INFO0004-2: 1vs1 Worms (Groups of 2)

Pr. Laurent MATHY, Sami BEN MARIEM, Gaulthier GAIN

April 4, 2023

1 Introduction

Worms is a series of 2D turn-based video games in which cartoon worms battle it out in a destructible landscape, with the goal of becoming the only surviving team.

In this project, you will implement a simplified version of this game in C++, using the SDL (Simple Direct media Layer) 2.0.14 library (at least). This library provides easy windowing and user input, timing and graphics rendering facilities.

The project must be done in groups of **2** students.

2 Gameplay

In this game, two players (human opponent) take turns controlling a worm with a health level and two different weapons. The goal is to use these weapons to kill the other player's worm and get the last surviving worm. Each player has 30 seconds to control his worm (movement, aiming, shooting, ...). The player loses the game if the health of his worm reaches 0. Figure 1 illustrates the beginning of a new round where each worm has its maximum health level (100) and the player has 30 seconds to perform its actions.

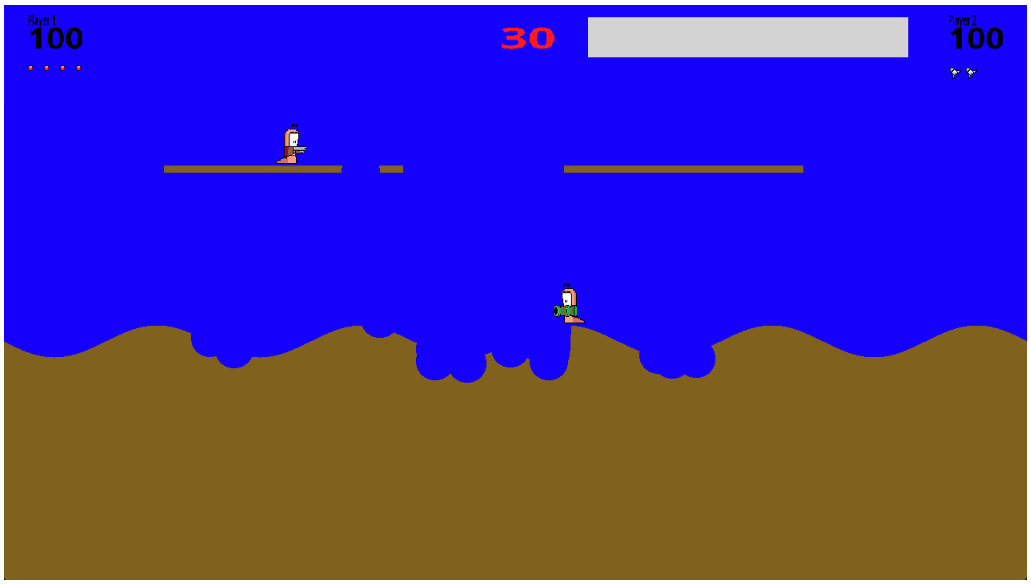


Figure 1: General view of the game. Each player controls a worm that can use various weapons to cause damage on the opposing player's worm. In addition to cause damage, some weapons can destroy the landscape.

2.1 Worms

Each worm is controlled in turn by one player for 30 seconds. Initially the worms spawn at fixed positions (see Section 4) and are subject to gravity which causes them to fall until they hit the ground. Their fall is slowed down by a "helicopter hat" which prevents them from being injured.

A worm can have 6 different states:

1. *Moving left* (on the ground);
2. *Moving right* (on the ground);
3. *Taking aim* (aiming mode);
4. *Flying*: when the worm uses a jetpack to have a **straight** vertical trajectory;
5. *Falling*: when the worm does not use a jetpack or when the worm does not collide with a platform or the ground; It is within this state that he can "move" in the air to the right or to the left,
6. *Resting*: the worm does not move and is located on a platform or the ground.

Note that each worm can only be in one of the six states at a time. In addition, when a worm falls or flies, it is not allowed to fire.

Controls Worm can be controlled by the player using different keys on the keyboard:

- The «**Left**»/«**Right**» keys: by holding down one of these two keys, the worm is moved respectively to the left or to the right at a specific velocity.
- The«**Up**» key: by holding down this key, the worm uses a jetpack and performs a straight vertical trajectory at a certain velocity. This key is also used to control the angle of the weapon (see below).
- The «**1**»/«**2**» keys: pressing these keys allows you to switch from one weapon to another. The first one selects the bazooka, the second selects the shotgun.
- The «**Space**» key: pressing this key enters the "aiming mode" or fires the current weapon (see Section 2.2.1).

2.2 Weapons and "aiming mode"

Each worm can shoot with two predefined weapons: a bazooka or a shotgun. Each weapon has its own characteristics:

- **Damage points**: These are the damage points caused on the opponent's worm when it is hit by the weapon's projectile.
- **A reload delay**: Once used, each weapon must be reloaded for a certain time before it can be used again.
- **A number of ammunition**: Each weapon has a certain amount of ammunition available per round. Once the ammunition is empty, it is no longer possible to fire with the current weapon.

The bazooka has also a radius of damage and is able to destroy the scenery. For simplicity, the damage caused remains constant with the radius. The shotgun causes more damage but has no radius area; it fires a bullet that prevents it from destroying scenery.

2.2.1 The "aiming mode"

When the player presses the «Space» key, the game enters *aiming mode*. In this mode, the player has to choose the shooting angle and velocity. First, the shooting angle is decided by pressing the «Up» or «Down» arrow key. Pressing the «Up» arrow key will aim the weapon towards the sky, while pressing the «Down» arrow key will aim it towards the ground.

After choosing the angle of fire, the player must decide on the speed of fire. To do this, simply press the "Space" key again. The value of the power bar increases gradually until it reaches a maximum threshold, then resets to 0, then increases again, etc. The player must press the "Space" key one last time to select the desired speed. The different values used for this mode are defined in Section 4.

2.3 Projectile

Each weapon contains a specific projectile, a rocket for the bazooka and a bullet for the shotgun. The projectiles that are thrown by the different entities are subject to gravity. Your game engine should properly deal with realistic projectile motion. Indeed, you should not only consider the gravity but also drag forces. The formula that you should use to compute the drag force applied on a projectile is the following:

$$\vec{F_d} = \frac{1}{2} * \rho * A * C_D * ||v||^2 * -\hat{v}$$

where

- ρ is the density of the air,
- A is the cross sectional area,
- C_D is the drag coefficient,
- $||v||$ is the norm of the speed vector, and
- $\hat{v}$ is the unit speed vector representing the direction of the projectile.

A projectile must be stopped and destroyed instantaneously if it collides with the scenery (the ground or platforms) or with a worm. Similarly, any projectile that is no longer visible on the screen must be destroyed.

2.4 Collision Detection

Your game engine must be able to handle collisions between objects.

Concerning the collision of projectiles with a worm, we do not expect very complex collision detection mechanisms using complex sprite meshes. Instead, you can use simple primitives like rectangles. However, we leave it to you to define the size, position and orientation of the primitive you will use for this task.

For the detection of collisions with the scenery, we expect your game to be somewhat realistic. For example, a worm must move according to the slope of the terrain (we do not ask you to tilt/rotate its sprite accordingly), the impact point of a rocket must destroy the scenery according to its radius of damage. For this task, it is up to you to think about how you can handle this feature. More information on ground/worm collisions is provided in Section 3.2.4.

3 Rendering

In this section, the various elements of the scenery as well as the entities and game objects are described.

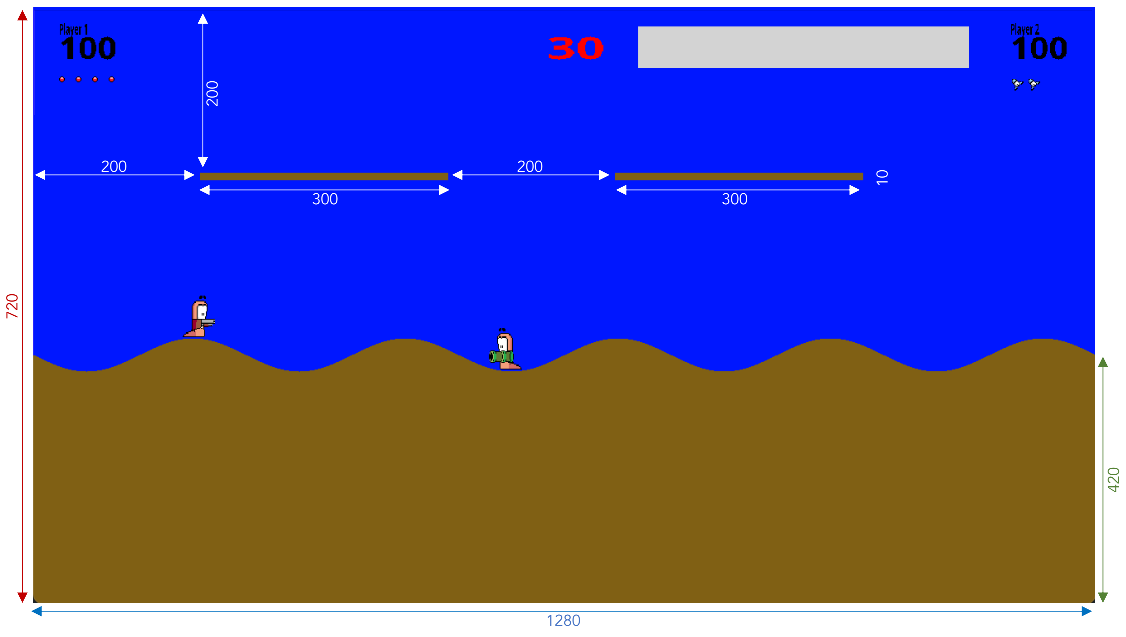


Figure 2: General Components.

Figure 2 shows the size of the components that make up the game scene. Regarding the color of components: **(1)** the background of the scene is blue `(0, 0, 255)`, and the two platforms and the ground are brown `(128, 96, 20)`. All metrics mentioned are in pixels.

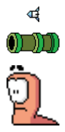
3.1 Game objects

For rendering of the game objects such as worms, weapons and projectiles, you will use the sprites that have been provided on eCampus.

However, you will still need to specify the size of the sprites to render correctly. We have (in pixels): **(1)** Worm renders at size `(30, 50)`, **(2)** both weapons render at size `(30, 15)`, **(3)** rocket projectile renders at size `(20, 10)` and finally **(4)** bullet projectile renders at size `(10, 5)`. By default, all provided sprites are oriented to the left. These ones must be flipped horizontally by your game engine when necessary.

3.1.1 Worm, Weapons & Projectiles

As represented in Figure 3, the sprite of worm that is given does not include any type of weapon. You are responsible for rendering a worm with a weapon when equipped and without a weapon when it falls or flies. In this figure, the distance vector between the center of the weapon sprite and the center of the worm's sprite is `(-10, 10)`.


(a) Separate view of a worm equipped with a bazooka and a rocket.

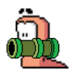

(b) Rendering of a worm equipped with a bazooka and a rocket (not visible).

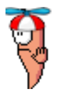
Figure 3: Rendering of a worm, a weapon & a rocket.

3.1.2 Worm states

Figure 4 illustrates how a worm should be rendered differently depending on its state. Figure 4a shows the two sprites that should be used to represent a worm moving to the left on the ground. The two sprites should be used to represent the steps that a worm is making when moving forward. Then, Figure 4b represents a worm flying with a jetpack. Figure 4c shows a worm falling. Finally, Figure 4d represents the state of a worm at rest.

(a) Worm moving on the ground.

(b) Worm flying using its jetpack.


(c) Worm falling.

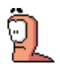
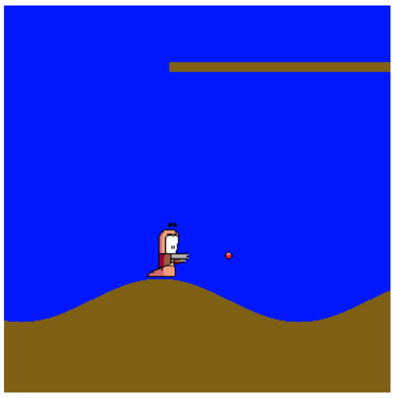

(d) The worm is at rest.

Figure 4: States of a worm.

As mentioned above, the worm as well as weapons and projectiles must be rendered differently (flipped horizontally) depending on their direction.

3.1.3 Projectiles

As represented in Figure 5, projectiles should be rotated and flipped appropriately depending on their current speed vector.


(a) Example of a projectile flipped and rotated when a worm is directed to the right.

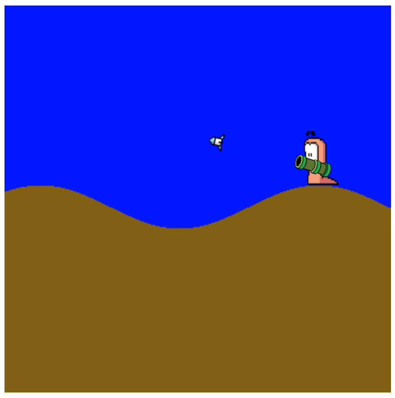

(b) Example of a projectile rotated (only) when a worm is directed to the left.

Figure 5: Examples of projectiles fired from weapons.

3.2 Scene elements

The scenery contains two platforms and a ground that are **fully destructible** when hit by a rocket (and not by a bullet). In addition, it also contains textual and visual elements which are described here.

3.2.1 Textual and visual elements

As shown in figure 6, there are 8 visual and textual elements. All these textual elements are black `(0,0,0)` except the timer which is red `(255,0,0)` and have a font size of 50 pixels. The font to use is `OpenSans-Bold.ttf` which is available on eCampus.

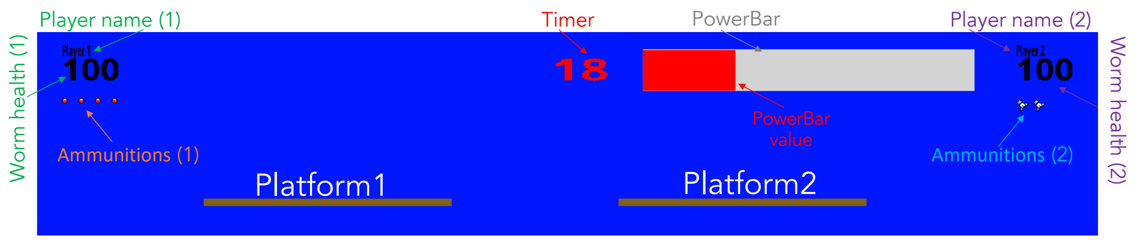


Figure 6: General Components.

As previously stated, all metrics below are in pixels[1]. In this figure, there are the following elements:

1) A timer that decreases with time. It has a size of `(70, 50)` and is located at coordinates `(620, 20)`.

2) The power bar linked to the current player. It has a size of `(400, 50)` and is located at coordinates `(730, 20)`.

3,4) The display of the names (by default player1 and player2) who constitute two different visual elements. These have an identical size `(35, 25)` but are located at different positions, namely `(30, 10)` for the first player and `(1180, 10)` for second player.

5,6) The health of each worm by player. As with the names, there are two different visuals. They have also the same size `(70, 50)` but are located at different positions, i.e. `(30, 20)` for first player and `(1180, 20)` for second player.

7,8) The last visual element represents the number of ammunition according to the chosen weapon. There are two of them, one per player, and they are decremented each time a projectile is shot. The size relies on the underlying sprite displayed (bullet or rocket), and the first displayable ammunition is located at the coordinates `(30, 80)` for the first player and `(1180, 80)` for the second player. Each ammunition is separated from the other by an offset of 20 (pixels) and is displayed at a $45°$ angle. If a worm has used up all of this ammunition, there is no more rendering.

[1]The size of the text elements given here represents the size of their bounding box.

3.2.2 The power bar

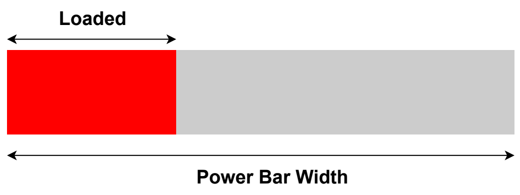


Figure 7: The power bar contains two different rectangles.

The power bar can be represented by using two rectangles of constant total area but with different colors. The two rectangles should be rendered next to each others with equal height.

The first rectangle, representing the loaded part of the power bar will be of RGB color `(255, 0, 0)`, with a width equal to

$$power\_bar\_width * (power\_bar\_value/max\_power\_bar\_value) \tag{1}$$

and the second rectangle will be of RGB color `(211, 211, 211)` and with a width equal to

$$power\_bar\_width - loaded\_power\_bar\_width \tag{2}$$

3.2.3 The two platforms

The two platforms are represented as rectangles each having a width of `300` pixels and a height of `10` pixels. Their origin (upper left corner) is respectively located at coordinates `(200, 200)` and `(700, 200)`. A worm can only access it by falling after using its jetpack and collisions must be handled as follows: (1) a worm can only move on the top of the platform; (2) if a worm collides with the bottom or one side (right or left) of the platform, it is blocked.

3.2.4 The ground

The ground is generated according to a sinusoid which is characterized by its maximum amplitude and its frequency. It can be written as:

$$y(x) = A\sin(2\pi f x + \varphi) = A\sin(\omega x + \varphi) \tag{3}$$

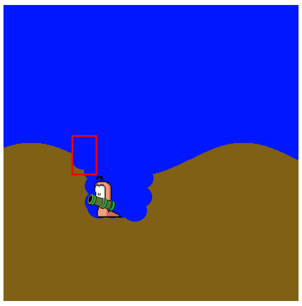
where:

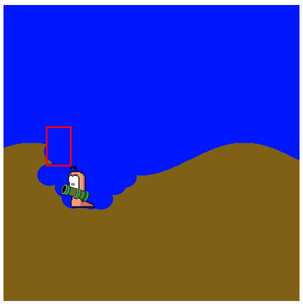
- A, the amplitude, the peak deviation of the function from zero.
- f, the frequency of the signal in hertz
- $\omega = 2\pi f$, the angular frequency, the rate of change of the function argument in units of radians per second.
- φ, the phase at the origin in radians.

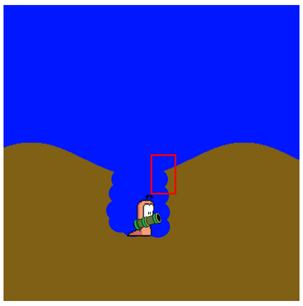
In the game, the frequency (f) is set to 5 and the amplitude (A) is set to 20. The phase φ can be set to 0.

In some cases, the destructible landscape can leave very particular shapes (hole, very steep slope, etc.) as shown in Figure 8. To manage the movement of a worm according to the slope of the ground, we use a basic approach which consists in defining a "virtual" rectangle above the worm. The (moving) worm is blocked if this rectangle collides with a part of the ground. If no collision is detected, the worm can climb the slope. However, if this virtual rectangle collides, only the use of its jetpack can help the worm get around the obstacle. You can assume that we will test your implementation with examples similar to those in Figure 8.

We **leave it to you to define** the size and the position of the primitive you will use for this task.


(a) Example 1 where the worm is blocked because of the scenery.


(b) Example 2 where the worm is blocked because of the scenery.


(c) Example 3 where the worm is blocked because of the scenery.

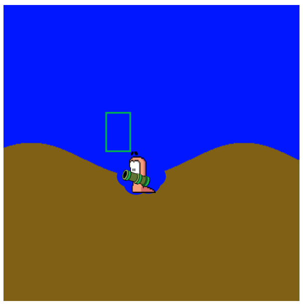

(d) In this case, the worm can move since no collision is detected.

Figure 8: Complex landscapes

4 Additional Parameters

We provide some additional parameters that you should use to ensure the good behaviour of your game. Positions are expressed as the coordinates at which the top left part of the sprites should be rendered. Delays are expressed in a number of frames. You are responsible for ensuring that your game renders at exactly **30** frames per second.

Parameters of Worm The actions of a worm can be parametrized using:

- **Movement & Positioning:**
  - Moving Velocity: `(0, ±75)` (in pixels/second)
  - Moving Up Velocity (flying with jetpack): `(0, -75)` (in pixels/second)
  - Falling Velocity: `(0, 100)` (in pixels/second)
  - Falling Starting Time[2]: 300 ms
    - Initial Position (**Player1**):[3] `(70, 0)` (in pixels)
    - Initial Position (**Player2**): `(1100, 0)` (in pixels)

[2]Time at which a worm is in the "falling" state and its sprite is changed accordingly.
[3]Worm must be flipped for Player1

- **Misc:**
  - Initial Weapon: "bazooka"
  - Shooting Angle Step: $\pm 10°$
  - Shooting Power Step: 10
  - Min Shooting Power: 0 pixel/second (related to power bar)
  - Max Shooting Power: 500 pixels/second (related to power bar)
  - Shooting Power Bar Update Time Step: 10ms
  - Initial Health: 100

Parameters of Weapons

- **Bazooka:**
  - Reloading delay: 30 seconds
  - Initial number of ammunition: 2
  - Damage points: 10
  - Radius of damage: 24 (pixels)
- **Shotgun:**
  - Reloading delay: 20 seconds
  - Initial number of ammunition: 4
  - Damage points: 20

Other useful parameters:

- Rocket Mass: 5 Kg
- Rocket cross-sectional area: 1 pixel
- Bullet Mass: 2 Kg
- Bullet cross-sectional area: 1 pixel
- Gravity: 500 pixels/second^2
- Air Density: 1.0 Kg/m^3
- Drag Coefficient: 0.001

If you have to use other constants, please define them and explain why you have used them (in your report or in your code).

5 Miscellaneous

In this section, we detail some gameplay and rendering behaviours that must be respected under penalty.

- A worm cannot leave the screen. When it reaches the limits of the window (left, right, and top), it is blocked. As for the bottom, if the ground below it is destroyed, the worm will fall into the void and be instantly killed;

- When two worms meet, they block each other's path without hurting each other. It is necessary to use the jetpack to bypass a worm;
- If a rocket is shot underneath a worm, the worm is injured if it is in the impact zone (the damage radius) and falls into the resulting crater;
- The projectiles do not pass through the screen. They are destroyed if they reach the limits of the window (left, right, bottom and top);
- When a player finishes his turn, the worm must return to its resting position. If the worm was in the air, it falls until it hits the ground and then it goes to the resting posting.
- The sprites of the worm, the weapons and the projectiles must be flipped horizontally according to the direction in which the worm moves.
- Once in the air (flying or falling), a worm cannot shoot and its weapon is hidden.
- A worm must injure itself if it is within the range of its own rocket.
- As soon as the health of a worm reaches 0, it disappears and the following message "Player X won the game!" appears on the screen, where "X" is the name of the player who won the game. All elements of the game are frozen (the timer also stops on the remaining time) and it is no longer possible to play.

6 Remarks

6.1 Report

You should then write a short report explaining your architecture and how you managed the collision and destruction of the scenery. In addition, you can explain your implementation choices if necessary. You should use static diagrams to represent the interactions between the different classes. Your report should be **2** pages long (maximum).

We would be grateful if you could give us an estimate of the time you spent on this work, and the main difficulties you encountered.

6.2 Readability

Your code must be readable:

- Make the organization of your code as obvious as possible.
- Use descriptive names for classes, functions and variables.
- Complement your self-documenting code with comments, where appropriate.
- Choose a coding convention, and stick to it. *Consistency* is key.

6.3 Robustness

Your code must be robust. The `const` keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, the program must run to completion **without crash**.

Your code should be reasonably efficient and should not leak resources.

A tool such as valgrind can help to check your code behaves properly.

6.4 Warnings

Your code must compile **without error or warning** with `g++ -std=c++17 -Wall` on the reference docker environment. However, we advise you to check your code also with `g++ -std=c++17 -Wall -Wextra`, `clang++ -std=c++17 -Wall -Wextra` or tools like `cppcheck`.

6.5 Evaluation

Your code will be evaluated based on all the above criteria. Failure to comply with any of the points mentioned in **bold face** can (and most often will) result in a mark of zero!

7 Submission

Projects must be submitted on the submission platform before **Monday, May 22th, 23:59:59 CET**. Late submissions will be accepted, but will receive a penalty of $2^n - 1$ points (/20), where n is the number of days after the deadline (each day started counting as a full day).

You will submit a `<GROUPID>.tar.xz` archive of a `<GROUPID>` folder containing your C++ source code, and all other files needed for your code to run, where `<GROUPID>` is your group ID on the submission platform. Your archive should also contain a `Makefile` to compile your project (with a simple `make`), your assets (sprites and font) and a **2** pages report in PDF format.

The submission platform will do basic checks on your submission, and you can submit multiple times, so check your submission early!