Integrative Project in Computer Science and Mathematics

420-204-RE Sect. 00001

Neon Control

By Jakob Thibodeau, Joshua Morency, Arnaud Davignon

15/02/2021

Index

1. Introduction

1.1. History

For this integrative project, we wanted to make a game that would display accurate physical principles and concepts in a way that would be fun and easy to understand. We had many ideas but came across an existing project that fit our objectives pretty well, a tricky game called C*leaning the system*. However, this game's goal was to be frustrating and hard, so we decided to modify our intents towards this project so it would better match a scientific and educational. Here is a brief overview of what our project is all about.

1.2. Summary of the project and game rules

Neon Control is a physics-based platformer game, which aims at helping anyone better understand the concepts of springs, collisions, gravity, and momentum. The player makes progress through a level by rotating around its axis a stick equipped with springs at both its ends. When a spring makes contact with a wall, floor, or ceiling, it causes the stick to bounce back. This creates opportunities to make the stick travel at whatever angle the player desires. The catch is these collisions are the only way to move and make progress in the game. Therefore, by angling the stick, the player must navigate rooms full of walls and obstacles to complete each level. This will prove to be a tough task, as the stick can sometimes bounce back in unexpected ways. The physics engine will use real-life physics equations to calculate movement and collisions and a HUD will explain, at all times, the force experienced by the stick while moving.

1.3. Objectives

This project, as mentioned earlier, aims at helping the player gain a better understanding of some key concepts in the field of physics, such as how springs being compressed can cause objects to bounce back, how gravity affects a body flying through the air, and how momentum affects collisions between two solid bodies (in this case, the stick itself and any other static element). The more the player participates in the game, the better their understanding of physics will be, making them even better at the game. Another key objective is to make the game fun to play. In order to keep a player engaged and wanting to learn more in this physics-driven game, its controls, premise and actual gameplay must be enticing and fun to experience.

1.4. Importance of the project

The importance of this project comes from its demonstration of physics concepts in a fun environment. To anyone who is interested in physics or who wants to pursue it at a college level, this project is an example of what can be done when applying those concepts. Neon Control will demonstrate concepts of conservation of energy, gravity, acceleration, collisions and forces. Learning these concepts in the context of a game could be the catalyst to an engineering career, or can at least spark the interest of the player. Learning a new subject in a classroom can be tedious, and even boring. However, when discovering the same concept in an different environment, such as that of a game, a player is more likely to find a genuine interest in it, generating a thirst for more of this new knowledge.

1.5. What will we learn?

This project is the perfect opportunity for us to apply concepts learned in our previous programming classes, such as threads, the javafx library, objects and classes, as well as notions learned in our previous physics classes, such as vectors, forces, elasticity and acceleration. We will also have to learn new skills in order to produce a fully functioning game. We will learn how to implement hitboxes to objects, how to detect collisions between objects, how to update the properties of an object at a set interval. We will also need to learn about Git and GitHub, since those platforms will be used to share the work done between all three us.

2. Design.

2.1. Problem

The game employs the use of physics – specifically mechanical motion – in order to determine the motion of the player character – the stick with springs at both ends. There are two forms of motion that must be considered: the motion which is a result of one of the springs colliding with a wall, and the motion which is a result of the side of the stick colliding with the wall. Either way, the total velocity must be split into its vertical and horizontal components.

2.2. Physics

2.2.1 Energy in the physics engine

The system that is our game is an ideal system; not only do laws like the conservation of energy apply, but friction is being ignored, both regarding contact points and air. Because of this, unless new energy is added to the system, then the total velocity of the stick is a fixed value. This is because of the law of conservation of energy. In essence, since the game is a closed system, any energy that is initially given to the stick will be conserved throughout the level, constantly going from kinetic energy – when in motion – to potential energy – when the stick slows due to gravity. Because of this, for any collision the total magnitude of the velocity vector will remain the same pre- and post-collision – only the direction of the sticks motion will change. A vector is a unit with a direction, so for this case a velocity is speed/motion but with a direction. It is often split into horizontal (x) and vertical (y) components.

2.2.2 Collisions with the springs

There are two ways to deal with the spring end of the stick colliding with the wall. It can be seen like a ball bouncing across. This means that if the stick is flying horizontally but gaining vertical velocity downwards due to gravity and is pointing straight up, if it collides with the ground then it would bounce upwards due to the vertical component of its velocity but continue sideways since the horizonal component was not affected. The other method – which will be employed – is by thinking of the stick like a double-ended pogo stick. In the scenario above, instead of the stick skimming across the ground, all the velocity – even the horizontal component – would be converted into upwards velocity – or in more generally, into a new velocity vector in the direction that the stick is pointing in. At the very least, the horizontal velocity would be stopped. It is as if the pogo stick landed and stopped, then pushed off again. Obviously, it does not actually stop, since that would not provide enough challenge to the player. This nullification of certain velocities provides the player with a modicum of control in a game where control is a luxury resource, while also simplifying code and math since the angle of the ground will have no effect on the direction the stick flies off in. In order to calculate the components of the post-collision vector, the orientation of the stick is needed. One possible formula for the new velocity is:

$$V_{new\ x} = \sqrt{V_{old\ x}^2 + V_{old\ y}^2}\cos\theta$$

$$V_{new\ y} = \sqrt{V_{old\ x}^2 + V_{old\ y}^2}\sin\theta$$

In these equations, θ is the angle representing the orientation of the stick. In essence, the formulae take the magnitude of the old vector – which is $\sqrt{V_{old\ x}^2 + V_{old\ y}^2}$ and multiply it by either the sine or cosine of the orientation angle. Obviously, these new components may not necessarily be pointing in the actual correct direction, so actions will need to be taken post calculation to change the sign of the new components to the correct one.

2.2.3 Collisions with the stick

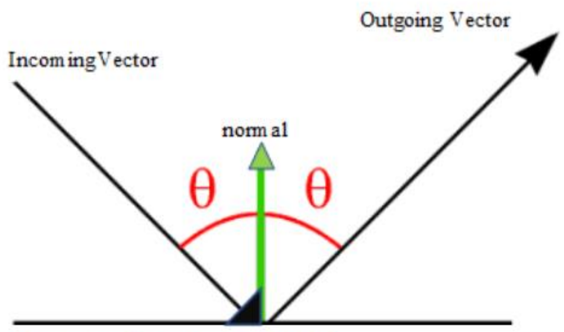


Diagram of a vector reflection[1]

When the side of the stick collides with a wall, it obviously will not bounce off the same way as if one of the ends with a spring collided with the wall. The physics behind this sort of collision is more like a reflection. Think about the effect of shining a laser into a mirror or a ball bouncing across the ground. In practice, when the stick collides, the angle between the stick and the normal vector will be the same as the angle between the stick post-the same normal vector. The normal vector quantity having a magnitude of 1 and pointing perpendicular to the surface of the object that it is on. So, a normal vector on a flat horizontal wall will be pointing straight up. The normal vector that must be considered is taken at the point of collision – although since we will be using only straight walls, all the vectors along the same face of the wall are the same. The relationship between the pre-collision vector of the stick and the post-collision one is:

$$V_{new} = V_{old} - 2(V_{old} \cdot n)n$$

Where n is the normal vector and V is the velocity vector of the stick, both in vector form. $(V_{old} \cdot n)$ is the dot product between the normal and velocity vectors, which results in a scalar quantity – a number that does not have a direction, a non-vector quantity – which is multiplied by 2 and by the normal vector. When multiplying a vector by a constant, both components are multiplied by said constant. This new value is subtracted from the pre-collision velocity – vector summation is done by adding/subtracting the individual components together. i.e., the *x* components get added together and the *y* components are added together.

2.2.4 Gravity

Meanwhile this entire time the stick is under the influence of gravity. This will give the stick a parabolic trajectory when in the air. Mathematically, this means that every time the program updates the velocity vector, a predetermined amount of vertical velocity will be subtracted from the *y* component of the velocity vector; however, in practice it would be added since Java/Netbeans considers moving downwards the positive *y* direction.

2.3. Algorithm

The algorithm developed to solve the scientific problem will, in this project, be used to update the location of the player character (the Spring Stick) in real time in a 2D panel (the level). The algorithm can be described using the following Pseudocode:

```
While(!levelComplete){
        If(collisonWithSpring){
                formula1(getVelocity(), getStickAngle(), getNormalVector());
                lauchSpringAnimation();
        }
        Else If(collisionWithStick)
                formula2(getVelocity(), getStickAngle(), getNormalAngle());
        Else
                formula3(getVelocity(), getStickAngle()
        setPos(newPosition);
        updateVelocityVector();
        checkWin();
}
```

The algorithm above explains the steps in which the program will calculate the trajectory of the moving Spring Stick. First of all, the program is set on a loop that will continuously run until the level is completed. The program will check if the level is completed at the end of each iteration of the loop.

Then, the program checks whether the Spring Stick collides one of its springs with a wall, and if so, calculates a new velocity vector to apply to the Spring Stick using formula 1. The program will also play a small 7 frame animation to show the spring bouncing and allow the player to react (see GUI section). Then, if no collision with the springs has been detected, the program checks if there is a collision between a wall and the side of the spring. If so, the calculation for the new velocity vector is made using formula 2. It the case where there is no collision, the program simply adds a predetermined double value to the y-component of the velocity vector to act as the force of gravity. This predetermined value is yet to be calculated, as it requires some testing based on runtime, realism and fun.

After updating the velocity vector of the Stick Spring, the program updates its position by applying the velocity vector to the current position of the Stick Spring as to simulate fluid movement as close as possible.

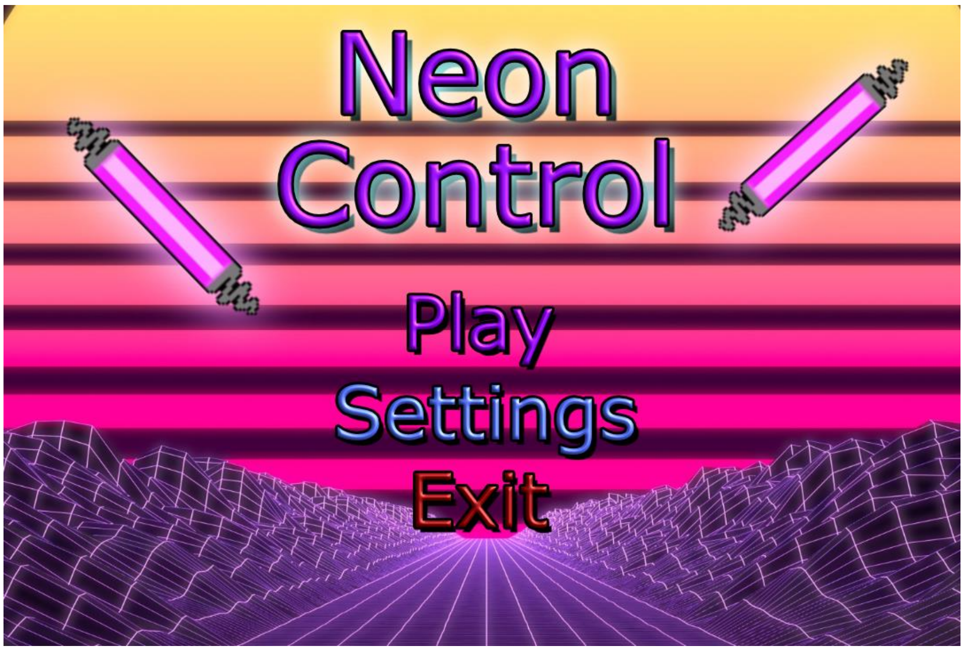
The efficiency of these algorithms, for the most part, are *O(n)*. This is because the algorithms do not need to sort, search, or generally do a lot of comparisons. The only comparison it must do is compare the position of the stick's hitbox with the hitboxes of every wall object on the level, until it found one that the stick has collided with or has checked every wall, but even then the program would only have an efficiency of *O(n)*, but only for this one task.

2.4. GUI

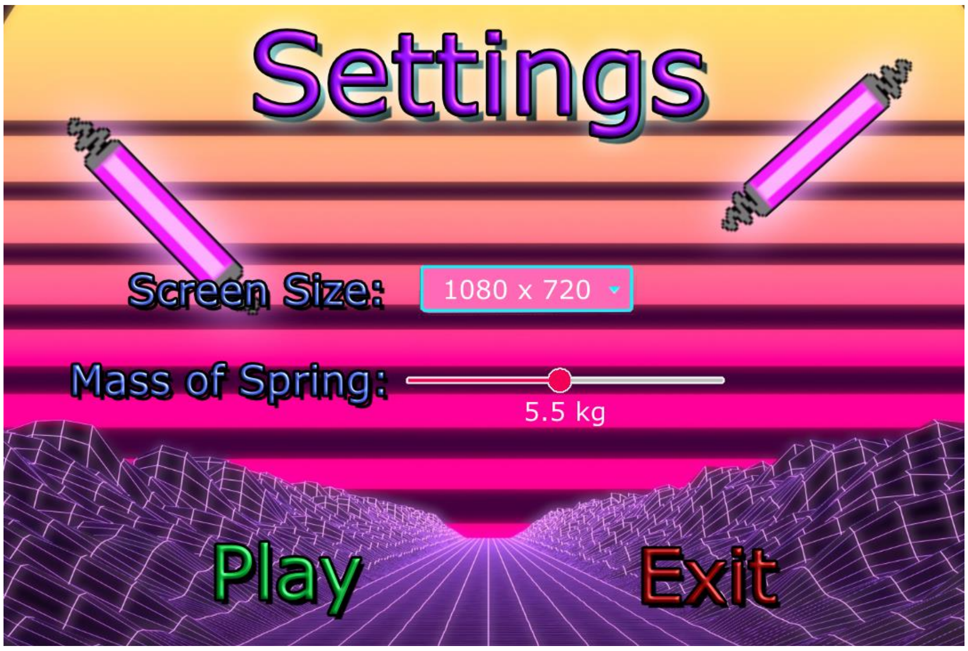
The GUI had to be designed for four major components: The main menu, the settings page the player character and the level graphic design. All these are made using a combination of custom designed graphics (created with Affinity Photo) and pictures found on the internet.

The esthetic theme chosen is “neon”. We found that this esthetic style is quite pleasing to the eyes and was not too hard to design.

2.4.1. Main Menu



2.4.2. Settings Menu

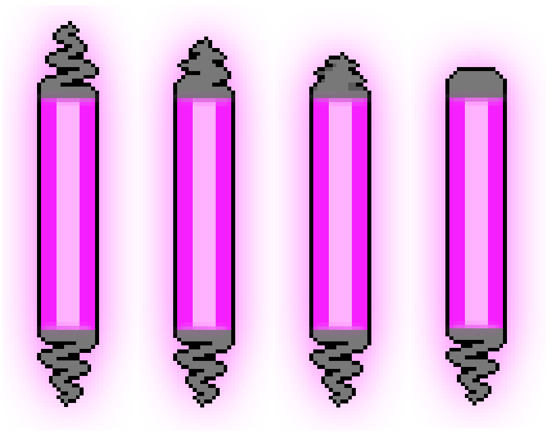


Screen Size: 1080 x 720

Mass of Spring: 5.5 kg

As the main component of the project is the physics, collisions and gameplay of the program, the options displayed here (Screen Size and Mass of Spring) are optional features. These features (and other features see section 3.3) may or may not be included in the final project.

2.4.3. Player character sprites

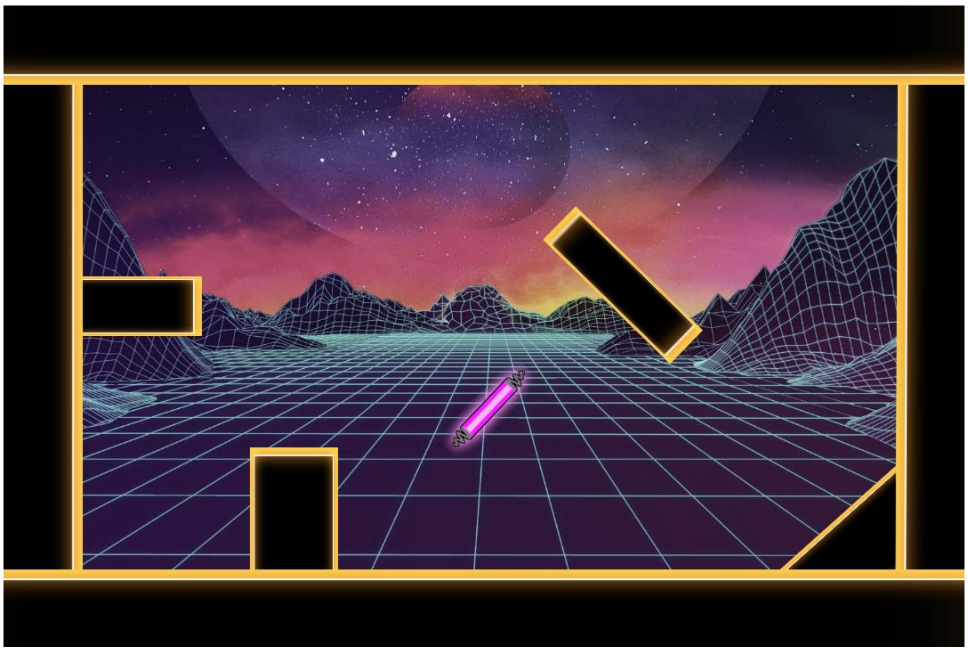


The first image on the left will be the main sprite used when the program is running. The other 3 sprites are used when one of the 2 springs collide with a wall to create a small 5 frames animation.

2.4.4. Wall sprite

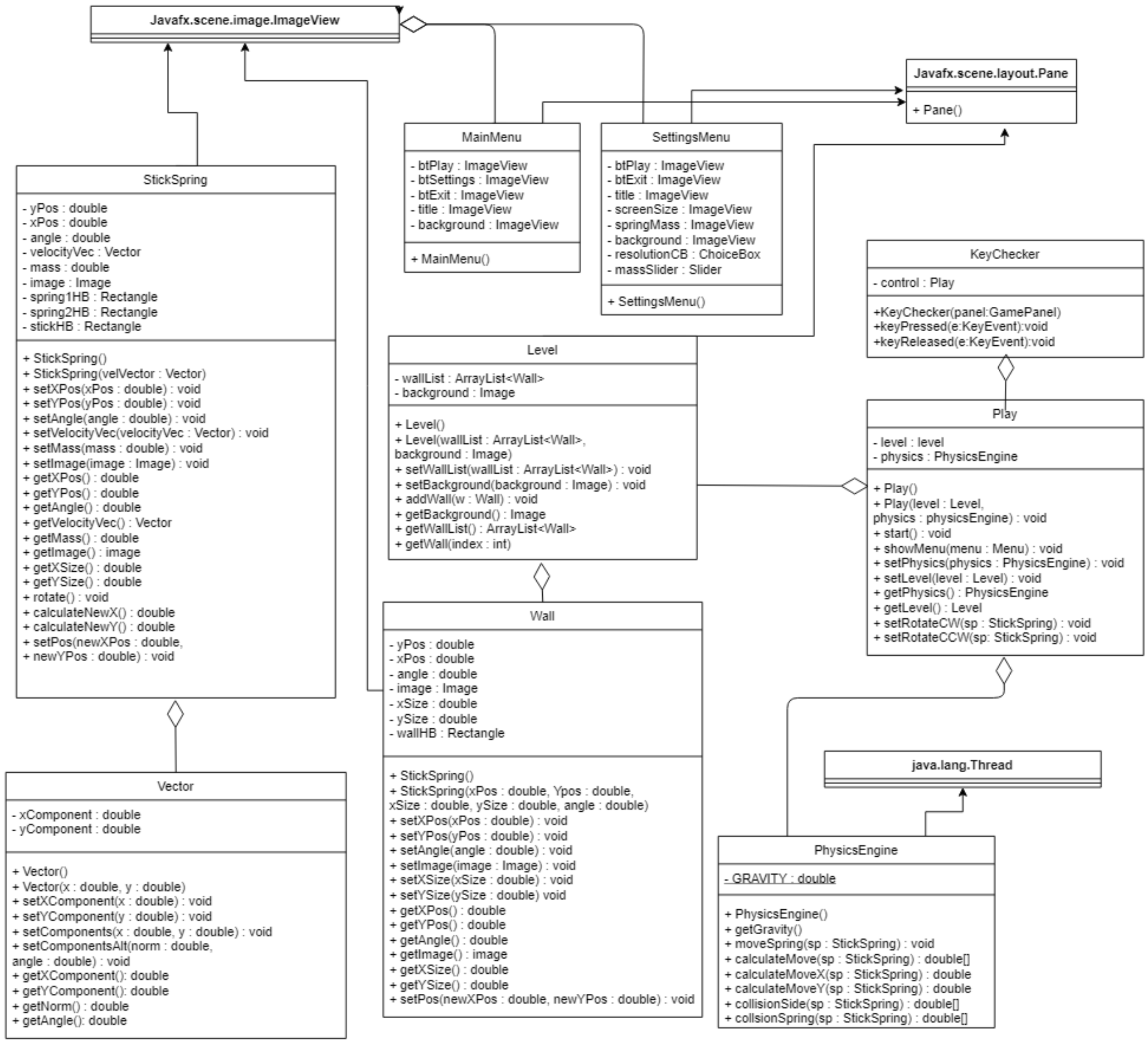
The wall displayed here is just a square, however, the image will be stretched, and the aspect ratio will change depending on the dimensions of the walls put in a level.

2.4.5 In game Level example



This layout is an example of what a level could be. The position and angle of the walls are subject to change. The spring is displayed here at an arbitrary position.

2.5. UML Diagram


Javafx.scene.image.ImageView
Javafx.scene.layout.Pane
+ Pane()
MainMenu
- btPlay : ImageView
- btSettings : ImageView
- btExit : ImageView
- title : ImageView
- background : ImageView
+ MainMenu()
SettingsMenu
- btPlay : ImageView
- btExit : ImageView
- title : ImageView
- screenSize : ImageView
- springMass : ImageView
- background : ImageView
- resolutionCB : ChoiceBox
- massSlider : Slider
+ SettingsMenu()
StickSpring
- yPos : double
- xPos : double
- angle : double
- velocityVec : Vector
- mass : double
- image : Image
- spring1HB : Rectangle
- spring2HB : Rectangle
- stickHB : Rectangle
+ StickSpring()
+ StickSpring(velVector : Vector)
+ setXPos(xPos : double) : void
+ setYPos(yPos : double) : void
+ setAngle(angle : double) : void
+ setVelocityVec(velocityVec : Vector) : void
+ setMass(mass : double) : void
+ setImage(image : Image) : void
+ getXPos() : double
+ getYPos() : double
+ getAngle() : double
+ getVelocityVec() : Vector
+ getMass() : double
+ getImage() : image
+ getXSize() : double
+ getYSize() : double
+ rotate() : void
+ calculateNewX() : double
+ calculateNewY() : double
+ setPos(newXPos : double,
+ newYPos : double) : void
KeyChecker
- control : Play
+KeyChecker(panel:GamePanel)
+keyPressed(e:KeyEvent):void
+keyReleased(e:KeyEvent):void
Level
- wallList : ArrayList<Wall>
- background : Image
+ Level()
+ Level(wallList : ArrayList<Wall>,
background : Image)
+ setWallList(wallList : ArrayList<Wall>) : void
+ setBackground(background : Image) : void
+ addWall(w : Wall) : void
+ getBackground() : Image
+ getWallList() : ArrayList<Wall>
+ getWall(index : int)
Play
- level : level
- physics : PhysicsEngine
+ Play()
+ Play(level : Level,
physics : physicsEngine) : void
+ start() : void
+ showMenu(menu : Menu) : void
+ setPhysics(physics : PhysicsEngine) : void
+ setLevel(level : Level) : void
+ getPhysics() : PhysicsEngine
+ getLevel() : Level
+ setRotateCW(sp : StickSpring) : void
+ setRotateCCW(sp: StickSpring) : void
Wall
- yPos : double
- xPos : double
- angle : double
- image : Image
- xSize : double
- ySize : double
- wallHB : Rectangle
+ StickSpring()
+ StickSpring(xPos : double, Ypos : double,
xSize : double, ySize : double, angle : double)
+ setXPos(xPos : double) : void
+ setYPos(yPos : double) : void
+ setAngle(angle : double) : void
+ setImage(image : Image) : void
+ setXSize(xSize : double) : void
+ setYSize(ySize : double) void
+ getXPos() : double
+ getYPos() : double
+ getAngle() : double
+ getImage() : image
+ getXSize() : double
+ getYSize() : double
+ setPos(newXPos : double, newYPos : double) : void
Vector
- xComponent : double
- yComponent : double
+ Vector()
+ Vector(x : double, y : double)
+ setXComponent(x : double) : void
+ setYComponent(y : double) : void
+ setComponents(x : double, y : double) : void
+ setComponentsAlt(norm : double,
angle : double) : void
+ getXComponent(): double
+ getYComponent(): double
+ getNorm() : double
+ getAngle(): double
java.lang.Thread
PhysicsEngine
- GRAVITY : double
+ PhysicsEngine()
+ getGravity()
+ moveSpring(sp : StickSpring) : void
+ calculateMove(sp : StickSpring) : double[]
+ calculateMoveX(sp : StickSpring) : double
+ calculateMoveY(sp : StickSpring) : double
+ collisionSide(sp : StickSpring) : double[]
+ collsionSpring(sp : StickSpring) : double[]

2.5.1. Vector class

The Vector class creates a vector object – an object with an *x* and *y* attribute. It is used to perform vector arithmetic. It contains constructors to either create the object using two double values as the *x* and *y* values, or to intake an angle and magnitude as double values. The class has setters and getters for both attributes. It also has a method to add two vectors together - which takes a vector as a parameter and outputs a new vector – and to perform the dot product of two vectors – which also takes a vector as input and outputs a new vector.

2.5.2. StickSpring class

The StickSpring class, which extends from javafx.scene.image.ImageView, will be used to create the object controlled by the player, the SpringStick. The StickSpring has a position (x,y) an angle (in degrees) a velocity vector, an image (see section 2.4.3), a mass, and 3 hitboxes (one for each spring and one for the stick), all of them instances of the javafx.scene.shape.Rectangle class. When moving the stickpring, there are actually 4 components which change positions: the StickSpring object itself and its 3 hitboxes. The StickSpring is really only a projection of the graphics while the hitboxes will be used to detect collisions with walls. The angle will be controlled by the player using the rotate() method, which will be called in the KeyChecker and Play class (see sections 2.5.6. and 2.5.7.). The velocity vector will be used to calculate the movement of the StickSpring and can only be changed by gravity and collisions (as it would in a perfect system). The mass will be used to calculate the effect of gravity.

2.5.3. Wall class

The Wall class is used to create the walls, ceiling, and floors that make up each level in the game. It contains constructors either to create a default wall object with no parameters, as well as one with defined angle, x and y position, and x and y size. The next few methods are getters and setters for these five parameters. The class also has a method to set and one that returns the image that is displayed inside the wall. Finally, the Wall class has a method, called setPosition that takes two parameters and is used to set both the x position and y position at once.

2.5.4. Level class

The Level class in used to group all the walls needed to create the layout of a level (example, see section 2.4.5.). A Level has an array of walls, which it displays on screen, and a background image, also displayed. Setters and getters for all walls and the background image are implemented.

2.5.5 PhysicsEngine class

The PhysicsEngine Class is the class used to calculate all movements and collisions of the SpringStick. It extends the Thread class. It will also serve to actually move the SpringStick when the program is running, using the moveSpring method. The calculateMove method will be used when there is no collision detected. It will find the modify the velocity vector of the SpringStick by a set GRAVITY constant. The GRAVITY constant will be set during playtesting. Then the new position on the SpringStick (based on the current position and the updated velocity vector) will be calculated and returned. The collisionSide method is used when a collision between the side of the StickSpring and a wall is detected. The calculation will be implemented following the physics formulas described in section 2.2.3. The collisionSpring method is used when a collsion is detected between one of the springs of the SpringStick and a wall. The calculation will be implemented following physics formulas described in section 2.2.2. Neither of the collisionSide or collisionSpring methods utilise the GRAVITY constant. In brief, the PhysicsEngine class defines the laws of physics used in the game.

2.5.6. Play class

The Play class extends is used to run the program. The Play class has 2 variables, a Level and a PhysicsEngine. The Play class will display the Main Menu, Settings Menu and the Level when prompted. It will use to calculations and methods of the PhysicsEngine class to move the SpringStick and update the game state in real-time.

2.5.7. KeyChecker class

The KeyChecker class' main purpose is to check to see if a key has been clicked. It extends the KeyAdapter class and overrides its methods to check if a key has been pressed and to check if a key has been released. It's needed as an intermediary between our Play class and

KeyAdapter, which is why the only attribute is an instance of the Play class. The pressed keys will permit the rotation of the SpringStick player character.

2.5.8 MainMenu and SettingsMenu classes

The MainMenu class and SettingsMenu class are used simply to generate the GUI and controls for both menus.

2.6. Timeline

Task	Date	Assigned Person	Notes
Choosing Idea	18/01/2021	All	Search Ideas…
Physics research	24/01/2021	Joshua	Find formulas for gravity, spring, collisions, etc…
Write introduction	24/01/2021	Arnaud	Introduction, how we came up with the idea, objectives, possible difficulties
Work on timeline	24/01/2021	All	Determine the tasks that have to be completed, when those tasks will be done and who will do them
Develop algorithm	01/02/2021	All	Come up with an algorithm in pseudocode. Determine complexity.
Design Main Menu and Setting graphics	01/02/2021	Jakob	Design Main menu and settings menu textures.
Write down optional features	01/02/2021	Arnaud	Start thinking of some optional features to add if we have enough time.
Write UML Diagram	05/02/2021	all	Write UML Diagram for all needed classes. Determine the interaction between classes
Write the features	08/02/2021	Arnaud	Main game features, as well as limitations and constraints

Conclusion	08/02/2021	Arnaud	Write the conclusion to the project
Create Wall and Stick Spring GUI	08/02/2021	Jakob	Design graphics for walls and player character, needed animations.
Oral Presentation	15/02/2021	All	First presentation of the idea of the project
Write vector class	15/02/2021	Joshua	
Write SpringStick class (player character)	21/02/2021	Arnaud	
Write PhysicsEngine class	21/02/2021	Joshua and Jakob	
Write Play Class	28/02/2021	Jakob	
Write Wall class	28/02/2021	Joshua	
Write Level class	07/03/2021	Arnaud	
Code KeyChecker Class	07/03/2021	Joshua	Class that checks if a key has been pressed
PlayTest and cleaning up code	15/03/2021	All	Testing the gameplay of the game and making sure everything runs smoothly
Code Main Menu GUI	22/03/2021	Jakob	
Code Setting Menu GUI	29/03/2021	Jakob	
Write the forces information display	29/03/2021	Joshua and Arnaud	
Additional features if possible	05/04/2021	All	Code the additional features if the time permits it
Write the final report	April	All	
Oral presentation	April	All	
Comprehensive Assessment	April	All	

2.7. Software used

2.7.1. IDE

For the coding part, we will be using Netbeans. This means we are using Java along with the JavaFX plugin. We choose this software and IDE since it is one that we are very comfortable with. We have worked with it for almost two years now, and so we have a pretty good understanding of its ins and outs.

2.7.2. Platform

We may also use GitHub to share our code between ourselves – in order to make it easier to sync up our work. The platform that we are coding on is Windows.For the GUI, we are using a graphical piece of software called Affinity photo. This tool will be used to design the stick player model, as well as the main menu's visuals.

2.7.3. Software of packages

The JavaFX library will be the main package used to display and run the game. The Java.lang package will be used for using Threads to run the movement of the player character.

3. Features

3.1. Explanation of all features

When running the program, a screen shows up displaying three options for the player to select: Play, Setting, and Exit. The Play button sends the player into the first level. The Setting button will mainly serve for some optional features, which will be explained shortly, but include options like turning on or off some background music and customizing the player model as well as changing its mass. The Exit button, obviously, would serve to exit to game by closing the window in which it takes place.

When the Play button is pressed, as mentioned above, the player begins the first level. The level is presented as a series of walls, ceilings, and floors, all of them being either horizontal or vertical. The player, through their keyboard, will have the ability to make the stick navigate through the level. Indeed, by pressing either the left and right arrow keys or the "a" and "d" keys, the stick will rotate around it center accordingly to input. That way, the plyer will be able to control the angle at which the stick hits the different surfaces.

The player will have to navigate through the level (level*s* if we have enough time to make more than one) and reach a predetermined ending point, at which they will have completed the level.

We also plan on displaying, in one of the corners of the screen, the velocity vector of the stick and updating it in real time, as the stick is flying through the air. This would be a way to make the game more educative by providing a better understanding of how vectors work and how they can be used to calculate the velocity of an object throughout its trajectory.

3.2. Limitations and constraints

It can prove difficult, using JavaFX, to create accurate trajectories, as it is made for very simple animations. We however plan on making a method that updates the stick's position as often as possible without being too demanding of the computer the program is running on. This will help us keep the trajectories as accurate as possible, since their position will be kept in check and re-evaluated almost constantly.

One of the main difficulties of this project will be identifying which part of the stick is colliding with a surface, since the sides of the stick do not bounce in the same way that the ends so. Indeed, only the ends are equipped with springs, making them way bouncier. Furthermore, as explained earlier, a "spring collision" will send the stick flying back in the direction it is pointing in, while a "side collision" will make the stick bounce as if reflected by the surface.

3.3. Optional Features

There are a few things that we would like to have enough time to incorporate, but the physics in themselves might end up taking too much of our time, so we will have to revaluate if we have enough time to add them further down in the creation process. First of all, we would like to be able to have more than one playable level. This would create additional challenges by placing the player in new environments so that they can find new ways to exploit the physics of our game. Secondly, we would like to be able to add an option on the settings menu where the player could change the mass of the stick as they wish. This would be an interesting feature, as changing the mass greatly affects how the spring interacts with its environment in terms of bouncing, velocity, and how it is affected by the gravitational force. Thirdly, in the "Settings" section on the menu, we would like to put an option to turn on or off a background soundtrack, as well as change its volume.

Another possible potential feature would be to give the player the ability to choose between different colors or skins for their player model. This would give a little visual variety that could make the experience more enjoyable.

4. Conclusion

In conclusion, this project has proven to require quite a bit of mathematics and physics, some of which we were not very familiar with, as well as a need to get more familiar with many aspects of JavaFX, since we plan on using it in ways different from what we've done with it before. For both of those reasons, this project will be a challenge, but one we more than welcome, as it will test our abilities to extents we have not met before.

To be able to complete this project within the assigned time, we will need to be rigorous in our work, always do our best, and encourage each other when working on the harder parts of the programming. We also must not set ourselves up for failure by aiming for objectives beyond our abilities. A project like this one can never be truly complete, as there is always a way to add to it. To prevent us doing this, we will have to do our best to stick to our preestablished schedule and only work beyond it once the project is up and functional. Only then may we work on some of the optional features presented in this document. We strongly believe that with this work method, we will accomplish, if not surpass, our initial goals and objectives.

5. References

1. Physics formula: https://gamedev.stackexchange.com/questions/23672/determine-resulting-angle-of-wall-collision
2. Main Menu and Level Backgrounds https://www.walpaperlist.com/2020/01/retro-neon-phone-wallpaper.html
3. Original Game inspiration: https://nasheik.itch.io/cleaning-the-system