Professor Purtee and Bella Save the World From AI Robots
(or something like that)

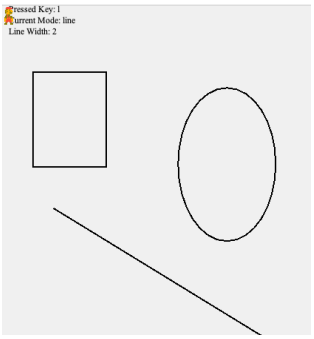
Design Document

Elvis Imamura
CSC 171 – Intro to Computer Science
Prof. Adam Purtee

Contents

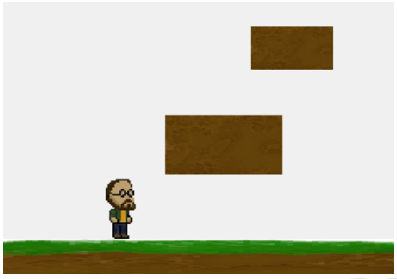
Overview

Design Philosophy and Version History



Before the project was formally assigned and before the ConnectTheDots homework, I experimented with creating a drawing program with the Graphics library that was able to draw different shapes and drag them across the screen. From this, I thought that it would be a good idea to create a buildable platformer, similar to MarioMaker (even though I've never played it before).

Due to the numerous different parameters involved in drawing swing Graphics, I thought it would be best to create my own vector-based implementation to simplify the process moving forward.



For a while, the mario sprite in the corner was the main character as I figured out how to implement motion. Once I had controls working, I decided to create some basic sprites and textures to make the game look more original.

After Bella made a guest appearance, I was inspired to add her to the game. She also was the first reason for adding the pathfinding mechanic.

Although the final gameplay is not very impressive, I am still very proud of the framework I've built. With the Level class, build mode, and navigable levels via doors, I would have been able to create a much more full-fledged game with multiple layers of levels given more time.

Admittedly, the levels I created so far are quite difficult (all are play-tested and possible). This is to maintain a certain level of interest since the game's mechanics are not very interesting.

Plans for Future Implementation and Resubmission

For the resubmission, I hope to add more levels, more story-based frames and animations, and sound effects. There are also a few minor motion bugs that need to be sorted out.

Class and Method Breakdown

Class PhysicsGame

This is the main class which contains most of the methods for operating the game. It contains the main **Window** (extends JFrame) and **Canvas** (extends JPanel) classes.

Inner Class Canvas

The canvas is the main driver which handles the displaying of all game elements. It uses **KeyListener**, **MouseListener**, and **MouseMotionListener** with custom keybindings and methods to detect clicks in specific areas or on specific objects.

There are two modes within the class: **buildMode** and play mode (buildMode = false). Build mode allows me (and the user, if interested) to fully customize levels with the loaded textures. This will be explained in more detail in the Notable Features.

Notable Methods

initLevel(Level level) initializes the given level, setting all of the onscreen ArrayLists equal to those in the level. As explained later in the **Level** class breakdown, Level stores all of the data for the game elements of each level, and the main class simply references a level for displaying.

paintComponent(Graphics g) overrides the original Graphics.paintComponent method and handles the displaying of all visual elements. It calls, but does not handle the motion and physics all game elements

buildPaint(Graphics g) handles the graphics for elements specific to **buildMode**, such as real-time updating of drawn elements.

translateMap(Vector2f dr), and other methods handle camera panning in the map by keeping track of the total translation as a vector called **mapTranslationVector.** All movable shapes **(ArrayList<ImageShape> movableShapes)** are translated by this vector to emulate side-scrolling behavior

package mypackage

mypackage contains all of the custom objects and methods behind the elements that make up the game. Its main directory includes the classes:

- **Vector2f**
- **Level**
- **Door**

and also contains the sub-package **shapes**, which includes:

- **VectorShape**
- **ImageShape**
- **VectorRect**
- **VectorLine**
- **VectorOval**

Class Vector2f

Vector2f (not to be confused with javax.vecmath.Vector2f) is the core object used to build all other shapes and functions used in the game. They are used to store location, motion, size, and directional data and is the most fundamental component. It contains all of the normal vector operations such as:

- **add(Vector2f other)**
- **sub(Vector2f other)**
- **magnitude()**
- **unitVector()**
- **scalarMult(float c)**

Class Level

Level stores all of the data for a playable level. VectorImages are distributed to ArrayLists which represent different components of the level.

- **environment**
    - Contains the “solid” elements within the map like floors and walls. The player and other **focusedShapes** collide with the elements in environment
- **enemies**
    - Contains the locations of all enemies. Causes level to reload when it detects a collision with the player
- **boundary**
    - Not currently in use, but is intended to be a clear border which reloads the map when touched by the player.
- **doors**
    - Contains the locations, textures, and destination levels for all doors in the level.
- **focusedShapes**
    - Holds all of the shapes affected by gravity and collideable with the environment **(enemies and mario/bella)**
- **movableShapes**
    - Holds all of the shapes that translate during mapTranslation **(environment, enemies, doors, mario/bella)**
- **draggableShapes**
    - Holds the shapes which are draggable in buildMode **(environment, enemies, doors, mario/bella)**

Notable Methods

save(String filename)
saves the current level as a text file to a given filename. The file contains fields for the filename, spawnpoint, environment, enemies, doors, boundary, and background image as shown.

```
maps/hoyt1
SPAWNPOINT
401.999512 467.985229
ENVIRONMENT
sprites/environment/floor2.png 1458.000000 95.000000 13.995300 594.985596
sprites/environment/floor2.png 57.000000 627.000000 13.995300 -28.021362
sprites/environment/floor2.png 154.000000 131.000000 707.996094 481.985291
sprites/environment/floor2.png 257.000000 284.000000 1214.997192 338.982666
sprites/environment/floor2.png 88.000000 755.000000 1832.028564 -48.021484
sprites/environment/floor2.png 87.000000 367.000000 1833.028564 705.985657
sprites/environment/floor2.png 86.000000 251.000000 1834.028564 994.979736
sprites/environment/floor2.png 1500.000000 143.000000 64.996094 -24.021362
sprites/environment/floor2.png 512.000000 163.000000 1408.010010 -43.021484
sprites/environment/floor2.png 793.000000 90.000000 1126.992676 1157.958862
sprites/environment/floor2.png 0.000000 1.000000 1495.014404 967.984253
sprites/environment/floor2.png 375.000000 52.000000 1459.012451 940.985596
sprites/environment/floor2.png 168.000000 809.000000 1026.992676 941.985596
sprites/environment/floor2.png 446.000000 88.000000 612.997253 1359.952271
sprites/environment/floor2.png 157.000000 815.000000 476.999512 936.985596
sprites/environment/floor2.png 156.000000 44.000000 902.994019 942.002197
sprites/environment/floor2.png 142.000000 798.000000 14.000000 657.000000
sprites/environment/floor2.png 142.000000 555.000000 14.000000 1423.000000
sprites/environment/floor2.png 1096.000000 154.000000 15.000000 1959.000000
sprites/environment/floor2.png 168.000000 375.000000 1027.000000 1738.000000
sprites/environment/floor2.png 220.000000 80.000000 135.000000 1107.000000
sprites/environment/floor2.png 250.000000 93.000000 274.000000 1365.000000
sprites/environment/floor2.png 181.000000 80.000000 118.000000 1665.000000
sprites/environment/floor2.png 250.000000 68.000000 594.000000 1684.000000
ENEMIES
88.210388 463.129395
1716.674805 1037.178345
164.000000 1831.000000
DOORS
hoyt2 sprites/doors/hoytdoor.png 146.000000 146.000000 671.000000 1537.000000
BOUNDARY
BACKGROUND
sprites/backgrounds/hoyt.png
maps/hoyt1 (END)
```

load(String filename)
reads a given save file and generates a level based on its contents.

Class Door

Door acts as a gateway between levels and serves as the main level/menu navigation method. The lobby and its level selection area uses doors to choose the starting chapter. It has an associated destination level which is set upon drawing, and also has an associated **ImageShape** for displaying it onscreen.

package mypackage.shapes

mypackage.shapes is the **Vector2f**-based implementation of swing graphical elements. The shapes implemented include VectorRect, VectorOval, VectorLine, VectorShape, and ImageShape. In order to be referenced onscreen, all shapes are fundamentally rectangles with two main vectors:

- **tail** represents the top-left corner location of the bounding rectangle and is referenced when dealing with translation and motion

- **vec** represents the size of the bounding rectangle from top-left to bottom-right and is referenced when dealing with rigid body motion to determine the extent of the shape.

The two most important and widely used shapes, **VectorShape**, and **ImageShape**, are explained below.

Class VectorShape

VectorShape is the parent class of all other shapes in the package and contains most of the methods used for object interaction, physics handling, and motion, including pathfinding. Pathfinding details will be explained in the Notable Features section.

Notable Methods: Object Interaction and Physics

contains(Vector2f point) returns a boolean value if a given point lies within the bounds of the shape.

detectCollision(VectorShape other) returns a Boolean value if any cmesh points (see Notable Features) lie within the bounds of another shape.

handleCollision(VectorShape other, boolean elastic) handles the collision between two objects. By using the cmesh, the method detects the direction of the collision and appropriately stops or bounces the object.

Notable Methods: Motion and Pathfinding

calculatePosition() calculates the spatial position of an object based on motion, updating the cmesh points.

calculateMotion(float dt) calculates the change in position and velocity over a given time interval **dt** based on the objects acceleration and velocity.

handleMovement() uses **movementDirection** (int values of −1, 0, 1) to determine the shape's horizontal movement direction. **movementDirection** is either increased or decreased by the move and stop methods, allowing the program to effectively handle multiple key inputs. It also handles the **facingRight** value which keeps track of the object's orientation, which is used when assigning animation frames.

moveLeft/Right() sets the movement for the shape by changing **movementDirection**, but does not actually handle movement.

stopLeft/Right() indicates an end in movement in a given direction by changing **movementDirection**, but does not actually handle movement.

jump() sets the objects vertical velocity to a preset value. The object returns to the ground due to gravitational acceleration

Class ImageShape

ImageShape takes care of the visual representation of solid VectorShapes as sprites. They are essentially represented as rectangles with an associated image. The animation functions are built-in and allow the ImageShape to cycle through a set of frames to show animation.

Notable Methods

updateFrame() cycles through the current set of frames and resets when it reaches the end

handleMotionFrames() uses **facingRight**, **movementDirection**, and other motion data to determine the appropriate set frames to display

Notable Features

buildMode

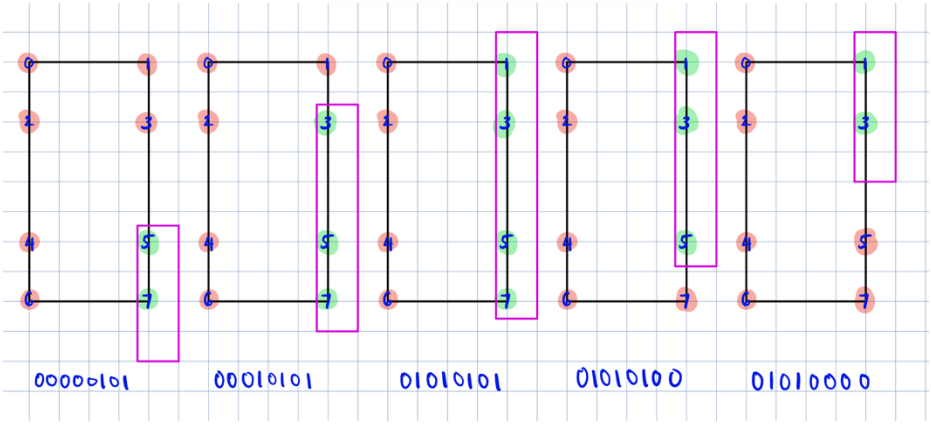
buildMode, triggered by the "B" key, was the method used to create all of the levels in the game. It keeps track of a few variables, **drawnShape** and **selectedShape** in order to allow interactions with specific drawn elements. There are a few **drawMode**s which enable different types of interaction with the level:

- **"e" : environment**
  - Draws environment elements and appends to environment list
- **"c" : character**
  - Either places an enemy or sets the level spawnpoint
- **"p" : door**
  - Draws a door and prompts an input to specify its destination
- **"d" : drag**
  - Drags the selected shape (checks all **draggableShape**) across the screen
- **"backspace" : delete**
  - Deletes **selectedShape**
- **"[/]" : texture cycle**
  - Cycles through textures for given drawMode and displays current texture at top left corner of screen
- **"\" : background cycle**
  - Cycles through available backgrounds
- **"l" : load**
  - Loads a saved level from input prompt
- **"s" : save**

- Saves current level to prompted location

Collision Detection and Handling

Collision handling was probably the most difficult feature to implement with the main issue being determining which face had collided. I decided to create a mesh of 8 points at the corners of each VectorShape, which I called the **cmesh** (collision mesh). The mesh is represented by a 8-digit binary short, and collisions mesh values for U/D/L/R collisions are saved for reference. Below is a visual example of the **LCollisions** set of shorts and how the collided points are represented



detectCollidedPoints(VectorShape other) determines which of the cmesh points lie within the target object, and returns the binary value for the collision.

handleCollision(VectorShape other, boolean elastic) uses the binary value to appropriately handle the collision based on the indicated direction by either bouncing or stopping on that axis.

Vector2f Implementation

As mentioned previously, all objects and shapes used in the game are fundamentally based on the **Vector2f** class. To see (almost) all of the vectors being used in the game, hit “V” on the keyboard to enter **vectorMode**, which displays most of the current vectors.

Pathfinding

The pathfinding methods allow a shape to track to another shape. In-game, they are applied to bella and the enemies. Pathfinding is based on a **pathfindVector** which points from the shape to its target, informing the direction and determining the movement of the shape. The pathfinding properties are customizable and include:

- **boundToTarget**
    - a boolean value which determines whether the shape teleports to the target or stops pathfinding when the maxSeparation is exceeded
- **pathfindVelocity**
    - the speed at which the shape moves when pathfinding
- **pathfindTarget**
    - the targeted shape
- **maxSeparation**
    - the trigger distance for the behaviors outlined in **boundToTarget**
- **pathfindBoundary**
    - the minimum distance within the target which shapes pathfind to

pathfind() simply uses the **pathfindVector** to determine the direction to move in. It also handles the behavior of the object based on **boundToTarget.**