COSC112: Final Project Program Specifications

Andrew Palena, Sergei Leonov, Biruk Tefera, Bilal Tariq

May 7th, 2024

Program Specifications

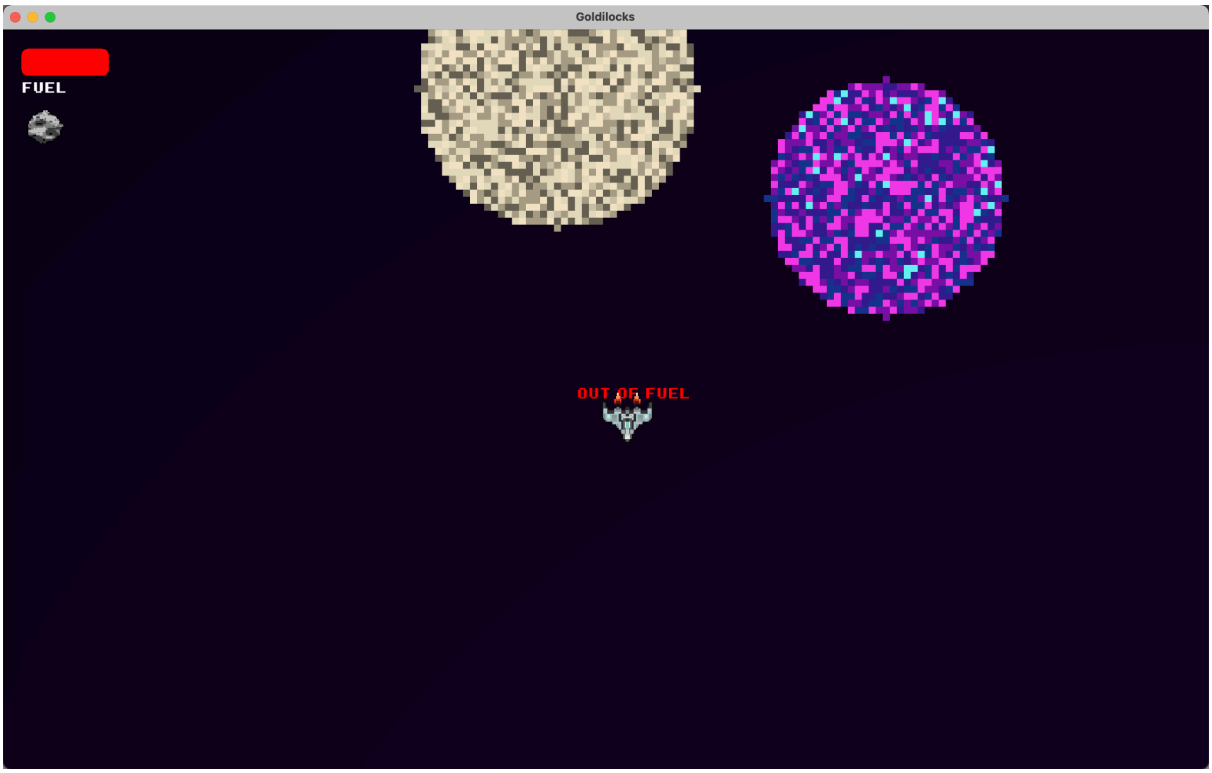
Project Overview

Our project is a 2D game set in a large space ecosystem. The player takes on the role of an astronaut exploring distant planets, throughout the ecosystem. Instead of just aimlessly exploring, the player wanders in search of the coveted “Goldilocks” Planet. This special planet takes its place in the world as the most resourceful object in space. Covered in precious natural resources and fuel, Goldilocks is the crown jewel of an astronaut's journey. However, to find Goldilocks the player must embark on a harrowing journey, through the depths of a treacherous space world. Along the way they will encounter threats such as asteroids, mobs, and critically low fuel, which they must mitigate if they wish to survive.

The game takes on a pixel art style with custom sprites and color patterns. The codebase is written in Java and utilizes the Java Swing graphics library to handle graphical rendering. All of the art is loaded in at runtime, and stored in a dedicated folder within the project repository. The background is the endless void of space. Each planet resides on top of the space background and are situated in unique static locations across the map. Asteroids and Characters, such as the main player populate the foreground, and have the ability to move across planets and through the space void. Characters, asteroids, and the spaceship the character uses to navigate are all loaded sprites. Planets, instead are loaded images, that are randomly generated using procedural generation. There are 5 main types of planets that populate the night sky: earth, lava, cyberpunk, stone, and sand. Each is randomly spawned with randomized size and textures.

Program Flow

Additionally, the game has multiple screens for the player to navigate through. When the program is first run, the user is greeted with the game start screen. There the user must use the mouse to either select the start game phase, or the quit phase - which exits the program. The game start phase makes use of a simple light theme, and two basic buttons. When the play button is selected, the user is then taken to the intro animation screen. On this page, a series of prompts takes the player through a basic fictional setup. The display is static, but the user must use the ‘k’ key on their keyboard to progress the setup guide. Upon each key press the user is greeted with another short message describing the basic concepts of the game and the fictional backstory. After the user has stepped through the entire introduction message, the next ‘k’ key press begins the game.



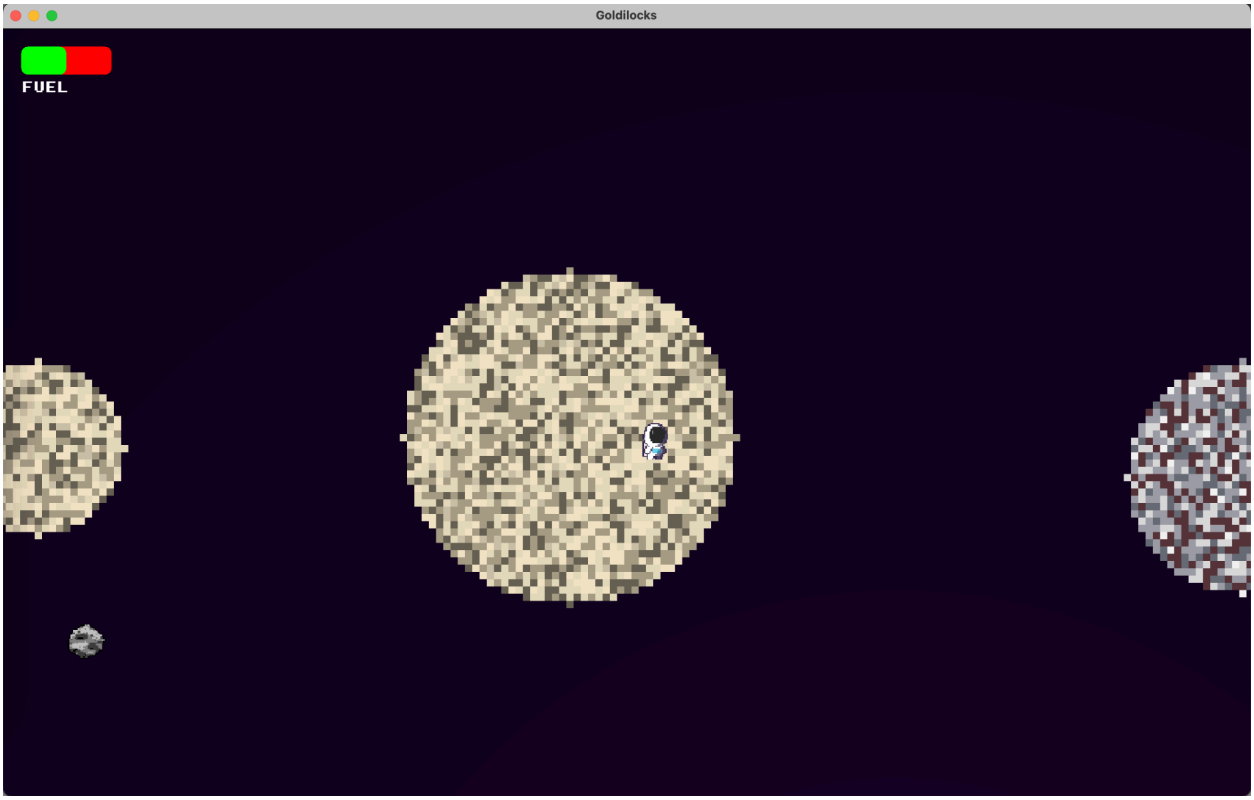
Once the game has started, the user must drive their spaceship around to find the Goldilocks planet. To fly the spaceship the user must use the ‘w’ ‘a’ ‘s’ ‘d’ keys on their

keyboard. Pressing and holding the keys will start the movement of the spaceship in the corresponding direction. ‘W’ propels the spaceship screen up, ‘a’ propels screen left, ‘d’ propels screen right, and ‘s’ propels the spaceship screen down. By using Java’s KeyListener Two direction keys can be held at once to propel the spaceship in the bisecting direction of the two directions. For example, if the ‘w’ and ‘d’ keys are held simultaneously, the spaceship will travel in the screen upward-right diagonal direction. To stop movement, the user must release the key pressed down. When the spaceship changes direction, the on-screen sprite will rotate accordingly to place the nose of the spaceship in the correct direction. The exception to this is if the spaceship is traveling in a diagonal direction. The sprite will take the orientation of the first key pressed in sequence.

While flying, the spaceship is consuming fuel. If the user does not attempt to replenish the reservoir, they will face a quick demise. To refuel, the user must first locate the asteroids flying around the world. Then, they must use their mouse to fire the spaceship’s cannon in the direction of the asteroids. By clicking a location on the screen, the user aims and fires cannonballs in the vector direction between the spaceship and the clicked location. The bullet

will travel in the specified direction infinitely unless it hits an asteroid. If the bullet collides with an asteroid the fuel for the spaceship will increase by 2 units. Therefore, if the player continues to destroy asteroids, they will be able to sponsor their exploratory efforts. Java's MouseListener is used to read the mouse clicks and initiate the shooting methods.

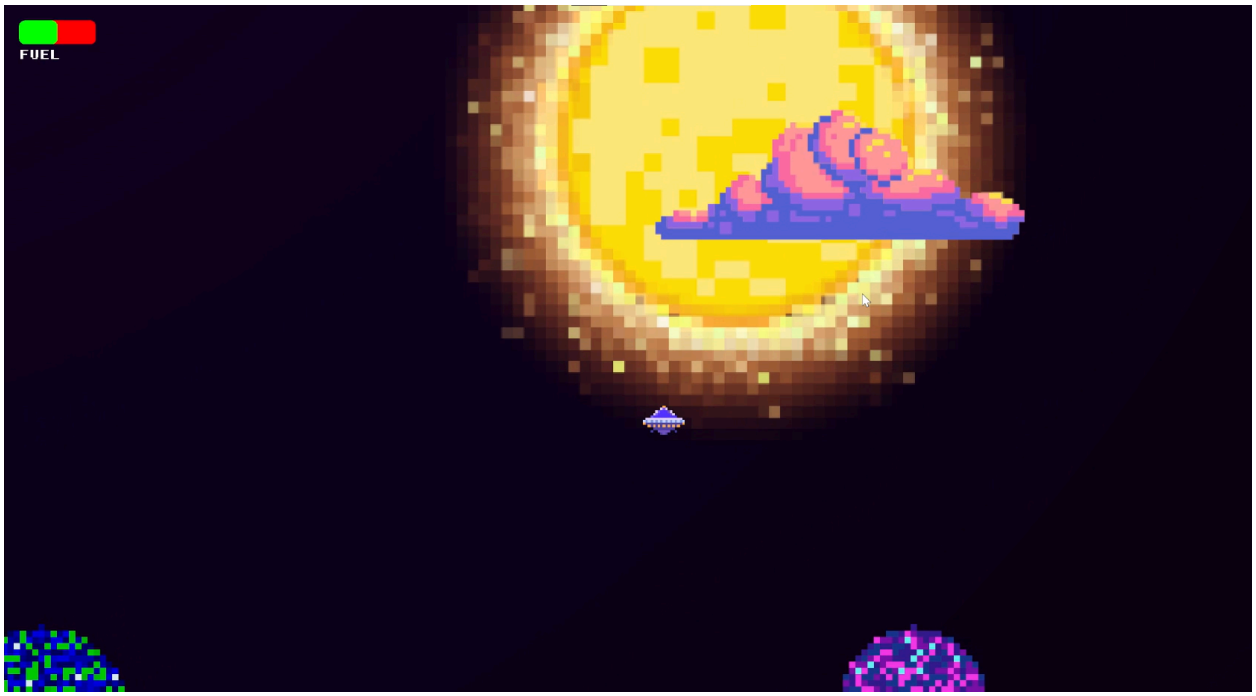
To explore a planet in greater detail, the user must first use the direction keys to



approach the planet in their spaceship. Once arrived, the in-game player departs the spaceship, and is free to navigate the terrain on foot. Their sprite changes to a human spacesuit and they can use the same direction keys to "walk" around the planet terrain. When the player is ready to get back in their spaceship, they can proceed to the edge of the planet, and will automatically reboard their spaceship. Once they reassume command of their spacecraft, all features and functions of the spaceship, as well as the fuel level, will be restored.

When the player finds the unique Goldilocks planet, the game will terminate and a game ending screen will appear. The game over music will be played, and similar to the intro

animation the user will be presented with message prompts from their wizard friend. Upon completion of the prompts, the program will terminate.



Classes:

1) **Pair**
    a) Similar to the programs we created for homeworks, Pair will be a holder class that will store the two double private variables to help us encode velocity, position, and acceleration
2) **Fields**
    a) Private double *x*
    b) Private double *y*
    c) These will store the values that we pass. Both private since we will have getter and setter methods to access them. Double because we need the floating point to set positions and velocities properly
3) **Constructor** (we have two of them, one with no arguments, and one with x and y)
    a) One with arguments:

b) Takes in two double arguments

c) Assigns them to private fields

d) One with no argument:

e) No arguments

f) Assigns 0 to both x and y arguments.

4) Methods

a) Getter and setter for both x and y fields (change the values or allow them to be read)

b) *Public void flipX()* and public void *flipY()* (void, no parameters) multiply the x or y value by -1 accordingly to change direction

c) *Public void multiply()* takes in a double scalar which the method will use to multiply x and y fields.

d) *Public Pair addPair()* takes in a Pair type argument and returns a new pair with x and y values added accordingly

e) All methods are non recursive non-static and public

5) Background

a) This class represents the background of the game, the universe. It includes all the asteroids, all the planets, and other stuff. We only have one instance of this object.

b) Fields:

i) - private int WIDTH

ii) - private int HEIGHT

iii) The above two represent the height and width of the background.

6) Methods:

a) - Constructor: public Background() this constructs the only instance of background.

b) - public drawBackground(Graphics g0ri): this draws the background as a gradient.

i)

ii)

7) Character

a) Character is a public abstract class in a standalone file. It is used to represent all life objects in the world. At the onset of the project we had envisioned a world full of life, however character has been limited to being extended by the Player class. It is used to represent the actions of lifeforms, such as movement, and drawing to the screen. No instances of Characters are created since it is an abstract class

b) Fields

i) Private boolean alive

(1) A boolean variable to represent the life status of the character

ii) Protected Pair position

(1) A Pair data structure to store the current position of the character in the world

iii) Protected Pair vel

(1) A Pair data structure to store the current velocity of the character in the world

iv) Protected Pair accel

(1) A Pair data structure to store the current acceleration of the character in the world

c) Constructor

i) Initializes all Pair fields to be Pairs that store (0,0) and sets the character to alive

ii) Only one basic constructor needed

d) Methods

i) Public abstract void draw(Graphics g)

(1) A method to be overridden by child classes to draw the character to the screen

ii) Public method so that the draw method can be accessed for all characters in Main

8) Moveable

a) Moveable is an interface to represent the non-static objects in the game. Moveable contains methods to update the position and move and stop movement of the object.

b) Methods

i) Public abstract void update(Game g, double time)

(1) A public method to be accessed in the Main class for each object in the game

(2) Updates the position of the object

ii) Public abstract void move(int dir)

(1) Method to be overridden and accessed in main to start movement of object

iii) Public abstract void stopMove(int dir)

(1) Method to be overridden and accessed in main to stop movement of object

9) Villager

a) Villager is a class that represents the starting villager in the game. One instance of villager is made at runtime, and it is removed from memory at program termination. The villager class contains the required fields and methods to constantly run the starting villager.

b) Fields

i) Public Pair position

(1) A pair data structure to store the villager's position in the world, public to be accessed from Main

ii) Public Pair velocity

(1) A pair data structure to store the villager's velocity while moving in the world, public to be accessed from Main

iii) Public Image village

(1) An Image object from the java.awt.Image package to store the image sprite of the villager

iv) Public boolean prompt

(1) A boolean value to store if the villager will be talking or not, made public to be accessed from Player and Main classes

v) Public Font font

(1) A Font datatype from the Java.awt.Font package to store the font for the villagers speech

c) Constructors

i) Public Villager(Pair position)

(1) Villager has one constructor that sets the position of the villager at the position passed in as a parameter, sets the velocity to a positive x direction value of 20. It also sets the villager image to an image sprite from the assets folder, and scales it to an appropriate size. The font is also initialized to the font passed in from the GameFont.ttf file

d) Methods

i) Public void flipImage()

(1) A method to flip the image sprite of the villager using the AffineTransform and AffineTransform packages from Java. It flips the image as a reflection over the y-axis so that the sprite faces the opposite direction

ii) Public void updateAI(Game g, double time)

(1) A method to move the village around the starter planet. If the villager is between bounds 100 and -200, move the villager according to its velocity. If outside the bounds, it flips the sprite and sets the velocity to be in the other direction. If the player is within 80 pixels, it sets the prompt to true so that the villager speaks

iii) Public void draw(Graphics g)

(1) A method to draw the villager to the screen. The method is public so that it can be accessed from the main class. If the prompt is set to true, it draws the corresponding prompt for the villager speaking saying “Hello, young one”

10) IntroScreen

a) IntroScreen is a class to present the starting screen of the game. It is instantiated upon the execution of the main method in the Main class. IntroScreen is drawn to the same dimensions as the Game is(system specific). IntroScreen is removed and deleted upon termination of the program.

b) Fields

i) Private int width

(1) A private variable to store the width of the display, to be accessed in IntroScreen

ii) Private int height

(1) A private variable to store the height of the display, to be accessed in IntroScreen

iii) Private Image bacImage

(1) A private variable to store the background image, to be accessed in IntroScreen

iv) Private Color menuButton1/menuButton2

(1) Colors from Java.awt.Color package to store the colors of the menu buttons

v) Int buttonx1

(1) int to store the x location of button1

vi) Int buttony1

(1) Int to store the y location of button1

vii) Int buttonx2

(1) Int to store the x location of button2

viii) Int buttony2

(1) Int to store the y location of button2

ix) Double widthBox

(1) Double to store the width of the 2 buttons

x) Double heightBox

(1) Double to store the height of the 2 buttons

xi) Public RoundRectangle2D playButton

(1) RoundRectangle2D object to store the rectangle polygon of the play button

xii) Public RoundRectangle2D quitButton

(1) RoundRectangle2D object to store the rectangle polygon of the quit button

c) Constructors

i) Public IntroScreen(int WIDTH, int HEIGHT)

(1) The constructor for the IntroScreen class initializes width and height to the passed parameters, then sets the x and y positions of the buttons to the desired locations. The play button is set 100 pixels above the quit button. The dimensions of the buttons are set around the x and y coordinates and the sizes are set to the width and height boxes. The background image is set to be the our desired background image we read in from the assets file.

d) Methods

i) Public void updateIntroScreen()

(1) An update method to check if the mouse is hovering over the play button and then recreates the boundaries of the play button, needs to be public to be called from the Main class

ii) Public void draw(Graphics g)

(1) A draw method which draws the entire intro screen display. The draw method draws the black background, then draws the background image to take the entire screen. Similarly the font for the intro screen is set to the GameFont read in from the “GameFont.ttf” file in the assets folder. Next the draw method draws in the other assets of the screen, including the two buttons with text. The draw method is public, and called from the Main class.

11) Game

a) The game class represents the actual running. Essentially, everything in the Space world is managed by the Game class. There is one instance of the Game class created, called steveGame and it is removed from memory at the termination of the program.

b) Fields

i) Public Player steve

(1) A public variable of type Player to store the main player of the game. Humorously named steve

ii) Public Villager villager

(1) A public variable of type Villager to store the villager on the starter planet. Made public to be accessed from Main class

iii) Public int rotate

(1) A variable to store the rotation value for the player. Public to be accessed in the moving methods of Main

iv) Pubic camera cam

(1) A variable to store the camera for the game. It is public because it must be accessed and moved with the player in the Main class.

v) Public IntroScreen introScreen

(1) A IntroScreen variable to store the introScreen for the game. Is public to be accessed in Main

vi) Public WorldNoise worldNoise

(1) A WorldNoise variable to store the values of the noise algorithm for procedurally placing random planets.

vii) Public Background background

(1) A Background variable to store the Background of the IntoScreen and animation.

viii) Public ArrayList<Planet> planets

(1) An ArrayList to store the planets. Planets are added and subtracted in other classes, so must be public

ix) Public ArrayList<Asteroid> asteroids

(1) An ArrayList to store the asteroids in the game, must be made public

x) Public Goldilocks goldilocks

(1) A public Goldilocks variable to store the goldilocks planet

xi) Public boolean isWinner

(1) Public variable that tells us if the player won the game by finding goldilocks

(2) Used to start the outro animation and music

c) Constructor

i) There is one constructor for the game class, which does most of the initial heavy lifting with setting up the game. The constructor first initializes the fields of steve, and villager at their specified locations, middle of the game. The camera is initialized right at the player and the planets are initialized to an empty ArrayList. Introscreen is created using the Main class program dimensions, along with IntroAnimation. Next the constructor initializes the planets based on the noise generation algorithm defined elsewhere in the document. To actually set the position of the planets the noise generation algorithm is applied to find the optimal positions for the planets. For a new planet, it is first checked if there is a good spot to make the planet. Then it sets the location of the new planet to that location. Simultaneously, it checks to ensure that the new planet is not overlapping any other planets in the ArrayList before adding the planet.

d) Method

i) Public boolean checkTotalOverlap(Planet a)

(1) A public method to check if a planet defined in the parameters overlaps and planets in the Game ArrayList. Returns true if it does, false if not.

ii) Public ArrayList<Asteroid> initAsteroids(int n)

(1) A public method to initialize n number of asteroids and add them to the ArrayList. It then returns the ArrayList. It is public because it needs to be accessed from the Main class.

iii) Public updateGame(double time)

(1) A public method to update all elements of the Game. It first checks if the player has overlapped Goldilocks, and sets isWinner to true if true. Then It updates positions of all elements, such as Villager, Players, and Camera. The method also loops through all of the Asteroids and checks to see if the player has destroyed them, initializes a new asteroid if needed. Finally, it sets the velocity of the player and camera to 0 if the spaceship is out of fuel.

iv) Public void drawPlayers(Graphics g)

(1) This public method is called in Main at repaint() and draws all elements of the game at the same instance. It draws the planets, fuel capacity, villager, asteroids, and bullets. If the player is being moved in a new direction, the method also uses the rotate method of g2d to rotate the player and redraw it at the corresponding location.

v) Public void drawPlanets(Graphics g)

(1) Draws all of the planets, to be called in drawPlayers()

12) Player

a) Player is a class which extends Character and implements Movable. Abstractly, it is a class which represents the main player on the screen. Player will override all methods from Character, which store the basic information about the player object. By implementing Movable, the player class is also a representation of an object which can move around the game. One instance of Player is made at runtime, and it is discarded at program termination.

b) Fields

i) Private Pair position

(1) A private Pair data structure to store the position of the player at a given position, only accessed from Player

ii) Private Pair velocity

(1) A private Pair data structure to store the velocity of the player at a given frame, only accessed from Player

iii) Private Image spaceship

(1) A private Image data structure from Java.awt.Image package to store the spaceship sprite, only accessed from Player

iv) Private Image astronaut

(1) A private image data structure from Java.awt.Image package to store the astronaut sprite, only accessed from the Player class so can be private

v) Private double fuelCapacity

(1) A private double variable to store the fuel capacity, time until the spaceship runs out of fuel/game is over. Only accessed in the Player class so can be private

vi) Public int fuelCollected

(1) A public int variable to store the amount of fuel collected from destroying asteroids in the Main class

vii) Private double currentFuel

(1) A private double variable to store the current fuel, how much time until the present spaceship state runs out of fuel/the game runs out of fuel. Is only accessed in the Player class and don't want it modified in Player class

viii) Private Rectangle bounds

(1) Private Rectangle object from the Java.awt.Rectangle package to store the bounds of the Player in the Main class. The variable is private because it is only accessed within the player class and bounds should not be modified by classes outside of the Player

ix) Public ArrayList<Bullet> bullets

(1) An ArrayList data type to store Bullet objects which “belong” to the player. Bullets are public because there are bullets added and accessed from the Main class.

x) Private boolean isSpaceship

(1) A boolean variable to store whether the player is a spaceship or an astronaut. Private because we do not want it overridden outside of the designated player class.

c) Methods

i) Public boolean getIsSpaceShip()

(1) A public method which returns the boolean field isSpaceShip()

ii) Public Rectangle getBounds()

(1) A public method to return the bounds of the player as a Rectangle object. Returns the bounds field. Public to be accessed from Main

iii) Public double getFuel()

(1) A public method to return the current fuel of the spaceShip at the present moment, public to be accessed from Main class

iv) Public void refuel()

(1) A public method to refuel the spaceShip fuel if fuel has been collected. If the current fuel exceeds fuel capacity, then fuel is set to capacity and fuel collected is decremented

v) Public void updateFuel()

(1) A public method to be accessed in Main class which decrements the fuel if the player is not on planet. Calls refuel method upon method call

vi) Public int getSpaceshipLevel()

(1) Returns the level of the spaceship, public to be accessed in Main

vii) Public double getX()

(1) Returns the double value of the x coordinate

viii) Public double getY()

(1) Returns the double value of the y coordinate

ix) Public void draw(Graphics2D g)

(1) Draws the spaceship image if the player is in their spaceship, draws the astronaut image if the player is not in the spaceship. Public so that is can be accessed from Main and Game classes.

x) Public void drawFuelCapacity(Graphics g)

(1) Draws the fuel capacity bar of the spaceship so that the user can view the current level of their fuel. The amount of fuel is displayed in green as a proportion of the whole bar. If the fuel is empty, the method prints a message that the spaceship is out of fuel in red at the spaceship position. Public so that it can be accessed by a call in Main and Game

xi) Public void move(int dir)

(1) Starts the player's movement in the direction of the key press. Sets the velocity to +-100 so that the player is constantly moving while the key is pressed in. Public to be accessed from the Main class

xii) Public void stopMove(int dir)

(1) Stops the player's movement in the direction of the key press. Sets the velocity to 0 so that the player's movement is halted on key press. Public to be accessed from the Main class

xiii) Public void update(Game g, double time)

(1) Checks if the player is in space or on planet, and sets the value of isSpaceship to the correct value. Then adds the position by the velocity times time since the last frame. Adjust the bounds of the player accordingly. Public to be accessed from the thread class in Main

xiv) Public boolean isOnPlanet(Game g)

(1) Checks if the player is on a planet by looping through all planets and checking if the player intersects one at the given frame. Accessed within player class and Main class.

xv) Public void shoot(double targetX, double targetY)

(1) Creates a new bullet and adds it to the Player's bullet list to be shot

13) Bullet

This class represents the bullet the spaceship shoots. There are multiple instances of this, whenever the player aims at an asteroid and clicks the left click an instance is created.

Fields

- Pair position: representing the position of the bullet.
- Pair velocity; representing the velocity of the bullet.
- Int radius; represents the radius of the bullet, all bullets have the same radius but useful for other implementations.
- Public boolean used: checks if the bullet has been used, false by default.
- Private Rectangle bounds: represents the bounds of the bullet.

Methods

- **Public Bullet(double x, double y, double targetX, double targetY):** this is the constructor which creates the bullets whenever we shoot.

- **Public void update(Game g, double time):** This method updates the position of the bullet by adding the velocity and multiplying time.

- **Public rectangle getBounds():** this is a getter method to get the bounds of the bullet, since it is private.

- **Public void draw(Graphics g):** this draws the bullet

14) Simplex Noise

This is an external class that generates the noise to be used to create the planets and the whole universe(where planets should be generated and where we should have stars, and other stuff). It has several fields and methods. Basically, It serves as a simplex noise generator. Take a look at the following link for specific details.
https://jvm-gaming.org/t/lwjgl-procedurally-generated-simplex-perlin-noise-2d-world/46571/2

15) SpaceObject

This is an abstract that represents all space objects such as: planets, asteroids, and other stuff. So the mentioned classes extend spaceObject. Since it is also an abstract class no instance of it is instantiated.

Fields

- **Public int radius:** this variable stores the radius of space objects. And all space objects have radius.
- **Public Pair position:** this represents the position of the space object. And all space objects have position x and y.
- **Public Pair Velocity:** this represents the velocity of the space object. And again all space objects have velocity since they move.
- **Public Pair accel:** this represents the acceleration of space objects which is used by some of the space objects but not all.

- **Rectangle Bounds:** this represents the bounds of the space object, this will be used to track if a player is on a space object.
- **Image texture:** this represents the image of the space object that is then used to draw the space object.

Methods:

- The methods in this class are just getters and setters for each field. So we have: setRadius, setVelocity, setAccel, getRadius, getRadis, getVelocity, getAccel…..
- In addition to the getters and the setters, we also have:
- **Public void draw(Graphics g):** which draws the space object, no matter what the space object is.
- **Public Pair getCenter():** returns the center, position wise, of the space object.

16) Planet

This class represents the planets we have in the entire universe. We will have a bunch of instances, many of them actually, of this. It also extends space objects since a planet is a space object.

Fields:

- **Public double[][] noiseMap:** this array stores the noise value of the universe.
- **Public int planet type:** this int represents the type of the planet could be 1, 2, 3, 4, 5, or 6 representing the different types of the planets in the universe.
- **Public Static int countPlanet:** This field keeps track of the number of planets in the universe.

Methods:

- **Public Planet(int initRadius):** this method creates all the planets in the universe, there are a bunch of them created when the program is run.
- **Public static getNumPlanets:** This method serves as a getter for countPlanets.
- **Public[][] getNoiseMap():** this serves as a getter for the noise map which will be used in other implementations.
- **Public boolean checkPlanetOverlap(Planet a):** this checks if the planet overlaps with the planet passed.

  //https://stackoverflow.com/questions/335600/collision-detection-between-two-images-in-java helped resolve this
- **Public double noise(int i, int j):** this generates a new noise for the universe.
- **Public boolean insideCircle(Pair center, Pair point):** this method is used to get a bunch of variables and check whether distance is less than or equal to radius.
- **Public void circularNoise():** this generates a circular noise using a nested for loop.
- **Private void makePNGfile() throws IOException:** this method makes a png file of the planets and stores them in an allocated folder since it is more efficient than drawing them manually.
- **Public draw(Graphics g):** draws the planets.
- **Public Color planetColorScheme(double alpha):** this class returns the colors of the planets based on its type.

17) Asteroid

This class extends SpaceObjectsa nd implements Movable. Semantically, it represents the asteroids to be shot. There will be a total of 200 asteroids which will be instantiated when the game is created.

Fields

- Since it extends SpaceObjects and implements Movabe, it has all their fields(look above in this document for specifics).
- Public boolean destroyed: stores whether or not the asteroid has yet been destroyed.
- Private Rectangle bounds: this stores the bounds of the asteroids. Later used for collision detection.
- Private Image texture: stores the image to be drawn;

Methods

- **Public Asteroid(int initRadius):** serves as a constructor **so** makes instances of asteroids with radius initRadius.
- Public void Draw(Graphics g): this draws the asteroids.
- Public void update(Game g, double time): updates the position of the asteroid and determines whether it is destroyed or not.
- Public Rectangle getBounds(): this returns the nouns of the asteroid.

18) Goldilocks

This class represents the final planet the player has to find. It extends the planet because it is a special type of planet. There will only be one instance of this and the player has to find it to complete the game.

Fields

- Since it extends the planet class it has all its fields.
- Private Image goldilocksPlanet: represents the image of the planet.

- Private Image villager: represents the image of the villager who will be on the goldilocks planet.

Methods

- **Public GoldiLocks():** serves as a constructor so creates the goldilocks planet
- **Public void draw(Graphics g):** this draws the goldiLocks planet.

19) StarterPlanet

This class represents the starter planet. The planet where the player spawns at. There will only be one instance of this throughout the whole game.

Fields

- **Image starterPlanet, cloud1, cloud2, castle:** this represents the images of the planet, cloud1, cloud2, and castle respectively.
- **Private cloud X1:** represents the x position of cloud1 which will always be at -350,
- **Private cloudX2:** represents the x position of cloud2 which will always be -100.
- **Private int VelocityX:** represents velocity of cloud1, always equal to 1.
- **Private int velocityX2:** represents velocity of cloud 2, always equal to 2.

Methods

- **Public StarterPlanet:** creates the starter planet, it is a constructor.
- **Public void draw(Graphics g):** draws starter planet.
- **Public void animateCloud:** Responsible for making the cloud1 animation.
- **Public void animateCloud2:** Responsible for making the cloud2 animation.

20) Outro

This class takes care of the outro animation, only one instance which will be created when the player completes the game.

Fields

- **Private int width:** represents the width of the outro screen.
- **Private int height:** represents the height of the outro screen.
- **Private image background:** stores the background image.
- **Private Font gameFont:** stores the game font.
- **Public int continueCounter:** stores the state of the outro for animation purpose.

Methods

- **Public Otro(int WIDTH, int HEIGHT):** Takes care of creating the outro screen, its a constructor.
- **Public draw(Graphics g):** handles drawing based on the counter to make the animation come to life

21) Camera

- This class represents the camera object which follows the player everywhere even when it gets out of bounds. This class also implements movable (look above for what's inherited from movable).

Fields

- **public Pair position, velocity;**

  The above fields represent the position and velocity of the camera since it moves to follow the player wherever he goes.

Methods

- **public Camera(Pair pos):** this method makes an instance of the camera, only one instance is created though. In this method the velocity and position are instantiated.

- **public void setPosition(Pair a):** this method sets the position to a.
- **Public move (int dir):** this method handles the movement of the camera whenever the player moves in whatever direction, that's why it takes in dir.
- **Public stopMove (int dir):** this method handles stopping movement of the camera whenever the player stops moving based on the direction that is why it takes in dir.
- **Public void update(Game g, Double time):** this method handles updating the position of the camera by adding velocity times time.

22) IntroAnimation

- This class represents the intro animation of the game.

Fields:

- **Private int width:** represents the width of the animation screen.
- **Private int height**: represents the height of the animation screen.
- **Private image background:** this represents the image of the background, its initialized from an external file source.
- **Private image textBox:** this is used as a textbox for the direction given by the initial wizard at the start of the screen.
- **Private image wizardDude:** this represents the wizard dude that gives the initial directions for the player.
- **Private Font gameFont:** this represents the fonts the text from the wizard are delivered.
- **Public int continueCounter**: this keeps track of which page we are on. So that the animation is rendered accordingly. It's set to 0 by default.

Methods:

- **Public IntroAnimation(int WIDTH, int HEIGHT):** This initializes all the fields of this class and draws the required images from external sources, we have the files in a separate folder.
- **Public void draw(Graphics g):** this method draws the important stuff to make the animation come to life.

23) Main

Fields:

- **public static final int START_SCREEN**
- **public static final int GAME_RUNNING**
- **public static final int GAME_OVER**
- **public static boolean isIntroScreen** variable keeps track of whether the intro screen should be drawn
- **public static boolean isIntroAnimation** variable keeps track of whether the intro animation should be drawn
- **Public static final int GAME_WIDTH** stores the width of the "infinite" world
- **Public static final int HEIGHT** stores the height of the "infinite" world
- **Public static Dimension screenDimension** stores the dimension of the screen of the user, used to fit the graphics for different resolutions

- **Public static int WIDTH** stores the pixel width of the user's monitor
- **Public static int HEIGHT** stores the pixel height of the user's monitor
- **Public static final int FPS** stores the framerate of the game
- **Game steveGame** stores an instance of the Game class
- **Public boolean isWinMusic** boolean that stores a value that tells if the win music was initialized, used to not initialize it more than once in the runner class

Methods:

- **Public Main()** constructor that initializes all the variables, adds key,mouse, and mouse motion listeners to JFrame, initializes the thread for the runner class, shows intro screen and plays intro music.
- **Public static void main(String[] args)** tester method
- **Public void keyPressed(KeyEvent e)** overridden method from the key listener interface. Used to determine the motion of the player using key inputs WASD.
- **Public void keyReleased(KeyEvent e)** overridden method from the key listener interface. Used to stop the motion of the player
- **Public void keyTyped(KeyEvent e)** overridden method from keylistener interface used to keep track of the user pressing K to move the animations along and play the music when intro animation is initialized
- **Public void mousePressed(MouseEvent e)** overridden method from mouse listener class that keeps track of when the mouse is pressed. If the user is on the intro screen, it detects whether a button is pressed and what should follow. Also, allows the user to shoot bullets when game is played
- **Public void mouseReleased(MouseEvent e)** overridden method that we had to include because of the mouselistener interface. Does not do anything
- **Public void mouseEntered(MouseEvent e)** overridden method that we had to include because of the mouselistener interface. Does not do anything
- **Public void mouseExited(MouseEvent e)** overridden method that we had to include because of the mouselistener interface. Does not do anything
- **Public void mouseClicked(MouseEvent e)** overridden method that we had to include because of the mouselistener interface. Does not do anything
- **Public void painComponent(Graphics g)** handles repainting of the appropriate components. When on intro screen, it draws intro screen components, when in game, calls the drawPlayers method in the Game class that draws player, background gradient, planets, asteroids. When player wins, draws the outro animation
- **Public void mouseDragged(mouseEvent e)** overridden method from mouseMotionListener interface that we did not use
- **Public void mouseMoved(MouseEvent e)** highlights the buttons when the cursor hovers over them and increases the size of the play button if the cursor is on it if the user is on the intro screen.

24) Runner

Fields: none

Methods:

- **Public void run()** runner method that has a while true loop that runs the game updates the intro screen to see if the user clicks the buttons. If the user is in the game, the method updates the game. If the player wins, it plays the music once so that it does not restart over again (boolean to track that). Also, calls the repaint method to draw components at each iteration after updating.

How Classes Interact