UNIVERSITY OF
GOTHENBURG

Contra: final report

David Campos Rodríguez

Game Engine Architecture
Master’s Program in Game Design and Technology
March 15, 2020

1 Introduction

This document is intended to describe the implementation of the 2D game *Contra* [4, 1], which we have implemented for the course *Game Engine Architecture* as part of our master's studies in the University of Gothenburg, using an architecture according to the principles depicted in [10]. The code with the implementation result, as well as a compiled executable for Windows and all the necessary DLLs, is attached to this document and publicly available online in [13].

KONAMI
CONTRA
THIS SEAL IS YOUR ASSURANCE THAT Nintendo HAS EVALUATED AND APPROVED THE QUALITY OF THIS PRODUCT.

Figure 1: Original cover in the USA version.

Contra is a computer game developed and published by Konami [5] in 1987. It is a *run and gun*[3] for up to two players, originally developed for arcade machines but published later, in 1988, for the well-known Nintendo Entertainment System (NES) [6]. It is a very successful saga which has produced several sequels along the years. As explained in the Japanese original merchandising of the game, the story of the game occurs in the future year 2633, when the *Red Falcon* alien organization has established a base in New Zeland planning to destroy humanity. Two commands, Bill Rizer and Lance Bean from the Contra unit of the Earth Marine Corp (a group of elite soldiers) are sent to the island in which the base is placed to find the real nature of the alien forces.

2 Scope

This section verses about the scope of the project and what was intended to implement and what not.

The original game possess up to 8 different stages which the players will be able to advance in order, from the outer jungle of the island to the alien's lair. Due to the time constraints for the players, it was not possible to implement such a large number of features. Instead, the scope of the project was reduce to hold only the first two stages (*Jungle* and *Base 1*) of the game, since they provide a good base for what could be the development of the complete game. The reference version employed is the American NES version, and all the features of the two selected levels are reproduced with as much fidelity as possible. The intended scope of the project included also some extension that was not present in the original game, as it could be, for example, *LAN multiplayer*, but due to the lack of time this was finally not implemented.

The original Japanese introduction animation, as well as the credits scene in which we can see the two soldiers getting into the helicopter, are both out of the scope of the project. The record of high scores and the example *gameplays* that run when no key is pressed in the menus of the original game are also out of scope. They are in scope, however, the main menu and the screens shown between stages.

3 Specification

In this section we will describe the main features to be implemented in the game from a user perspective.

In general, the game supports two players with a different set of controls on each of the two stages. Each player starts the game with 2 additional lives (3 in total), after both players have lost all their lives, the game ends. Only keyboards is supported as a controller. An score system per player is present but visible only in the screens between the levels.

3.1 Menus

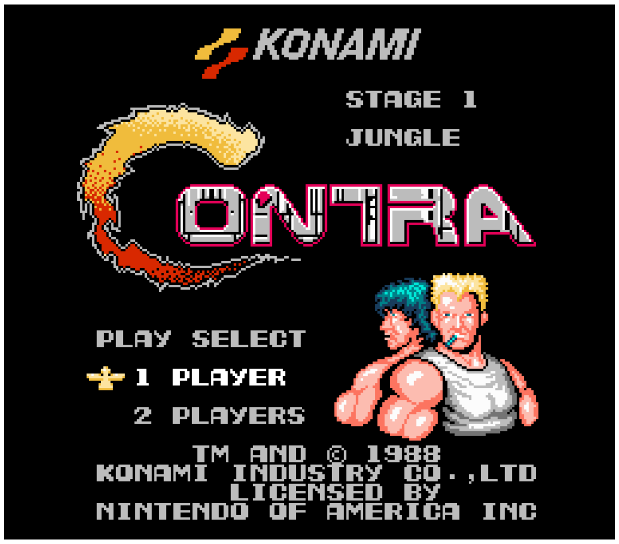

Figure 2: Main menu of the game

The project implements a main menu like the one shown in the figure 2, the player can use the keyboard arrows to choose between one or two players, after the background music ends, the game will start for the selected number of players.

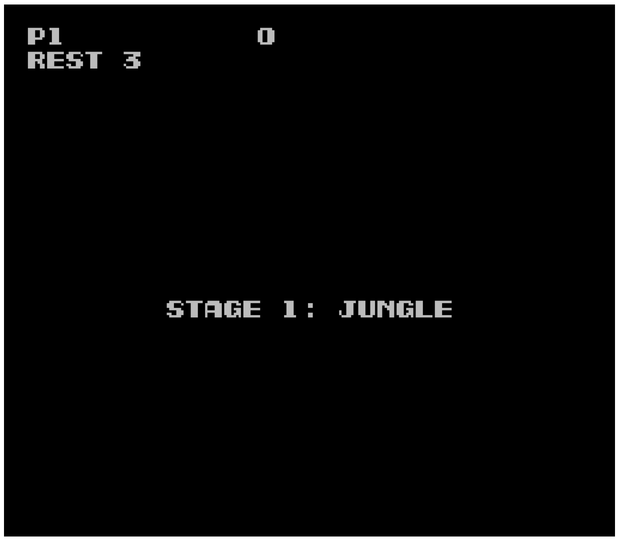

Figure 3: Screen before each level of the game

Before each level, an scene like the one in the figure 3 is displayed, informing the players about their current score, the amount of remaining lives and the current level to be played. The game moves to the next screen automatically or, alternatively, the player can press the *enter* key to skip.

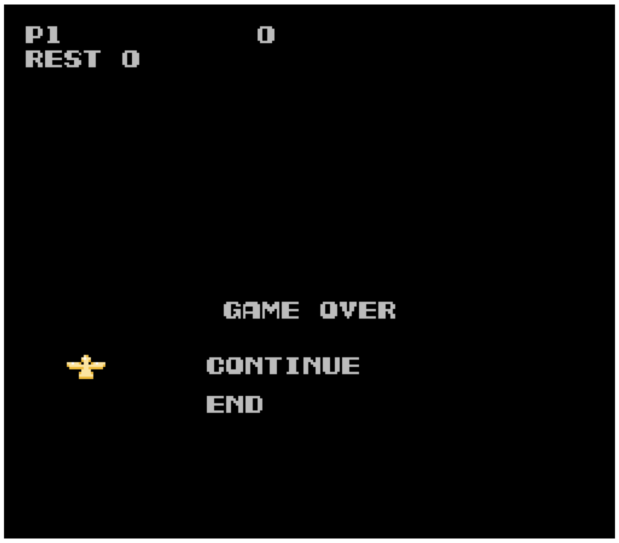

Figure 4: Menu of “game over”

If the players lose all their lives for the first time, the screen displayed in the figure 4 comes up giving the players the choice to continue trying from the level they were in or to give up and go back to the main menu. However, after the second time players will be directed to the main menu and obligated to restart from stage 1.

I added a small credits window, as shown in the figure 5, at the end of the game.

P1 13700
REST 3

CONGRATULATIONS, YOU HAVE WON!

VERSION AUTHOR:
DAVID CAMPOS RODRIGUEZ

ORIGINAL GAME BY KONAMI

THANKS FOR PLAYING

Figure 5: Final credits

3.2 Stage 1: Jungle

The first stage is a horizontal-scrolling stage in which the player can move across different platforms. The camera will follow the player as it advances forward (to the right of the screen) always keeping the character before the half of the screen in single mode and in the first three quarters in multiplayer. The character can never go back out of the screen but he can move backwards inside it, if a player is behind the other one will not be able to advance so both players are always on the screen. The background stars and the water (visible in figure 6) are animated.

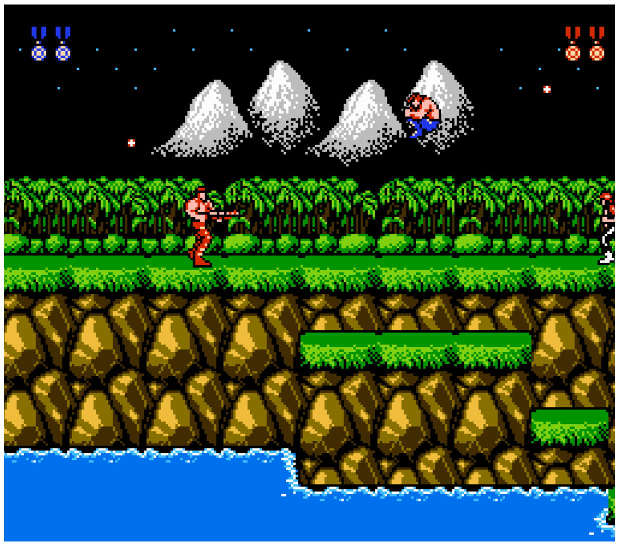

Figure 6: Screenshot from *Stage 1: Jungle*

The player can run, jump or lay down and shoot from any of this positions. Combinations of the up/down arrows and the right/left ones can be employed to shoot in diagonal directions too. The players can keep down and press the jump button to let themselves fall into lower platforms (but not into the void). At the start of the level there is a section of water at the bottom, if the player falls here the soldier will be able to swim and shoot from there, even diving to avoid getting hit, but it will not be able to jump anymore. Later in the level the water disappears, causing the player to lose a life in case of falling out of the screen through the wholes between the platforms.

3.2.1 Enemies

Along their way though the level, the following entities will actively try to kill the main characters:

- **Ledders** (figure 7): these high-rank enemies can be found standing, pointing at the player with their guns, or hidden behind the bushes, waiting for the right moment to shoot. They cannot be shot when they are hiding (although they can be spotted) and their shots fly with precision in the direction of the player. When faced frontally their bullets can easily be dodged by lying down.

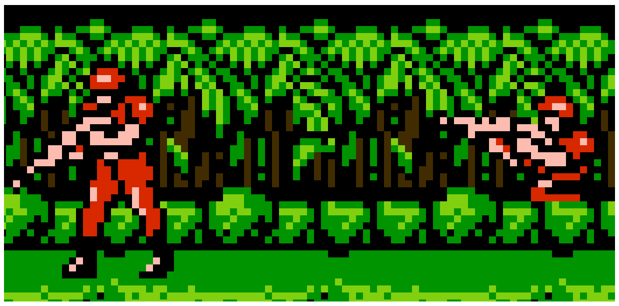
Figure 7: Two Ledders ready to shoot

- **Greeders** (fig. 8): enemies that run across the level equipped with knives and will cause the soldiers to lose a life if they reach them. When a Greeder reaches the border of a platform, he is very likely to jump in an attempt to reach the following one. There is a small chance, however, for them to turn around and continue in the opposite direction.

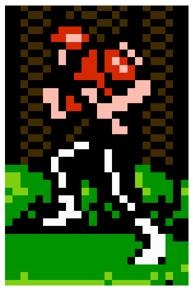
Figure 8: Run, Greeder, run!

- **Rotating canons** (fig. 9): these cannons remain covered until the soldiers approach them. Once they uncover, they will slowly rotate around trying to follow our protagonists and shooting at them whenever they are in the right position. They can be killed by shooting them several times with any weapon.

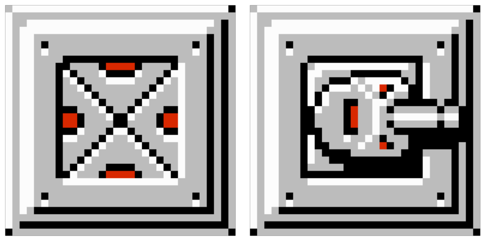
Figure 9: Rotating cannon covered and uncovered

- **Gulcans** (fig. 10): Gulcans are deadly canons, hidden underground waiting for the players to approach in order to come up and shoot them. Their shooting rate is much faster than the one of the rotating canons but their angles of rotation are limited. They can also be killed the same way the rotating canons are.

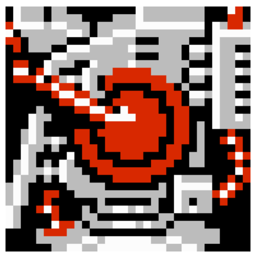
Figure 10: A Gulcan aiming 30º up from the floor

- **Explosive bridges** (fig. 11): The two bridges of the level are equipped with remote-controlled explosives which will be detonated when the player approaches them. Their explosion does not kill the player, however, but it may cause the soldier to fall into the river bellow.

Figure 11: *Wrong jump, Bill*

- **Defense wall** (fig. 12): big metal structure which protects the entrance to the first base. It can be considered the final boss of the first stage. Two blaster cannons fire constantly from it as the player approaches the wall. Upon destruction of the main door, which can be achieved by repeatedly shooting at it, the wall will explode letting the soldiers pass and finishing the level.

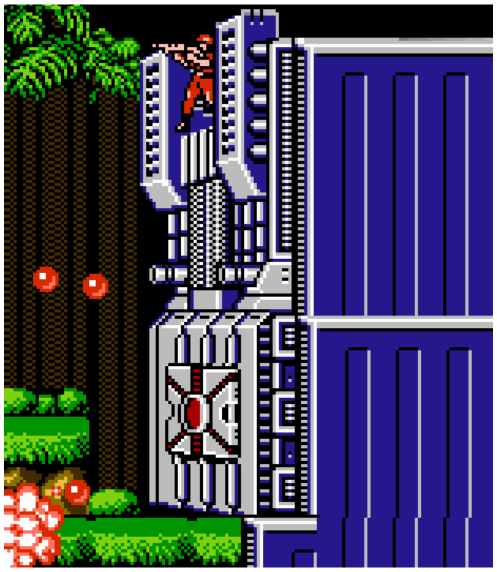

Figure 12: The defense wall protects the entrance into the base 1

3.2.2 Pickups / weapons

Along the level, the player has the option to shoot at some special entities which will provide her with different pickups, while most of them are different weapons, there is also a special pickup which will increase the speed of the bullets shot by the player, allowing also for a faster, more continuous rate of shooting (as the number of bullets in screen is limited).

There are two kind of pickup holders that the player can find:

- **Protected core** (figure 13): a core in the shape of the Red Falcon logo brights protected by two opening and closing metal doors. When the doors are open, the core can be shoot to drop the pickup to the player.

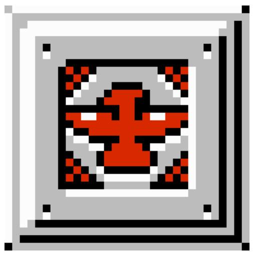

Figure 13: A shinny core with the Red Falcon logo hiding some pickup.

- **Flying capsule** (figure 14): this capsules will cross the screen from left to right following a sinusoidal path through the air. Shoot them to obtain the pickup they hide inside.

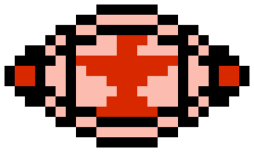

Figure 14: A flying capsule hiding some pickup.

Each pickup is identified by a letter, while the already-mentioned rapid fire one (identified by *R*) causes an upgrade to the current weapon (however, it does not affect the laser), all the other pickups provide different weapons for the player to use. The possible weapons the player can hold are as following:

- **Rifle** (default): this is the weapon the soldiers will spawn with. It is a manual weapon which fires in a straight line, up to 4 bullets in screen.
- **Machine gun** (M): this is a semiautomatic weapon which will shoot up to 6 bullets in screen.
- **Fire gun** (F): weapon able to throw flames which will slowly fly describing circles into the shooting direction. Really useful to hit rotating canons when lying down.
- **Spread gun** (S): this is considered by the fans as the best weapon of the Contra series. The gun shoots a group of 5 bullets at once (up to 10 on the screen) which spread covering a big portion of the screen.
- **Laser gun** (L): this weapon shoots a hot laser beam which will continue through enemies allowing to kill several enemies in a row. The main difference with all the other weapons is that the laser resets its beam each time it is shot, so it can be harder to use effectively. It is not affected by picking the rapid fire.

3.3 Stage 2: Base 1

The second stage employs a new projection (shown in figure 15) which creates the illusion of a 3D, perspective camera following the player from behind as she travels across the base. The base is divided in different rooms, each one initially blocked by a laser beam which can be turned off by shooting at the cores in the back of the room. The cores are provided with two doors which open and close protecting them at regular intervals and have the appearance shown in figure 16.

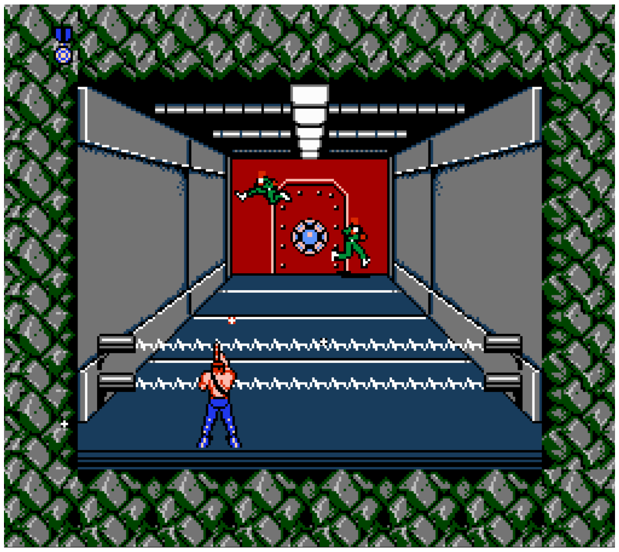

Figure 15: The second stage has a pseudo-perspective camera

The character can move left and right, lay down on the floor, jump or advance forward. However, trying to advance before the laser has been switched off will electrify the soldier causing him to be vulnerable for a short period of time.

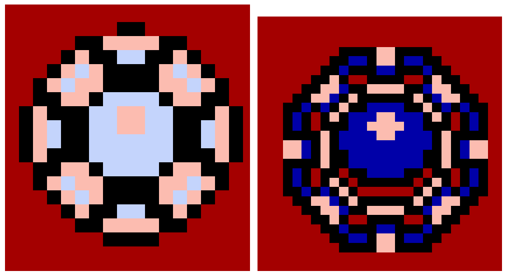

Figure 16: Cores in the second stage

In this levels we find the following types of enemies:

- **Core canons** (figure 17): these canons can be found sometimes in the back wall of the rooms of the base, aimed at protecting the cores. Like the cores, they have protective doors which open and close periodically. They shoot precise bullets at the player, but they can be avoided by jumping or laying down.

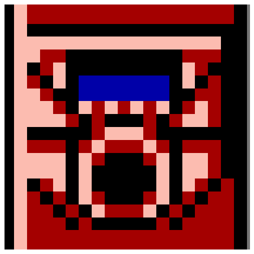

Figure 17: Core canons

- **Ledders** (figure 18): already present in the previous level, but now they will cross the screen from one side to the other, sometimes shooting continuously and sometimes stopping for a while to shoot before moving. Sometimes, they will cross the screen jumping in an attempt to dodge the player bullets. Apart from rifles, some Ledders can port explosive capsules which they will throw at the player, causing her to die even if laying down. Red Ledders can be found carrying pickups that they will drop when killed.

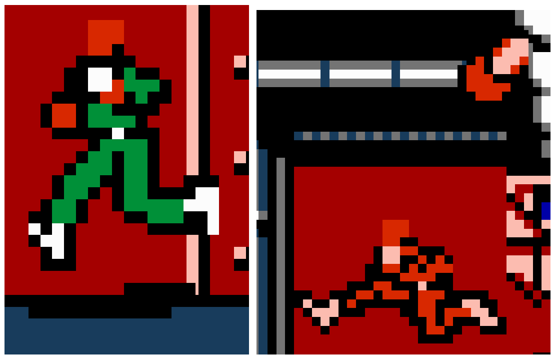

Figure 18: At the left, a Ledder runs from right to left, at the right, a Ledder which drops pickups jumping and throwing a explosive capsule

- **Darrs** (figure 19): rolling explosives which are shot from the back of the room. The player can shoot them to make them explode before they reach the front of the screen or, instead, try to dodge them by jumping or moving laterally.

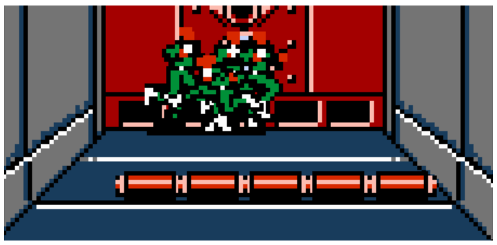

Figure 19: Five *Darrs* come rolling towards the player

- **Garmakilma**: final boss of the level, reached after all the previous rooms have been cleared. When facing the boss the player will not be able to lay down anymore. The real boss, the Garmakilma eye (shown in figure 21), will be hidden until the four cores in the screen are destroyed. To guard this cores two triple canons as the ones displayed in the figure 20 shoot groups of three bullets and cover themselves periodically.

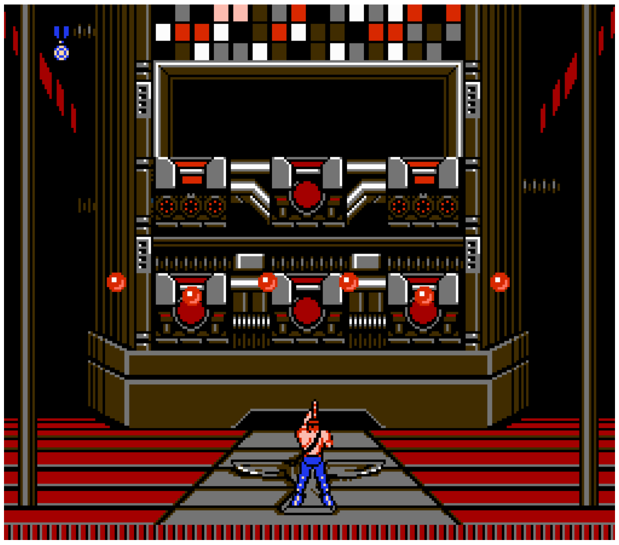

Figure 20: Garmakilma, final boos of the second stage

Once the four cores have been destroyed the eye will show up, shooting the player with big rings of light which can be shot back to make them explode.

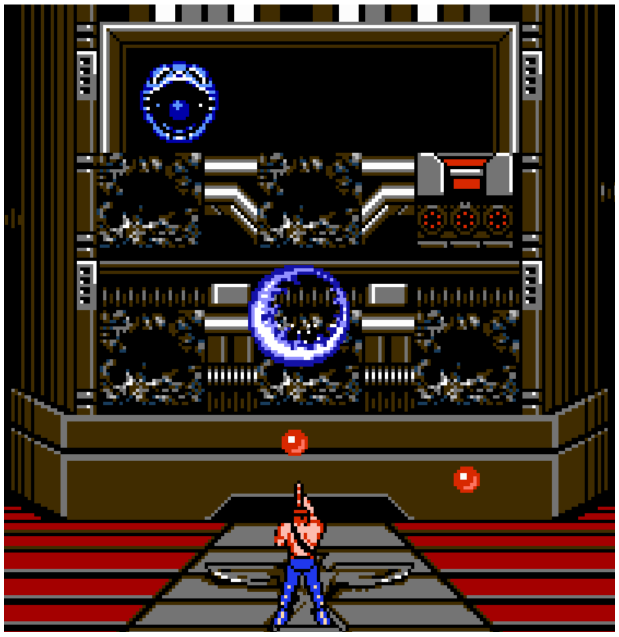

Figure 21: The Garmakilma Eye shooting light rings

Once the Garmakilma eye is destroyed, the stage finishes and the soldiers will go into the elevators on the sides to get up and out of the base. This is where our version ends and shows the final credits.

4 General Overview

In this section we will try to give, briefly, some general concepts about the architecture of the game.

Our system is an *entity-component system* [9], in which our entities are the game objects. However, the implementation is not pure and, although entities are expected to contain no *business logic* at all and provide only the logic required to identify them and manage their components, in our case we include the management of the position inside the game object instead of, for example, a `Transform` component like Unity does [2]. Despite of this, our implementation is nearly pure and almost all the game logic is implemented in the components managed by the game objects. Some concrete game objects, however, have been extended by being considered special, as they are the `Game`, `BaseScene` and the different types of `Level`. In section 6 we will talk a bit more about the composition of game objects in our game.

Our system also implements a basic messaging system – driven also by the own game objects – which is used, for example, to modify the scores or communicate the menus with the main game class. This system is based on the well-known *Observer* pattern [8].

To improve performance and enhance data locality object pools are used. However, given the tight time and that no performance issues at all were found during development (right now the game runs at over 150FPS on almost any tested computer) their use has been in fact limited to some extent, in order to speed up the development a bit[1]. They are mainly use to store the bullets which the enemies and players shoot, having a general bullet pool for the enemies, one for each player and weapon, and some special one on the bosses which need bullets with a special setup. Again, more on the improvement possibilities will be discussed in section 6.

The figure 22 shows a UML [12] class diagram aimed at providing a quick overview to the structure of the program in general lines. The main thing to notice is how the `Game` saves a reference to the current scene, which can be a `Level`, as shown in the diagram, or could be, for example, a menu instead. The scene manages the objects on it and the level extends this functionality by providing a queue of objects to add

[1]A lot of the instantiation and creation occurs, still, at the start of the level, which saves some time during the game after all.

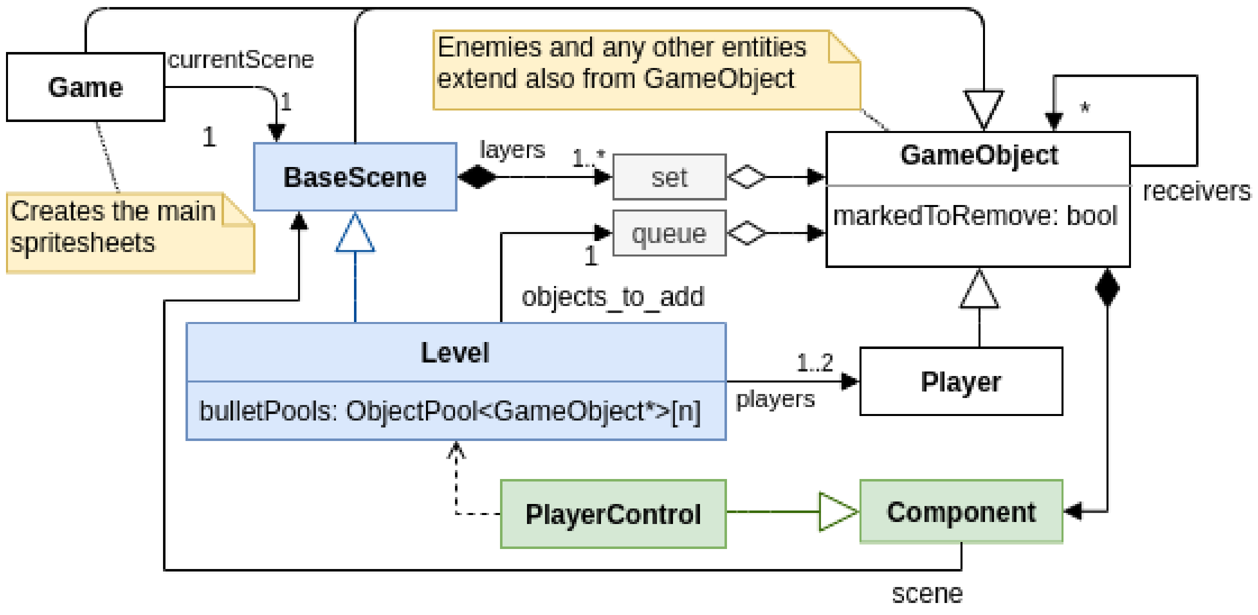

Figure 22: UML class diagram of the main parts of the system

(more on this in section 5.3). Notice also the association from `GameObject` to itself, which is the key part of the messaging system.

5 Implementation

In this section we go into more detail about each of the subsystems in the game and how they are implemented.

5.1 Rendering

The rendering corresponds to the part of the system relative to drawing on the screen. To this purpose we have implemented two different components which are able to draw images into the screen based on a sprite-sheet, represented by the class `Sprite`. The sprite-sheets for the players and enemies are loaded on creation by the `Game` class, while the sprites for the background of the scenes are created per scene and some other punctual sprite-sheets are lazy-loaded (like the sprite-sheet for the explosive bridges). The two render components we have are `SimpleRenderer` and `AnimationRenderer`, which we can observe in the UML class diagram [12] in figure 23.

On the one hand, the `SimpleRenderer` works, as it indicates, in a simple way. It just implements the `Update` method so each time it is called for update it draws the associated part of the sprite-sheet in the adequate position based on the game-object position (the attributes to achieve this have been omitted from the diagram, but they are just the coordinates for the rectangle on the sprite-sheet, the size of the rectangle in the window and an anchor which marks the exact point where the object position point must be inside the drawing. The camera coordinates inside the scene are managed by `BaseScene` (as this class is in charge of drawing the background too), which exposes a public getter to access them. This generates some (small) dependency from the renderers to the scene.

On the other hand, the `AnimationRenderer` works with a slightly more complicated logic. It is able to manage animations composed by rows of adjacent frames with the same size inside the sprite sheet. It exposes several methods to ex-

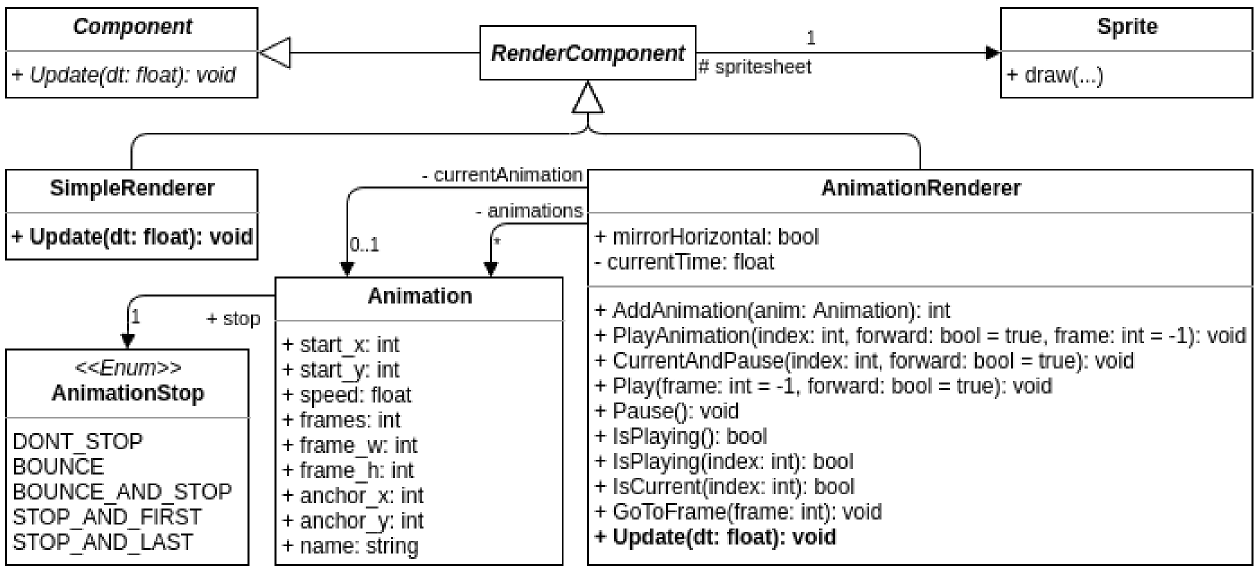

Figure 23: UML class diagram which describes most of the design of the rendering system

change and manipulate the animations in course as well as it provides the options to flip them horizontally. It also supports a handful of different configurations on how to act when reaching the end of an animation. The possible behaviours are as follows:

- `DONT_STOP`: when reaching the last frame of the animation, we will continue from the start again, looping it until external pause.
- `BOUNCE`: when reaching the last frame of the animation, we will go backwards until the first frame and then start forward again.
- `BOUNCE_AND_STOP`: when reaching the last frame of the animation, go backwards until the first frame and then stop. The current implementation does not handle correctly starting the animation backwards (since it was not needed).
- `STOP_AND_FIRST`: once we reach the last frame of the animation, change to the first one and pause.
- `STOP_AND_LAST`: once we reach the last frame, stay paused on that frame.

This simple but powerful animation system allowed for a much faster development, as a big part of the workload for this project was on the large amount of animations needed.

5.2 Collision

To detect collision, we have a series of components we call *colliders*. Only box colliders have been required in the implementation of the game, but the design employed is oriented to facilitate the addition of new colliders in the future. In each frame, colliders check collisions with other colliders registered on their *check layer*. Notice that a collider can be registered in one layer but check collisions with colliders from another layer, this is used in the game, for example to make bullets check collision with the player but not among themselves. The collision check is performed without ensuring whether other colliders have been already updated for this frame or not. This could cause some lack of precission but it works more than fine for this project. Improvement on this will be discussed in section 6.

The figure 24 shows a UML [12] class diagram with the main elements that compose our collision system. The main idea is to have a

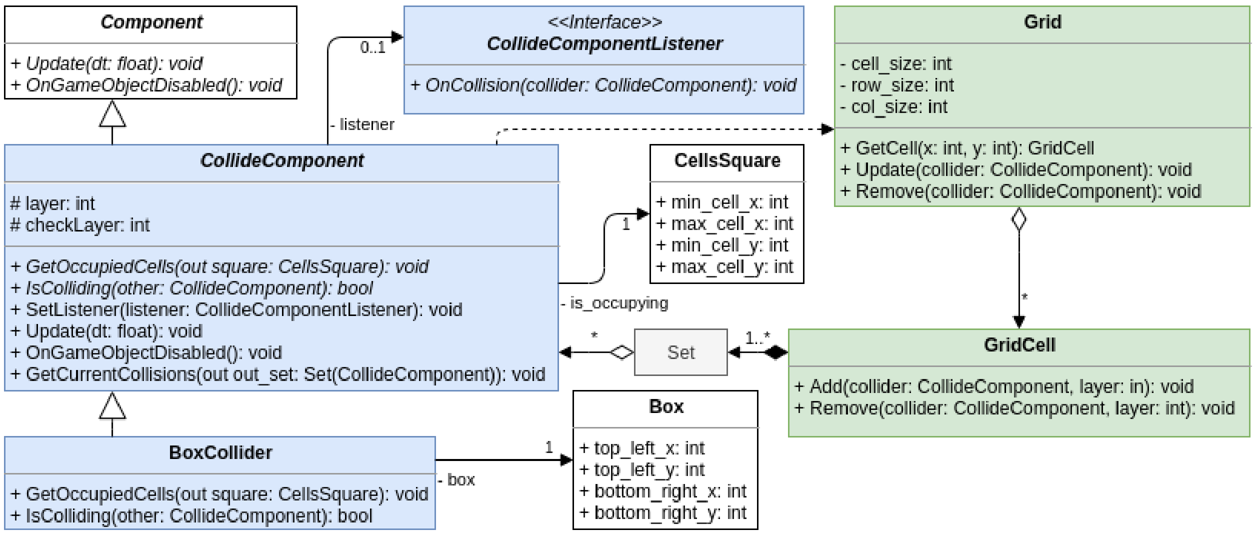
Component
+ Update(dt: float): void
+ OnGameObjectDisabled(): void
<<Interface>>
CollideComponentListener
+ OnCollision(collider: CollideComponent): void
0..1
- listener
Grid
- cell_size: int
- row_size: int
- col_size: int
+ GetCell(x: int, y: int): GridCell
+ Update(collider: CollideComponent): void
+ Remove(collider: CollideComponent): void
CollideComponent
# layer: int
# checkLayer: int
+ GetOccupiedCells(out square: CellsSquare): void
+ IsColliding(other: CollideComponent): bool
+ SetListener(listener: CollideComponentListener): void
+ Update(dt: float): void
+ OnGameObjectDisabled(): void
+ GetCurrentCollisions(out out_set: Set(CollideComponent)): void
CellsSquare
+ min_cell_x: int
+ max_cell_x: int
+ min_cell_y: int
+ max_cell_y: int
1
- is_occupying
*
Set
1..*
*
GridCell
+ Add(collider: CollideComponent, layer: in): void
+ Remove(collider: CollideComponent, layer: int): void
Box
+ top_left_x: int
+ top_left_y: int
+ bottom_right_x: int
+ bottom_right_y: int
1
- box
BoxCollider
+ GetOccupiedCells(out square: CellsSquare): void
+ IsColliding(other: CollideComponent): bool

Figure 24: UML class diagram of the most relevant parts of the collision system

base, abstract `CollideComponent` class which receives the update call and implements a Template pattern by calling two abstract methods `CollideComponent::GetOccupiedCells` and `CollideComponent::IsColliding` to check collisions with other possible colliding `CollideComponent`'s.

To speed up the collision detection, instead of a brute-force approach we use a grid. A grid is an spatial data structure which stores the elements into cells depending on their positions in the game space. Our grid is created by the level and a reference to it is provided to the colliders on creation. The `CollideComponent` class stores the currently occupied cells and makes the proper calls to update the grid on each update. It also listens to certain events of the life-cycle of the game object the component is attached to, like `OnGameObjectDisabled`, to make sure to remove the collider from the grid whenever it is not needed anymore.

To get more specific around the update of the colliders we provide also a UML sequence diagram [12] in the figure 25. In this diagram, the collider being update is named `col` and its lifeline has been duplicated (it is drawn twice), one for the base class `CollideComponent` and one for the concrete one `BoxCollider`, to ease the reading. Please, keep in mind both lifelines correspond, in fact, to the same object. Some steps of the process have been skipped so the main idea of the sequence is as clear as possible.

Through the use of the template pattern [8] we obtain a hierarchy of colliders which can be easily extendible to new types of colliders (for example, if we want to add a new *circle collider*).

For other components to be able to check the current colliding colliders at any moment, the method `GetCurrentCollisions` has been extracted from the update and made publicly available. To speed up the execution over multiple calls, the `Grid` class keeps a map per frame with all the collisions that have been already detected in the same frame (the corresponding methods and attributes, however, are not displayed in the figures 24 and 25). This cache gets cleared before the update of the game objects by the `BaseScene` class.

The notification of the detected collisions is done, once again, through the implementation of an observer pattern. The observer interface is called `CollideComponentListener` and can be implemented by any component that desires to listen to collisions on the component. The collider only allows to have one listener as maximum at any time, however this has been more than

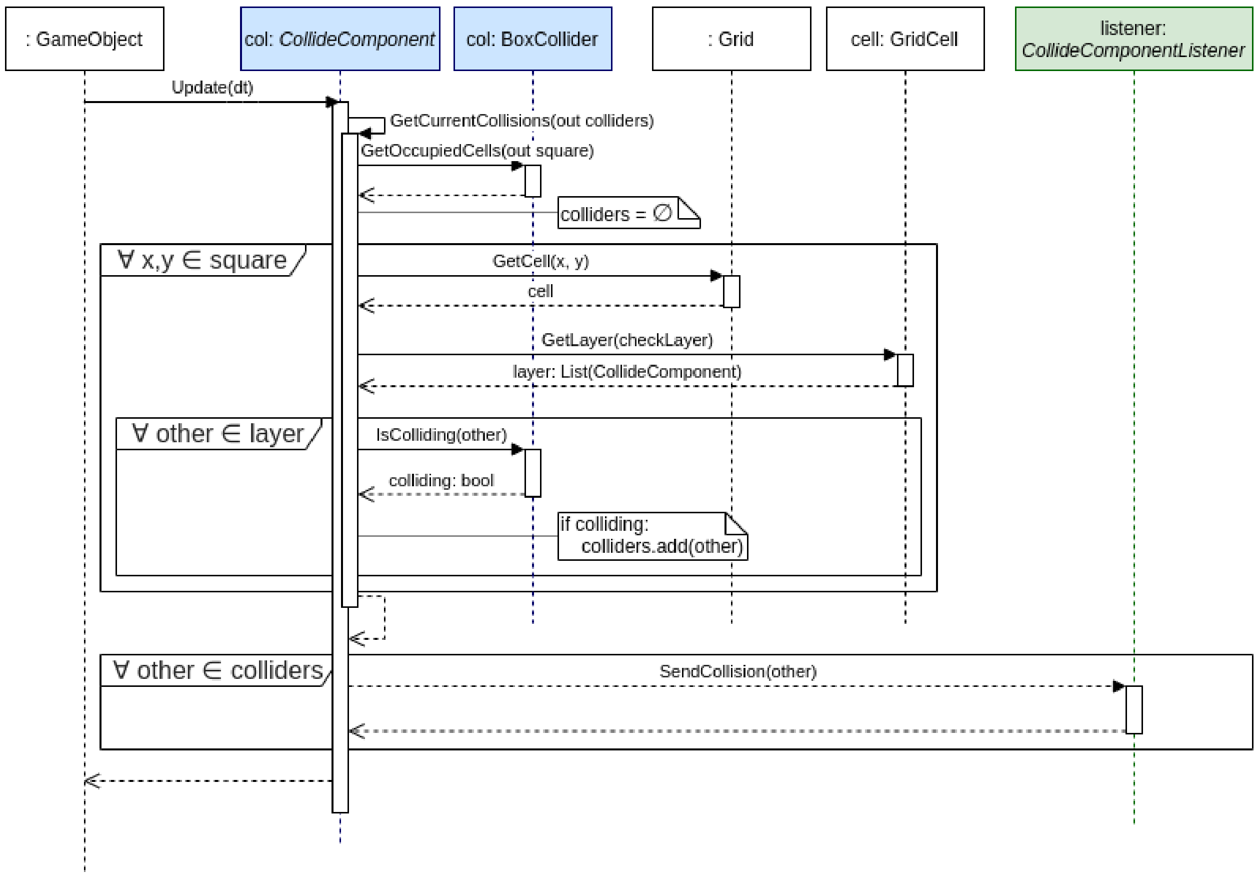

Figure 25: UML sequence diagram of the (simplified) interaction for collision update. Notice that the object `col` has been unrolled into two lifelines: one for the abstract, base class `CollideComponent` and another one for `BoxCollider`. However, it is just one object.

enough in our case.

5.3 Scene object management

The BaseScene class is, as its name indicates, a base class for all the scenes of our game. It is able to draw a background (which can be animated by providing a space shift for the animation and the time length for each frame) and manage an arbitrary amount of "rendering layers". Each "rendering layer" is a set of non-repeated game objects which will be called for update on each frame. The layers are ensured to be updated in order, so their main purpose is to establish an order of update for the render components between the objects, performing this way a *painter's algorithm* implementation.

To avoid data races inside the layers (as sets in C++11 are not safe to be iterated over addition or removal), the BaseScene administers the addition and removal of the game objects to the layers. It exposes the methods AddGameObject and RemoveGameObject in order to do so. The AddGameObject method introduces the game object into an unsorted queue which will be flushed into the layers before the start of the next frame, after all updates have been finished. The RemoveGameObject method is no more than a wrapper that sets the markedToRemove attribute of the GameObject to true. At the end of the frame, all objects marked to removed are removed from their layers and, depending on their onRemoval parameter, can be also destroyed.

5.4 Level loading

The structure for the levels of the project are stored in YAML files [7]. Whenever a NEXT_LEVEL message is received by Game, a current level counter is increased and the folder data/level{counter}/level.yaml is opened to load the level. Once the level is loaded (it has been created), it is initiated and replaces the current scene of the game[2].

The level YAML file contains information about whether to find the background image and the music, as well as how to make the animation of the background and some other relevant data. Apart from that, the file contains lists for each type of elements in the level with the necessary parameters to create them. While some of the elements are directly visible (like the *Ledder* objects), other entities do not have a visual representation (like the *Greeder spawner* objects, which spawn *Greeders* while they are on the loaded limits past the sides of the screen). It is out of the scope of this document to explain in detail the structure of this files and what each of the options does. However, the files are reasonably understandable and, with the help of a look at the code, it should be easy to figure out what everything does.

Depending on the type of level, the initiation of and addition of the created game objects to the different "rendering" layers in BaseScene for them to be updated occurs in a different way: while HorizontalScrollingLevel creates a queue in which it sorts the game object by their spawning positions (in order to go including them into their layers as the camera, which can only move in one direction, reaches them), the PerspectiveLevel follows a cycle of initiating and later killing and cleaning of each screen the player goes through. To do so, it keeps a list of behaviour components implementing a Killable interface, which allows to call Kill on all of them once the *core* for the current screen is destroyed, destroying them out (clearing the screen) after some animation time.

5.5 Other

There are some other systems we do not want to go into detail with, but that are important to give some nuances about.

The player system has certain level of com-

[2]Of course, the previous "current scene" is properly destroyed when this transition happens.

plexity caused by the different controls relative to each of the types of levels. The players are controlled by the `PlayerControl` components. `PlayerControl` is, however, an abstract class which implements a template pattern [8], providing virtual methods for the child classes `PlayerControlScrolling` and `PlayerControlPerspective` to implement which will be used during update. To implement the shooting, since the behaviour needs to be different depending on the weapon being carried, an strategy pattern [8] was used, extracting the firing behaviour into an external `Weapon` class which encapsulates how to initiate the bullets and add them into the scene.

The player stats on the game (the current weapon, the number of lives and the score) need to be able to be rolled back to their previous state when we decide to continue a level, in order to do this we employ a kind of memento pattern [8] in which we store this part of the state into an external object so we can easily replace it by any other when needed.

All the objects that fall in the game use the `Gravity` component. This component stores a vertical speed for the object and tracks its acceleration and relation of the object with the floors. To do this, the levels can create a `Floor` object, which is read from an image and works like a mask to mark the pixels which are water, floor or air. There are two types of floors: the floors that allow to fall through and the ones that do not (the player falls through the floors by pressing down and then jump). The floor of the level can be updated: for example, the exploding bridges update the floor in the pixels they occupy to be air after each explosion. The perspective levels do not have a floor at all, this is not a problem for the gravity component as it also supports a *base floor*, which is just a y value which will be always consider as floor by the gravity component no matter if the floor object is instantiated or not. The perspective level manipulates this base floor value to reproduce the fake perspective illusion.

Rotating canons (and *Gulcans*) shoot in hard-coded angles; this is intentional. The first implementation, which used `atan2` to calculate the exact shooting direction broke some of the options available to the player in the original game. After some testing we came to the conclusion angles on the original game are not exactly proportional for all the shooting directions, so the easiest solution to be as loyal as possible was to just hard-code the values for each direction.

6 Possible improvements and future work

In this section we would like to mention some problems we acknowledge in our system, and propose some possible solutions which were not implemented due to the time constraints and the evolution of the code along the development.

First, object pools are not used to performance improvement, in fact. The bullet pools are used for the bullets but they are very small and they do not make a great performance contribution. This is this way because performance was never a problem given the small requirements of the project compared to computational capacities nowadays. To improve its usage, all game objects could be stored in object pools when the level is created, so they are all stored sequentially in memory, getting a better data locality during the update. For further data locality, the components themselves could be stored in pools.

The behaviour components of the project support, in many cases, way too many responsibilities and could be split into smaller, more reusable and maintainable components. For example, many if not all the enemies behaviour components are in the end state machines, which usually are simply implemented through the usage of an state variable and a `switch`. The state machine could be abstracted into a single, unified class for better code maintenance. Another example, related to this part of the game too, are the lives or the shooting of the enemies:

while sharing sometimes some components, in general there is quite a relevant amount of repeated functionality which could take advantage of some extraction into abstract, generic components for live management and shooting. We are not satisfied, either, with the coupling in the design. Despite of having kept care of not coupling excessively some concrete parts of the system, some classes are way more coupled that we really would like them to be and some refactoring in this sense could clearly enhance the neatness of our code.

Hierarchical (composited) game objects are another case of multiples solutions to the same problem. They are not so common but some are present, examples would be the `BaseScene` and the final bosses. To improve this and ease future extension of the system all this functionality could be abstracted into a generic model of composition for the game objects (implementing the composite pattern [8]) so all the nested game objects in our program follow the same single system of composition.

Colliders currently store the cells they occupy and need to keep care of removing themselves from the cells when they are disabled or destroyed. This is a problem carried from a previous, reused implementation which probably is not necessary. The grid could be, instead, completely cleared per frame and then repopulated again. This would avoid a handful of problems: detecting collisions twice, non-updated collisions, need for removal when disabled, etc. Also, the detection of the collision between box colliders is performed in the `IsColliding` method by checking the other collider is a `BoxCollider`, it is not hard to see that this solution does not scale well with the introduction of new classes of colliders[3]. To do so, it would maybe be better to implement a mediator pattern [8] with a mediator which knows how to check collisions between different colliders instead of having them doing it for themselves.

The instantiating of the level could benefit from the use of abstract factory in some cases, like instantiating the player, to better decouple from the internal logic dependent on the type of level and facilitating to add new *families* of components for new level types in the future.

Text is being written with direct calls to the engine function to draw text, a text component could be added to encapsulate this functionality and provide more flexibility.

It would be really beneficial too to change the way the input system works to make it more abstract. Right now it makes use of hard-coded values for the different actions and it is a bit limited. It would be nice to have a more flexible input system which allowed not only to configure the controls by the user, but also abstracted the concrete input device facilitating future implementation of controller support.

Finally, the factor the pixels in the art are multiplied by in size (to be displayed on the screen) is determined by the constant `PIXELS_ZOOM`. The program has a big list of usages of this constant (every time measures have been made relative to the art but need to be converted to program coordinates), which makes it a really big problem to modify it into a dynamic selection of the scale (needed, for example, to get a nice full-screen behaviour on any size of screen). The solution we propose to this would be simply putting the time of going case by case replacing the usage of the constant by the proper measure. For example, a big chunk of the cases are just hardcoded dimensions for colliders and visual references, which could easily work just by resetting their scale to 1 (essentially, removing `PIXELS_ZOOM`) and implementing the scale in the kernel, in the own sprite objects, so they just get drawn with a dynamic scaling configured in a single point in the end. However, there are other cases where this value is used in a static manner, and some more work

[3]Imagine adding a circle collider: now the box collider needs logic to check collision with a circle collider, and the circle collider needs to check collision with box colliders, but both of these algorithms are, in the end, the same one.

would be required to make them initiate in run-time before getting rid of the constant.

7 Conclusions

Despite of all the room for improvement, we consider we made a reasonably good job for the time available. The original game has been fairly faithfully recreated (at least the two first levels of it) and, while having a lot of refactor in the code needed, it is not a giant, unmanageable mess and it allows for a fair level of maintenance and extension. Of course, the absolutely first thing needed in vision of future work would be the implementation of automated testing (unit tests at least [11]), but this was clearly out of the scope of this course.

In the end, we believe the project shows a basic understanding of how a game engine works in general lines and a set of resources we can make use of and get deeper in to make a more robust system in a future, hypothetical, less-time-constrained scenario.

References

[1] Anonymous, *Contra (NES) - online game* in `retrogames.cz`. Visited on March 15th.
`https://www.retrogames.cz/play_022-NES.php`

[2] Anonymous, *Unity - Scripting API: Transform* on Unity documentation. Visited on March 15th.
`https://docs.unity3d.com/ScriptReference/Transform.html`

[3] Bielby, M., *The YS Complete Guide To Shoot-'em-ups Part II*, Your Sinclair, 1990.

[4] Collaborative, *Contra (video game)* on Wikipedia. Visited on March 15th.
`https://en.wikipedia.org/wiki/Contra_(video_game)`

[5] Collaborative, *Konami* on Wikipedia. Visited on March 15th.
`https://en.wikipedia.org/wiki/Konami`

[6] Collaborative, *Nintendo Enterainment System* on Wikipedia. Visited on March 15th.
`https://en.wikipedia.org/wiki/Nintendo_Entertainment_System`

[7] Collaborative, *The Official YAML Web Site*. Visited on March 15th.
`https://yaml.org/`

[8] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; ("Gang of Four"), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] Martin, A. *Entity Systems are the Future of MMOG Development*. Visited on March 15th.
`http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-dev`

[10] Nystrom, B. *Game Programming Patterns*. Online. 2014.

[11] Osherove, R., *The Art of Unit Testing: With Examples In C#*. Manning Publications, 2013.

[12] Rumbaugh, J.; Jacobson, I.; Booch, G. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1998.

[13] Campos Rodríguez, D.; *Contra*, GIT repository for this project in GitHub.
`https://github.com/david-campos/Contra`.