gamecast	`l-`	Player removed	player's ID
Message	`lm`	Game lobby chat	chat message
Gamecast	`lm`	Game lobby chat	chat message
Message	`ls`	Start game	–
Gamecast	`ls`	Game started	RNG seed and game resolution
Broadcast	`sr`	Game no longer free	game's ID
Message	`gc`	Player's action	timestamp, player's ID, action
Gamecast	`gc`	Player's action	timestamp, player's ID, action
Message	`gt`	Player's robot's coords	player's ID, coords
Gamecast	`gx`	(Out of sync – game ends)	"Out of sync" message
Message	`gm`	In-game chat	chat message
Gamecast	`gm`	In-game chat	chat message
Message	`gn`	In-game team chat	chat message
Gamecast	`gn`	In-game team chat	chat message
Message	`gx`	Abort game	–
Gamecast	`gx`	Game aborted	"Aborted by a client" message

3.4 Data structures

Besides trivial data structures and structures provided by the C++ Standard Template Library it was necessary to implement a few more for use in special situations. One such situation is storing information about games and clients at both server and client with a low time complexity and other is managing quick interval queries to the game map.

3.4.1 Hashmap wrapper, hashmap-vector wrapper

Lets discuss the situation we have at the server. Every client and game on the server and every player in a game receives its own ID which uniquely identifies it. IDs are assigned incrementally from 1 to a very high number and then again, skipping used ones. Naturally, we wish to be able to quickly retrieve any of these by its ID. Therefore using a binary search tree seems to be a good idea. To encapsulate all operations involving assigning the ID and freeing it correctly, STL balanced binary search tree `map<int, T>` is wrapped into a data structure `WrapMap<T>`. Lists of games and clients are implemented this way in both client and server part of application. This and the following data structure are implemented in the `template.hpp` file. The average complexity of STL map is not affected by this wrap.

Table with time complexity of required operations follows. Listing means iterating over the whole list.

Operation	Add	Remove	Get pointer by ID	Listing
Complexity	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

The situation with players is even more complicated. Not only that players receive IDs, but they also may have a specified index in the list of players' slots and it is useful to be able to access players by this index. For this case even more sophisticated structure was created: `WrapBoth<T>` extends the `WrapMap<T>` by a STL vector and accessing by index is done using the vector while accessing by ID is using the WrapMap. This way a good time complexity is guaranteed for most operations.

Table with time complexity of required operations follows, n is the number of players in the game. By index and by ID means retrieving the pointer, listing means iterating over the whole list.

Operation	Add	Remove	By index	By ID	Listing
Complexity	$O(\log n)$	$O(n)/O(\log n)$ *	$O(1)$	$O(\log n)$	$O(n)$

* $O(n)$ in case of unlimited players (shrinking vector needs linear time), otherwise $O(\log n)$

3.4.2 Map two-layer quadrant tree

The game map was first kept only as three layers of images and, with constant time for determining a pixel value, this way was sufficient for all required computations until AI scripts support was implemented. The reason why the need for a new approach emerged was the difference in artificial and human perception of the map. While human player recognizes large areas with or without soil or rock in virtually no time, if an artificial player had to do the same recognition pixel by pixel, it would need an unbearable amount of time. Obviously the AI may need this kind of recognitions very often and for this purpose a quadrant tree is built during the game initialization. It is implemented in `quadtreetemplate.hpp` and `quadtree.hpp`.

In a quad tree, the given rectangular area is recursively divided into four quadrants until these are homogenous. In our case, homogenous means that the whole area is filled with rock or with mud or is empty. Every node of this quad tree has two integer values – soil coverage and rock coverage, expressed in pixels. In a leaf, these values are either zero or equal to the size of the given rectangle. In non-leaf nodes, this value is easily computed as the sum of sub-nodes' values during backtrack, thus the whole structure can be built in a single pass. If n is number of pixels in the map, the quadrant tree can be built with time complexity $O(n \log n)$ – there are at most $\log n$ levels of nodes and checking homogenity of a node is linear to its size. This complexity is not crucial as it is performed only once before the game starts.

While the rock layer is immutable during the game, the soil coverage must be updated when a robot ditches or a shot explodes, but this does not demand too much time because these changes are always only minor. For these purposes, the quad tree structure has a method for updating a rectangle.

Methods provided by this structure are following:

- Homogenity test: checks whether the given rectangle is homogenous in following ways: whole empty, soil only, rock only, no rock, no empty.
- Coverage: checks how big part of the given rectangle is covered by either rock or anything (rock or mud).

All these methods have a good time complexity $O(\log n\sqrt{k})$ where n is the number of pixels in the map and k is the longer side of the rectangle. These methods will be mentioned in Section 4.2.

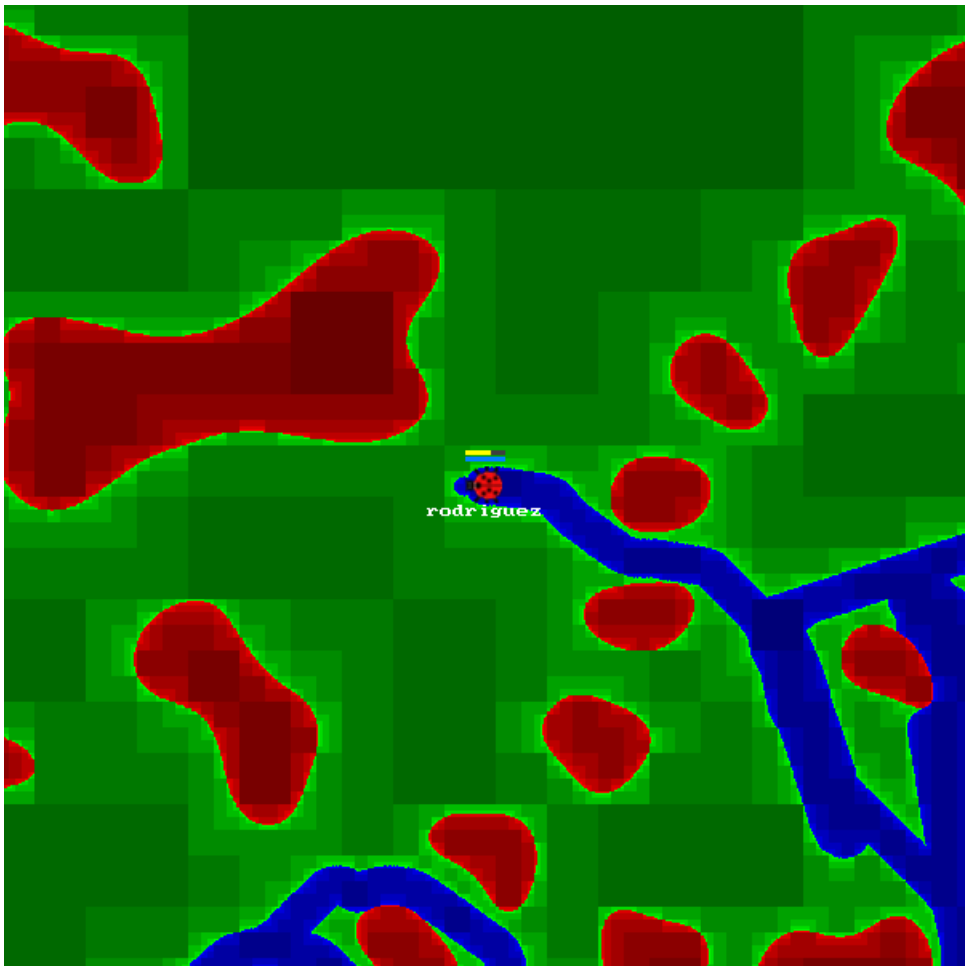


Figure 3.6: Map represented in the quad tree
Each node refers to a square in the map.
The lighter the square is, the lower is its leaf node in the structure.

Chapter 4

Artificial players

In a lot of action or strategic games artificial players ("computers", bots) may be present in the game. Sometimes it is even possible to choose between a number of levels of intelligence, but these intelligences are usually hard-coded and the user have no option of altering it except for changing the source code (in case of open source programs). In Ditchers, not only that artificial intelligence is easily alterable, but the user should feel encouraged to do so or even to write his own AI script.

4.1 Basics

Among many scripting languages I chose to use Lua[6]. This language is very easy to learn, has many intriguing features that are especially valuable in case of AI scripts (associative arrays, coroutines) and also has a very lightweight interpreter. Lua is used in a number of well known games – World of Warcraft, SimCity 4, FarCry and more.

AI scripts are saved in the directory `scripts` and each scripts has its own subdirectory. The file `main.lua` must be present in this directory. The directory may also contain any number of other `.lua` files and to use them they must be loaded using the Lua `require` command which has similar function as the C preprocessor `include` command. The `require` path is set to the script's directory. There must be a function `main` implemented in `main.lua` or in any file required by it.

When the game is launched, an interpreter is loaded for every artificial player and the scripts are initiated. During this game, when players' input is demanded by the application the `main` function is called. The expected

result is a bit mask of six main action keys, the same as the one that is sent to server.

4.2 Interface

Besides easy creation of a script it is necessary to have a good interface between the gameplay and the script. By good I mean easy to use and providing the as similar options to a human player has as possible. This may sometimes be quite a problem due to obvious differences between human and AI approach and therefore compromises have to be made.

There is a number of interface functions in the application that may be called by a script. These functions provide information about the game state – map, robots, shots etc. and usage of these functions is the only way game information may be retrieved. Lua allows functions to return multiple results or even unspecified number of results, and this feature is widely used. Some functions would not return values in case the scripted robot should not know the requested information. This happens if it refers to a part of map or another object that is not in sight of the robot.

List of interface functions, divided into several categories, follows. Some of these functions return quite raw data and there are methods in pre-implemented scripts that use them so the user does not have to reinvent the wheel, he might include these methods into his own script.

4.2.1 Game settings

These functions return application or game constants, thus might be used only once during initialization. The reason why application constants are retrieved this way is that some of them may change slightly in another version and a well-written script should adapt without the need of rewriting.

- `getradius()` – returns integer, integer – radius of a robot and hole appearing after robot destruction
- `getrotcount()` – returns integer – number of possible rotational directions
- `getmaxgrave()` – returns integer – how many loops does a stain last at the point of destruction and also how long does it take after stain disappearing before robot respawn

- `getmaxprotection()` – returns integer – how many loops is a robot invulnerable after respawn
- `getweaponscount()` – returns integer – number of weapon types
- `getshottype(integer index)` – returns 9 integers carrying information about the indexed shot type:
  1. how much energy the shot drains from a firing robot
  2. how long does it take to reload after shooting (in game loops)
  3. speed (pixels per turn)
  4. ditch – how big hole it creates when explodes (radius of hole in pixels), negative number means creating a terrain instead of removing
  5. splash – distance from the explosion needed to avoid damage
  6. damage – how much health it drains from hit robots, negative number means restoring health
  7. energy damage – how much energy it drains from hit robots, negative number means restoring energy
  8. bounce count – number of bounces before exploding
  9. live time – how long does the shot exist before automatic explosion (in game loops)
- `getmaxstatus()` – returns integer, integer – maximal energy of robot, maximal health of robot
- `getmapinfo()` – returns integer, integer, boolean – map width in pixels, map height in pixels, whether map is torus
- `getsight()` – returns integer – number of pixels from view center (middle of robot) to view border
- `getplayerscount()` – returns integer – number of players in the game
- `getteamscount()` – returns integer – number of teams

4.2.2 Game state

These functions return variables that may change during game.

- `gettime()` – returns integer – number of loops since the start of the game

- `gethomes()` – returns table of records containing information about visible homes; each home fills 2 rows of the table with doubles, coordinates of the home

- `getdeaths()` – returns table of records containing information about visible death stains; each stain fills 3 rows of the table with two doubles and an integer, coordinates of the stain and how fresh the stain is (appears with value returned by getmaxgrave() and goes to zero)

- `getshots()` – returns table of records containing information about visible shots; each shot fills 8 rows of the table:

1	boolean	whether shot was fired by an enemy
2, 3	double, double	coordinates
4	integer	speed (pixels per turn)
5	integer	angle
6, 7	double, double	direction vector (normalized)
8	integer	index of shot type

- `putchat(boolean teamonly, string message)` – returns boolean – whether the message was sent successfully, function puts a message to game chat; if no teams are set and teamonly is set, message is not accepted. Every player has a letter limit for team-only messages; it is 8 at the start of the game, is increased by 1 every loop and decreased by length of a sent team message.

- `getchat()` – returns table of records containing information about chat messages sent from beginning to previous loop; each message fills 3 rows of the table:

1	string	message content
2	integer	time (in game loops) when the message was sent
3	integer	index of the player who sent the message – is -1 if the message is not to teammates only (the idea is that public messages don't contain important data)

- `getlog()` – returns table of records containing information about log entries made from beginning to previous loop; each entry fills 4 rows of the table:

1	integer	time (in game loops) when the entry was made
2	integer	who killed someone (player index)
3	integer	who was killed (player index)
4	integer	what weapon was used (shot type index)

- `getlog(integer timestamp)` – as previous, but contains only entries made at time timestamp (if set too big, changed to time of last loop)

- `getlog(integer timestamp1, integer timestamp2)` – as previous, but contains only entries made in interval [timestamp1, timestamp2]

- `at(integer x, integer y)` – returns integer code of terrain at point x,y; if map is a torus, x and y does not have to be in the map size rectangle.
  Terrain codes:
  nil point not visible, cannot decide
  0 free
  1 soil
  2 rock or not in the map (non-toroid)

- `hom(integers x1, y1, x2, y2, integer code)` – returns integer information whether the rectangle is homogenous according to code, returns 1 for true, 0 for false and nil if is not whole visible. If map is a torus, x1, y1 might be bigger than x2, y2 resp., the rectangle lays over map edge; in non-toroid maps, such rectangle would be homogenous (zero surface). Homogenity codes:
  0 ..... free only
  1 ..... terrain only
  2 ..... rock only
  3 ..... free or terrain, but no rock
  4 ..... terrain or rock, but no free

- `cov(integer x1, y1, x2, y2, integer code)` – returns integer computing how many pixels of the given rectangle are covered with anything – mud or rock. Returns nil if not whole rectangle is visible.

- `covr(integer x1, y1, x2, y2, integer code)` – returns double between 0 and 1 computing what part of the given rectangle is covered with anything – mud or rock. Returns nil if not whole rectangle is visible.

- `rock(integer x1, y1, x2, y2, integer code)` – returns integer computing how many pixels of the given rectangle are covered with rock. Returns nil if not whole rectangle is visible.

- `rockr(integer x1, y1, x2, y2, integer code)` – returns double between 0 and 1 computing what part of the given rectangle is covered with rock. Returns nil if not whole rectangle is visible.

4.2.3 Robots status

These functions return information about scripted robot or other robots

- `getmydesignation()` – for the scripted robot, returns integer – player index string – name integer – team index
- `getdesignation(integer index)` – for the robot specified by index, returns string – name integer – team index
- `getvisible(integer index)` – returns boolean – whether indexed robot is alive and in sight
- `getmyinfo()` – for the scripted robot, returns:

double, double	coordinates
double, double	direction vector (correlated to angle)
integer	angle in range [0, 360) from 0 = "up" clockwise
double	speed
integer, integer	current health and energy
integer	returns how long the robot will be protected (after respawn)
integer	returns how long the robot is dead / until respawn
integer	returns current weapon's index
integer	returns how long until a new shot may be released

- `getinfo(integer index)` – for the indexed robot, if it is visible, returns:

double, double	coordinates
double, double	direction vector (correlated to angle)
integer	angle in range [0, 360) from 0 = "up" clockwise
double	speed
integer, integer	current health and energy
boolean	returns whether the robot is protected (after respawn)

4.3 Implemented scripts

4.3.1 Generic

There are several useful scripts in the `_generic` directory. It contains the `main` function but it does not do any real work, it is not a complete script, there are only some auxiliary functions. Some, like those for acquiring data or setting action mask, should be adapted by any script, some other, like path finding, are rather examples of more sophisticated algorithms. List of functions grouped by files follows.

- `init.lua` contains example of initialization part of the script. Initializes application and game constants, scripted robot's attributes and a few more useful global variables. Commands are outside any function thus will be performed during script initialization – upon game start. Some less essential initialization is done in the `acquire.lua` and `mask.lua`.

- `acquire.lua` contains functions for acquiring objects in the sight of the scripted robot– robots, shots, homes, death stains and also chat and log messages. Global variables set by these functions are initialized here. These functions may be used in every loop to refresh the sight.

- `aux.lua` contains some very basic useful mathematical functions like signum, absolute value and distance of points.

- `mask.lua` contains functions for comfortable setting of the action mask.

- `main.lua` contains the example of the `main` function with necessary inclusions using the `require` command. In the `main` function scripted robot's attributes and objects in sight are updated, game time is acquired and the actionmask resetted. After a gap for the intelligence itself some saving of useful data and returning the action mask is performed.

- `path.lua` contains non-essential functions making the movement in the map easier. There are some auxiliary functions like going straightly from one point to another, but the longest and most important function is for finding a path from one point to another. This function uses a special approximative version of Dijkstra algorithm [3]. For this function, `blocks.lua` and `heap.lua` have to be included.

- `blocks.lua` contains functions for approximating the terrain by a grid of blocks. These blocks are either impassable if containing rock or passable with a various levels of soil coverage. The visible part around initial position is checked upon game start and then only a newly uncovered line at the view border is updated when moving. These blocks are then used to compute shortest path whenever needed. A preview how do scripted robots see the map is in Figure 4.1.

- `heap.lua` implements binary heap data structure. It is required by the path finding function, but is usable for any other applications as well.

4.3.2 Stupid

This is a very, very simple script. The returned action mask is achieved by random virtual pressing and releasing of keys. This script is certainly not recommended for any serious use, it is rather an example to show how little is required to add a script to the game.

4.3.3 Crumbs

A robot using this script is capable of basic exploring and fighting, but its path finding method is too simple. It moves in a random direction while it can and remembers every point of direction change. When it has low energy, it returns in the manner of visiting those points in reverse order. The Crumbs title refers to those points in the meaning that the robot drops bread crumbs on every corner while exploring and returns by picking them.

4.3.4 Pathfinder

This script actively uses the pathfinding script described in 4.3.1. Therefore, if it knows a sufficiently big part of the map, it can find a path from any to any place – except for cases when it is not possible or if pathways are extremely tight (it is a limitation of the algorithm). Though approximated, the algorithm may need more time than available to find the path; for these reasons, coroutines are used in the script.

Coroutines allow script to stop executing at a point and reenter at the same point. This is exactly what we need – searching the path may take several seconds and it would be very difficult to implement the algorithm the way it would not need too much time in one loop if we could not use

coroutines. There is an unpleasant limitation to Lua coroutines – when inside a coroutine, interface functions cannot be called. Thus it is wise to realize it when outlining the script as it is necessary to acquire all required game data before entering a coroutine.

Using coroutines in the script is quite easy to implement and there are several examples in the Pathfinder script.

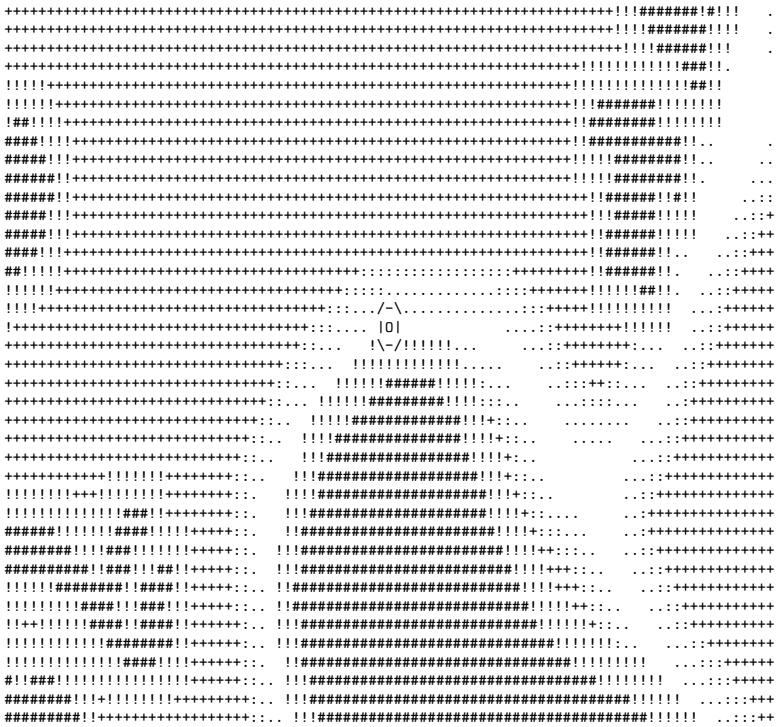


Figure 4.1: Printout of how the pathfinding script approximates the map

Chapter 5

Installation

The application may be compiled and installed on Linux. It requires installing development versions of the following necessary libraries:

- SDL: `libsdl1.2-dev`
- SDL_image: `libsdl-image1.2-dev`
- SDL_net: `libsdl-net1.2-dev`
- SDL_gfx: `libsdl-gfx1.2-dev`
- png: `libpng12-dev`
- guichan: `libguichan-dev`
- TinyXML: `tinyxml-dev`
- boost_filesystem: `libboost-filesystem-dev`
- lua5.1: `liblua5.1-0-dev`

The `make` and `g++` compiler are also needed. After running `make` in `ditcher` and `ditchs` directories the client and the server are created in respective directories. The `install.sh` script does all the work – checks for library dependencies, compiles the project and installs it.

Furthermore, the game had such a positive response in the community that Jan Jeroným Zvánovec [5] decided to create Debian packages.

The application has its own homepage [2] with a link to the newest version available.

Chapter 6

Corrolary

6.1 Comparison to similar projects

Here I try to compare Ditchers with the most similar action games I found.

6.1.1 Tunneler

The old and legendary Tunneler was overcame in every way. Ditchers has better graphics, more game features like choice of weapons, network game and artificial players.

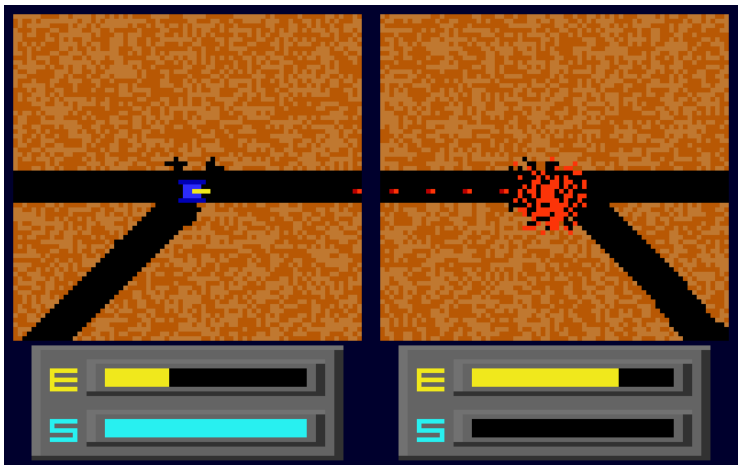


Figure 6.1: Tunneler

6.1.2 GM Tunneler

The GM Tunneler is a remake of the old one, and it very strictly sticks to the model. Thus, when it comes to evaluation, similar limitations as those of the old Tunneler are found. Single weapon, only local game, only two players and no artificial ones. In Ditchers, playability is the one of main goals and Tunneler was rather an inspiration, and it greatly overcomes GM Tunneler in many ways.

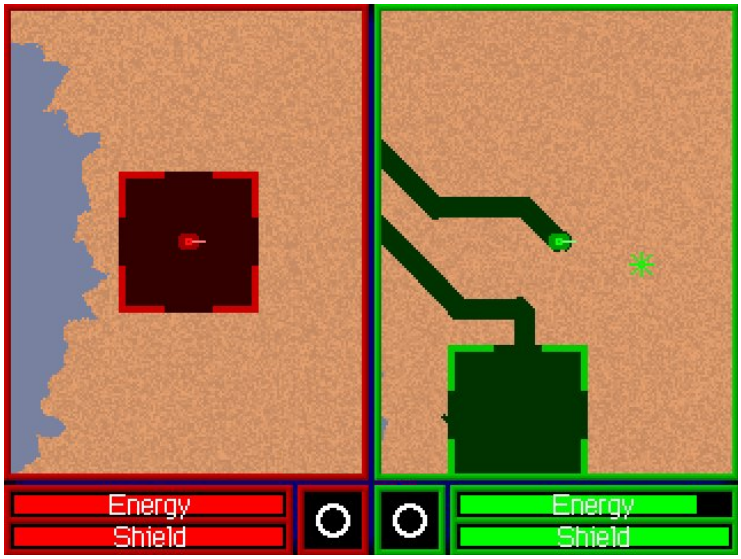


Figure 6.2: GM Tunneler

6.1.3 Liero

Liero is not really a remake of Tunneler though it is similar in the idea of digging in the ground. Liero resembles Worms in real time and it has many nice features – it has a large choice of weapons, it is very action and supports network game. The artificial intelligence is very weak, though. It rotates weapons rather randomly and has no sophisticated methods of locating the opponent. Furthermore, there is no option of programming it.

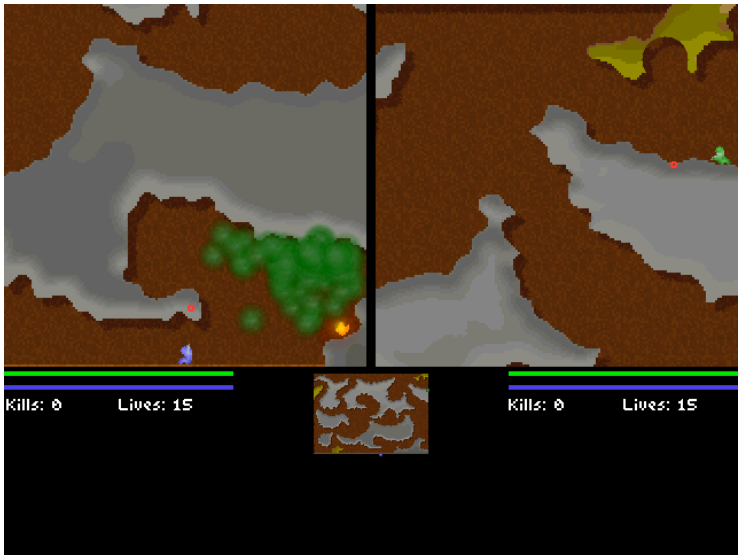


Figure 6.3: Liero

6.2 Project evaluation

The composite goal of the project was to create a playable action game based on principles of Tunneler with possibility of programming artificial players. All these tasks were fulfilled: the resemblance to Tunneler is obvious, game is well-playable (with its multiple weapons, network game and nice graphics) and scripts for artificial players are quite easy to create.

A somewhat missing feature is a debugging environment for AI scripts. It would be also fine to enhance the network session control so the game could be played not only over LAN.

6.3 Future of the project

The project is working fine, the game is playable and AI scripts are well-programmable, but it certainly does not mean the project is finished – finished projects are mostly those whose development was abandoned rather than considered complete. Here is a list of a few ideas that would be nice to implement to enhance the gameplay or the whole project.

- Images used in the game are quite nice, but they look somewhat simplistic due to lack of animation. It would be cute to animate robots

movement and shooting as well as shots and explosions.

- There is no music nor sound effects in the application and the gameplay would certainly seem more professional and enjoyable if there were some.
- The "Capture the flag" game mode, quite usual in action games, would be great to enable.
- As noted in 6.2, a debugging environment for AI scripting would be useful.
- If the project becomes even more popular it will be a good motivation to try offering packages into official repositories.

Bibliography

[1] *C++ Language reference*,
`http://www.research.att.com/~bs/C++.html`

[2] *Ditchers*,
`http://www.ditchers.sourceforge.net`

[3] M. DeLoura: *Game Programming Gems Series*, 2000.

[4] *Guichan*,
`http://guichan.sourceforge.net/`

[5] *Debianí balíčky pro kopáčovou hru – Ditchers*,
`http://web.zvano.net/drupal6/node/11`

[6] R. Ierusalimschy, L. H. de Figueiredo, W. Celes: *Lua 5.1 Reference Manual*

[7] J. Hall: *Programming Linux Games*, Linux Journal Press, 2001.

[8] *SDL Library*,
`http://libsdl.org/`

[9] *Tunneler by Geoffrey Silverton (DOS, 1991)*,
`http://members.chello.at/theodor.lauppert/games/tunneler.htm`

Appendix A

CD content

There is a CD included in the work. Its content is the following:

- The application itself in the `ditchers.tar.gz` file, in the `ditchers` directory and in Debian packages in the `packages` directory.
- Documentation generated by doxygen in the `doxygen` directory.
- Source codes of used libraries in the `libs` directory.
- This work in the `bc` directory.
- Last but not least, the file `ditcher.ogv`, a video from the game.