Berzerk

[redacted]

Game Engine Architecture, TDA572

A.A. 2016/17

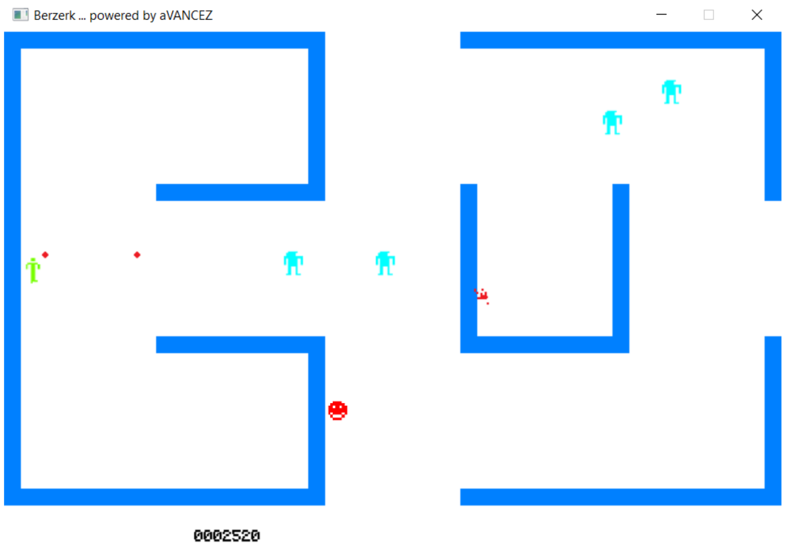

Figure 1: A picture of the game. The *player* can move and shoot in 8 directions. Randomly spawned *robots* track the player and can shoot in 8 directions. After 15 seconds or when there are no robots left a *sentry* spawns.

1 Introduction

Berzerk is an arcade game which was published in 1980 by Stern Electronics. [3] This report describes the implementation of a clone of this game. The architecture is based on the guidelines illustrated in [6].

The objective of the game is to get as much points as possible by destroying robots without getting killed yourself.

2 Controls

W, A, S, D	8-directional movement
$\uparrow, \downarrow, \leftarrow, \rightarrow$	8-directional shooting
F1	Display help on console
F2	Enable/Disable Godmode
F3	Enable/Disable if the robots die on wall hit
ESC	Quit game

3 Specifications

Berzerk consists of different interacting objects which are described in this section.

Player. The player is the controllable green man. At the start of the game and every time he loses a life he is positioned by the western exit of the room. He can move and shoot in 8 directions and has three lives.
The player loses a life when:

1. he touches a wall.
2. he is hit by a bullet.
3. he touches the sentry.
4. he touches a robot.

Robot. There are 3 different kinds of robots differentiated by their color.

1. Yellow: <500 Points. The yellow robots cannot shoot any bullets.

2. Red: From 500-1500 Points. The red robots can shoot one bullet. This counts for all robots in the room, e.g. there can be only one robot bullet on the screen at a time.

3. Cyan: >1500 Points. The cyan robots can shoot two bullets. This counts for all robots in the room, e.g. there can be only two robot bullets on the screen at a time.

The robots can move in 4 directions by randomly tracking either the x- or y-coordinate of the player. They are able to shoot in 8 directions. Robots move half as fast as the player.
Robots lose their life when:

1. they touch a wall.

2. they are hit by a bullet (this included bullets from other robots).

3. they touch the sentry.

4. they touch another robot.

Bullet. Bullets are fired by the player (green bullets) and the robots (red bullets). Bullets get destroyed by hitting any kind of object.
Sentry. The sentry spawns when there is no robot left in the current room or after 15 seconds. It can move through walls and is indestructible. The sentry will always follow the player. The horizontal speed is always half as fast as the vertical speed. As long as there are robots left in the room, the sentry moves half as fast as the player but as soon as all robots are destroyed it speeds up to match the player.
The end of the game. The game ends as soon as the player has no lives left. The goal is to gain as much points as possible.

4 Overview

The game architecture is based on lab4 of this course. The `GameObjects` contain different `Components` which implement the corresponding logic. The game objects take care of receiving messages which are delegated to the corresponding component. There are four different components. The `BehaviourComponent` implements the logic of the game object, e.g. movement. The `RenderComponent`

renders the game object on the screen. The `CollideComponent` simply notifies once two game objects collide with each other. The AiCollideComponent has the same function as the `CollideComponent` but sends out another message. It is solely used for the robot-wall detection.
The game class takes care of creating the game objects with their respective components as well as updating them.

5 Implementation

5.1 Game Update, Object Pool, Game Object, Components

Just as in lab4 the update pattern has been used. In the `Main` class the `Game` object is created and the `update(...)` function is called which takes care of updating the game objects and their components.
An `Object Pool` is used to pre-allocate a number of game objects in order to avoid memory allocation during the game loop.
The `GameObject`s can be seen as skeletons. They consist of the state and a collection of components. The components implement the real functionality of the game object. The game object is also responsible of the message communication.
There are different `Component`s for different applications. The `BehaviourComponent` implements the logic of a game object, the `RenderComponent` is responsible for the drawing of a game object and the `CollideComponent` informs if an collision between two game objects has occurred. For more information on these classes please refer to the documentation of lab4.
The `AiCollideComponent` is used for the collision between a `Robot` and a `Wall`. If the `AiCollideComponent` registers a hit the robot stops moving into a wall. The `AiCollideComponent` is deactivated by default because it is somewhat buggy. This is due to lack of time and because the original game in fact does not have that kind of AI.

5.2 Player

The `PlayerBehaviourComponent` handles the input and logic of this object. The player is moved accordingly and `Bullet`s are created whenever necessary.

5.3 Robot

Whenever a `Robot` is initialized the type of this robot is checked via a message. This is done so that the correct sprite is loaded since there are three different robot types depending on the current score.
The `RobotBehaviourComponent` is responsible for the AI. A robot will always move towards the player by randomly tracking the player's x- or y-Value. In the `update(...)` function the tracking stage is set and the robot then moves accordingly.
Robots can fire in 8 directions and will always fire once the player is aligned in any of these directions. For the 45 degree alignment simply the difference between the robot's and player's x and y position is calculated.

5.4 Bullet

The bullets are simple objects. Depending on the set `shootDirection` the movement is calculated. `Robot`s as well as the `Player` use this object with different sprites.

5.5 Sentry

The `Sentry` is rendered outside the screen as long as it is the `despawn` state. Once it receives the `spawn` message it will be moved into the screen.
In the `SentryBehaviourComponent` the movement and speed of the sentry are implemented. The sentry always moves half as fast in x-direction than in y-direction. Furthermore it moves half as fast as the player as long as there are still robots left in the room. It speeds up to the player's speed as soon as there are no robots left.

5.6 Room

Each room consists of a number of `Wall` tiles.
The `RoomBehaviourComponent` implements the maze generation. First the fixed layout is created which never changes, see Figure 2.

The maze generation is the same as in the original game. Each room has an x- and y-coordinate between 0 and 256. Every time the player leaves the room the coordinates are adapted. For the maze generation a pseudo random generator, the linear congruential generator, is used. [5] The seed of

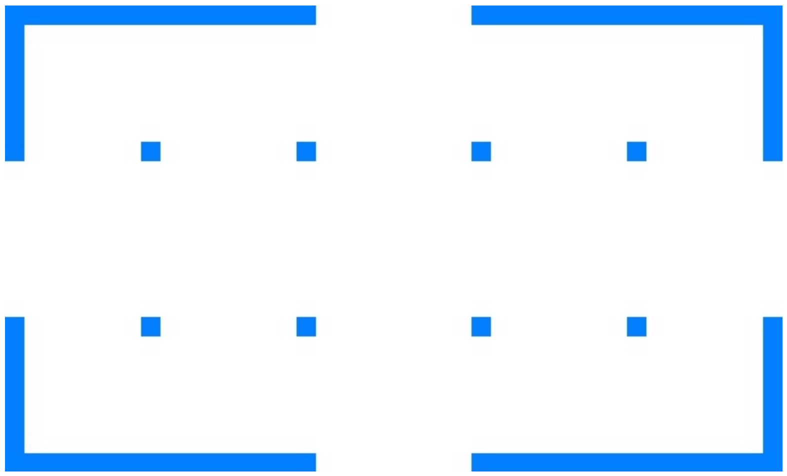

Figure 2: The fixed layout for every `Room`. There are 8 pillars evenly distributed in the center of the room.

the generator is computed as follows:
$seed = ((roomX + roomY) * 7) + 12627$
A new pseudo random number is created with the seed for each of the 8 pillars in the room:
$newSeed = (seed * 7) + 12627$
The high 8 bits of the number are used as the random value, this means the result is right-shifted by 8. [1] The random number generation is done twice before it is assigned as the pillar value. Once the pillar as an assigned value, the direction (i.e. the pillar value) is checked and the walls are drawn. This process is visualized in Figure 3 and the code can be seen in Listing 1.
This process guarantees the same rooms every time the game is started and also allows a revisiting of rooms.

Listing 1: Code for generating a pseudo random maze.

```
void generateRoom(int roomNr) {
seed = (roomNr * 7) + 12627;
        int x = 144;
        int y = 144;
        for (auto row = 0; row < 2; row++) {
                for (auto col = 0; col < 4; col++) {
                        generateRandomNumber(seed);
                        int pillarValue =
                           generateRandomNumber(seed);
                        int wallDir = pillarValue & 3;
                        drawWall(x, y, wallDir);
                        //PILLAR SETTING
                        x += 144;
                        if (col == 3) {
                                x = 144;
                        }
                }
                y += 144;
        }
}
int generateRandomNumber(int newSeed) {
        seed = (newSeed * 7) + 12627;
        return seed >> 8;
}
```

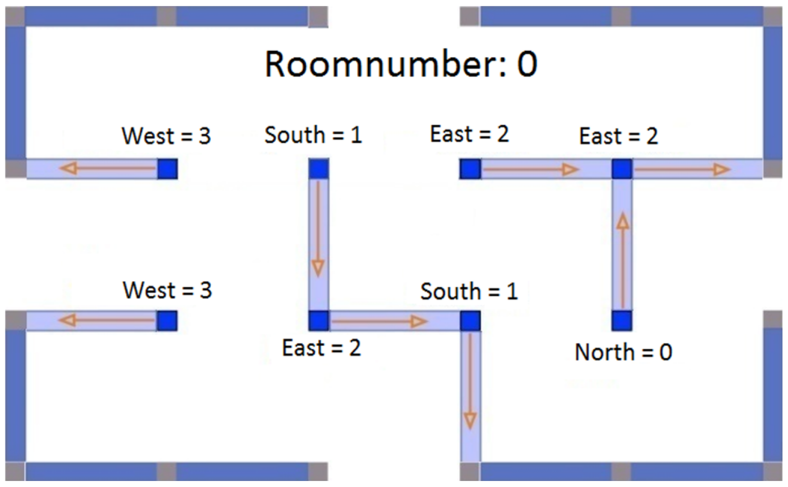

Figure 3: Maze generation of a room. A pseudo random number is assigned to each pillar which gives the draw direction. [2]

5.7 RobotSpawn

First of all, some rules have to established regarding the spawning:

1. A maximum of 11 robots can spawn in one room

2. At least one robot must spawn in a room
3. Robots do not spawn on top of each other (or in collision distance)
4. There can spawn a maximum of 2 robots per spawn square
5. Robots cannot spawn in the entry squares

As seen in Figure 4, robots can only spawn in the red spawn squares. The `RoomBehaviourComponent` first of all sets these squares. Each time a new room is visited, the number of robots to spawn is set randomly. Afterwards each robot gets placed in a random spawn square. The rules are checked before the current robot is assigned to the square. The code for spawning the robots can be seen in Listing 2.

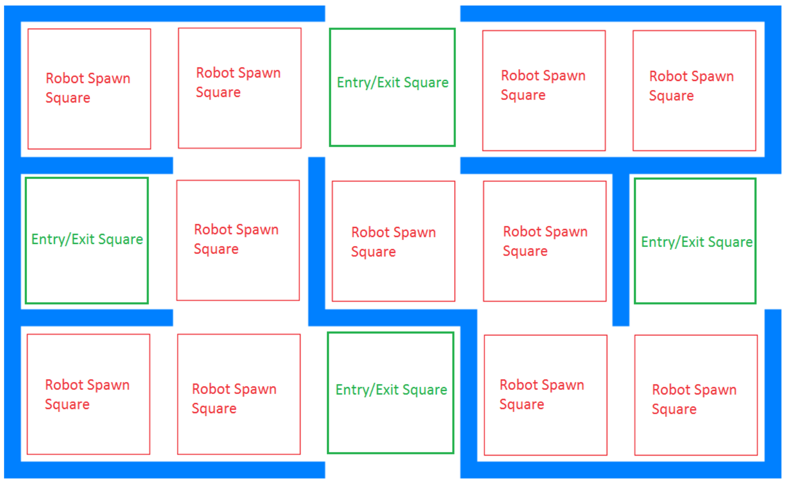

Figure 4: `Robots` can only spawn in the red spawn squares and according to the rules set in Enumeration 5.7

Listing 2: Spawing the robots according to the rules.

```
for (int i = no_of_robots; i > 0; i--) {
                SDL_Rect spawnZone = getRandomRectangle();
                offsetX = std::rand() % 92 + 1;
                offsetY = std::rand() % 88 + 1;
                x = spawnZone.x + offsetX;
                y = spawnZone.y + offsetY;
                spawnPerQuadrat = std::rand() % 2 + 1;
                if (spawnPerQuadrat == 1 || i == 1) {
                        Robot* robot = robots_pool->
                           firstAvailable();
                        robot->botType = checkBotType();
                        robot->init(x, y);
                        game_objects->insert(robot);
                }
                else if (spawnPerQuadrat == 2) {
                        for (int j = 0; j < 2; j++) {
                                Robot* robot = robots_pool->
                                   firstAvailable();
                                robot->botType = checkBotType
                                   ();
                                robot->init(x, y);
                                game_objects->insert(robot);
                                SDL_Rect spawned{ x, y, 20,
                                   24 };
                                SDL_Rect newRobot =
                                   getNewRobotPosition(
                                   spawnZone, spawned);
                                x = newRobot.x;
                                y = newRobot.y;
                        }
                        i--;
                }
        }
```

5.8 Particle System

The extension implemented for this game is a particle system. Every time a robot is hit it explodes in 50 particles.
There are two classes responsible for the particle effect: `ParticlePool` and `Particle`. For the storage of the particles an array of `Particle`s was chosen. Each time a robot gets hit, the `ParticlePool` is created. By iterating through the particle array it is checked if a particle is currently in use. If not, the particle is initialized with a random move direction and a little position offset, as seen in Listing 3. The `ParticlePool` has an`update(...)` method which simply iterates through the array and calls the `update(...)` method of each `Particle`. The `Particle`s themselves have a `draw()` method which renders the current particle. They also have a `init(...)` and `update(...)` method for initializing and updating the current particle.
I have tried implementing the particle system with a free list. Although it

works (i.e. no error), the particles all get rendered on top of each other and move in the same directions. The source code for this implementation can be seen in the *../objects/ParticleSystem_freeList* directory. Both implementations of the particle system are based on the Object Pool pattern as described in [6].

Listing 3: Creation of the `ParticlePool`

```
void ParticlePool::create(double x, double y, double xVel,
    double yVel, int lifetime) {
        srand(time(NULL));

        for (int i = 0; i < POOL_SIZE; i++) {
                if (!particles[i].inUse()) {
                        angle = std::rand() % 360 + 1;
                        particles[i].init(sprite, x + std::
                            rand() % 10, y + std::rand() % 10,
                             xVel * cos(angle), yVel * sin(
                            angle), lifetime);
                }
        }
}
```

5.9 Game

The `Game` class takes care of the creation of the game objects with their respective components. The game objects are created and pre-allocated to the corresponding object pools. After the creation of a game object it is inserted into the game objects std::set. [4]
The `update(...)` function calls the `update(...)` methods of each game object in the set. Here it is also checked if the godmode or the robot-wall collision are enabled or disabled and the corresponding actions are taken.
The `draw()` function is responsible for drawing the score, life sprite, bonus (if applicable) and game over message (if applicable) to the screen.
With the help of the `receive(...)` method the `Game)` object also calculated the current score and bonus.

6 Conclusions

The particle system has the major downside that upon creating a particle it is possible to iterate through the entire array to find an unused particle. This can cause a performance drop for large particle systems. For this game the performance is stable since there are only 50 particles in each system. Nystrom proposed the implementation for a particle system with a free list. [6] I have tried implementing it that way but could not eradicate the problem that all particles render on the same spot.

The messaging system can get quite confusing for people new to the source code. The observer pattern as described by Nystrom could have been used to make the messaging between objects more clear. [6]

The robot-wall collision with the `AiCollideComponent` could have been solved better with more time on the hands. Another solution for this problem could be the following: In the `update(...)` method of the `RobotBehaviourComponent` it should be checked if the current robot collides with a wall before the movement. If so, the position of the robot is set back to the previous position where it does not collide with the wall and the move method returns false. Otherwise the move function returns true and the robot can move.

References

[1] Berzerk. `http://www.drcjt.myzen.co.uk/Berzerk.html`. Accessed: 2017-03.

[2] Berzerk maze generation. `http://www.robotron2084guidebook.com/home/games/berzerk/mazegenerator/`. Accessed: 2017-03.

[3] Berzerk (video game). `https://en.wikipedia.org/wiki/Berzerk_(video_game)`. Accessed: 2017-03.

[4] std::set. `http://en.cppreference.com/w/cpp/container/set`. Accessed: 2017-02.

[5] Aycock, J. *Retrogame Archeology Exploring Old Computer Games.* Springer International Publishing, 2016.

[6] Nystrom, B. Game programming patterns. Online, 2014.