# Space Hunter

Tyler Waltner

# The Goal:

The goal of the game is to collect all of the tokens (gold spheres) that spawn into the maze and reach the finish line before you run out of lives. Reaching the tokens fills up your fuel and they also double in function as checkpoints for the game. This game I would say is heavily inspired by the CoolMathGames “World’s Hardest Game” but essentially its in Space, with gravity. (which really doesn’t make sense i know haha)

# Star Class

Star class:

The star class has 4 instance variables: starX, starY, starWidth, starHeight

And are pretty self explanatory…

I make a array of 400 randomly generated stars varying in location / size in the App class and in the AnimationTimer class I draw them with the gc.fillOval().

Just a nice touch to make the background look less bland.

# Block Class (extends Star)

Block class extends the Star class and stores 4 variables.

X, Y: indicating top corner of Block

Height, Width: indicating the height / width

Is used to create wall and borders in the game

# Token Class (extends Star)

The token class stores the various tokens located throughout the map.

Special Methods:

setDisplayTrue() & setDisplayFalse() change setDisplay boolean variable which is used to determine if the Token should still display. (once we collect we don’t need to display it anymore.)

Contains isColliding() method to return boolean and detect collision between token & player.

# SafeZone Class (extends Star)

Extends Star class, like Token class also has isColliding() to detect collision with player. Stores the x, y, height, and width of the green safe zone.

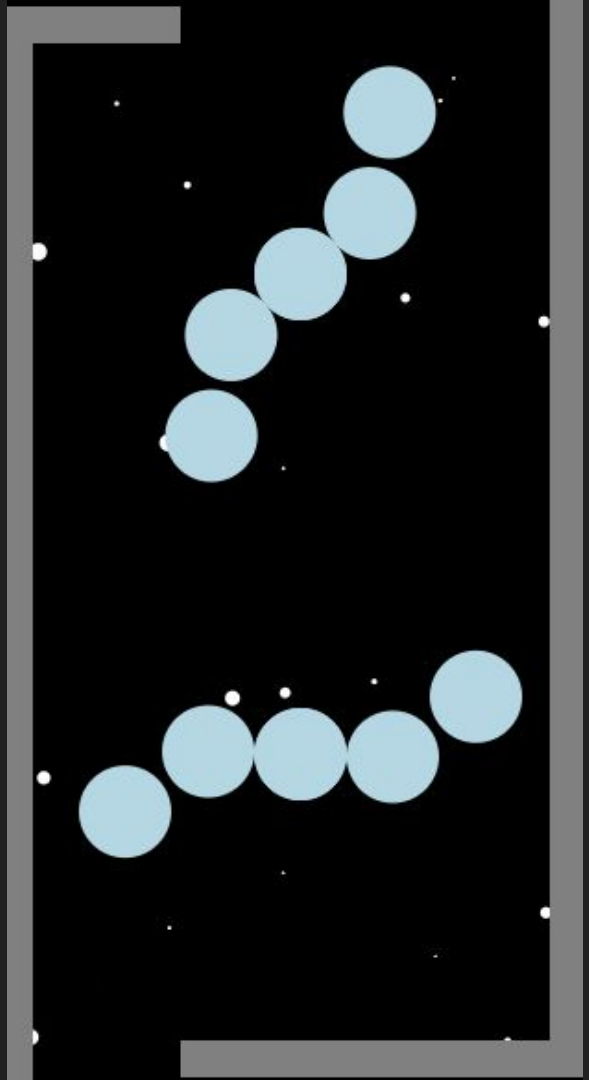
# EnemyCirclePath Class

Variables: pathRadius, centerX, centerY, double speed

We want this object to orbit around a center so we have an update():

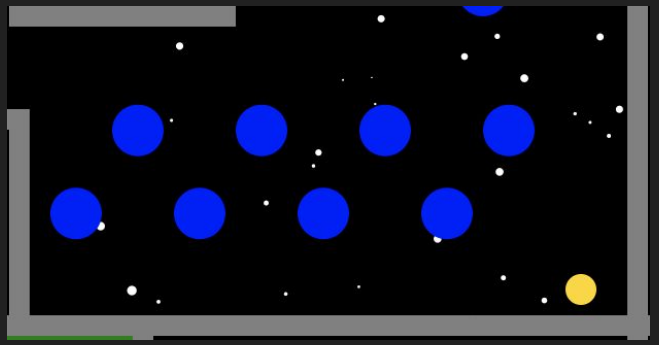
```
public void update() {
    x = centerX + pathRadius * Math.cos(speed * time);
    y = centerY + pathRadius * Math.sin(speed * time);
}
```

*Also has an isColliding method to check for collisions.



# EnemyVertPath Class (extends EnemyCirclePath)

Overrides update() as this object wants to move up & down instead of in a circle.

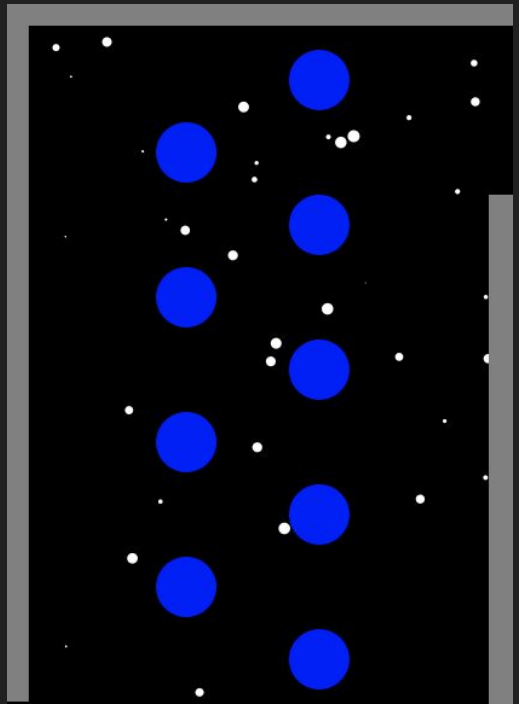


```
@Override
public void update(double time) {
        x = centerX;
        y = centerY + pathRadius * Math.sin(speed * time);
}
```

# EnemyHorzPath Class

Overrides update() as this object wants to move side to siede instead of in a circle.

```
@Override
public void update(double time) {
        x = centerX + pathRadius * Math.sin(speed * time);
        y = centerY;
}
```



# Player Class

Variables: x, y, prevX, prevY, gravity, canvasWidth, canvasHeight, size, maxSpeed, timeWNotPressed, xVelocity, yVelocity, left, right, top

All variables are pretty self explanatory except for:

prevX/Y: previous X/Y position

timeWNotPressed: increases by +1 every 60th of a second that we go without holding down ‘W’ (up button)

Left, right, top: booleans that indicate whether the player is pressing the keys to move in said direction. (used to animate thrusters)

# Player Class methods

setX, setY, getX, getY, getSize, getTop, getLeft, getRight - self explanatory 1 line

update() & ifColliding() handle actual movement

The way I coded my game the only actual “solid” objects in my game are the walls. Tokens, Enemies, SafeZone, Stars, all can be passed through by player. (though there is other logic in the AnimationTimer class that makes this unnoticeable to the user Ex: resetting the player location once they die)

# Player Class methods - update()

Takes 2 inputs: Set<String> input, ArrayList<Block> Arr

The hash map is used to figure out which keys are being pressed by the user so we can update the x & y velocities and change the appropriate boolean variable for the key being pressed. (top, left, right)

For the gravity, we take the total amount of time that ‘W’ has not been pressed (int timeWNotPressed) and if the Player is NOT on the ground && it’s Math.abs(yVelocity) < the maxSpeed then we apply the gravity to yVelocity through: yVelocity += timeWNotPressed * gravity;

To add friction I just multiply respective axes (axis plural) by 0.95 if their movement button isn’t being pressed. If (W isnt being pressed) yVelocity * 0.95, if (A and D aren’t being pressed) xVelocity * 0.95

# Player Class methods - ifColliding

+

Takes the array list of Blocks as input: ifColliding(ArrayList<Block> Arr) and iterates through the blocks, calculating whether or not we will collide to any of the Blocks (walls) in the array list given our current trajectory. (using x & y velocities) As soon as we detect even a single block is overlapping, we break from the loop iterating through the blocks and go through a series of logic to determine what the course of action should be.

If we will collide, we reset the respective axis' velocity back to 0 and we put the player to it's previous x or y value through variable prevX & prevY.

If we won't collide we let the Player's x & y values be updated respectively through nextX & nextY.

# App Class

The App Class is the Main class and contains lots of the setup for the program such as:

- Usual javafx setup
- Star[] for creating the stars in the background
- Creating the GUI: pause button, fuel counter, lives left counter, score counter
- Hash map to receive keyboard inputs
- Initialization of various objects (Blocks, Enemies, Tokens, SafeZone)
- AnimationTimer anonymous class (where actual animation occurs)

# App Class important methods

Add methods: circlePathsAdd(), vertPathsAdd(), horzPathsAdd() streamlines process of initializing enemies.

Display passive Objects: displayBlocks(), displayTokens(), displaySafeZone()

Display enemies: displayHorzPath(), displayVertPath(), displayCirclePath()

Display player: displayPlayer(): handles displaying player & thrusters through player.getTop(), player.getLeft(), & player.getRight()

Death method: playerDie(): negates lives by 1, resets fuel back to full, resets player location back to last token collected (or original start position)

# AnimationTime Class


The AnimationTime class is where all of the actual animation happens. A new is animated every 60th of a second so for Objects I stored them in arrays / ArrayLists such as Star, Blocks, Enemies, & Tokens and are iterated through and displayed.

Much of the logic of the program is done in the AnimationTime class.

Every frame, our fuel goes down by a small increment. Once Player runs out of fuel, the playerDie() is called and a message is displayed.

A “YOU WIN” message is displayed when you reach the safe zone and a “GAME OVER” message is displayed when you die and have no more lives left. (this.stop(); is run for either case)

# AnimationTime Class continue

When we are iterating through and displaying all of these objects, we also have to run some logic. Examples below:

For every enemy we check if player is colliding. If so, we run the playerDie() and display a death msg.

For every token we check if the player has collected it. If so, we stop displaying it, add 50pts to the score counter to make it so the next time the Player dies, they will respawn at their last collected token.

# And yeah that’s about everything!

Thanks! :-)