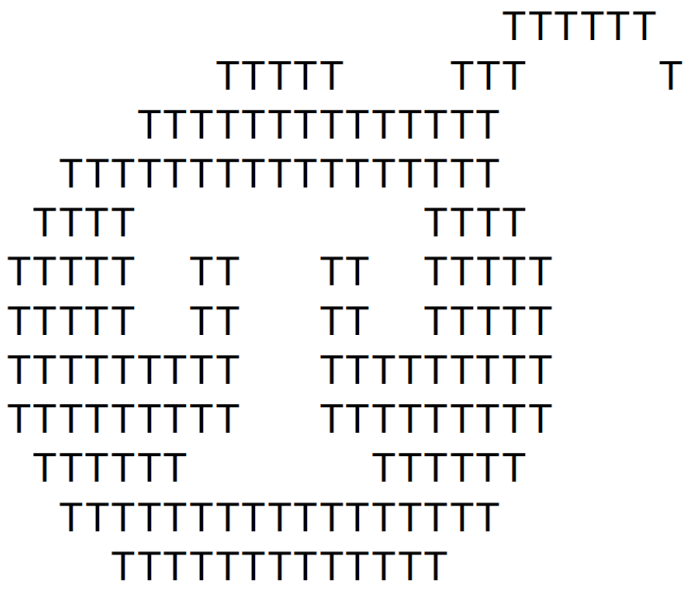

Turbo-Bomber

Tim Gevers, Jonas Krug, Hendrik Sieck

January 06, 2016

Contents

1 Specification

The main goal is to implement a game like "Bomberman". The player plays in a quadratic world and tries to clear its way with clever placed bombs. Some walls contain items that increase several properties of the player and its utilities.

1.1 Criteria for evaluation

1.1.1 Read from files

The game uses files to be configurable and modifiable.

1.1.2 Random item drops

When a tile is destroyed by a player it is possible that the wall drops an item. An algorithm chooses based on probabilties which item is then dropped. The items are generally *Power Ups*.

1.1.3 Read keys from the keyboard

For a fast game keyboard input is required. The input is taken directly without any buffering. (Does also not require pressing the Return-key.)

1.1.4 Move player

The player is able to walk around in the map to place bombs.

1.1.5 Animations

The game will be outputted to the command line via a common used library called *ncurses*. This includes also animations and a simple text based user interface.

1.1.6 Place bombs

Bombs can be placed on empty tiles of the map. They have simple detonation physics which gives them the ability to destroy walls and harm players.

1.1.7 Destructable map

The map consists of several destructable walls which may be destroyed by bombs. When these walls are destroyed they may drop an item.

1.1.8 Game loop

To be able to interact with the player inputs and play animations the game has a main loop. This main loop will process several events (e.g. keypresses) and actions.

2 Design

2.1 Structure of the game

The game has a main information stream. This stream starts at the user. The user provides informations to the program which are especially keyboard inputs. These keyboard events are processed and moved to the *gameplay*-module. This module is the central module which consists of the complete game logic. The *gameplay*-module generates a data structure that is consumed by the *graphics*-module. A framebuffer (of the *ncurses*-library) is then filled by the *graphics*-module. The module also frequently updates the framebuffer to produce some animations.

The complete information stream is processed each frame. As a result each frame keyboard events are processed, the game logic (*gameplay*-module) reacts to this events and the *graphics*-module displays the game on the screen.

2.2 Important data structures

2.2.1 Properties of a map tile

- type of the tile (wall, destructable wall, floor)
- player that stands on the tile
- explosion animation informations
- bomb that is placed on the tile
- random generated item (*Power Ups*) of the tile
- timing informations for items and explosions

2.2.2 Properties of a player

- health points
- movement cooldown (to decrease movement speed)
- position on the map
- amount of placeable bombs
- amount of currently placed bombs
- current item
- item usage timing (duration, used time) (some items are limited in time)

3 Conventions

3.1 Coding style

Use *Allman*-style with the following changes:

- Do not write whitespaces after *while*, *if* or other control statements.
- Do use tabulators as indentation.
- Do not use single line statements (e.g. in *while*, *if*). Always use (curly) brackets around these statements.

3.2 Nomenclature/Naming

3.2.1 Functions

Names of a function in the code should have the following nomenclature:

```
<module name>_<part of the module>_<action to perform>();
```

module name Each module consists of one source file. It has the name of the module. If the module has multiple words the words are separated by "-"-characters, in the function name "_"-characters. An example file name would be: "image-loader.c". An example function name can be found in the *action to perform*-paragraph.

part of the module A module can be splitted into multiple parts. The parts are summarized in a single module file. If a part has multiple words the words are separated by "_"-characters. An example function name can be found in the next paragraph. The part can be omitted if the module only has one part.

action to perform The action describes what a function will do with the part in the module. If the action has multiple words the words are separated by characters. An example function name would be: "image_loader_file_informations_print();".

3.2.2 Variables

Do not use global variables. Variables needed in a module can be declared as "static"-variables at the head of the source file. Variables should always be initialized with a default value (at compile time). Variables have the following nomenclature:

```
static int <module name>_<part of the module>_<property name> = 0;
```

module name and *part of the module* are described ahead.

property name The property which the variable holds the value for. If the property name has multiple words the words are separated by characters. An example function name would be: "static int image_loader_file_informations_count = 0;".

3.3 Other

1. Do not upload binaries.
2. Do describe the commits usefully.
3. Do not commit multiple features in one commit. Split them into several commits.
4. Do not use special characters (whitespaces included):
   - Do not use special characters in the program (code).
   - Do not use special characters in directory names.
   - Do not use special characters in file names.
   - Do not use special characters in commit descriptions.
5. Do use English as general language.
6. Do always document functions. Use the Doxygen documentation style. (http://www.stack.nl/~dimitri/doxygen/manual/index.html)
7. Do document code when the code does not express its meaning itself.