Vanier College

Animal's revenge

Project report

MAZE

Ethan Tran 2270000
Anton Lisunov 2246745
Zachary Tremblay 2244682
Mackenzie Rouchdy 2268182

VANIER
CÉGEP/COLLEGE

Program Development in a Graphical Environment
420-203-RE

Sleiman Rabah

December 18, 2023

1 Documentation

1.1 Overview

Animal's revenge is an immersive interactive graphical physics simulator centered around projectile motion and rigid body dynamics (RBD). Upon startup, users are greeted by a home screen with a variety of buttons and menus, determining where they want to go and what they want to do. User launch a projectile at a structure that was built with a drag and drop system allowing users to have invoke their creativity and design unique structures. Users have full control over the type of projectiles, their size, density, shape, etc. and are able to save the layout as a level and load it for later use. The projectile collisions with the obstacles will cause a ripple (domino) effect with the obstacles that it collides, one causing the movement of the next. The movement of these interconnected bodies under the action of external forces (RBD) acting upon one another was implemented with the help of a physics engine. The mathematical algorithms necessary to implement projectile motion, kinematics, and rigid body dynamics include the Pythagorean theorem, trigonometric functions, and the derivation of acceleration from velocity.

1.2 Implementation

1.2.1 Unimplemented features

Our team implemented all major features that were planned.

1.2.2 Features

- Projectile motion in 2D: Users will be able to launch projectiles along a specific initial velocity vector. They will be able to manipulate that vector in the simulator (customizing its direction, velocity magnitude, and angle).
- Drag-and-drop obstacles: Users are able to spawn in obstacles into the simulator based on selected parameters. These parameters include, the shape, the texture, the friction, the rotation and the size. By pressing `CTRL` + `LMB`, the users will spawn an obstacle with the specified parameters at the position of their cursor.
- Rigid-body dynamics: The application effectively simulates highly sophisticated rigid-body dynamics and forces between obstacles and projectiles with the help of a physics engine.
- Kinematics Graphs: When a user launches the projectile by clicking on the launch button, a window with 3 tabs of kinematics graphs appears, including y position, velocity v_y, and acceleration a_y versus time t, which describes the motion of the projectile.

- Custom projectiles: Users are given the ability to create and customize their very own projectile. They are able to choose the shape of the projectile, the size and the colour. Furthermore, users are also able to customize non-physical properties of the projectile such as its restitution (how bouncy something is) and its density. Allowing the user to have these options gives them to ability to learn about projectile motion and how changing such parameters can affect it.

- Parameters customization: Users are able to open the Parameters window to change some of the parameters of the simulator. These include the value of gravity and whether the graphs are turned on or off.

- Save and load: Users can save a certain layout of obstacles into a file (this is essentially a preset level), such that they can load that same layout in the future even if they close the application.

- Settings: User can modify background, music and sound style and volume. Those modification are automatically saved, and after exiting and restarting program, users preferences will be saved.

1.2.3 Implementation of major features

Projectile motion in 2D: In FXGL, actions are defined using the `UserAction` class [1]. The class which models the launching of a projectile can be found in `LaunchAction` in the `actions` package. When the user initiates the action by right-clicking mouse, we obtain the position of the mouse and check if the action was initiated with the mouse touching the launcher. If it is, then on every frame where the mouse is held down following that, a vector (represented by a `Rectangle`) whose position is set at the bottom left of the screen will be scaled by a factor equal to the distance between the cursor position and the bottom-left of the screen, and rotated by an angle such that its end will touch the cursor. From Figure 1, 2 variables are known: the adjacent side (which is simply equal to the `x` position of the mouse), and the opposite side (which is the difference between the height of the screen and the `y` position of the mouse since it is measured from the top). One knows from the Pythagorean theorem [5] that $c^2 = a^2 + b^2$. From this, we obtain the hypotenuse by $c = \sqrt{a^2 + b^2}$, where c is the hypotenuse. From trigonometric functions [6], one knows that $\tan\theta = \frac{a}{b}$, $\sin\theta = \frac{a}{c}$, and $\cos\theta = \frac{b}{c}$. The angle θ can be found using $\theta = \tan^{-1}(\frac{a}{b})$. The vector body is then rotated by an angle of $90 - \theta$ because the `setRotate` method rotates from the vertical axis, and we use `setScaleY` by a factor c to scale the height of the vector body to perfectly line up with the cursor. Once the user clicks the `launch` button, the process is reversed, as we are now trying to find the the initial x velocity (adjacent side, or b in Figure 1), and the initial y velocity (opposite side, or a) given the hypotenuse c (using `getScaleY`) and the angle of rotation θ (using `getRotate`). To find the initial v_x and v_y, we use $b = c\cos\theta$ and $a = c\sin\theta$. Once this information is obtained, we can spawn the projectile in `Factory` and set its linear x velocity to v_x, and its linear y velocity to v_y. Had the project been a pure and simple projectile motion simulator (without RBD), it would have been feasible to

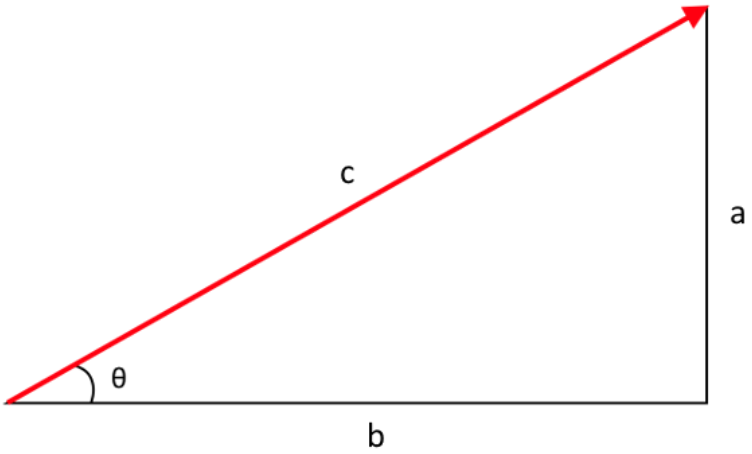


Figure 1: A right-angle triangle with angle with sides a, b, c

simulate projectile motion by calculating the path and using a `PathTransition` to move the projectile. However, given the need to integrate projectile motion with RBD in FXGL, there were no alternatives to our approach.

Rigid body dynamics (RBD): FXGL's game world utilizes an entity-component system (ECS) [1]. In FXGL, the `Entity` class is nothing but a generic class that represents any object in the game world. Components model behaviors, and multiple components can be attached to a single entity. Adding a component is also similar to adding a method. Components allow us to make an entity do something, thereby defining entity behavior. FXGL uses JBox2D's physics engine under the hood to simulate rigid body dynamics. FXGL's API allows interacting with JBox2D through the use of a predefined `PhysicsComponent`, which embeds all the necessary logic to make an `Entity` a physics entity, such as applying external forces, torques, setting linear velocity, density, etc. Through the use of this physics engine, we are able to simulate realistic movement of multiples bodies under the the action of external forces and torques, as well as handle linear momentum conservation (and energy dissipation) for colliding bodies. There were no alternatives to using FXGL alongside JavaFX to simulate rigid body dynamics. FXGL is the only library that is a super-set of JavaFX and contains a physics engine. RBD involves highly complex frame-per-frame calculations for collision detection, resolution, rotation, force, torque, etc. JavaFX is not optimised for handling so many calculations at one time. In addition, simulating RBD without a physics engine is a nearly impossible task: the sheer number and complexity of the calculations for all the objects at one time would render the task impossible.

Kinematics Graphs: The kinematics graphs were implemented via a `Timeline` that is attached to an `EventHandler` method that would update the chart every 0.1s with the projectile's motion data. To get its y position from the bottom of the greet, we do: $h_{screen} - h_{entity} - y_{entity}$, to get the starting y position as 0, and positive moving upwards. To get velocity, we invert the value returned by `getVelocityY`, because it returns a value positive increasing downwards, and convert to cm to match the unit of the graphs. One

knows that acceleration is the derivative of velocity with respect to time [3]:

$$\vec{a}(t) = \lim_{\Delta t \to 0} \frac{\vec{v}(t + \Delta t) - \vec{v}(t)}{\Delta t} = \frac{d\vec{v}(t)}{dt}$$

Since `Timeline` is only updated every 0.1s, one knows the time interval $\Delta t = 0.1$s. Thus, to calculate the instantaneous acceleration, it suffices to subtract the velocity in the previous 0.1s to the current velocity and divide by 0.1s (assumed elastic ball stays in touch with ground). However, in the context of projectile motion, we know that acceleration in the y direction is $a_y = -g = -9.81 m/s^2$ (or whatever value a_y the user sets in parameters). Because the ball bounces, the only time this acceleration changes is when **a)** the projectile bounces off the floor or **b)** the projectile rebounds from the ceiling. The only time either of those conditions is true is when v_y changes direction. From this, it follows that to update the acceleration graph, we need to check if the ball's v_y component changes direction: if it is, then calculate the instantaneous velocity at the point. If it is not, put the value of gravity as a_y. To the best of our knowledge, there were no alternatives to this approach. We searched the documentation thoroughly and we came across methods that gave access to an entity's position and velocity, but not its acceleration, therefore we used those to get kinematics data. What results of this computation is a graph that approximately looks like:

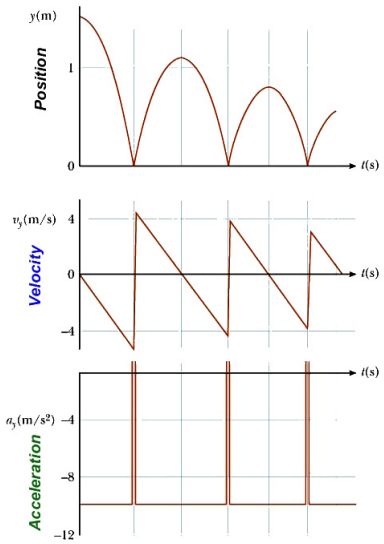


Figure 2: Kinematics of a bouncing ball

Drag-and-drop obstacles: The action of spawning and dragging obstacles in the scene can be found in the `DragAction` class in the `actions` package. When the user initiates the obstacle creation action via `CTRL` + `LMB` in the main window, we check if the user is creating a new obstacle (if `CTRL` is held down) or dragging an existing one (mouse must be touching an obstacle). If the user is creating an obstacle, we fetch the data from the right-hand panel (size, rotation, friction, shape, and texture) and use it to create a new instance of the `Obstacle` class. Using this class, we use the `spawn` method inside the `Factory`. In FXGL, to pass data to the `spawn` method of an entity factory, one must use

FXGL's `SpawnData` object. We pass the `Obstacle` object to the `SpawnData` (contains all necessary spawning data), which is then passed as an argument to the `spawn` method inside `Factory`. Inside the `Factory`, we then parse the data from the `Obstacle` (shape, size, rotation, friction, etc.), spawn an entity and attach a `PhysicsComponent` with the embedded data to it. For every following frame where the `LMB` is held down, we obtain the difference in position between the obstacle's position and the mouse position (while taking into account offsets, because `getY` and `getX` return position of top-left corner) and set its linear velocity v_y and v_x. This has the effect moving the obstacle to the mouse in a smooth fashion because its linear velocity decreases as the obstacle approaches closer and closer to the mouse position. When the user releases `LMB`, we deselect the entity that was dragged so that next time the action is initiated, it does not re-drag the same entity. We chose this approach instead of implementing an actual drag-and-drop system where the user would click on the desired obstacle and drag the cursor to the desired location, because the alternative was impossible. Since UI elements are not entities in the game world, we create invisible static "wall" entities that cover the area of the the right-hand panel and the menu bar at the top. This makes dragging obstacle out of the wall impossible, so we settled with the `CTRL` + `LMB` approach instead.

Custom Projectiles: Custom projectiles were implemented by taking each custom projectile as a shape and implementing the `Serializable` interface [4]. The implementation of this feature consists of three parts. The first part is the creation of the projectile. This is done on the screen where the user is greeted with a user interface built for allowing the user to create projectiles with a variety of features. The second part consists of saving the object that is created. This is done by taking each field of the custom projectile as a string and then serializing those fields on the user's computer. The third part is accessing the shapes the user has saved from the computer and deploying them back into the application. There are two possible ways a user can do this, either the user enters a simulation just after creating his custom projectile, or he creates the custom projectile, closes the application and then re-opens it. In the case of the former, the user will be greeted by a list of `Custom Projectile`s that he just created on a previous screen when they click on the `Custom Projectile` button. In the case of the ladder, the user must import their `Custom Projectile`s via a file on their computer. This is simply done via a button in the `Custom Projectile` dialogue box. Once the user clicks on the file, it is deserialized and placed into the dialog box. Once these three parts are complete the user can now use the Custom Projectile in their simulation. Moreover, once the Simulator has access to the projectile the user created, it spawns a new custom projectile based on its shape. When spawned, physics are added via the `Physics Component` from FXGL in order to properly simulate collisions and other physics related properties. An alternative to this implementation would be to save all the fields inside a text file and modify values by reading and writing to the text field. Each object would be saved in a `txt`, `JSON`, database, or other format which would then be saved to the users computer. This approach offers the benefit of allowing users to directly access and modify the data stored in these external files. Unlike serialized data composed of byte streams, this stored data is human-readable and editable, facilitating easier user interaction. However, implementing this solution isn't as straightforward as serializing objects. There's

a risk of unintended modifications by users that could disrupt the file's intended structure. Serialized objects, adhering strictly to the implemented Java class structure, offer protection against accidental alterations to the file's structure.

Save and load: The implementation of the save and load feature was done with the help of Java's `Serializable` interface [4]. The `Level` class is the class responsible for saving data regarding all obstacles within a level, and does so via a reference to a `List` of `Obstacle`s. In FXGL, the only way to store data inside entities is by attaching components to them (components not only model behaviors, but also store data [1]). As such, the class `ObstacleComponent` is necessary to store a reference to `Obstacle` object with its data, which allows us to obtain the instance of the `Obstacle` object that was used to create the obstacle whenever the user tries to save the level. Whenever the user clicks on the `save` button in the `Simulator` screen, we loop over all of the entities of type `Obstacle`, get the entities' current position and angle, and set appropriate fields in the `Obstacle` (this is because the position and rotation of an obstacle is not necessarily the same as it was when the user first spawned in the obstacle). When the user tries to load a file containing a `Level`, the application attempts to deserialize it and load the `SimulatorController` alongside the deserialized `Level`. An alternative approach to this method would be to use external data storage mechanisms such as databases, `JSON` files, or other file formats to save and load the game state, rather than directly embedding serialized data into files. The advantage of this solution would be that users could directly edit and change the data contained inside the externally stored files. This data would be readable and understandable by humans, whereas serialized data contains streams of bytes which humans cannot directly manipulate. The con is this solution is that it is not as easily implementable as serializing the objects, and the user could accidentally modify the files in a way that it was not designed to. `Serializable` objects follow a the strict structure of the implemented Java class, therefore there are is no way the user could break the structure of the file by modifying it.

1.3 Discussion

1.3.1 Acquired skills

While working on this project many skills were developed. These skills not only consisted of programming but also in teamwork. As this project is impossible to do alone, it was key that we worked concurrently and effectively in a team environment. We each learned that being part of a team means direct and clear communication and strong planning which only became stronger as the project went on. Although we each learned a lot of team skills, there was also an abundant knowledge of programming concepts that were learned and improved upon. For instance, before commencing the project, most of us had limited experience with JavaFX and FXML. However, by working on this project we each greatly improved our skills in both those fields. Moreover, we took this opportunity to learn about saving and loading files to a user's computer using the `Serializable` interface, that had to be researched as none of us had any prior experience using it. Furthermore, we also took

on the challenge of implementing FXGL, something else none of us have ever worked with before. We were forced to learn to integrate FXGL with JavaFX, something that proved to be quite a challenge. Not only did implementing FXGL teach us about collision handling, input handling, and resource loading, but it also taught us to work under pressure, as we had to learn a completely new library in a limited amount of time. This is a skill that can not only be applied in programming but also in life, making this experience very important. It also reinforced our knowledge of mechanics (kinematics), as we had to research relationships between position, velocity, and acceleration.

1.3.2 Challenges

While working with FXGL, we encountered many troubles. The biggest problem was the highly limited learning resources. Finding comprehensive learning resources such as tutorials, books, and videos specifically focused on FXGL was extremely challenging, as FXGL is an extremely small and niche library (not very widely used) The FXGL library was written by a single person, so almost all of the information regarding its usage came from a single source. Figuring out how to implement the UI with FXML and integrating that with FXGL's game world proved to be a very difficult issue. The reason for this is because there are no scenes in FXGL, only a game world. To add user interfaces to an FXGL app, one most port the scenes graphs generated with FXML to FXGL's custom `UI` objects, and lay that over the game world. Learning in FXGL library was difficult and time-consuming because FXGL works very differently than JavaFX. It has a different structure from what we are used to (completely different from material learned in class). We also encountered challenges not related to programming in our project, the main one being communication between the members. A member wasn't active at all for a certain period of time and wasn't responding to the messages in our team's group discussion. There was also another member who had wasn't able to come to class for a period of time because of a medical operation. To ensure a smooth implementation of our project, we needed a lot of communication between the members and this lack of availability hindered our progress.

1.3.3 Individual discussions

- Ethan Tran: I was responsible for setting up the project and linking the JavaFX project with FXGL. I was mainly responsible for launching the projectile, implementing the logic for resetting the scene, saving and loading levels, and making sure the rigid-body dynamics worked well (this involved implementing a custom entity `Factory` to spawn entities and ensuring functionality with FXGL's `PhysicsComponent`s and entity-component system). I also helped out with bugs, refactoring, and improvements in other areas that popped up as we worked on the project. For example, I made improved the logic for spawning obstacles by making sure to spawn the entities with offsets such that they are centered at the mouse position, and I improved the logic for the kinematics graphs and implemented the velocity derivation change consideration

for the acceleration graphs.

- Mackenzie Rouchdy: In my part of the project, I was tasked with implementing the custom projectile and any features related to custom projectiles. This includes the implementation of the properties for the custom projectiles such as size, colour, and so forth. This also includes the saving and loading of the projectiles to and from the user's computer. Furthermore, I was also the primary member of the team to work on the implementation of the graphs as the logic behind the displacement and velocity graphs were added by me.

- Zachary Tremblay: My main tasks in the project were the implementation of everything relating to the drag-and-drop building system and the Parameters window for the simulator. The Parameters window touched on two other tasks, namely the Simulator and the graphs. I also contributed to some other parts like the implementation of the warning dialogue boxes and the visual appearance of the simulator. In all honesty, due to my medical situation, I fell behind schedule. Some of my teammates helped me in some part of the implementation, especially in understanding FXGL.

- Anton Lisunov: My part in the project were the implementation of setting and every feature related to it. My part includes creating of background and music with sound player. Also it includes of saving features that is separated from custom projectile and level saving. Sadly, It wasn't the most important part of the project.

1.4 Java/JavaFX Libraries

- FXGL [1]: Physics engine that helped simulate rigid-body dynamics and projectile motion

- FontAwesomeFX [2]: JavaFX library that imports icons that are used for buttons

1.5 Future work

In the future, we want to optimize and improve the physical behavior of objects, we also want to add more preset projectiles and levels, as well as the ability to add your own background and music. Moreover, the ability to add projectiles trails that can be seen as the projectile travels is also a visual implementation that we would like to implement in a future version of the application. Such trails would simple closely follow behind the projectile and would be chosen by the user. Furthermore, the addition of an angle label is also something that would be heavily considered in the future. It would essentially mimic a protractor, measuring the angle between the ground and the launcher. This would give the user more information about their projectile launch and how it compares to others. Another feature that could possibly be implemented is preset levels.

2 User Guide

Home screen: This is the page users are greeted with when they launch the application. There are 4 buttons which lead to different places and functions:

1. The `New` button loads the simulator screen with a fresh new level (no obstacles)
2. The `Load` button opens a file chooser and allows the user to load a previously saved level. Once the file containing the level has been chosen, the application will load the simulator screen with that level preset
3. The `Custom Projectile` button will open the custom projectile maker page, which allows users to create their own custom projectiles.
4. The `Settings` button will lead the user to the settings page, allowing users to change music, sound effects, background, etc.

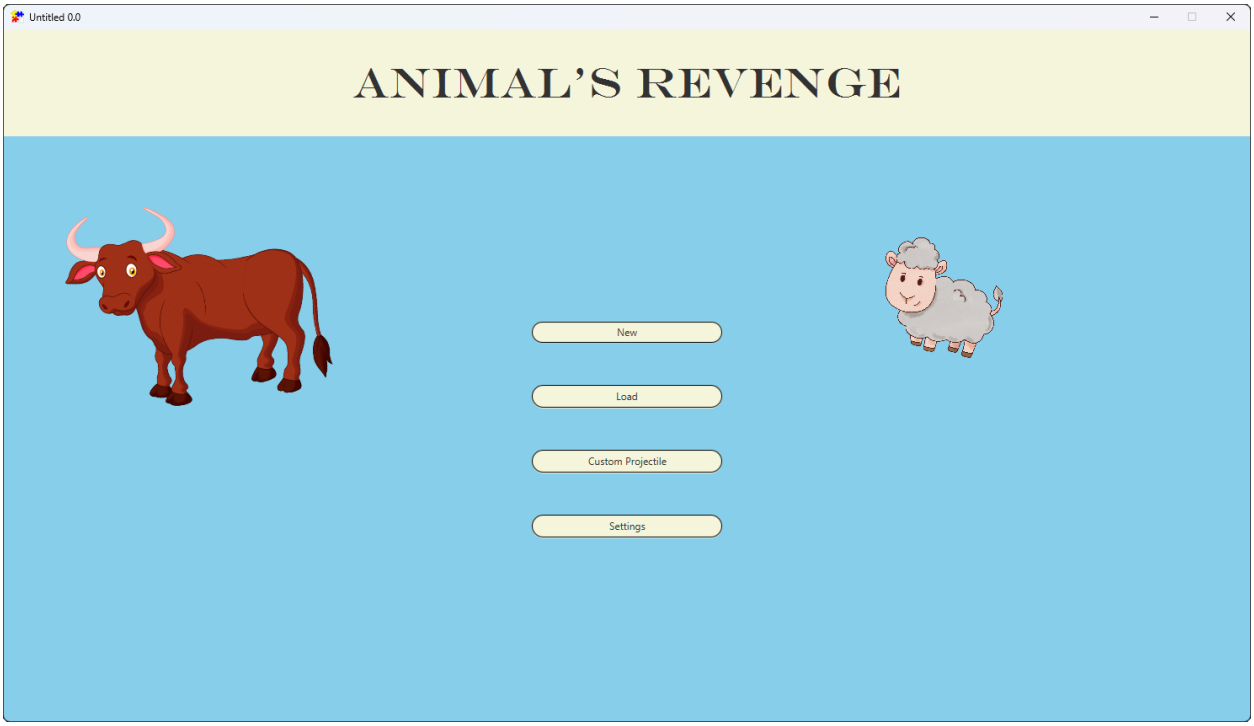


Figure 3: Home screen

Simulator screen: Users land on this page by either going on `New` or `Load` from the home screen. The 4 buttons in the top left are as follows (from left to right):

1. `Home`: Returns to the home page
2. `Launch`: Launches the projectile with the given velocity vector from the launcher. This also opens a popup with the kinematics graphs (if not disabled in parameters). See Figure 5 and Figure 6.
3. `Reset`: Removes all obstacles and projectiles from the layout, and reset the velocity vector to 0
4. `Save`: Saves the current layout of obstacles by opening a file chooser, allowing users to choose where they want to save the file.

In the top right there are 2 buttons: `Projectile` and `Parameters`. The former opens a popup that allows the user to choose custom projectiles (See 7a). The latter opens a popup with the parameters page, allowing users to change the value of gravity and disable the popup of the kinematics graphs when the user clicks on `Launch` (See 7b). The right-hand panel allows the user choose the shape and the type of the obstacle (this also changes density). The gray area around the obstacle indicates which shape and type has been selected. To select one, the user simply clicks on one. The second portion of the right-hand panel (after the obstacle selection) allows the user to change the size, the rotation, and the friction coefficient of the spawned obstacles. The user can do this by either interacting with the slider or by changing the value of the text field (slider & text fields are linked, changing one will change the other). Inputting values that are invalid or outside the accepted range of values will trigger appropriate warning messages (See Figure 9). The ticks under the slider (and warning messages) indicate the accepted range of values for a field. The value of these fields will affect the obstacles spawned when the user holds `LMB` + `CTRL`.

In main pane of the simulator screen, the user can do 4 things:

- Click right-click `RMB` on the launcher, hold then drag to manipulate the vector magnitude and direction. Release `RMB` to stop changing the velocity vector. This is the vector that will determine the launch angle and speed of the projectile when `Launch` button is clicked
- Click left-click `LMB` on an obstacle and hold to drag it around. Release `LMB` to let go of the selected obstacle
- Click left-click `LMB` and hold control button `CTRL` to create a new obstacle (with correct shape, type, size, rotation, friction). Hold `LMB` and `CTRL` to drag newly spawned object, and release to let go of the obstacles (See Figure 8)
- Hold backspace `BACKSPACE` and drag cursor around to delete any objects that the cursor touches

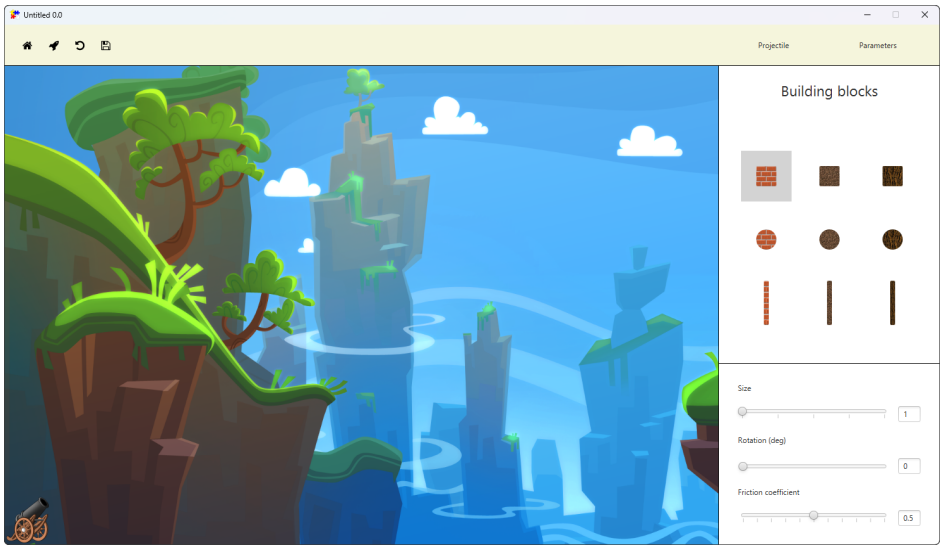


Figure 4: Simulator screen (new level, no obstacles)

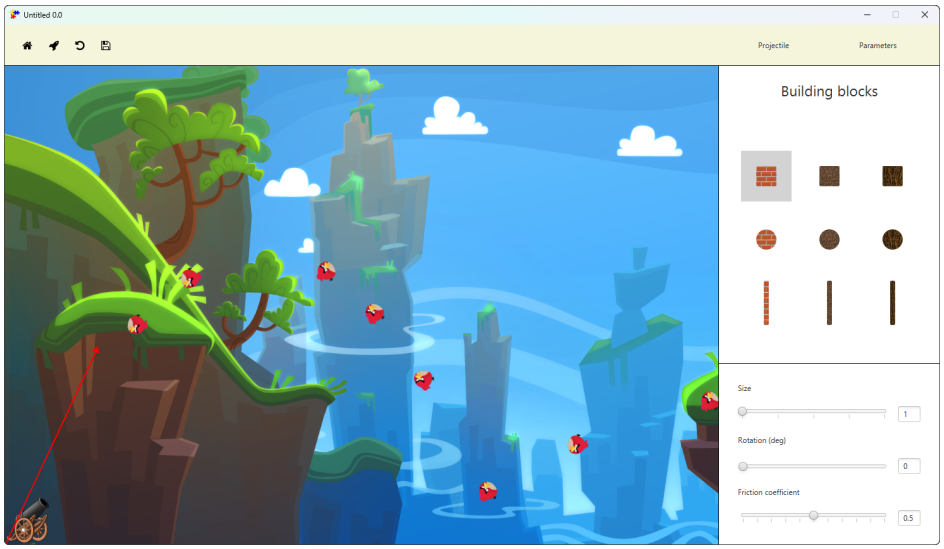
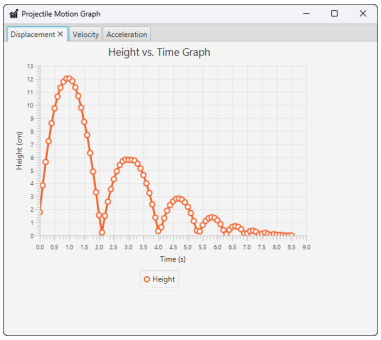
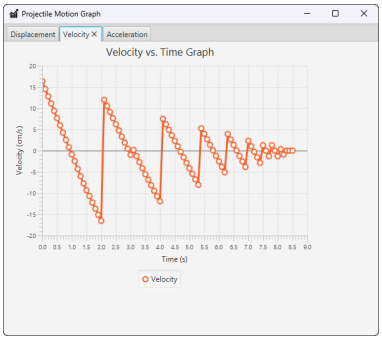


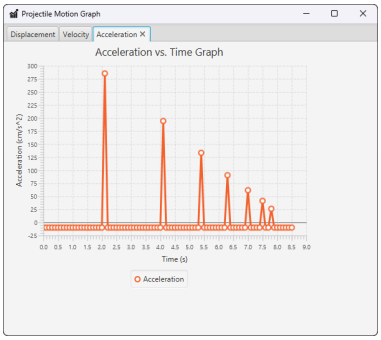
Figure 5: Launching multiple projectiles in the simulator screen



(a) Position y vs. time t



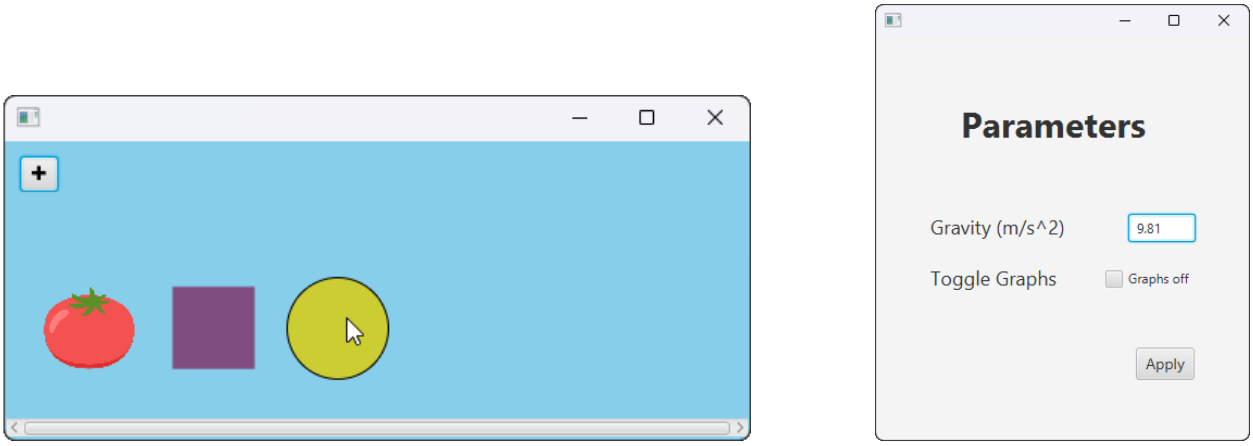
(b) Velocity v_y vs. time t



(c) Acceleration a_y vs. time t

Figure 6: Kinematics graphs

The projectile selection pop up allows the user to choose which projectile he would like to select. To select a projectile, simply click on the projectile hovered by the cursor. A black outline will indicate which projectile the user is currently hovering on. To add projectiles to the list of projectiles from local files, click on the button in the top left with the `+` icon. In the parameters page, the user can enter a number representing the value of gravity in m/s^2 that will be set, where a positive number indicates gravity acting downwards, and a negative number indicates gravity pulling objects upwards. There is a check box that can be toggled on/off to disable/enable the graphs popup. To save the parameters, the user must click on `Apply` before closing the window.



(a) Projectile selection popup **(b)** Parameters popup

Figure 7: Popups in the simulator page

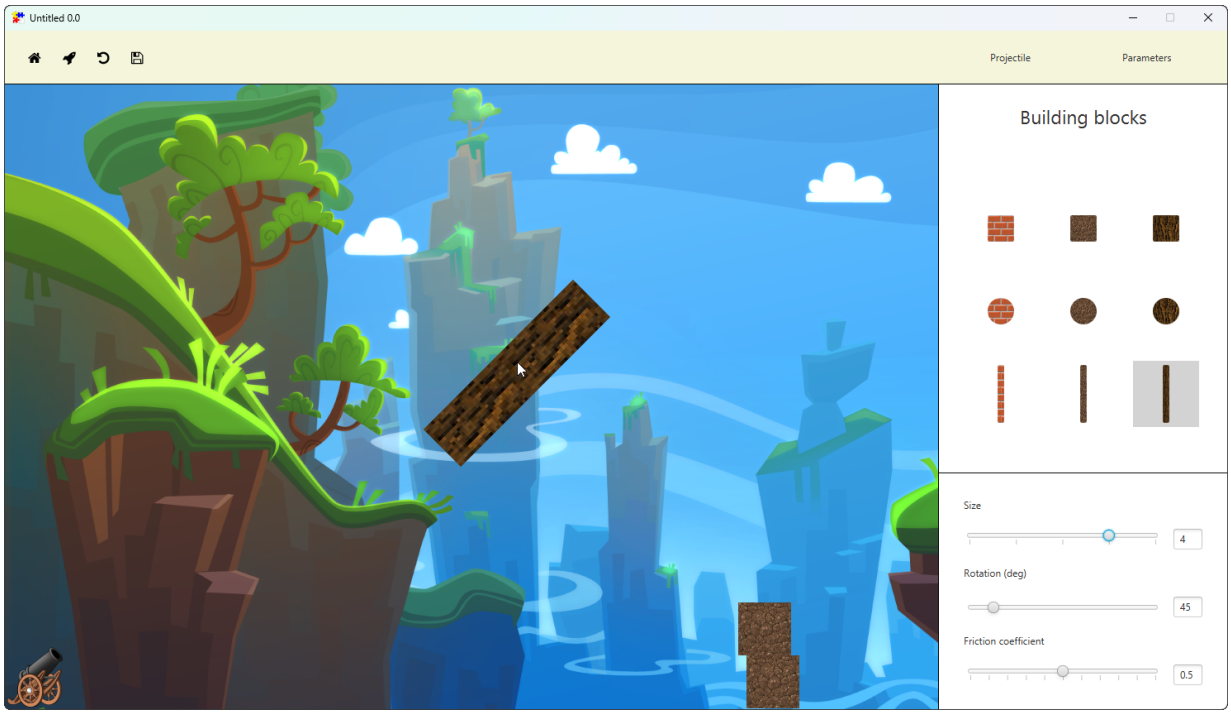
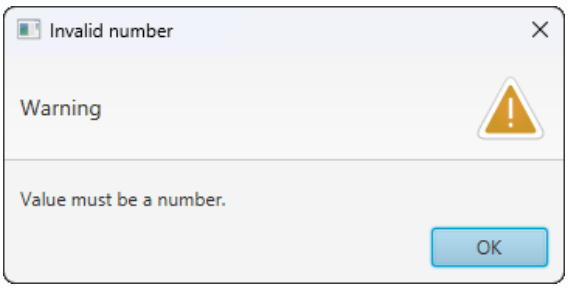
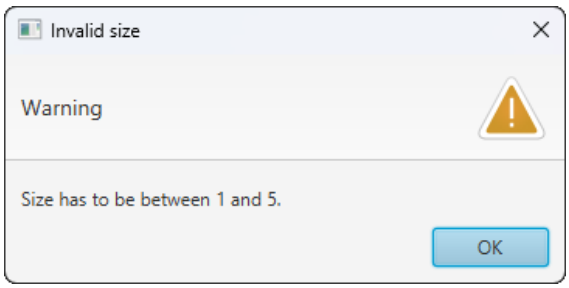


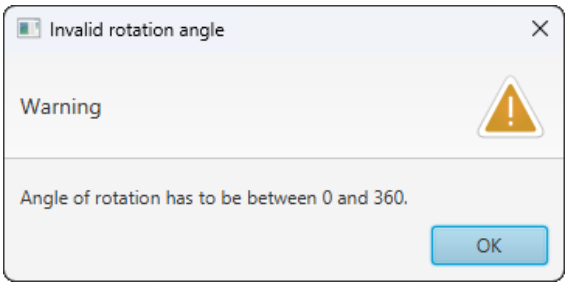
Figure 8: An example of a user creating an obstacle with size, rotation, friction, and type



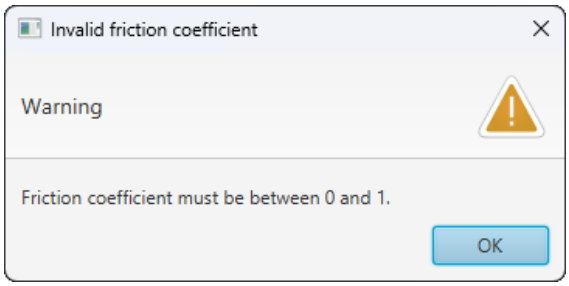
(a) Invalid number format exception



(b) Number out of range exception for size



(c) Number out of range exception for rotation



(d) Number out of range exception for friction

Figure 9: Warning message thrown in the simulator screen

Projectile creation screen: This is the page where users are given the ability to create their own projectiles. Users have the ability to change five properties pertaining to the projectile:

1. The `Shape` feature gives the choice of choosing between a square and a circle. Once a shape is chosen it will immediately appear in the center of the screen, waiting to be influenced by other parameters
2. The `Size` slider allows users to pick a specified size. The values have a minimum and a maximum such that every object has a reasonable size. The size always updates in real time, allowing users to see how big the size they choose appears.
3. The `Colour` picker allows users to open up a popup containing many preset colours, it also allows users to create a custom colour with RGB or colour codes. Similarly to the size slider, once a colour is chosen it updates in real time.
4. The `Restitution` Slider allows user to change the restitution of the projectile.
5. The `Density` Slider allows user to change the density of the projectile.
6. The `Image` Button allows users to pick an image from their computer and paste it onto their projectile, upon clicking.
7. The `Save` Button allows users to save all of their changes into a Custom Projectile file object and a path of their choice.

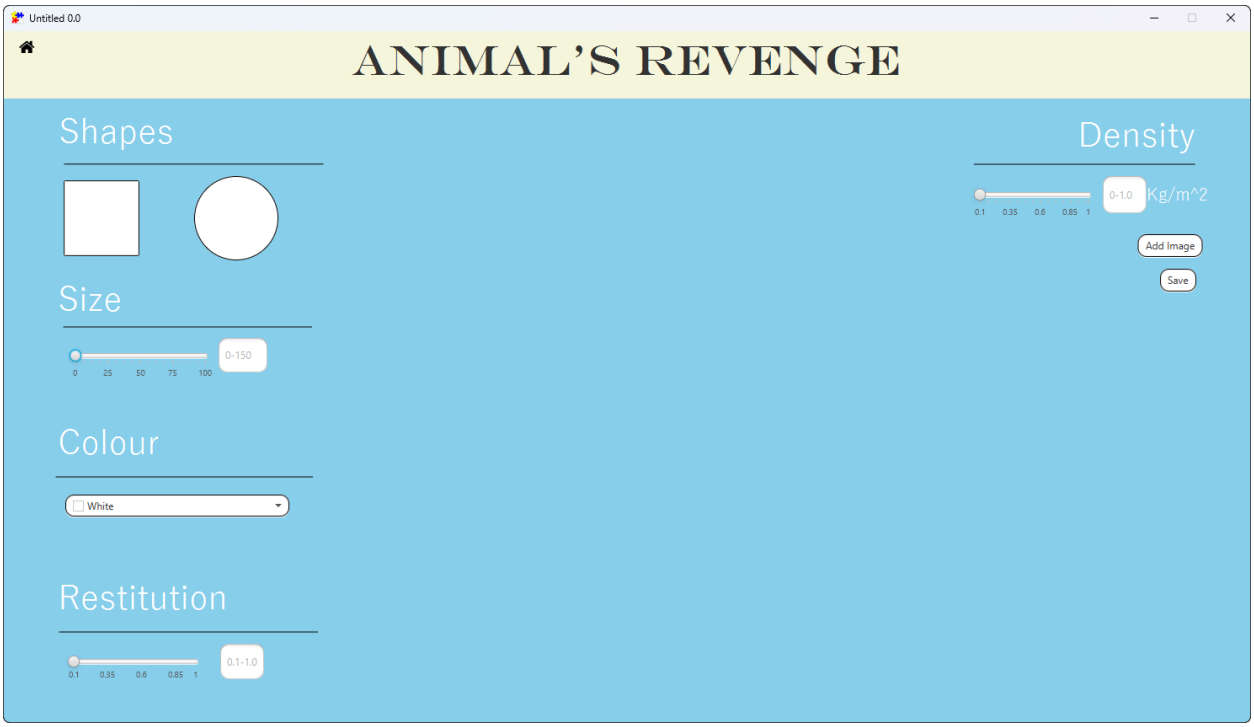
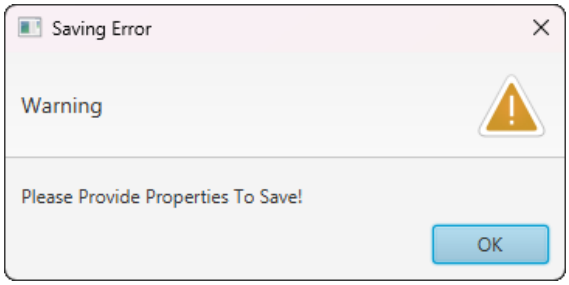


Figure 10: Projectile selection screen

(a) Invalid number format exception



(b) No valid properties have been selected

Figure 11: Warning message thrown in the Custom Projectile screen

Settings: This page gives users the ability to change sound, music and background. Users can change five properties of settings and they are saved automatically, even if user exit the program (See Figure 12):

1. The `Background type` picker gives the choice of choosing between a few background that are initially added in project.
2. The `Background color` picker allows users to pick a specified color of background.
3. The `Sound volume` slider allows users change sound volume in real time.
4. The `Music type` picker let user to pick any kind of music that already exist in the project.

5. The `Music Volume` slider allows user to change volume of music.
6. The `Back` Button allows users return to home screen.

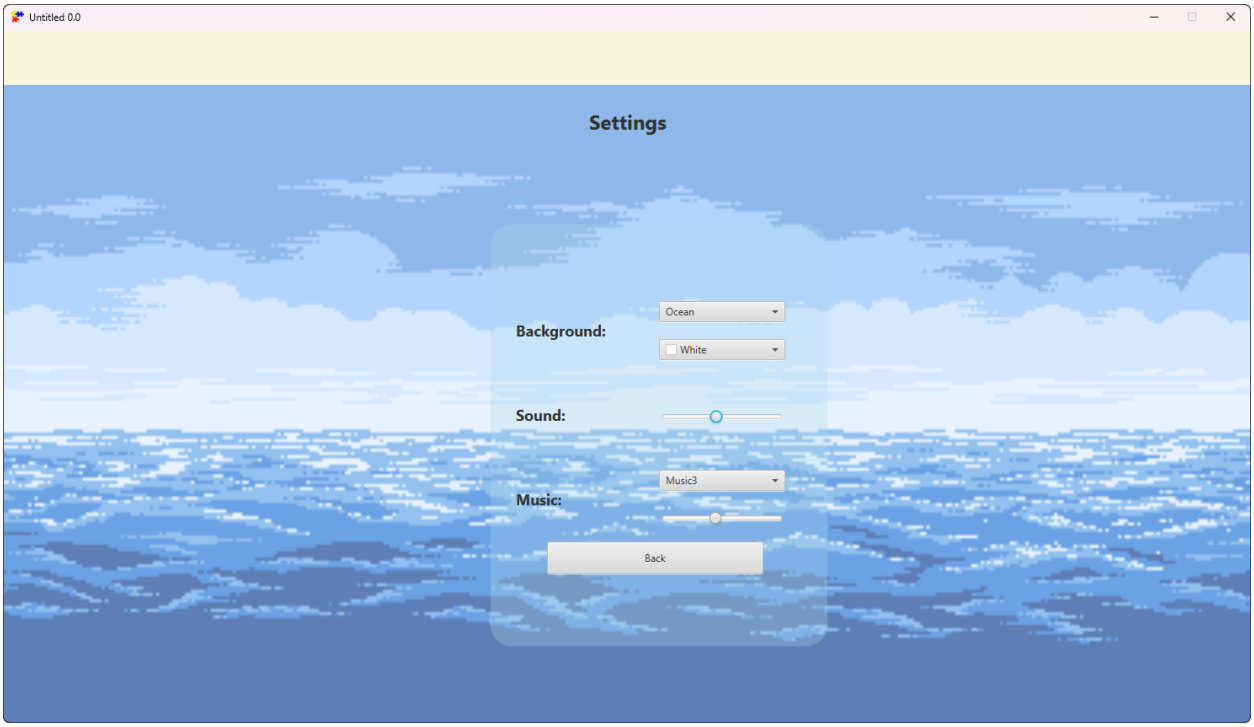
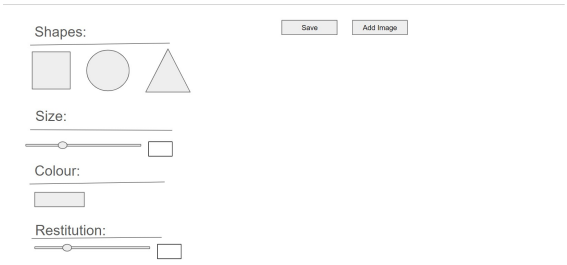


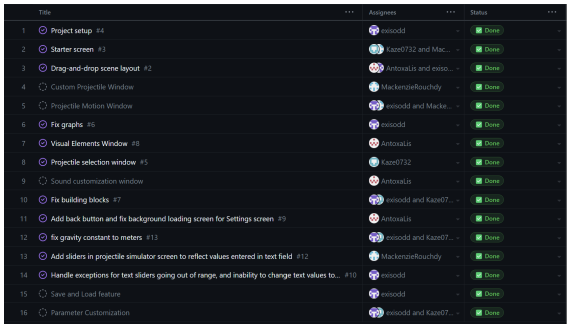
Figure 12: Setting screen

3 Appendix

3.1 Notes



(a) Initial wireframe for projectile screen



(b) Team backlog in GitHub

Figure 13: Notes

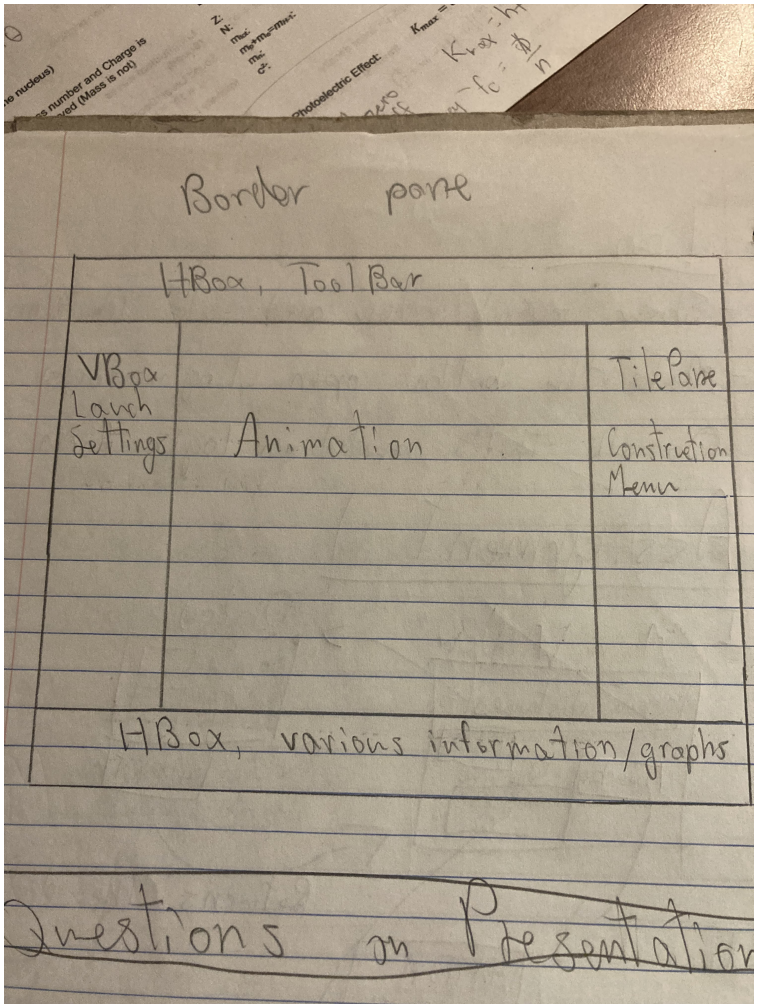
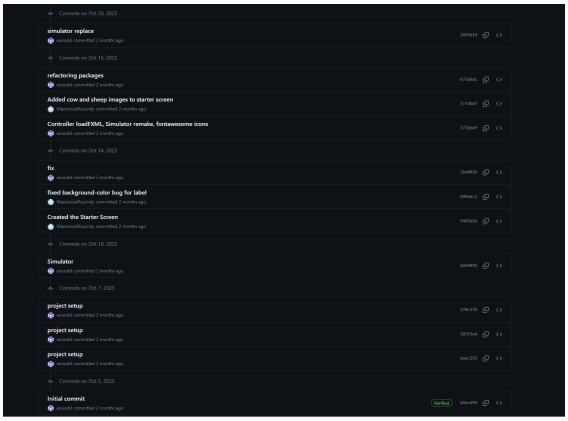


Figure 14: Initial wireframe for the simulator screen

3.2 GitHub commits



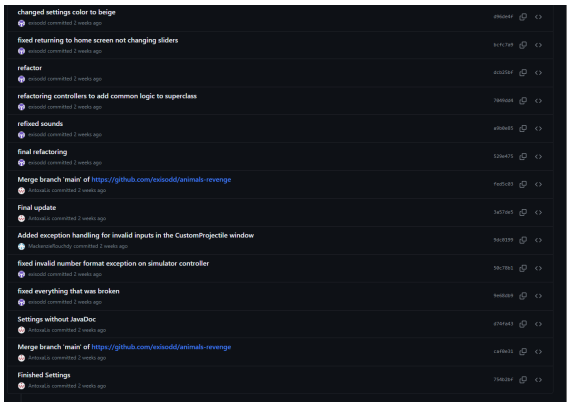


Figure 15: GitHub commits

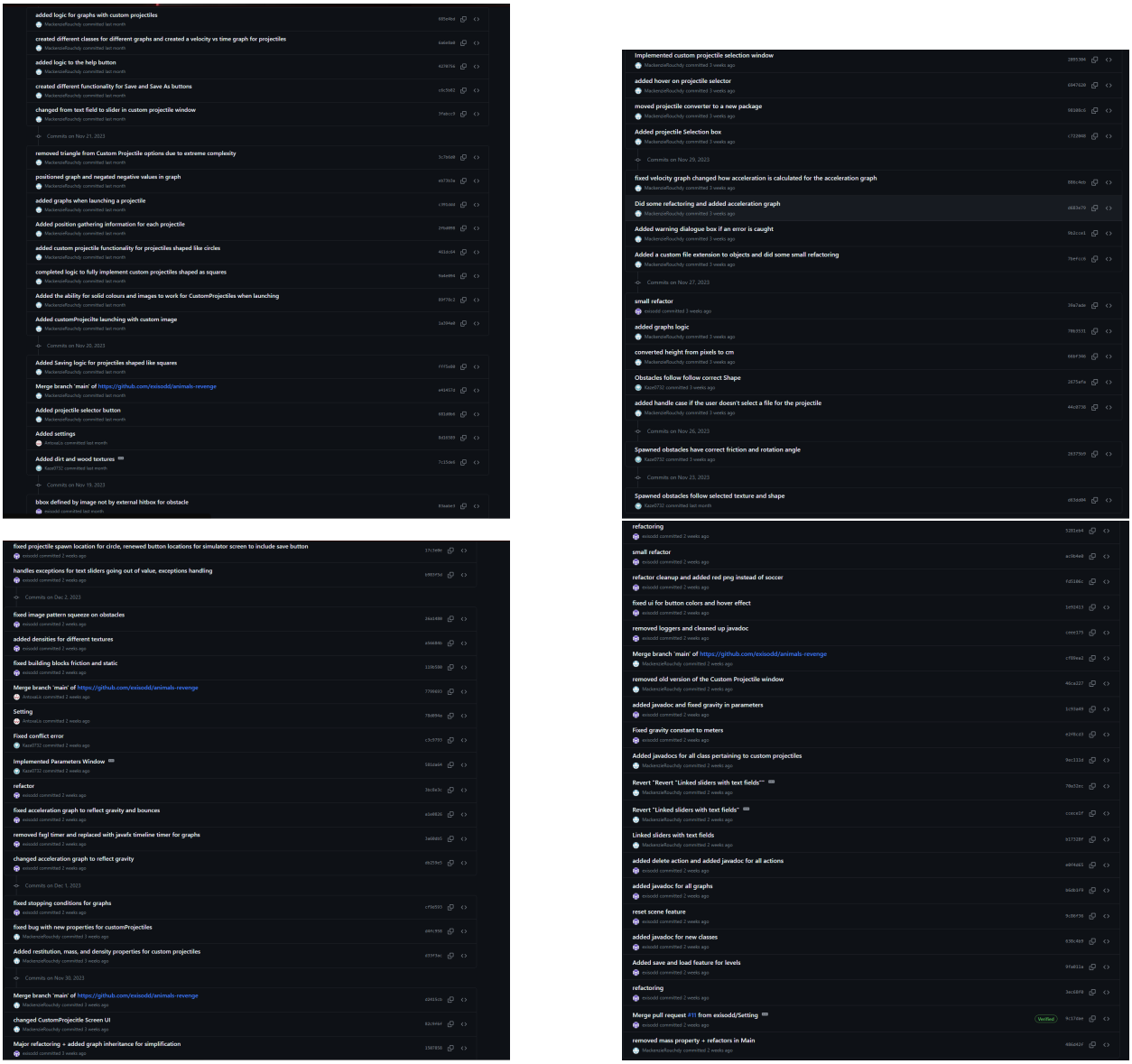


Figure 16: GitHub commits

3.3 References

[1] Almas Baimagambetov. *Learn JavaFX Game and app development: with FXGL 17*. Apress, 2022.

[2] Jens Deters. *FontAwesomeFX*. https://mvnrepository.com/artifact/de.jensd. [Online; accessed 6-November-2023]. 2013.

[3] William Moebs, Samuel J. Ling, and Jeff Sanny. "Acceleration vector". In: *University Physics Volume 1*. Houston, Texas: OpenStax, 2016. URL: https://openstax.org/books/university-physics-volume-1/pages/4-2-acceleration-vector.

[4] Oracle Corporation. *Serializable Interface (Java Platform SE 8 API Specification)*. [Online; accessed 11-December-2023]. 2014. URL: https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html.

[5] Wikipedia. *Pythagorean theorem*. https://en.wikipedia.org/wiki/Pythagorean_theorem. [Online; accessed 9-December-2023]. 2023.

[6] Wikipedia. *Trigonometric functions*. https://en.wikipedia.org/wiki/Trigonometric_functions. [Online; accessed 9-December-2023]. 2023.