Lieburo

Table of contents

1. Overview

Lieburo is a split-screen two players' combat PvP game. It is in the spirit of Worms and Liero. Players have different weapons that have different effects as well as a jetpack for flying around. The gamefield is static but has some specialities like invertible gravity. Lieburo has an irritating background music as well as lots of other sound effects.

The game uses Box2D for the physics modeling and SFML for the graphics.

2. Software structure

2.1. Class: Game

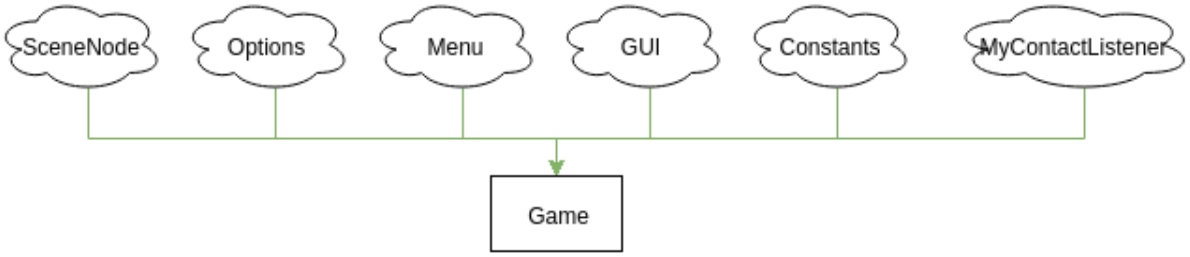

Figure 1: Game. Green arrows are compiler dependencies.

Game is the class that wraps together all other classes: for example calls the menu dialog or calls an updater for all the game objects. The instance of this class is responsible for the gameplay. Thus it is necessary for almost all other classes to have a reference to this instance as it is the link between all other classes and the user.

Game includes all other classes as well as Box2D and SFML libraries.

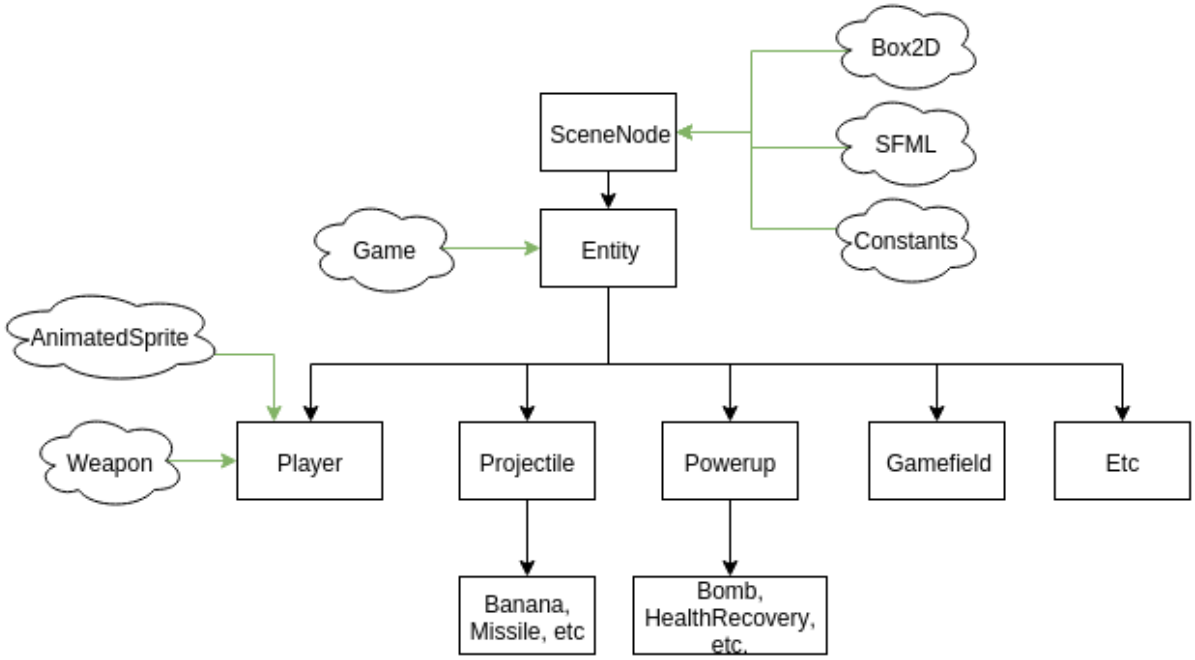

Figure 2: SceneNode. Compiler dependencies are green and direct inheritances are black arrows.

SceneNode is the base class for storing the game entities. It includes generalized methods for updating, drawing and storing the entities. Using SceneNode allows us to keep the game loop clean by only calling update and drawing once.

It includes Box2D and SFML libraries.

2.2.1. Subclass: Entity : SceneNode (abstract)

Entity is the base class for all independent, visible objects with a physical model in the game. It has more specific methods especially made for entities than SceneNode. It holds the data common for all entities, most important of which as follows: physical body, sprite, status (alive/dead), the game and world it is used in etc.

2.2.1.1. Subclass: Player : Entity

This self-explanatory class is responsible for handling the players. It has lots of methods and information thus it is the most complex Entity. In addition to Entity it includes

2.2.1.2. Subclass: Projectile : Entity (abstract)

Projectiles are the basically the objects that are shot from weapons and cause damage to players. Projectile is an abstract class, but it was considered practical to wrap all projectiles together as entity was considered too general for this. It has one method called fragment which proved very useful.

2.2.1.2.n Subclasses: Banana, Bullet, Missile, Shrapnel etc : Projectile

These classes are the concrete projectiles. Excluding Shrapnel they are all very special and have specific methodalities, but are still similar at the base level (physics model, sprite etc.).

Shrapnel is the class used for objects that are created when eg. a missile explodes. It has an effective constructor for creating many small flying objects that don't need very special methodalities.

2.3. Class: Weapon (abstract)

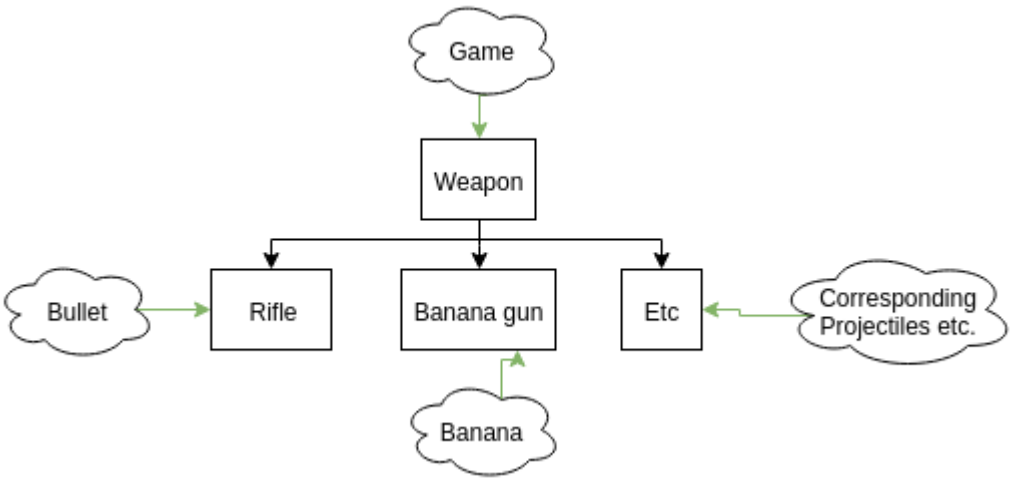

Figure 3: SceneNode. Compiler dependencies are green and direct inheritances are black arrows.

Weapon is the class from which all weapons inherit from. It has a an effective general constructor used by its descendants as well as some other general methods such as for actually launching projectiles etc.

2.3.1.n Subclasses: BananaGun, Rifle, MissileLauncher etc : Weapon

These classes hold the corresponding weapons and their specific data and methods.

2.4. Class: MyContactListener

This class holds overloaded methods for the Box2D collision callbacks. The collision callbacks were considered the easiest way to implement collision detection. Also they are great performance-wise because no multiple checks are required.

2.5. File: Constants

This file includes some constant values used by different classes in the game. These values could be stored in other classes too, but it was decided that these are the most general and user-dependant ones thus needing an own storing place.

2.6. Class: Menu

File contains variables to store navigation words and information for font etc. required for text.

2.7. Class: Options

Similar to Menu file but in much larger scale since it contains a lot more information. Contains a large map for almost all keys on keyboard.

2.8. Class: Sounds

File contains information for playing music.

3. Software logic and the most important methods

3.1. Software logic

The main software logic consists of game loops calling appropriate methods from instances of different classes in the game. The game logic is illustrated in Figure 4.

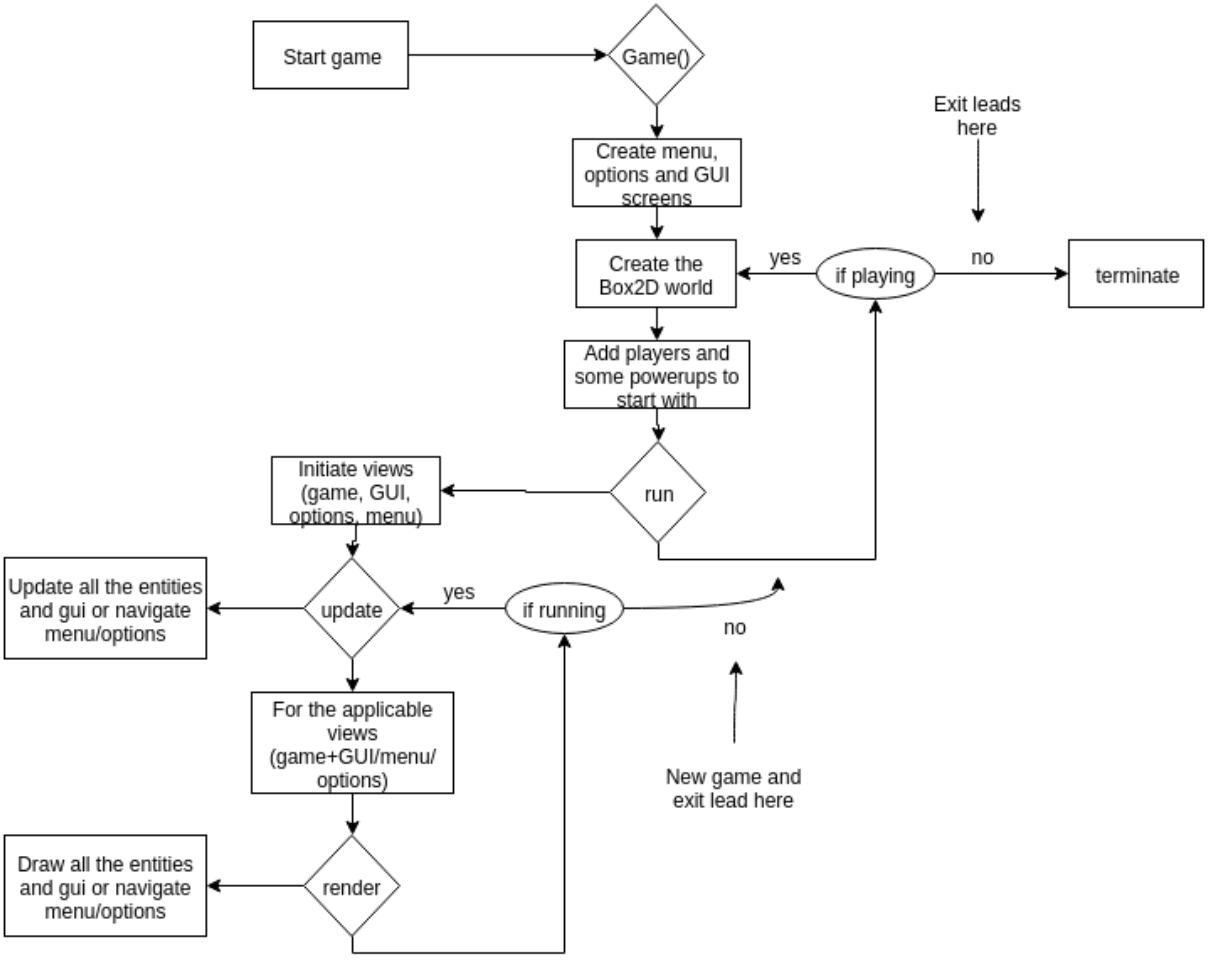

Figure 4: Game logic.

3.2. Most important method interfaces

3.2.1. Game

- The constructor is used to automatically start the game. It creates the game world, menus, players and some powerups. It then calls the run-method.
- The run method is used to drive the main game loop. It periodically updates all the game objects and draws the menu screen or the game views as per the current status.
- The update method checks for the state of the game ie. is the menu screen shown or not. If it is, it calls the appropriate methods for navigating the menu and options and pauses the game physics. If not, it calls the methods for updating the physical world of the game, handling user inputs and adding random powerups at times.
- The render method first draws the static gamefield, then all the entities and finally the gui by calling the appropriate methods.
- The limitPlayerCamera limits the camera view to the bounds of the gamefield.
- The createRandomPowerup method was needed because the game is more interesting if there are new powerups appearing during the playtime.
- The destructor removes all the entities and cleans up the game world.

3.2.2. SceneNode

- The methods are used to modify the vector of the game entities. The names are self-explanatory. They offer an easy way to access the desired entities (or all of them) and performs simple tasks for them efficiently.
- It should be noted that if an entity is removed from the vector, the possible iterating loop must be broken as the iterator invalidates.
- Only alive entities are stored, updated and drawn. Our original plan was to store a given amount of entities as a reserve in the vector, but it turned out to be too complex and gave poor

3.2.2.1. Entity

- The draw method is used to synchronize the entity's visual sprite with the entity's body and then draw the sprite using SFML's draw method. There is only one exception which is that players aren't drawn by this method but rather that this method calls player's own draw method. This system proved better as the number of separately drawn sprites with the player started increasing.
- The baseConstructor method was created to ease up the creation of new Entities. It can handle the most common and fundamental operations such as creating the body and sprite from a given texture.

3.2.2.1.1. Player

- The constructor is used to create the player. It doesn't use the baseConstructor method as there are some special features such as animated sprite and so on. As it can be seen, there are many sprites initialized such as the weapon and the aim dot.

This is where the special draw method comes in handy for drawing them all. Also a foot sensor is added to check whether the player is standing on a gamefield object.

- The aim and movePlayerX methods ease the use of user input. In movePlayerX (which moves the player horizontally) it gave best results to change the player's position a bit per update as well as to give it just a tiny amount of linear velocity. If the movement was done only with linear velocity, using the jetpack would not have been very pleasant. There is a direction indicator (int) and a previous direction indicator (int) which control the mirroring of the sprite.
- The jump method handles the vertical movement. As the foot sensor makes contact with the gamefield, a counter is increased/decreased. If there are contacts the player can only jump. After a given time when there is no ground contact the jetpack becomes available. The jetpack has a speed limit because it would otherwise fly like a rocket.
- The fire method sends the current weapon a call to launch a projectile.
- The update method is used to position all the player's accessory sprites correctly (drawing them is left in the draw method). It also handles the player's lives and respawns. A waitingForRespawn flag was needed to query the respawn command from the player. While waiting, the player's fixture is set as sensor to take away the nasty invisible box that is somewhere in the gamfield. The spillBlood method is called here with a private variable as an argument. It can't be called in any contact-related methods such as startContact (which would otherwise seem intuitive: spill blood at the same time as reduce hp) as per the limitations of Box2D.
- As mentioned earlier in 3.2.2.1. the player has a separate draw method for drawing all of its private accessories in the desired order. This is because the Entity's draw method doesn't have to know about the player's private sprites and so on.
- ScrollWeapons iterates through the list of the player's weapons and changes the weapon sprite accordingly.
- SetCommands and resetCommands load the stored commands from text files.
- HandleUserInput is called every time the game is updated (~60Hz). Although the user input must be disabled while waiting for respawn (to disable any projectiles from being shot by accident flying from nowhere) or for some other reason.
- The spillBlood method throws around tiny red droplets representing blood. It uses a simple random generator for the creation point and the flying direction. It was also decided that the blood can only collide with the gamefield.

3.2.1.1. Gamefield

- The Gamefield is generated from a text file where each line represents certain kind of shape. Each line is constructed from shape or type, texture path and then various parameters which are explained in detail in Gamefield.cpp. Gamefield initialization is also explained little bit more in detail in the Gamefield.cpp. In the gamefield file from where gamefield is initialized (text file) has to have also the boundaries that's why there are also non-friction rectangles for the ceiling and walls.

3.2.1.2. Projectile

- The fragment method is used by fragmenting projectiles. This generalized method was created to be able to easily add small flying and possibly exploding objects with

a damage. A common method was superior to others as fragments are generally all the same but with different sprites, damages and lifetimes.
- In the subclasses there are startContact methods that generally reduce contact's hp if it is a player.
- The way that all Projectiles is created is by first making a shared pointer of the corresponding projectile type and then dynamically cast it to SceneNode to attach it to the entity vector. This was a rather hard task as the dynamic cast was a bit hard to come up with. Although this technique has later proved very useful.

3.2.1.3 Banana

- In the update method the lifetime is counted. If it is near end, the banana changes its sprite to explosion and fragments.

3.2.1.4. Bullet

- No specialities, just plain, flying hp reducer on hit.

3.2.1.5 Missile

- The seek method tries to find its path to the opposing player. The missile gets the target's number (1 or 2) as an constructing argument from the missile launcher. For a smooth controlled flight the missile is being affected by a constant force in the direction of the player. The body is rotated to be tangential to the velocity. In order to be able to fly somewhat well the maximum velocity had to be limited. The missile has a short arming time while it flies to the direction it is shot before seeking to look more realistic. To not be too powerful weapon it stops seeking near the target to be able to be dodged.

3.2.1.6. Shrapnel

- The constructor was made as general as possible in order to easily add the shrapnels by the fragment method in the Projectile class. They don't have any specialities but were optimized more as being easy to create.

3.2.1.7 Powerup

- Powerups constructor doesn't use the baseConstructor from the entity beacause it needs little tweaks and one more parameter than the baseConstructor. Basically it creates a body,fixture and sprite and adds the all together. The position is random so powerups just pops-up randomly within the gamefield boundaries. Powerup's constructor also need a bool type parameter to determine if the body is static or dynamic. All powerups which derive from this class use the same constructor.
- Powerup also has isActive() and setActive(bool status) methods to check and change the powerups status.

3.2.1.8 GravityPU

- GravityPU powerups changes the world's gravity by random between 0 and 15. startContact method checks if the contact is with player if true and alive is also true it calls the first changeGravity method. Sets the status as active, makes the sprite invisible and the fixture as sensor so no collision is made after the first contact.

- After the lifetime is full it calls the second changeGravity method which is overloaded and sets the gravity to original and then destroys the body.

3.2.1.9 GravityInverter

- Basically the same as GravityPU instead it turns the gravity upside down.

3.2.1.10 HealthRecovery

- Same logic as two previous ones. Instead of changing the gravity it calls the players updateHP method and adds health.

3.2.1.11 Bomb

- Bomb powerups logic is also almost the same. In the startContact method it calls bombExplosion method and once again updates HP only this time with negative value.
- bombExplosion method uses Box2D AABB Query so it creates area around the bombs body and applies linear impulse to all bodies in that area. This was made to simulate a real bomb behavior. bombExplosion also disables user input for a little while if player is in that query area.

3.2.3. Weapon

- The constructor is generalized to assign the variables common to all weapons.
- The launchProjectile method gives the given projectile the given flight parameters.
- The multiple getters are mostly used by the GUI in order to update the ammo bar accordingly.

3.2.3.1. Banana gun

- The shoot method shoots the given projectiles (Banana) with specific parameters. It checks the re-fire and reload times.

3.2.3.2. Missile launcher

- Similar to banana gun.

3.2.3.3. Rifle

- Similar to the previous.

3.2.4. MyContactListener

- BeginContact is an overloaded method from the Box2D collision callbacks library. The idea is that it is possible to find out the entity instances of the colliding fixtures by storing them in the user data of the colliding fixtures' bodies. That way we can call the entities' startContact (see chapter 3.2.2.1.) method with the other entity as an argument. The original idea was to also pass the type of the colliding entity to the startContact as another argument but it proved easier not to do this. Instead the startContact method resolves the type with typeid method. As the Box2D bodies'

user data is of type void*, they must be casted as an Entity* before passing them as an argument.
    - This method is also used to update the status whether a player can jump or not by using the fixture user data value. This was discussed more in chapter 3.2.2.1.1.
- EndContact is used used to update the status whether a player can jump or not.

3.2.5. GUI

- createBar and createText helper methods for avoiding the repetition of setting positions, sizes and colors of GUI-elements.
- Separate draw method because GUI-elements don't have any physical movement and thus aren't included in SceneNode.
- update method for gathering numeric values from players and updating the text and visual bars accordingly.

3.2.6. Menu

- Variables hold data for navigation texts and for program to know when to display menu.
- methods allow user to navigate with arrow keys.
- Getters are for Game class
- Constructor and setPosition() contain all information required for text.
- Move up and down are for navigating.
- draw() draws words to the screen.

3.2.7. Options

- Since this class is quite the same as Menu class, only different abilities are written here.
- Options allows user to change keys and also to remember them so that the player doesn't need to change them every time.
    - It writes them to a file.
- Contains information for player
    - All keys and also return key "P" for returning back to menu from game.
- Contains reset for keys
- A bit of this class is stored in Player class to make the programming a bit easier. Keys for player are stored in Player class and changing them happens also there.
- Move (up, down, left right) allow user to navigate and they are implemented with indexing.
- setText() makes words to display correctly.
- Huge lines of code went to a large map which contains keyboard information. It makes changing keys possible since it contains almost every key on keyboard.

3.2.8. Sounds

- Constructor contains information for music files.
- Variables contain the files themselves.
- methods play, pause and stop work as normal music player buttons and are quite self explaining methods.
- Get-methods are for Game class to know which audio track to play.

4. Using the software

The game has two 3rd party dependencies: Box2D and SFML that should be installed in order to run the software. Once installed, all that needs to be done is “make run” in the corresponding folder

Playing the game has a simple objective: shoot the other player as much as you can. The controls can be assigned in the main menu that shows up upon initialization and by pressing “P” during the game. Navigate to the control to change, press “Enter” and then the button you want to use for that action.

Test different weapons and look for special objects in the game field. Some can for example change the gravity or give you more health.

5. Testing

Testing was mainly conducted alongside development. As new features were added, the software was ran multiple times simulating different scenarios. If everything seemed to work fine, the software was run with Valgrind to check for memory leaks. For investigating runtime errors such as segmentations faults gdb was used. It allowed for easy backtracing of problematic code. The following chapters describe the tests used for different parts of the game:

5.1. Game Logic

Main screen doesn’t require a lot of testing. Proper methodality was tested by pressing different keys and trying to make the program to fail. We didn’t notice any mistakes in that.

Options was tested with in a similar way but it included changing uncommon keys such as selecting every button to do the same thing. Game still works as expected, meaning that it doesn’t fail even though it’s not possible to play anymore. Resetting was also tested with different keys and everything works as it should. Another tested thing in options was to check that the user can’t change the keys which are hard coded such as P which pauses the game and returns to main screen.

We also used other students to try to play the game and each and everyone could navigate to options to change the keys and to actually check which buttons does which action. Therefore, we didn't see the need to add more instructions.

Sounds were tested by pressing new game, resume game and returning back to menu with rapid time intervals and listening whether it works or not.

Game loop, update and render methods etc. were tested by playing the game and by trying to spot any odd behaviour. It was noted that all the sprites initially moved at a very low speed and were lagging much. This was found to be caused by the fact that the update thus the Box2D's step method were running at about 500Hz. This was fixed by reducing the frequency to 60Hz by a delay loop.

5.2. Entities

Entities were tested in the game by creating large amounts of them. We had problems with the destruction of the Box2D bodies, but they were solved by setting a separate destructor for the projectiles. The entity class itself has an empty destructor so it doesn't destroy any bodies.

The initial design to create all the needed entities in advance soon provided too heavy to achieve reasonable performance.

5.2.1. Player

Testing was mainly done by playing and setting some informative prints in different spots and playing the game. One thing that was noticed was that it was possible for the players to respawn outside the playing area. This was fixed by setting a 100px marginal to the edges of the gamefield in the respawning area. It is though still possible that depending on the locations of the gamefield objects for the player to respawn "trapped". We had also problems related to moving the player and mirroring the sprite as the direction changed, but they were fixed. Other problems were mainly gameplay issues, such as too strong jetpack or too slow aiming speed etc.

5.2.2. Projectiles

Projectiles were mainly tested by shooting large amounts of them all over the place. Initially some segfaults were noted which turned out to come from the removal of the entities from the entity vector thus invalidating the iterators which hasn't been taken into account. A simple break statement fixed this.

It was also noted that there is a possibility for the projectile to explode right in the shooter's face. This was diagnosed to be caused by the projectiles being created so close to the player that their fixtures were actually touching thus calling the startContact method and exploding. A fix was made by setting a small time delay for the projectiles before which they can't explode. This makes it possible for some odd behaviour if the projectiles are shot at near distances. As the projectiles aren't also shot from the center of the player but from a

small distance (from the point where the aim dot lies) it is also possible to shoot through walls. This was later considered as neat gameplay feature and thus left in the game.

5.3. Weapons

Tested with normal gameplay. There were some initial problems with changing the weapon sprite and positioning it so that it always points to a consistent direction ie. turns with the player and so on. This was fixed by mirroring the sprite and doing some sign changes in the update method.

5.4. Collision callbacks

Collision callback are handled by the MyContactListener class. Tested with normal gameplay with increased amount of collisions. We had many segfaults in the test phase as the user data of the colliding objects must be converted from void* to Entity*. A static cast seemed to fix the problem. The passing of an Entity* argument to the startContact method proved surprisingly elegant.

5.5. Game

The methods of the Game class were tested in normal gameplay. In the limitPlayerCamera method there was a minor flaw that resulted in a slightly misaligned viewports. It was fixed with a carefully selected constant offset.

5.6. GUI

Most of the GUI testing was done by visual inspection ie. what looks good and what doesn't. There was problem with health bar at first by not reducing to zero if less health was left than the damage done by the weapon. This was fixed by using clamping method. The GUI-elements were at first hardcoded which turned out to be a bad idea when resolution of the game was changed. This was fixed by calculating their relative positions from screen width and height.

6. Work log

6.1. Responsibilities

6.1.1. Joel Lavikainen

- Graphical User Interface a.k.a statusbar containing healthbars, current ammunition, reloading, lives and other informative text such as respawn reminder
- Basic game loop
    - Setting up sane game loop with limited update rate of 60 FPS
- SceneNode implementation
- Animation system integration

6.1.2. Alvar Martti

- Base class Entity for all game entities and full compatibility with SceneNode and game logic (update, render).
    - Player class
    - Projectile class
        - +Projectiles Missile, Banana, Bullet, Shrapnel
    - Initial development of Gamefield class
- Base class Weapon
    - +Weapons Banana Gun, Missile Launcher and Rifle
- Few methods in Game class that relate to the current state of the Player (such as getPlayer(int)) and the parts where Entities are created and destroyed safely(a conceptual model for Game() and ~Game()).
- Base structure for Project plan and documentation
    - +Documentation of entities, weapons, collision callbacks and game logic and their testing.
- Collision callbacks
- Moving the options and menu methods to their corresponding classes from the Game class.

6.1.3. Samuli Mononen

- Options and Menu classes for showing main screen and ability to change keys
    - Writes keys to a file and therefore remembers them
- Text files for keys
- Selecting and finding suitable audio and font
- Split screen
- Menu and game screen sounds
- Part of player class because a bit of options are stored there (e.g. resetCommans() method and getCommands())
- A few things in Game class such as update since I constructed how the screens are viewed.

6.1.4. Noah Nettey

- Powerups
- Game area

6.2. Worklog

6.2.1. Week 44

All: Initial project meeting. We decided the topic and made some initial plans on what the game structure could be like. 1,5h.

Samuli: Studying SFML and trying to understand main methodalities. Since it was a bit difficult to understand the whole scope of the program. Also division of work made it more difficult for me because I didn't implement the basic methodalities.

Alvar: Writing parts of the project plan and drawing diagrams. 2h.

Joel: Reading SFML game development book and browsing iForce2D website for Box2D tutorial to get a grasp of usage. 2h.

6.2.2. Week 45

Joel: Implementing base structure for project. Projectile, Entity, Game and SceneNode classes. Basic game loop. Reading SFML game development book. 3h.

Alvar: Finishing the project plan, reading Box2D tutorials from iForce2D and the SFML game development book. 8h.

6.2.3. Week 46

Joel, Alvar, Samuli: Project meeting, 2h. Things discussed:

- Box2D collision callbacks can only set flags indicating that a collision has occurred. (This was later abandoned as a bad idea and used the startContact methods instead, see chapter 3.2.2.1.)
- Keyboard presses only set flags that are later interpreted. Such flag could be for example in the Player class a "jump" flag. (This idea was also abandoned later, see method handleUserInput in 3.2.2.1.1.)
- Adding music was considered good.
- The most important goal at the moment was decided to get the code compile. This would enable us to actually see what we are doing.

Samuli: Time used roughly 4 hours to study basic methodality for menu screen and to implement it. At the moment player can only move between different navigation words but after this it's going to be a lot faster to make it look more appropriate and to make the navigation words to actually work. I had difficulties with using separate files to work together but it was just a lack of coding experience. I also modified the README file to be more informative.

Joel: Implemented inheritance between Entity and SceneNode. SceneNode update-method implemented. Turning Game into a singleton and allowing proper closing of the window. Modifications to game loop to utilize SFML:Time methodality. Studying structure of game loops from internet. 4h.

Alvar: Implemented the early main.cpp and created the base for makefile. Refined naming conventions in Game.cpp as well as added synchronize method(deprived). Added Player class with a test image. 4h.

6.2.4. Week 47

Samuli & Alvar: Little menu tweaks and merged player_test branch to work with master branch. Now menu works with already implemented game methodalities. Estimated time 1h.

Samuli: Implemented Options class and made it to work with existing program. Most problems occurred when I wanted to use less lines of code e.g. prevent copying my own code but didn't remember how to use templates. We'll see whether this is an issue we want to pursue. A little bit more important thing would be to edit the Game class so that it contains Menu and Options classes because then we wouldn't need to worry about passing information from one method to another. Estimated time 2h.

Joel: Reading on GUI implementations and looking at example code from internet. 1h.

Alvar: Added Early stages of Banana, BananaGun, Weapon and MyContactListener classes. Trying to get the code compile with all the different preprocessor directives. Tried to improve the game logic by the parts of scenenode, update and rendering. Added protected variables to entity to make the class more inheritable. Enhanced the user interface for the user to be able to shoot bananas. Added Constants.h for simple storage of constants. Merged the player_test branch with master to add the basic structure of a movable player that shoots bananas. Added aim dot and fine-tuned the shooting system. Added a base structure for the gamefield that could easily be made more complex. 22h.

Noah: Started to work on the gamefield and to design in more detail what it would look like. 4h

6.2.5. Week 48

Samuli: Time used approximately 4 hours for research and coding split screen. The most difficult part was to understand our other team members' code and to know where to put implement the split screen. I also had problems with showing everything in right dimensions but after a little struggle everything is now in condition. On top of that the Menu and Options screens/classes are now part of Game class which made it easier to use them in the actual game file. Now I could remove all useless parameters from methods making the code much nicer.

Joel: Adding status bar view to window. Implementing the first version of player health bars. Healthbars are defined in game class which feels cluttered, decided to move them into separate class later. 3h.

Alvar: Added jump, jetpack, another player and made bananas explode when their lifetime has exceeded. Added the draw method to Player to ease up the drawing of all the player accessories. 15h.

Noah: Created the gamefield only made from boxes and circle and triangles created the world’s background and modified few other textures with gimp. 8h

6.2.6. Week 49

Samuli: Renewed the options menu layout to be more intuitive to use. Still quite many problems with user input e.g. getting user's commands to the sfml Keyboard class. Also changed both players' buttons to a lot more convenient and at the moment the game can actually be played. Time used over 6 hours, mostly with user input.

Also more implementations to selectable keys. Problem with getting input commands solved with 100 lines of code to insert values to a map which now knows almost all possible keys. After 4-5 hours I finally got the keys to show on the screen and to actually change. Now I only needed to make the changes to work a bit differently, because at that moment the user needed to press the key all the time when user wanted to change it, which wasn't very user-friendly. This was solved making a loop which doesn't end if users hasn't selected proper key. Now Key changing works. It doesn't save to a file, but it's not necessary and maybe not even wanted. The buttons doesn't also flicker etc. which could make it a bit easier to understand.

Keys are now stored in text file and changes will preserve after quitting the game. I encountered many problems with renaming files and actually what I needed to do was to change fstream to ofstream and ifstream. Also editing text files was more difficult than I expected because I couldn't find a method which would do the job easily. Time used 4 hours of which most went to study file handling and especially writing to a file.

Joel: Moving the GUI elements into a separate class. Implementing info about ammunition, health and jetpack status. Had to implement few getter-methods into Player class for data acquisition. Decided to draw GUI-elements far away from actual playfield. Separate GUI class integrated surprisingly well with existing code. 4h.

Alvar: Added rifle that shoots bullets and a missile launcher that shoots missiles that seek to the other player. Added animated weapons. Removed compiler warnings by declaring unused variables. Added fragmenting bananas and a base for all fragments and shrapnels. Created a new folder for textures. Tweaked missile. Made the Projectile- related methods more elegant by removing repetition as well as improving the collision callback handling. As per the performance issue removed all non-alive entities from the game world. 19h.

Noah: Created all the powerups and took a little while to get everything working nicely together. First idea was to make every powerups with their own constructors but then realized that is unnecessary and then created the Powerup class as a base class for all other powerups to use the same constructor. The hardest part was to get the bombs query area to work correctly and to apply the force so it would have some real effect on the player and other dynmaic objects. 16h

6.2.7. Week 50

Samuli: Added ability to reset buttons. Quite straightforward because it was almost the same as to change the buttons. However, I have a couple difficulties with printing everything on the screen since I changed the buttons in a bit different way. Nothing serious though. Also

included making the code a bit more reliable because keys still changed when they shouldn't. Time spent 2 hours. Added center bar and edited menu for future implementations. 20 mins.

As my last work for this project I implemented background music and Sounds class for it which plays certain audio track depending whether player is playing or in menu. It was a bit difficult to find suitable audio tracks due to copyright issues but I managed to find proper ones which are allowed to use in this product with mentioning credits. I also updated key changing so that the selected key now changes its colour to red when user is changing it.

Joel: Added simple animation system based on sf::Sprite which allows for simple integration into existing system we had and ease of use. Animation system is based on tutorial at SFML wiki. Also made the Player animated and changed graphics. Also made total GUI revamp with less number and more intuitive graphics. Ammunition and reloading time is now implemented as one simple bar graphic. Lives are also shown on the screen. Small text to remind about the key used for respawning. Started to implement laser weapon. Documenting self-implemented classes. 12h.

Alvar: Fixed jumping to work from the improper use of collision masks. Set the player to respawn properly as well as starting the new game. Added blood effects. Added a camera limiter to limit the views to the gamefield’s edges. Moved menu and options’ methods to their own classes from Game. Noted errors on the players’ bodies sizes and made them actually work. Some final refractoring. Project documentation: Figures 1-4 and chapters for the classes and logic being responsible for or the main author. 22h.

9.12. All: Project meeting. We divided work for finishing the program and fixed screen resolutions and added new power-ups. 2 hours.