# Chrono2D

# a physics-centered platformer game

Mohamed Ahmed Bouha
Benjamin Dupuis
Paul Moreau
Damien Poquillon

Project coordinator: Kiwon Um

# Introduction

The goal of our project was to develop a playable video game leveraging a rigid body physics engine. We chose this project as it is a direct and interesting application of course concepts like collision handling, combined with the creative freedom associated with the creation of a video game (a genre that all of us enjoy and play on a regular basis).

For our project, we chose to create a 2D platformer. While it is a classic genre, it provides a well documented and rich environment, while still being rather simplistic. This last point was important to us, as we all have no-experience in game development, and wanted our project to be realizable in the short time-frame of the course.

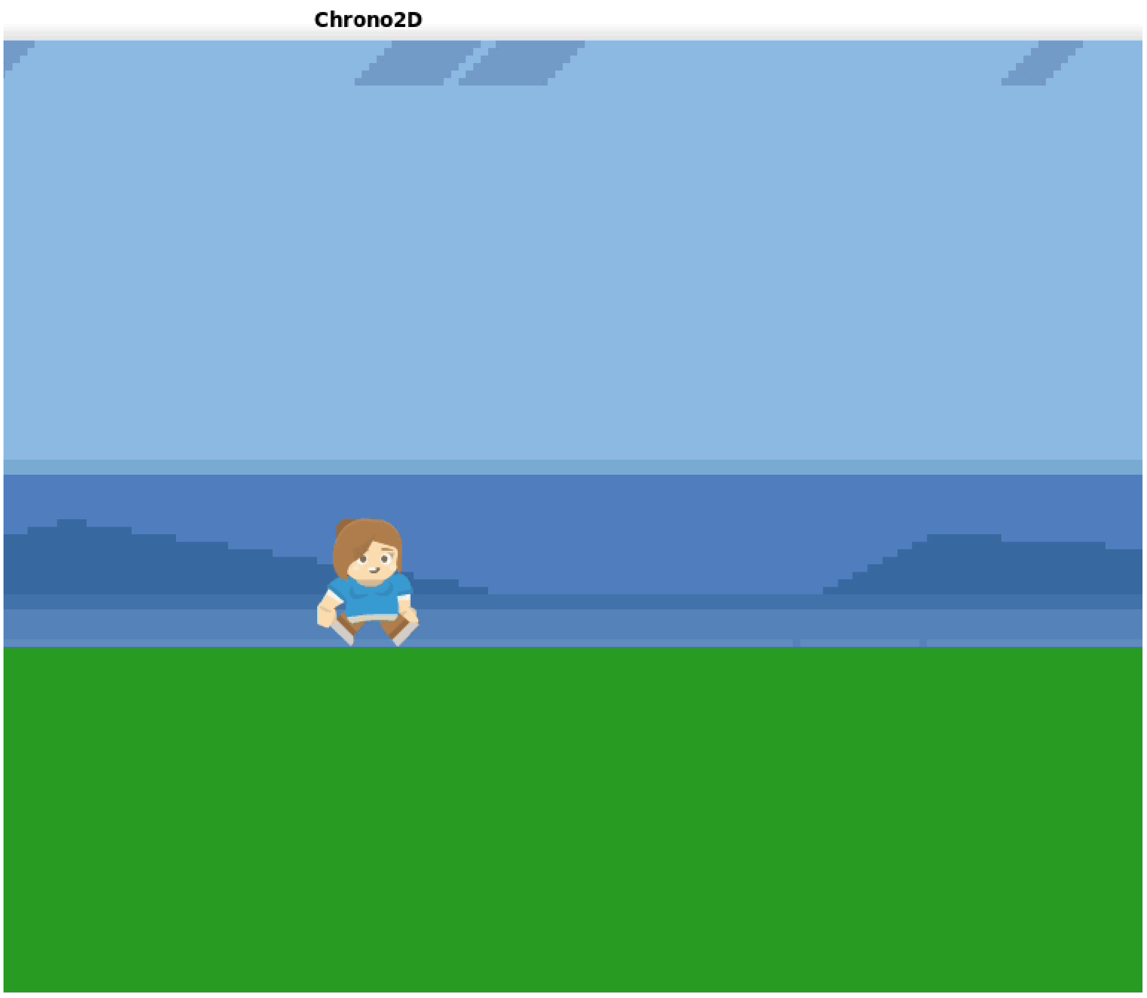
To add a unique twist and create interesting puzzles, we introduced a core mechanic: the ability for the player to freeze time for all surrounding objects. This transforms static environments into dynamic ones, where the player can halt objects mid-air to create platforms or clear paths.

This report details our journey from concept to final implementation. We will discuss our choice of technologies (C++ with SFML/Box2D), the implementation of our core gameplay mechanics, the challenges we faced, and how our final result compares to our initial vision.

# Our final game

Our final game is a 2D platformer where players must navigate levels filled with physics-based challenges and puzzles. The core mechanics revolve around interacting with dynamic rigid bodies and using a time-freeze ability to manipulate the environment. This time-freeze mechanic lets players pause the world's physics at critical moments, allowing creative solutions to otherwise impossible situations.

Each level is built from a variety of objects that form the building blocks of our world — balances that are anchored to the world and react to the player's weight and also other dynamic objects, ropes that sway under the player's weight, dynamic objects that can be pushed or stacked, and trampolines that launch the player or other objects into the air. With just these simple elements, we designed multiple levels offering fun and engaging puzzles.



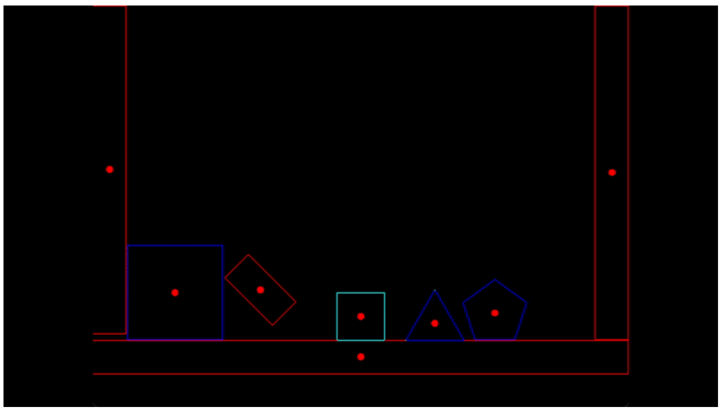
# Initial technologies and design choices

The first step we had to take was to choose the libraries that we would use for our project. Given our choice of making a platformer, this decision broke down into two key areas: the physics simulation itself, and the rendering and user interaction layer.

Our first option was to use a fully-fledged game engine built for C# or C++ like Unity or Godot. However, two of our goals with this project were to develop our ability to work as a team on a development project and to apply the concepts seen in class. We felt like using an engine would make it harder for us to fully grasp the base low-level physics for rigid bodies, as well as not allowing us to focus as much on the code itself, given most of the work with these engines is done in the software itself.

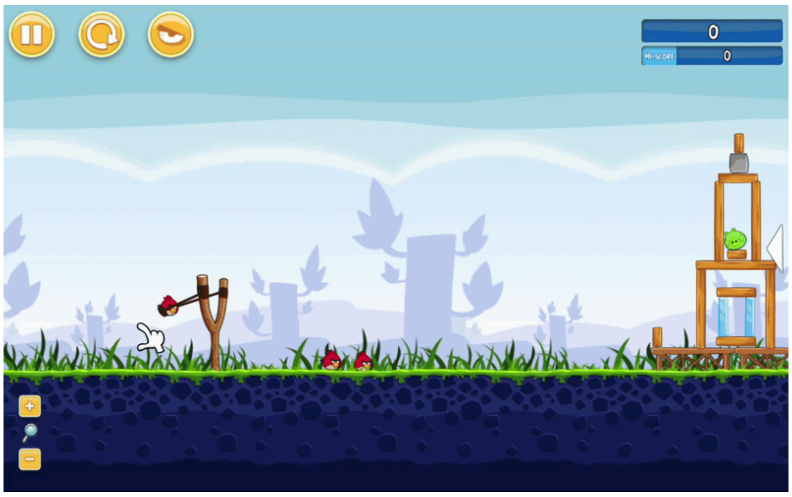
## The physics core: Box2D

We briefly considered implementing a minimal physics engine from scratch. At the beginning, we decided to explore both this approach and the use of Box2D (discussed below). However, a week of development, while making us confident in our ability to develop the game engine from scratch, made it clear that developing a stable and robust engine would consume the large majority of our project time.


Our own Physics engine after a week of development
The code for the this engine is in the "ECS" branch

This engine uses Position based dynamics and was inspired from this paper (PBD Paper), The core concept is that we treat each point as its own physics entity and apply changes in velocity and position as we see fit. These changes come from multiple factors for example gravity, collision resolution and constraints. The constraints are mainly to keep an object's structure intact (force 2 points to never be more than d distance apart). Combining all of these aspects into a physics solver gives us our physics engine, and to resolve collisions we used the Separating Axis Theorem. And the entire engine was wrapped in an Entity Component System (for maintainability). We ended up not using this version because we understood that if we wanted to have a more robust physics solver it would take too long and leave us with not that much time for level design even though it showed promising results for simple simulations.

So we finally settled on Box2D, a mature, open-source, 2D rigid body simulator. This choice is low level-enough for us to fully grasp what is going on, while still significantly speeding up the development process. Box2D is a game simulation engine, meaning it does not have any rendering capability - an issue we will have to address - .



While Box2D is mainly popular amongst hobbyists, one famous series used it:
most of the Angry Birds games are made in Box2D

## The Visual & Interactive layers: SFML

For rendering graphics, managing the game window, and handling user input, we selected SFML (Simple and Fast Multimedia Library). As novices, we once again wanted to avoid the high-level abstractions of a full-fledged game engine, which might obscure the underlying mechanics we were tasked with exploring.

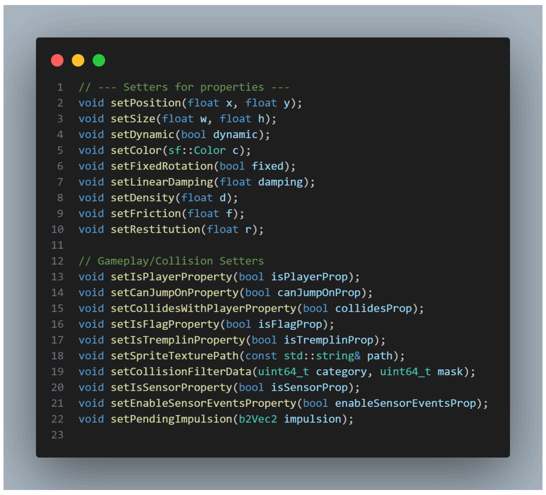
SFML provided the ground: a quite popular C++ API that gives direct control over the render loop and input polling, without imposing a rigid architecture. This gave us the freedom (and the challenge) to design our own system for bridging the gap between the physics simulation and the visual output.
Ultimately, the combination of Box2D and SFML formed the bedrock of our project. Box2D would be the 'brain' of our world, calculating the physical consequences of every action, while SFML would be its 'body', displaying the world to the player and relaying their commands. This decision defined our first technical challenge: to design and implement a system that would link these two distinct libraries.

# Technological details

With our core technologies chosen, our next major task was to build a system architecture that would allow them to communicate effectively. This meant designing and implementing the basics that would hold our game together. The following systems represent our primary technical contributions to the project, beyond simply using the libraries to create a functional game framework.

## Object abstraction (GameObject Class)

Our most significant contribution is the *GameObject* class (game_object.hpp, game_object.cpp). The challenge of using separate physics and rendering libraries is mainly about keeping their representations of the world in sync. A naive approach might involve parallel arrays one for Box2D bodies, one for SFML sprites, but it quickly becomes difficult to manage. Our *GameObject* class was designed to solve this problem and, crucially, to streamline the level creation process. It acts as a wrapper, unifying an entity's physical, visual, and gameplay properties into a single, easy-to-use interface.



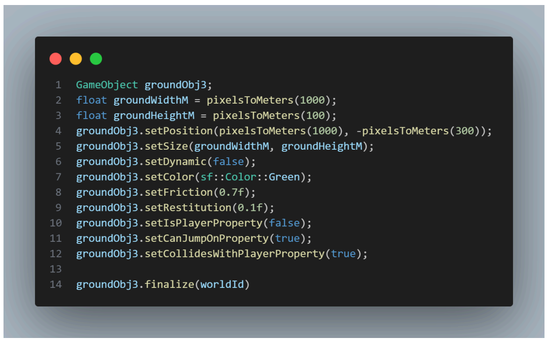
```cpp
// --- Setters for properties ---
void setPosition(float x, float y);
void setSize(float w, float h);
void setDynamic(bool dynamic);
void setColor(sf::Color c);
void setFixedRotation(bool fixed);
void setLinearDamping(float damping);
void setDensity(float d);
void setFriction(float f);
void setRestitution(float r);

// Gameplay/Collision Setters
void setIsPlayerProperty(bool isPlayerProp);
void setCanJumpOnProperty(bool canJumpOnProp);
void setCollidesWithPlayerProperty(bool collidesProp);
void setIsFlagProperty(bool isFlagProp);
void setIsTremplinProperty(bool isTremplinProp);
void setSpriteTexturePath(const std::string& path);
void setCollisionFilterData(uint64_t category, uint64_t mask);
void setIsSensorProperty(bool isSensorProp);
void setEnableSensorEventsProperty(bool enableSensorEventsProp);
void setPendingImpulsion(b2Vec2 impulsion);
```

All the *GameObject*'s setters

To be more precise, it implements the following features:

1. **Unified Entity Management**: Every game entity (player, platforms, boxes, flags) is a GameObject
2. **Physics Visual Synchronization (detailed below)**: Automatically keeps SFML visuals in sync with Box2D physics
3. **Flexible Configuration**: Property-based setup before finalization
4. **Animation Support**: Built-in sprite animation system for player and other entities
5. **Collision Management (detailed below)**: Sophisticated collision filtering system



```cpp
GameObject groundObj3;
float groundWidthM = pixelsToMeters(1000);
float groundHeightM = pixelsToMeters(100);
groundObj3.setPosition(pixelsToMeters(1000), -pixelsToMeters(300));
groundObj3.setSize(groundWidthM, groundHeightM);
groundObj3.setDynamic(false);
groundObj3.setColor(sf::Color::Green);
groundObj3.setFriction(0.7f);
groundObj3.setRestitution(0.1f);
groundObj3.setIsPlayerProperty(false);
groundObj3.setCanJumpOnProperty(true);
groundObj3.setCollidesWithPlayerProperty(true);

groundObj3.finalize(worldId)
```

We used it to create a high-level, declarative system for object creation. To create an object for a map, one simply has to:

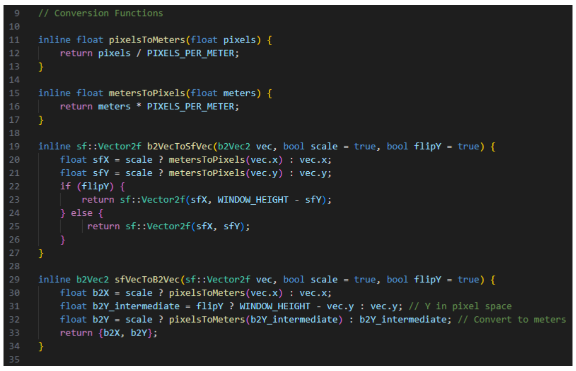
1. Instantiate a GameObject.
2. Call a series of intuitive setter methods to configure its properties (*screenshot above*)
3. Call *finalize()* once, which handles all the boilerplate of creating and linking the underlying Box2D and SFML objects.

## Physics / graphics synchronization

As mentioned in the previous part, a fundamental task was ensuring the visual world accurately reflected the physics simulation. This was achieved through two key components:

**Coordinate System Management:**

We established a global scale factor, *PIXELS_PER_METER*, to translate between Box2D's meter-based units and SFML's pixel-based units. Our utility functions also handle the Y-axis inversion between SFML's top-left origin and Box2D's bottom-left origin.



```cpp
// Conversion Functions

inline float pixelsToMeters(float pixels) {
    return pixels / PIXELS_PER_METER;
}

inline float metersToPixels(float meters) {
    return meters * PIXELS_PER_METER;
}

inline sf::Vector2f b2VecToSfVec(b2Vec2 vec, bool scale = true, bool flipY = true) {
    float sfX = scale ? metersToPixels(vec.x) : vec.x;
    float sfY = scale ? metersToPixels(vec.y) : vec.y;
    if (flipY) {
        return sf::Vector2f(sfX, WINDOW_HEIGHT - sfY);
    } else {
        return sf::Vector2f(sfX, sfY);
    }
}

inline b2Vec2 sfVecToB2Vec(sf::Vector2f vec, bool scale = true, bool flipY = true) {
    float b2X = scale ? pixelsToMeters(vec.x) : vec.x;
    float b2Y_intermediate = flipY ? WINDOW_HEIGHT - vec.y : vec.y; // Y in pixel space
    float b2Y = scale ? pixelsToMeters(b2Y_intermediate) : b2Y_intermediate; // Convert to meters
    return {b2X, b2Y};
}
```

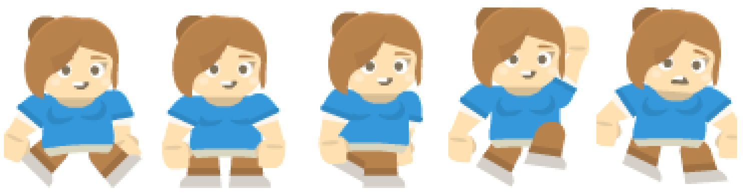
**Fixed Timestep for Physics:**

To ensure simulation stability and determinism, we run our physics world with a fixed timestep (*UPDATE_DELTA* of 1/60th of a second). This decouples the physics from the rendering framerate, preventing issues where a faster or slower computer could alter the simulation's outcome.

**Synchronization and Rendering:**

Our main game loop handles the synchronization: In each frame, after the physics step, we iterate through every *GameObject* and call its *updateShape()* method. This function retrieves the latest position and rotation from the object's body and applies it to its corresponding SFML shape.. This ensures the visuals are a direct representation of the physics state.

**Animation and Sound:**



*The different animation frames for our main character*

Our animation and sound systems are also decoupled from the physics step. The *updatePlayerAnimation()* method and sound logic are driven by the rendering frame's delta time. This allows for smooth animations and continuous audio, even if the physics simulation were to lag or run at a different rate.

## Collision and interaction logic

Box2D is very good at simulating collisions, but making those collisions meaningful for gameplay requires a more sophisticated system. Following the principles from our course, we leveraged Box2D's powerful collision filtering system, but we still tried to abstract  its complexity away.

Instead of forcing our level designers to manually manage the hexadecimal category and mask bits, we integrated this logic into our GameObject class. By setting simple boolean properties like *setIsPlayerProperty(true)*, *setCanJumpOnProperty(true)*, or *setIsFlagProperty(true)*, the object is automatically assigned the correct filter data during the *finalize()* step.

This allowed us to easily implement several key interaction types:

- **Solid vs. Non-Solid**: We can define objects that the player can pass through but other physics objects cannot, or vice versa.
- **Jumpable Surfaces**: The canJumpOn property flags surfaces that will reset the player's isGrounded state, a crucial part of our custom player controller.
- **Sensors for Gameplay Events**: Our flag and trampoline objects are marked as "sensors". This Box2D feature means they detect contact without producing a physical collision response. Our game loop

listens for *b2SensorEvents* and, when the player enters the flag's sensor, triggers the level-completion logic. This system is robust, efficient, and cleanly separates gameplay logic from pure physics.

## Easy level creation

Finally, all these systems come together in our approach to level creation. While we aimed to design a custom file format (like JSON or XML) and write a parser, the lack of time had us opt for a procedural method where each level is a self-contained C++ function (ex. *loadMap1()*, *loadMap2()*).

To make this creation easy, we introduced *Object Primitives* (like *Rope, Box, Flag, Balance …*). Those are functions that create commonly used game objects with predefined configurations.

While not perfect, this design choice had several advantages for a project of this scale:

- **Power and Flexibility**: We could use the full power of C++ loops, functions, variables to construct our levels. This was really useful for creating repetitive patterns, like a tall stack of boxes which are composed of many small, interconnected GameObjects.
- **Rapid Iteration**: Modifying a level was as simple as changing a few lines of C++ code and recompiling. There was no need to maintain a separate data file and a parser in parallel.
- **Compile-Time Safety**: The C++ compiler automatically validated our level-creation code, catching errors before the game was even run.

Our *GameObject* abstraction was the key to making this approach viable. It kept the level-creation code clean, high-level, and focused on design, while hiding the complex, low-level API calls required to actually build the world.

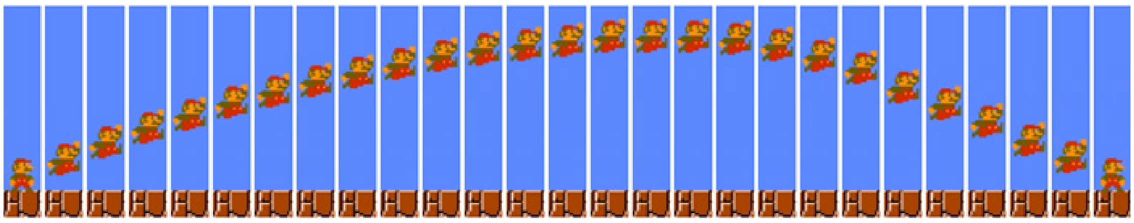
# Gameplay mechanics & implementation

## Player control: pursuit of "Game Feel"

To make the game the controls the most enjoyable and comfortable as possible, we adjusted the player movements using Box2D physics to have satisfying interactions.

**Horizontal Movement:**

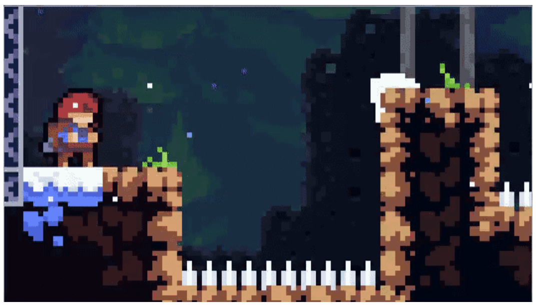
Instead of velocity manipulation, we add a force in the wanted direction. To make the movement more natural, that force is lower when the player doesn't touch the ground. Moreover, that force is only applied if the player isn't already moving at their maximum speed. Finally to make controls as responsive as possible we add an additional force if the player changes direction or when the player is grounded and they release the movement keys.



**Jump:**

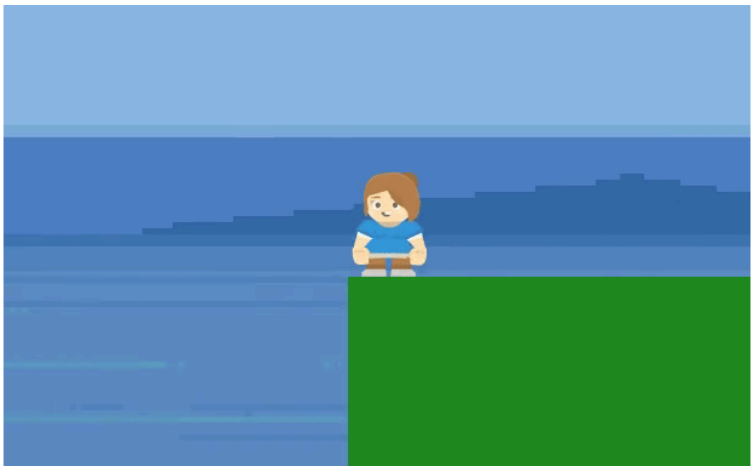
To make the jump feel natural, we couldn't just make a realistic jump and add an upward force. In fact, in the vast majority of games, the player falls faster than it jumps. Moreover, the player usually jumps higher if the Jump Key is pressed longer, which means when the character is already in the air  To resolve this problem we added a custom gravity factor for the player. That custom gravity changes depending on different factors: it stays the same if the Jump key is held, it becomes stronger when the player reaches the height limit of the jump or if they release the Jump Key. That way the controls feel more responsive.

We also added some functionalities to make the game more forgiving. First, we added a Jump Buffer Time which allows the player to jump even if they pressed the Jump Key a little too soon.



We also experimented with implementing a Coyote Time which would allow the player to jump even after they quit the ground. However, we encountered some problems with Box2D 's contact detection which made the functionality harder to implement.

Finally, we fine-tuned all these variables to achieve a result that felt right to us: high enough to keep the gameplay fun, but not so high as to break the environmental physics or prevent us from designing interesting puzzles.



**Level reset:**

We implemented a level reset system that activates either when the player falls below a certain point or manually by pressing the R key. This reset returns both the player and all objects to their initial positions. We chose to give the player this control to let them retry a level without needing to die, avoid potential soft locks, and freely experiment with different ways to complete a level.

## The time-freeze mechanic

One of the central features of our game is the time-freeze mechanic, which gives players the ability to pause the entire physics simulation at any moment by pressing the "F" key. When activated, all dynamic objects in the world—such as ropes, balances, trampolines, and boxes—are instantly frozen in place. The physics solver is paused, and we store the full state of the world (positions, velocities, and forces) at the moment of freezing. When the player presses "F" again, the simulation resumes exactly where it left off, continuing seamlessly as if no time had passed.

To enhance player feedback, time freeze is accompanied by visual and audio cues: the screen takes on a bluish hue, and a distinct sound effect signals the transition. While time is frozen, only the player character remains active. All other objects are treated as static for collision purposes, allowing the player to move, jump, and interact with the frozen world. This enables unique puzzle-solving opportunities—for example, freezing a falling box mid-air to use it as a temporary platform, or stopping a swinging rope at just the right angle to cross a gap. This ability is the core mechanic that makes our game unique and original.

## Interactive Primitives (rope, trampoline, ...)

The primitives are at the core of our game's design. They serve as versatile building blocks that we can easily place in different levels to create engaging gameplay scenarios. By abstracting these elements, we can quickly build and customize levels without needing to manually script each interaction. We have implemented five key primitives, each with its own functionality and role within the game.

**- Rigid/Static Rectangle:**

This is the most basic primitive. It represents a rectangle with configurable size, color, and physics properties (such as restitution, friction, and density). It can act as a static platform, or as a dynamic block that the player can push or interact with. These rectangles form the basis of most level geometry and obstacles.

**-Rope:**

The rope is built from a sequence of small dynamic rectangles connected by Box2D joints, anchored at both ends. The number of segments and their size can be adjusted to create ropes with different levels of flexibility and complexity. The rope reacts to the player or not depending on the flags we set.

**-Balance:**

The balance is essentially a dynamic rectangle with its center point anchored to the world using a joint. This setup allows it to pivot like a seesaw, reacting to the weight of the player or other dynamic objects placed on it.

**-Trampoline:**

The trampoline is represented visually by a texture and bounded by a collision box. When the player or any other dynamic object lands on a trampoline, our collision logic calculates and applies an upward force, launching the player into the air.

**-Flag:**

The flag is a simple static object marked by a texture. When the player collides with it, it triggers the transition to the next level.

The time freeze mechanic from earlier applies to all these objects. The player can freeze any dynamic object at any time, fixing its position and pausing the physics simulation until the unfreeze.

## Design choices (images, sounds, ...)

As for the visual design of our game, we initially planned to draw the sprites ourselves; however, apart from the trampoline, we ended up using sprites found online. For the overall aesthetic, we chose simple visuals with a mix of pixel art and a colorful playful style to match the lighthearted and fun nature of the gameplay.

As for the sound design, such as sound effects and music, we also couldn't make enough time to create our own even though that was one of our objectives. As a result we decided to pick music and sound effects from games we liked and that we think suited our project. Notably the music is from Celeste (*Celeste Original Soundtrack - 03 - Resurrections* ) and the jump sound comes from Mario.

# Other challenges & our solutions

We were definitely constrained by the time and there were plenty of things we wished to add. Although we mainly focused on gameplay, movement and physics, we would have liked to improve the general look of our game, by working more on the textures, the animations and the background music. This isn't very hard to do: we just needed more time to really work on those features to express our creativity.

Another aspect that challenged us was the level creation. We tried implementing a level editor to streamline level creation and make it easy to make and edit levels but that proved too time consuming so we opted instead to focus on the look of our game in the final week and just implement the levels statically. This allowed us to improve the visual quality of the game, as we added many visual aspects in the last week that really polished our game and made it much easier to look at but this trade-off meant less levels in the final demo

# Conclusion

We are very happy with the final outcome of our project: a simple, fun, and interactive 2D platformer that combines rigid body physics with a very original time-freeze mechanic to create engaging and varied gameplay. The levels make good use of dynamic elements like ropes, balances, trampolines, and movable blocks, and we're especially proud of how effectively we integrated rigid bodies — the main goal of this project.

Attempting to implement our own physics engine at the start taught us a lot about position-based dynamics, constraints, and collision handling. This experience gave us a deeper understanding of how a physics engine works internally and helped us navigate and use Box2D much more effectively. We realized that many of the methods we tried were similar to those in Box2D, though Box2D's solver was clearly more stable for complex simulations.

A lot of effort went into small but meaningful details, particularly the player's movement. We fine-tuned aspects like walking and jumping to make sure they felt fluid and responsive, like in a polished 2D platformer. Overall, we're proud of the work we put in and the game we created.