Fly or Die
Converting The Game From SFML to Unity

Inge Olaisen

Faculty of Arts
Department of Game Design
Bachelor’s Thesis in Game Design, 15 hp
Program: Game Design and Programming
Supervisors: Mikael Fridenfalk, Hans Svensson
 Examiner: Masaki Hayashi
June, 2015
Abstract

Fly or Die was a first year project that became fairly successful. It won the Gamer’s Choice Award at the Swedish Game Awards 2013 and has been on display in many locations, both in Sweden and abroad. Recreating the game and porting it from SFML to Unity was sparked by winning the award and by all the attention that the game received prior to the conversion process. The final version requires the Unity 5 Personal Edition and Microsoft Visual Studio 2013. The conversion process uses the old code base as a template and attempts to incorporate the same coding principles. In the new versions, all levels are built using many of the 2D features that Unity 5 offers, including that the game can be played with two to four players, and that some of the levels are enhanced or changed by adding additional graphics or game play enhancements. An approach to how the levels have been created and a comparison between the new and the old version is presented, including the differences, and why it was necessary to change certain aspects of some of the levels. The author thus recommends porting a game from a previous platform to Unity 5 in order to make it playable on a larger number of platforms, since the advantages of using an engine like Unity far outweigh the disadvantages.
# Table of Contents

1 Introduction ............................................................................................................................. 1

2.1 Background .......................................................................................................................... 2

2.2 Purpose ............................................................................................................................. 2

3 Materials and Method.............................................................................................................. 3

3.1 Conversion Process .......................................................................................................... 3

3.2 Tools Used ........................................................................................................................ 3

3.3 Project Plan ...................................................................................................................... 4

4 Results ..................................................................................................................................... 5

4.1 SFML with Box2D Compared to Unity 5 Physics ........................................................... 5

4.2 Creating the Aesthetic Feel of the Game ......................................................................... 5

4.3 Lobby ............................................................................................................................... 5

4.4 Swamp Level .................................................................................................................... 7

4.5 Military Level ................................................................................................................... 9

4.6 Lava Factory Level ......................................................................................................... 10

4.7 Windmill Level .............................................................................................................. 11

4.8 Score Screen ................................................................................................................... 12

4.9 Invisible and Generic Changes ....................................................................................... 13

5 Discussion ............................................................................................................................. 14

5.1 SFML and Box2D Compared to Unity Physics ............................................................. 14

5.2 Creating the Aesthetic Feel of the Game ....................................................................... 15

5.3 Swamp Level .................................................................................................................. 16

5.4 Military Level ................................................................................................................... 16

5.5 Lava Level ...................................................................................................................... 18

5.6 Windmill Level .............................................................................................................. 19

5.7 The Score Screen .......................................................................................................... 20

5.8 Invisible and Generic Changes ....................................................................................... 21

6 Conclusion ............................................................................................................................. 22

References ................................................................................................................................ 23
List of Figures

Figure 1: Screenshot of the lobby. Unity version (A), SFML version (B) ............................... 6
Figure 2: Screenshot of the swamp level. Unity version (A), SFML version (B) ...................... 7
Figure 3: Illustration of how Hinge Joints and Distance Joints are connected ......................... 8
Figure 4: Screenshot of the military level. Unity version (A), SFML version (B) ..................... 9
Figure 5: Screenshot of the lava level. Unity version (A), SFML version (B) ........................... 10
Figure 6: Screenshot of the windmill level. Unity version (A), SFML version (B) ................. 11
Figure 7: Screenshot of the score screen. Unity version (A), SFML version (B) ...................... 12
Figure 8: Visual representation of the collider and settings of Rigidbody 2D in Unity ............ 15
Figure 9: Image box selection screen in Unity (Sprite Editor) ............................................. 17
Figure 10: Overview over buttons (A) and pistons (B) in the lava level ............................... 18
1 Introduction

Fly or Die was originally a game that a team I was part of developed as first year students. It was developed during a course where the students were creating games with different types of input devices. The original game was created with the SFML and Box2D libraries and written in C++. 

The original team behind Fly or Die SFML was:
Jonatan Keil – Game designer
Anders Hedström – Artist
Maximilian Lund – Artist
Joar Hedvall – Programmer
Johan Wittrock – Programmer
Inge Olaisen – Programmer

What we ended up with was a four player game of Tag. The rules are not the same, but the idea behind it is very much the same. As a player, you pass a bomb to other players and try not to have the bomb when the timer runs out. When that happens and the bomb blows up, the player that had the bomb in the end loses the round. The game was well received at the Gotland Game Conference 2013, even though it did not win any prizes. The game was submitted to Swedish Game Awards and was nominated for Best Execution. We ended up winning a different category, called Gamer’s Choice. Although Gamer’s Choice was not the prize we hoped to win, it was still a great feeling to get an award for the game.

Winning that award is one of the reasons why I wanted to start up with Fly or Die again. Because it won an award and was nominated, it shows that the game has promise. On top of that, the game has been displayed in various locations and exhibitions across the world, including Tokyo Game Show and GameX (ComiCon). I have, since we first created the game, felt like the game has been a lost opportunity. Seeing that it would be fairly easy to convert and continue with the development of the game in Unity, I decided to test and see how it would turn out. Because the other members in the original team considered themselves done with the project or have moved on to other games, I took it upon myself to start the conversion process. I have already asked for and received permission to use the graphics and the music.
2.1 Background

The original Fly or Die game was designed to be a local game where four players are battling against each other in 2D. It was inspired by the game Tag and the mechanic in the game was called Hot Potato. Hot Potato mechanic means that the player gets an item they do not really want, and wish to pass that item to a different player. In Fly or Die, the item is a bomb, which will explode when the level timer runs out. One play-through consists of four levels each lasting 54.6 seconds. The timer was set to be in sync with the music. The four levels were chosen because each of the characters has their respective home world. The players that survive the round receive 5 points each; the player that blows up does not. Every time the bomb is passed to a player, their score is deducted by one point. After four rounds, a winner is determined and the score screen is displayed revealing the results.

SFML was chosen for the old project because it was a library that we were familiar with at the time. The reason for converting the game is that SFML is limited to the PC, Linux, and Mac platform. Unity supports multiple platforms including consoles. A game like Fly or Die might have a better chance of success on a console rather than on PC, Mac and Linux. Fly or Die requires the use of a joystick, which is why a console might be a better fit. Creating additional levels and adding network support is also much easier, compared to a project written in C++, which is the language used in SFML. Unity is a game engine that I have learned to use, which made the choice clear. There are other engines out there but I did not want to start the course with learning a new engine.

2.2 Purpose

The aim of the project is to see if it is possible to recreate the game from the C++ language to the Unity game engine within the timeframe of seven weeks. The end goal for the game is to keep the existing gameplay and game experience intact. How was the conversion process done in order to keep the gameplay and game experience intact? The conversion process is not a direct conversion between C++ and Unity; reason being that C++ is a programming language whereas Unity is a game engine. Conversion in this context is about recreating the game in Unity more than a true conversion process. The old code serves as a template and contains an idea of how to solve certain problems, rather than using tools or programs to translate or convert the code directly.

Developing any kind of game comes with certain problems and challenges. These problems and challenges differ from project to project. Learning from potential challenges that might occur during development is knowledge that can be applied from one project to the next. There are some problems that are project specific and some problems are common for most projects. These problems differ in severities and take different shapes. Examples of this can be that some projects run out of time, or that team members do not work as well together as one might hope. Working alone means that there are no team conflicts, but a single developer has other obstacles to deal with. This involves solving problems that occur with graphical assets, sound assets and even level and design ideas that initially were thought of by others in the previous team. It also means that one person needs to rely on solving every problem themselves. It does not mean that it is impossible to ask for outside help, but for the most part it might be quicker to get help from the internet rather than asking a specific person. In this paper I want to examine what challenges a single person developer would face, and for specific problems, how those problems were solved.
3 Materials and Method

When recreating the game, the original is needed to compare the Unity and SFML versions. I also wanted to use some of the original source code as a template for what values and ideas to use in the game. Since Unity uses JavaScript or C#, the C++ code that the SFML version uses is not directly translatable. However, the C++ code is a valuable source to see what ideas we used, learn from possible mistakes and try to fix those mistakes. There are also major differences between the SFML project and the Unity Project; the Unity project gives you many visual tools for placing collision boxes and graphical assets directly into the scene. SFML does not have that and everything needs to be coded in. When the SFML project was created, we talked about having a level editor, which would have made it easier to create levels, but because of time constraints, we did not do that.

3.1 Conversion Process

Converting the game from SFML to Unity did not mean I could transfer the old code directly over to Unity. The old codebase is supposed to be used to compare values for the most part. Values, although vague, refer to variables like torque, forces and gravity. Unity has since version 4.3 used the Box2D physics engine. Going into the project, the assumption is that the Box2D library and the Unity implementation are very similar. The SFML project was also used to loosely plan the structure of the new Fly or Die game.

3.2 Tools Used

Unity 5 Personal Edition

Unity is a game engine that is made to easily develop games. It has an interface where you work visually in a 3D or 2D space, depending on the setup of the game you want to create. If an object needs to do something in a scene, you need to instruct that object as to what it needs to do. These instructions are written in Boo, C# or JavaScript.

Microsoft Visual Studio 13

Visual Studio is an IDE that enables the user to write code. It has syntax highlighting that makes it easier to see that you have written correctly.

Adobe Photoshop CC

Photoshop is an image editor. It will be used to scale images to correct size if they are too big. For this project, the program will not be used to create any graphics, as the graphics has been created already.

SFML

Simple and Fast Media Library is a free library used for C++. This library enables the programmer to place images on screen and move them around. It uses OpenGL and is a cross platform library that can be used with Linux, Mac OS and Windows. The original Fly or Die game used this library.

Box2D

Box2D is a C++ library made for simulation of a physics world, mostly to be used in games, but it has other applications as well. It was used in the original Fly or Die game to simulate physics like collisions, explosions and movement. Unity adopted Box2D physics into their engine after 4.3.

Microsoft Xbox 360 Controllers

Four Xbox 360 joysticks were used to develop the game in order to test the game and to test that the spawning system worked as intended.
3.3 Project Plan
For this project there was no written project plan, but there was an overall idea on how to approach this project. The idea was to start with the lobby, continue on with the swamp level, military level, lava level, windmill level and then finally the score screen. Roughly the plan looked like this:

**Week 1**
Replicating movement from the old game.
Setting up the lobby with collisions.
Creating the basic player

**Week 2**
Creating a spawning system.
Adding joystick controls for the four players.

**Week 3**
Create swamp level.
Create military level.
Create lava level.
Create windmill level.

**Week 4**
Create the score screen.

**Week 5 and Onward**
Polishing, bug fixing and play testing.
4 Results

4.1 SFML with Box2D Compared to Unity 5 Physics
The first part of the process was to create a single level, which was called the lobby. The lobby does not have a function other than spawning the players and making sure the game can be started from there. This level was then used to recreate the same movement as was in the game.

Box2D physics engine setup has several stages in order to be properly initialized. The physics world needs to be setup with arbitrary values. These values define how many pixels you have on a meter and the gravity scale. These values in turn determine mass, which means how many kilos an object weighs in order to push or affect them in different ways. As part of the setup, there are also settings for how often the physics should be updated.

In Unity 5, a great deal of the physics has been abstracted away. The developer does not have to specify the amounts of updates that the engine needs since that is handled internally in the Unity Engine. There is a difference when programming towards the physics engine. Unity has two different update methods, one called Update and a second called FixedUpdate. The difference between the two is that the Update method is called every frame, but the FixedUpdate method is not. The default setting for the physics engine update frequency is 50 times per second. The FixedUpdate method can update more or less frequently depending on the computer the program runs on. Faster computer has a higher update rate compared to a slower computer.

4.2 Creating the Aesthetic Feel of the Game
The game feel was created using the same ideas in the physics engine. The physics bodies are acting in a similar fashion in both Unity and SFML. The movement was recreated by looking at the code in the SFML version of Fly or Die and the same principles were applied on how the movement was controlled. The code itself could not be used directly, but the idea behind the movement was the same. This was done by applying torque to the physical bodies, which made the physical bodies turn. Torque alone was not enough; the physical body also needed a force in order for it to move. The direction that it moves is decided by the direction it was facing.

Once that was done, the next part was to create the collisions with the walls. A wall collision spawns a certain animation if the player has a certain speed, it plays a sound and it also shows the animation. Although the old SFML code itself could not be used directly, the SFML version was the source to understanding the collisions and how it worked. It illustrated what part of the collision engine could be used in order to spawn the animation in the right place.

The collision system in Unity 5 and in Box2D that was used with the SFML version has similar programming paradigms. The collider in both versions has a first contact point which served as the point where the collision was spawned.

4.3 Lobby
The lobby is the entry point to the Fly or Die game and it serves two main functions. The first function is to make the player familiar with the controls of the game. The second function is to start the game. The lobby in the new version also serves as the place where the players can choose how many players there are. Originally, in the SFML version of the game, it was set to have four players. In the Unity version, the players choose how many they want to be; ranging from two to four players. This is done by the players pressing start on their Xbox 360 controller. The game also recognizes how many joysticks are connected, if there are four Xbox controllers connected, four players will automatically spawn.
Figure 1: Screenshot of the lobby. Unity version (A), SFML version (B)

The main difference between the two lobbies is the buttons. The SFML version was hard-coded with four buttons (see Figure 1-b), one for each player. These buttons are situated in the lobby scene to indicate that the player is ready, as seen in Figure 1-a and Figure 1-b. In the new version, the buttons spawn according to the number of players. If there are two players wanting to play, only two buttons are shown in the lobby (see Figure 1-a). Another change was the game mode text, to show the players which game mode is active. The picture of the Xbox 360 controller in the background is also smaller than in the original. In order for the menu to be visible, the picture of the controller in the background was removed and added as an overlay. The joystick information box disappears when the menu is open and appears again when the menu closes.
4.4 Swamp Level
The next thing to build was the swamp level.

A

Figure 2: Screenshot of the swamp level. Unity version (A), SFML version (B)

B

The level, in both the new (Figure 2-a) and original game (Figure 2-b), is very much alike graphically. The difference however is under the hood of the physics engine. One difference is that the tire and rope is connected in a different manner. In the SFML version, the rope connected to the tire is rotating towards a specific point; the origin of the rope is set to the top so that it rotates around the top of the image. Figure 3 shows the Unity version, but the top-most connection point is where the rope is rotating around in the SFML version:
In Unity, the physics engine was utilized for everything, and there are several types of joints and hinges. First, a GameObject with a Hinge Joint 2D component was created. The rope object consisted of a Distance Joint 2D which connected to the Hinge Joint. A second Distance Joint 2D was then utilized to connect the first Distance Joint to the wheel. Figure 3 illustrates the connection points that were utilized in the game.

Other than that, the rest of the level used the same ground principles. The mushrooms when they are hit, bounces the players back in both versions. The same Physics engine ideas were used, with an impulse force that drives the player back in a certain direction and rotates them by adding torque.

**Figure 3:** Illustration of how Hinge Joints and Distance Joints are connected
4.5 Military Level
The military level has a different setup than some of the other levels. There are four mines placed adjacent to each other. The cannon rotate around the circle base, but it does not serve any function other than being a decoration and an obstacle for the players to move around.

Figure 4: Screenshot of the military level. Unity version (A), SFML version (B)

The difference between the levels is in the different elements and is not visible. The different mines have the same code based ideas, but the design decision was to make the top mines send the players mostly to the sides, and the top left mine would send the player to the right. The mines below the top left and top right are sent in a direction depending on where on the mine they hit. If they are hit on the bottom of the mine, the player is sent down towards left or right.
The other difference is the rotor. In the SFML version (Figure 4-b), people were sent upwards and rotating. This was changed in the Unity version (Figure 4-a) so that the players are not sent upwards but rather get stuck and sent back and forth until the force sends them to the left or right sides. The reason behind this was that the physics engine did not have the same features, and implementing the same ideas did not turn out the same as the SFML version.

4.6 Lava Factory Level

Figure 5: Screenshot of the lava level. Unity version (A), SFML version (B)

This level was updated with some new graphical elements. For instance, the lava on the bottom had smoke (Figure 5-b) that was animated through a sprite that was moving alongside the lava. In the Unity version, the built-in particle system was used to create a smoke effect (Figure 5-a), in addition to the sprite sheet. The lava animation on the bottom was changed, be-
cause it did not tile properly. To fix this, the lava sprite was copied three times. The sprites where then tiled next to each other and the middle copy of that sprite was flipped horizontally. This made the sprite overlap making it seamless. The level timer was also moved so that it is now stuck on the piston that comes down when touching the top left button on the level. Also, the buttons were placed a little further out compared to the original level. The buttons and level are otherwise the same in both versions of the game.

4.7 Windmill Level

The windmill level is more or less identical in both versions. The visual difference is in the clouds in the background, they are more prominent in the Unity version (Figure 6-a) and use more screen space compared to the SFML version (Figure 6-b) of the game. The collision

**Figure 6:** Screenshot of the windmill level. Unity version (A), SFML version (B)
boxes are a bit different. In the SFML version we were able to only use rectangles and circles. In Unity 5, with the access of a polygon collider, the collision boxes can be drawn on the graphical assets directly, making the collision boxes more accurate. An added feature is that the players can change the direction of the windmill blades, which was not in the old version.

4.8 Score Screen
The scoring screen underwent some changes in the Unity version. The reason for this was that the level looked more static than the rest of the levels. The text of the scoreboard changed as well, because it was harder to distinguish between the numbers one and seven. In the SFML version, this was solved by replacing the number one with a lowercase l. This was not done in the Unity version. Figure 7-a and 7-b shows the old font in the Unity version as well as the SFML version. The old version of the font was used to illustrate the problem, that players often mistook one for seven and vice versa.

![Figure 7: Screenshot of the score screen. Unity version (A), SFML version (B)](image-url)
The avatars in the new version enter the scene one by one in sequence. They did that in the old version as well, but they flew in from the sides. Now they fly up from the bottom of the screen and are moving and showing their different smoke particles while hovering over their respective scoreboard. The screen flashes and the winner or winners get a cup above their respective wing and they once more move up so the first place is placed higher than the second, third and fourth position, like a podium.

4.9 Invisible and Generic Changes
The SFML game was based upon the idea of having four players. The new version was created to be more flexible. The thought behind this was in case the game would be networked, the spawn system needed to be more flexible. A flexible spawn system means that you create one template for a character. Depending on which number the controller of the player is, the player turns out to be the player assigned to that controller. One reason for that was that the game would dynamically be able to connect and disconnect players from the game. This feature was later removed, because there were a lot of problems with that philosophy that the players are instead removed when the level changes and not during the game. Players could still join whenever they wanted to even if the game was still in progress. Players disconnecting also added a challenge to what to do if the player with the bomb disconnects.

The variables for player movement were multiplied by delta time. Delta time was not used in the old SFML project because SFML comes with a setting that locks the frame rate of the game to 60 frames per second. By locking the frame rate, the delta time is approximately the same. In the Unity project, the delta time was added in to compensate for slower computers and for those who have more than 60 frames per second.

Of the generic changes, the Unity engine comes with a particle system. This was utilized on the players smoke effect. Each engine on each of the characters had their own particle emitter that emitted smoke, each colored with their respective color. This was not greatly changed, but it was somewhat different from the SFML version compared to the Unity version.

An added feature to the game was a new game mode called “Capture the Bomb”. Capture the Bomb game mode is the normal mode inversed. The person with the bomb is the one that gets all the points. When the timer runs out, the person with the bomb is awarded points and the other players lose the round. This game mode was not present in the original version of the game. The in-game menu that was added was made to accompany this change in the new game, as well adding a way for the players to select how many players to spawn at the start of a game session.
5 Discussion

5.1 SFML and Box2D Compared to Unity Physics
When I first started with the physics in Unity, it was a pleasant surprise to see how compatible the two physics systems were. The principle behind how you moved an object forward was very similar. In the beginning, I used the same values as the SFML project. After some testing, I discovered that delta time was missing. Delta time is used as a measurement of how much time the previous frame took. It is a value used to make sure that the game runs the same way on each computer, no matter how fast or slow it is. Since the old game was locked at 60 frames per second, the delta time was more or less the same. If the frame rate drops below 60 the game seems slower; if the frame rate increases, everything happens faster. This was something that needed to be fixed in the Unity project and delta time was therefore added to the calculation. The initial values needed to be multiplied because delta time is usually a very low value. If the initial value before delta time was 0.5, it would now be around 30 instead. After doing that, the game was tested on a second computer and the result was the same on both.

The big difference between Box2D and Unity is not so much how they appear in the finished game, but more how you set the two up. Box2D does not have a graphical interface; Unity, on the other hand, has a graphical setup. Box2D setup involves setting up a physical world and this world exists only in numbers. In this imaginary world, you have different physics properties like gravity, how much air resistance there is and the scale of the world. All of these properties are used to calculate the physics of the game. Scale is used to calculate the mass of every object. In this imaginary world you need to add circular and rectangular objects. These objects are invisible in this world. In order to be able to see them you need to add images that get the position of these invisible objects. This setup is done through code. Box2D does have a visual debugging tool so that you can show on the screen the different invisible bodies. It was easier at the time we made the original game to simply attach an image to the bodies.

Setting up the physics system in Unity is far easier. It already has visual representations for the collision boxes you want. Everything is done through the graphical user interface (GUI) that Unity has to offer. This simplifies the process and instead of guessing where the positions of the objects are in the scene, you simply add box-, circle- or polygon colliders to the images you put on the screen. From there, you can set the physics properties for each object in the scene. The physics properties can be mass, which is the density and weight of an object. Another property is the rotation resistance and how much the physical object is affected by gravity. Figure 8 illustrates the circle collider which is the green circle and to the right the settings of the Rigidbody 2D which is the body in the physics world. The settings are the same as you have in Box2D, but the difference is that you have the settings in a GUI.
Having a GUI to setup every object that the player can collide with in a scene makes the process much faster. In SFML the process consisted of: inserting an image, having colliders around it, and then placing it in the scene. The differentiation between rigid bodies and colliders was confusing when I first learned to use Unity. After having worked with colliders and bodies a while, it is easier to understand the difference. Colliders can be set to allow passage of physical objects or not. If you allow physical objects to pass through it, it becomes a trigger instead. Triggers were not used in the game, because triggers are only used to set off events once the player comes in contact with the trigger collider. Having a collider to be a non-trigger object makes it collidable and physical objects will not pass that collider. When talking about bodies in this context, it is about physical representations of objects, a physical body. Unity has several types of Rigidbodies, ten in total, which makes it difficult to know which Rigidbody that you should use. For the most part, it was Rigidbody 2D, but on the swamp level three different kinds of bodies were used, which I will come back to.

5.2 Creating the Aesthetic Feel of the Game
Recreating a certain movement in a game is a difficult task. It requires a lot of testing of the old and new versions of the game, testing different settings and re-iterate. It took several iterations before the movement was approximately the same between the old and the new version. Recreating the movement was the first thing that was done in the project. At first it was without the delta time, which made the values lower and the movement seemed to be replicated on one computer, but on another it was different. Adding the delta time, as discussed earlier, fixed those problems, but that meant that the values that was used in the old project, could no longer be approximated to the same values. This made it more difficult to recreate the game experience. Creating collisions and the rest of the game feeling was comparatively straightforward, and there were no major obstacles in that process.
5.3 Swamp Level

The biggest problem with the swamp level was to get the tire and rope right. Since the SFML project and Unity project has a different code language and their setup is different, it was impossible to use the SFML project as an inspiration. For the tire and rope setup, I needed to find a different way of solving this. A Hinge Joint 2D was used, which is something you can connect Rigidbodies to. Once the Hinge Joint 2D created, the base of the rope needed to be attached to the Hinge Joint 2D by using a Distance Joint. Distance Joints are used to rotate objects around each other. The wheel then needed to be able to lock onto the rope and the whole rope and wheel was connected and worked as intended. Initially, I thought a Hinge Joint 2D and a Distance Joint 2D connected to the base and the rope connected to the wheel would be sufficient. This was not the case. The wheel just fell down when the Distance Joint 2D was removed from the wheel. It turned out that the wheel and rope needed to be connected to each other in order for this to work properly.

Another problem that occurred when making the level was to get the spotlights in the background to work as they did in the original version. When writing code that has to do with angles in Unity, it uses quaternions. A quaternion is a calculation of angles based upon four numbers, x, y, z and w respectively. When it was used it in the past, I could still add an angle to the transform of the object and it would use normal angles to rotate around the origin of the object. Since this was different from the previous versions of Unity I had to find a different solution to the problem. The solution was to use a method called EulerAngles, which created a quaternion from the x, y and z angles. The quaternion could be used directly to tell Unity which angles should be turned and by how much.

The rest of the level was fairly easy. The mushrooms only needed to have a scale change added once they were hit by the player. When the player hits the mushroom, all that was needed was to add a force that pushed the player away whilst rotating them in a random direction. To do this, the original code for the SFML project was used as a template. A force and torque was added to the different bodies in order to be pushed away and rotated at the same time. Force can be described as something that pushes the object and torque can be used to rotate an object. I could not use the settings or the code from the old project, but it was possible to apply the same principles in a Unity script. The principle was adding a force that pushed outwards, like an explosion, and a torque value that rotated the object either counter-clockwise or clockwise. The result was fairly similar compared to how it was in the original game. One of the few differences in the level is where the mushrooms are placed, and where the frogs in the background are placed.

In Unity, there is a feature called prefabs. Prefabs is a way to copy objects that are the same, but with the ability to change their position and rotation while keeping all the other properties the same, including scripts that are attached. The mushroom was created as a prefab and copied into the scene using this technique.

5.4 Military Level

Constructing the military level and basing it on the original was not as easy as I had hoped. The first challenge was getting the rotor blades animating correctly. Reason was that the sprite sheet that was used was not properly aligned, once I started setting up how Unity should get the images from the sprite sheet I had to manually select what should be shown. The problem was that the selection boxes in Unity were not of the same size. These size differences made the animation look like it was jumping up and down. Once the problem was identified, it was necessary to make sure the boxes were the same size, but I was still not able to get it right. It was fixed by aligning the bottom of the selection boxes with the bottom of each image and making sure the height and width of the different rotor blades was the same (see Figure 9).
After doing that it looked a lot better. The below image is an illustration of how the selection boxes look:

![Image box selection screen in Unity (Sprite Editor)](image)

**Figure 9:** Image box selection screen in Unity (Sprite Editor)

The next challenge was to get the rotor blades to function the same way as in the original game. The setting in the SFML project was to create an implosion, as opposed to an explosion. The effect was that once characters collided with the rotor blades, they were sucked in and bounced upwards with a rotation, making it hard to escape. When playing around with the same setting in Unity, I was not able to create the same effect. Several settings were tested without being able to recreate the behavior. There was no solution for this problem other than creating a different effect. The effect created was using a setting where the players would be sucked in. The players would then be going back and forth on the rotor blades and be stuck until the force threw the player out to one of the sides. This made the rotor blades become less troublesome than they were before and it felt more natural.

The rest of the level was not that difficult. The mines were easy to create, although it took a little work to get them working correctly. When the first mine was created, it was saved as a prefab and then copied and placed several times. There was one problem which needed to be solved in code, which was to select the direction of the blast. For the two topmost mines, the blast would send the player to the right if they hit the left mine and vice versa. The two lower mines take the impact angle and use that data to send them back in the direction where they came from.
5.5 Lava Level

Until the lava level, the amount of time spent on planning the levels was minimal. I knew going into making the level it would be complicated to get it all to work well together. The different elements in the level that the players can interact with were what made it complicated. The challenge was how to deal with the buttons. The level contains three buttons; Figure 10-a shows the top left button as Button A, the top right button as Button B and the bottom right button as Button C. Button A controls Piston E (Figure 10-b) and Button B controls Piston D (Figure 10-b). Button C controls the lava flow as shown in Figure 5-a. Button B is independent from buttons A and C. Buttons A and C, however, are linked together because they are both activating an obstacle that occupies the same space. They are linked in such a way that if the player pushes either one, both are set in a pushed state and will be in that state until the
cool down is over. A pushed state means that it cannot be reactivated until the cool down has run out. A cool down is a concept where you have a timer that either counts up or down. I normally use one where you count down from the amount of seconds that I need. The counter is deducted every frame with the delta time. Once the counter reaches zero or below zero, the event you want to take place happens. In some cases, if it is a repeated event, the timer is reset and the counter starts over. In the case of the buttons, as described above, the cool down makes sure the buttons are in the pushed state for the duration of the cool down. Once that timer reaches zero, the button is reset to a pushable state and ready to be used again.

The problem that I faced was that the script I made did not work as intended. I started with a fairly complicated setup where I had a button manager. The job that the button manager had was to keep track of the different states that the buttons were in. As an example, if Button A was pushed, the event that Button A was going to perform happened. Graphically, Button A is set to a pushed state. Button C, which Button A was linked to, was deactivated and set in a graphically pushed state as well. After creating that script, I had to leave the button manager to work on other things at that time. When revisiting the problem, I found a solution that worked much better. Where Button A and Button C could send messages directly to each other and I could just remove the button manager completely. The buttons were able to communicate with each other, which was an easier solution than the button manager.

As mentioned earlier, the level has two pistons. Piston D moves from the top and downwards and Piston E moves from the bottom and upwards. When I started creating a script for how those two pistons should behave, the thought was to have one script that took care of both cases. The script that was created did take care of both, but it was much more complicated than making just a single script for each case. Because the solution worked, I did not redo the script and divide them simply because the alternative was better.

Compared to the buttons and pistons, the rest of the level was easier to create. The lava flow, as shown in Figure 5-a, was just a matter of gradually raising the lava. The script controlling the lava has several checks to see if the lava has reached a certain y-value. Once it reaches that value it stays there until the cool down counter reaches zero and it gradually starts moving down towards the bottom of the screen again. The lava was also updated with a particle effect to simulate smoke arising from the lava. This was done to make it feel more realistic. Later on, I added some orange particles as well, which are barely noticeable, but it also adds to the realism in the level.

5.6 Windmill Level
Of the playable levels, the windmill level was left last. There are several reasons for this, but first and foremost, it is the simplest level in the game mechanics wise. In the beginning, the intention was to not really change anything about the level, but then after having tested the game somewhat, the level felt dull compared to the others. When making the level, the new level was more or less identical compared to the old, apart from some small details. The collision boxes were more accurate compared to the original.

The second change was the placement of the timer that shows how much time there is left until the bomb blows up. The timer that was used in the old version looked displaced, and it is easier in Unity to set a placement that looks better since you can visually move it around. It was not that difficult in the old project either, but we tried to keep every detail, like the timer and scoreboards, the same on all levels. We wanted players to know where the timers were, but we noticed it was more the music than the timer itself that made the players understand when the bomb was about to blow. Moving the clock makes it more visible.
The third change in the level was the background clouds. In the old version, they were lower in the horizon, making them more anonymous. Initially, it was my memory that served me wrong; I thought the clouds were that high up. After a quick comparison, I realized they were indeed lower. During this comparison, I also realized the level was a bit more static compared to the other levels. I hoped having the clouds higher up and more prominent would make the level more interesting. As Figure 6-a and 6-b illustrates, the cloud placement didn't make much of a difference, but I decided to keep them as they were. I also wanted the players to have an animate object in the level. Since the windmill blades were already moving, they could be made interactive. The first instinct was to make the windmill blades into a wheel, because that is more or less how it should function. Once the body type was set to be a wheel it just fell down and started rolling to the right. This behavior made sense because a physical representation of a wheel will as in real life, follow gravity and fall down; unless it is attached to something.

From past experience in the swamp level, I knew a Hinge Joint 2D could be used in this case. After setting the parameters for the Hinge Joint 2D, which should be kinematic, the wheel was kept in place. Kinematic is a setting that tells a physical body that it should be set in place and never to be moved, like a wall in a house. In this case, the hinge should be where the wheel rotates around, so once it was set in place, the windmill blades just stood still. Turns out that in order for a wheel to move, it needs to be pushed; it needs a force. The thought behind it is that for instance if you want to have a believable 2D side scrolling game with wheels, the wheels is what drives everything forward by applying a force to the wheel. This holds true, even if you do not want to use the Wheel Joint 2D in Unity as wheels for a car. Once the motor was enabled on the Wheel Joint 2D, the wheel spun around, but the players were not able to slow it down or affect it. The motor was shut off to allow for the player interaction, however, this made the windmill blades come to a standstill. The solution to the whole problem was to turn on the motor once, then turn it off again. If the blades then stop spinning the motor will kick in and get it going again.

5.7 The Score Screen
The score screen was rewritten from scratch. The original code base was not used or looked at. The reasoning behind this is that the old score screen was a bit static and felt dead. The only thing the old screen had going for it was the way the players flew in and the background clouds were animated, but otherwise it felt very static and life-less. In the new version, the player’s avatars rotate a little bit from left to right to make them less static. The changes where you can see which placement the different players had by letting them fly higher up was something that was implemented more or less last minute based upon feedback from other players.

Implementing this level needed a lot more work and code than was anticipated from the start. The first problem that I encountered was figuring out how many players were active in the round. This would not have been an issue if I could keep the players as a prefab in this scene. When the lobby scene starts, it contains several Prefabs that are kept for the duration of the game. These prefabs are deleted in the score screen, because if they are not deleted in this scene, a duplicate would be present in the lobby when the game starts the lobby again. Circling back to the problem described above, by deleting the players I needed another way to keep track of the players and the score they had. I solved this using an array of Booleans and an array that kept track of the scores of the different players. This was connected to an identifier so that in code it is possible to find out which player had what score and if they played that round or not.
The next step was to find out who won the game. To accomplish this, the scores needed to be sorted and for that I used a sorting algorithm called bubble sort. Bubble sort sorted the values in descending order, so the one with the highest score was first in the array. The one with the second highest was number two in the array and so forth. Bubble sort was chosen because the maximum size of the list to sort was four. With such a low number, it is computationally cheap to use and easy to write in code. An un-optimized version of bubble sort would have twelve iterations as a maximum, and a computer can go through those loops without it being noticeable. That said, an optimized version was used, a worst case scenario in that case would be six iterations in total. Once the scores were sorted, figuring out who won, who was on second place, etc. was very straightforward. The problem arose when trying to find out if two or three players had the same score. All of those cases were manually written down, even if a better solution is likely to exist.

5.8 Invisible and Generic Changes
One of the biggest changes compared to the old game is that the game now features a way for the players to spawn while others are playing. The old game had a set number of four players. The new project allows from two to four players; if two players start the game, the third and fourth player can join in even if the game is in progress. This was done for two reasons, but mainly because there are not many who have four joysticks available to them. The second reason is in case of network support, which should allow the players to spawn when connecting to a server. Of the generic changes, the timers on the different levels are showing milliseconds as well as seconds. In the old version, milliseconds were only shown when the level time has less than ten seconds left. This was not a deliberate choice; at first I thought it was like that in the old version, but forgot that it was only when the timer reached less than ten seconds.

Capture the Bomb is a game mode that I wanted to add to the original game but never had the time to do. In the Unity project, adding the new game mode was not that difficult. The requirement for adding this game mode was to have a menu to allow for the players to select the game mode they want to play. In this menu, the players could then choose how many players they were going to play and also select the game mode they want to play. Adding game mode and menu did not take as long as I thought it would. By using a built-in method in Unity called UserPrefs, it was easy to store the game mode. It saves the settings if you quit the game, and that was what I wanted.
6 Conclusion

When starting the project, the original idea was to use the old source code as a template. After attempting to use the old code as a template, it quickly became apparent that it would not be possible. There were instances where I did use the old code as reference for certain values, however the old code was missing the delta time and that made the old values unusable. It was also clear that no code would be directly usable as the two languages are not compatible. I did, however, think that I could translate some of the ideas into scripts for Unity, but that was not possible either. Unity and SFML with C++ operate on two different levels. Unity has abstracted away much of the complicated coding required and provides tools to aid the developer. There are tools like the sprite editor, visual collision boxes and visual placement. SFML, on the other hand, does not have that and everything needs to be coded in manually. The differences between the language and usage of the two different systems were so different that I had to change how to approach this project. The change was that I did not look at the old code, but rather used the old game as a visual reference. Even if the approach to the project was not what I had in mind when I started the project, it still was successful. The game was recreated and because the engine had a lot of visual tools the development went much faster than expected. I was able to even add some features to the game after the main part of the game was finished, which were the in-game menu and the game mode.

Looking at the project and what was done, there are some changes that I would do. One example is to better plan the whole project, I had a plan for the project initially, but it was only a rough sketch. Taking the time to sit down and plan a project sounds unnecessary but writing down a specific plan even if you are working alone helps. My plan was all in my head, and even if I succeeded in recreating the game, it was sometimes difficult to keep track of what to do next and why. Sometimes it was more going back and forth fixing some code here and there rather than getting something done first, then move on to the next. Even if it in my case turned out well and I was able to get all the features I wanted into the game, proper planning could have made it faster.

The game play was not replicated fully, but it was very close to the old game. The input was the same, but how the player’s avatar moved around in the game world is a little different. There are values that can be changed to get it closer, but playtesting has shown that only the original members of the team notice any difference. When further developing the game, this is something that will be tweaked in co-operation with earlier members of the team.

What I have learned from this project is that it is possible to recreate a game in Unity. The old project will merely serve as a visual guidance. I suggest using the old project to plan the new project more thoroughly. Using an old project to learn a new game engine is a great approach. Even though I knew how to use Unity, I have learned a lot during this process. Unity is free for personal use and it is only when earning more than $100,000 (Unity3d.com) in a fiscal year that it starts to cost money. Apart from the learning curve there are no drawbacks in using a game engine like Unity. The learning curve is also substantially lower compared to C++. Unity may not be the right choice for every game. It depends on the type of game the developer wants to make and the knowledge of the developer. For my project, the Unity engine worked out great and I am happy with the result.
References

Web-pages


