‘PacWars’ Game Documentation C0452

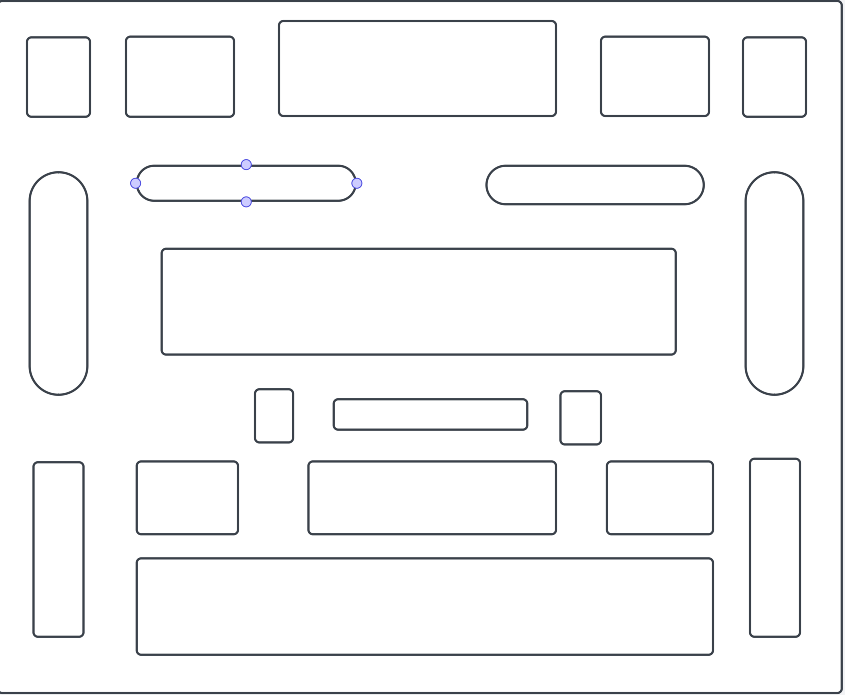
[22202188@bucks.ac.uk](mailto:22202188@bucks.ac.uk) - Mark Thwaite

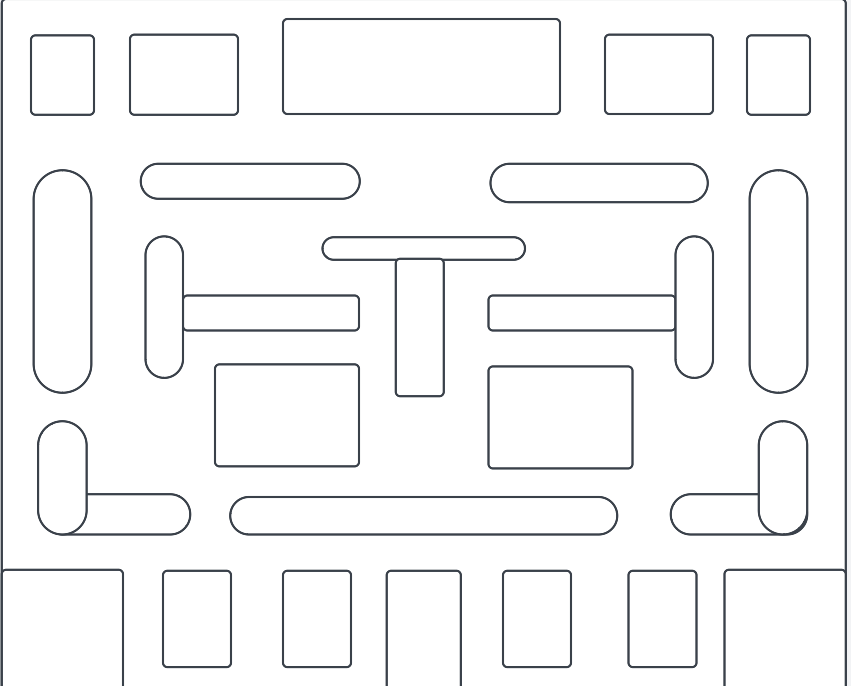
[22201368@bucks.ac.uk](mailto:22201368@bucks.ac.uk) - Lewis Burke

First team meeting -

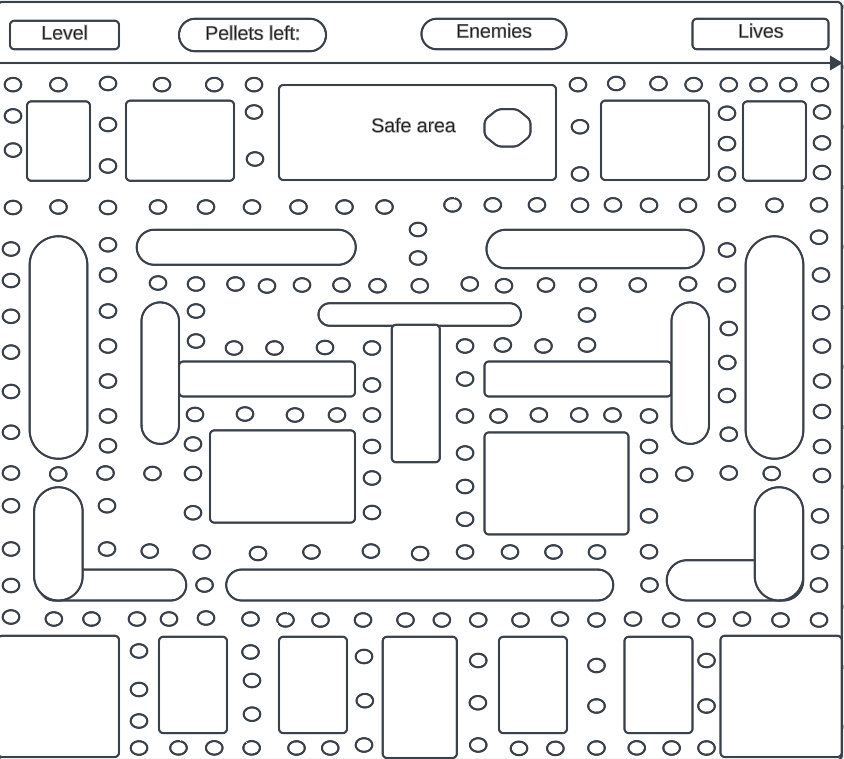
We discussed what type of game we would want to make, we shortlisted it to 3 different games, Space invaders, Pac man and Tetris. After some discussion we ultimately decided to go with Pac man. We discussed our team roles and decided that Mark would focus mostly on coding, Lewis would do the design and documentation. End of the first meeting.

Team meeting 2-

Using Lucid chart, Lewis designed the first level of our game and we discussed it and all agreed it was a good starting off point and Lewis designed more, Mark helped. Since Mark needed a design to work off, Lewis designed a new one. 



We designed another level and we were happy with it, it had a cleaner look and looked challenging for the player.

Team meeting 3 - 

Lewis added a level number, how many collectibles are left (pellets), how many enemies are on your current level, and your lives.

collectibles (pellets)

 Player

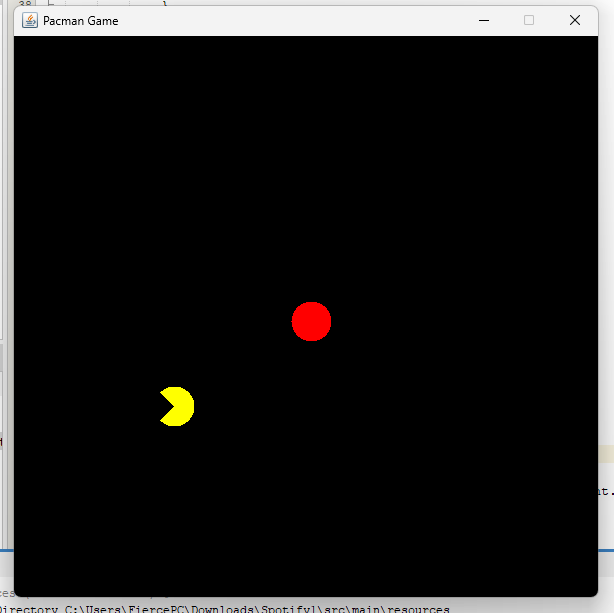
 enemy

 power up

Team meeting 4 -

Mark starts working on our game and creates our first character.

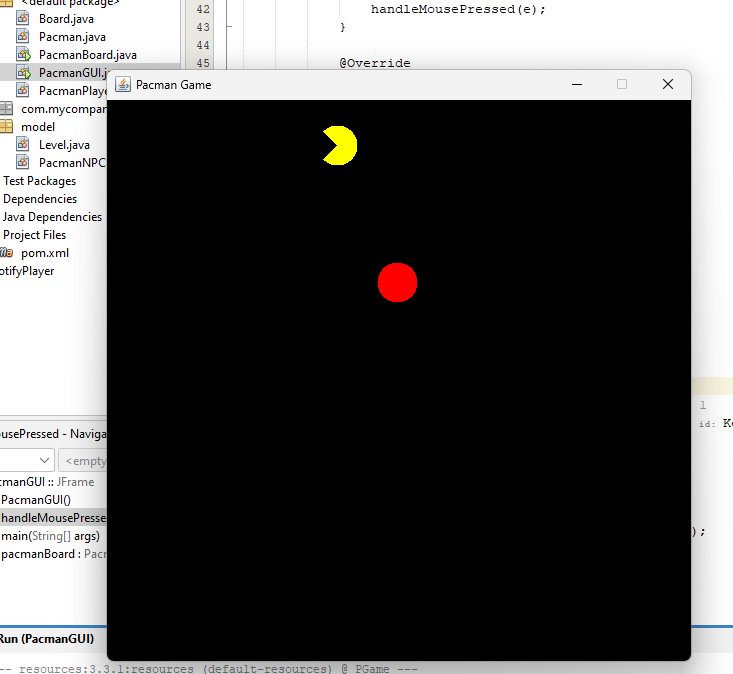




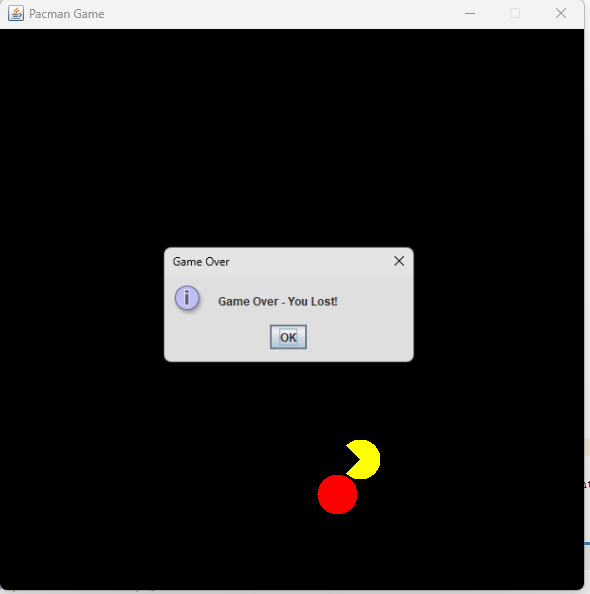
He then adds our enemy, an AI that will chase the player.

Team meeting 5 -

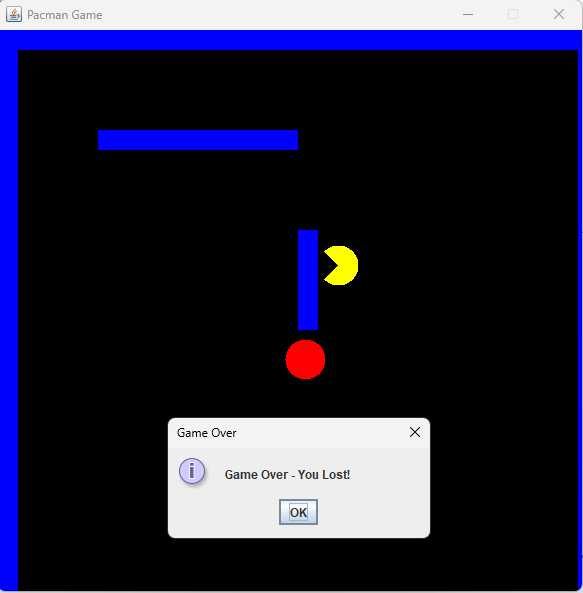
Mark successfully made it so the enemy AI will chase the player correctly at a relatively competitive speed.



We had trouble with the player's movement as any key would make it move left but we resolved this issue after a few rewrites in the code. We then ran into another problem, when the enemy touches the player the game would close, this is obviously not ideal so Mark added a game over screen.



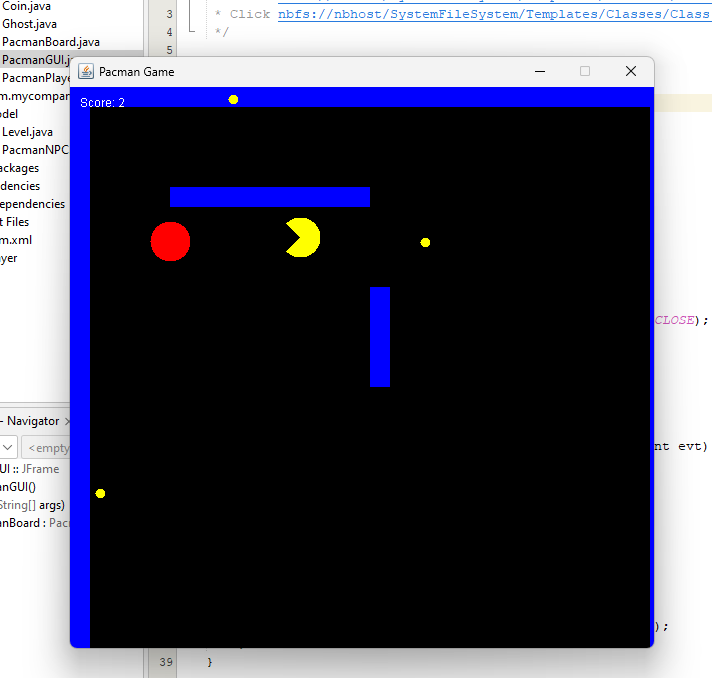
Team meeting 6 -

We started designing the map in the game, Mark adding the first two walls.

We ran into a few issues with the walls, initially when the player hit the wall the game would end and our enemy AI would spawn into the wall and get stuck. After messing around with the code, Mark fixed the issue

Seeks fastest route to player

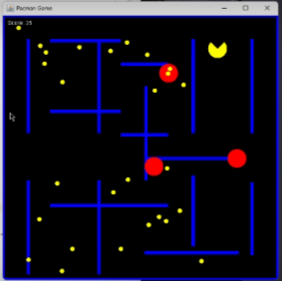
Team meeting 7 - Hey, at least we have coins.

We talked about our game design and changed our game from a level based game to a score based game.

We added a score in the top left and collectibles around the map. We ran into the issue of our collectibles spawning in the wall.

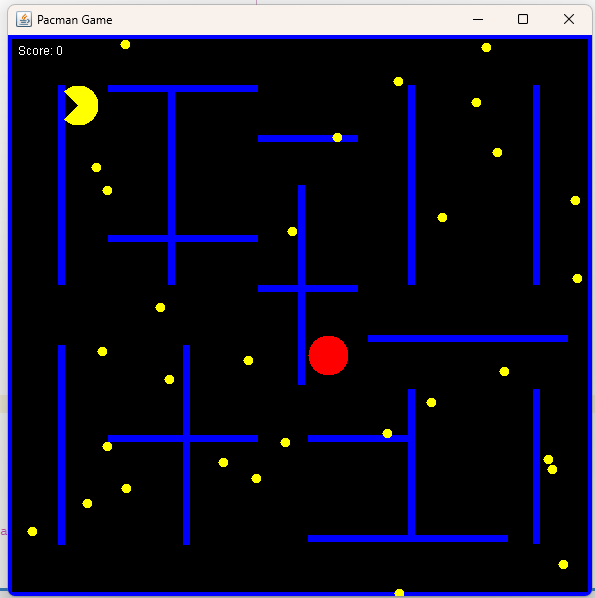
Team meeting - 8 -

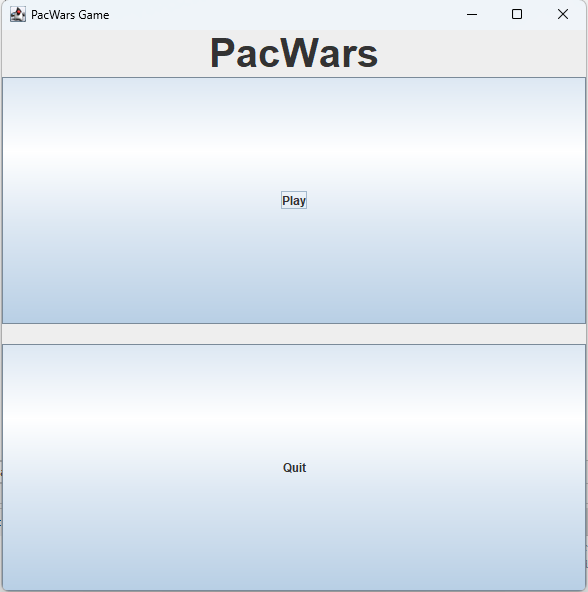
Mark cleaned up the game a bit and added more walls



He moved the walls so it resembled a map, which was extremely tedious. He also added more enemy AIs. He coded so when you reached a certain score, more enemies will spawn, the total number of enemies being four.

Final changes for the game:



Menu screen:  


This is all the code for the gamer:  
  
**PacmanGUI class**:  
  
import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class PacmanGUI extends JFrame {

private PacmanBoard pacmanBoard;

private MenuPanel menuPanel;

public PacmanGUI() {

menuPanel = new MenuPanel();

add(menuPanel);

setSize(600, 600);

setLocationRelativeTo(null);

setTitle("Pacman Game");

menuPanel.getPlayButton().addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

startGame();

}

});

menuPanel.getQuitButton().addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.exit(0);

}

});

}

private void startGame() {

remove(menuPanel);

pacmanBoard = new PacmanBoard();

add(pacmanBoard);

addKeyListener(new java.awt.event.KeyAdapter() {

public void keyPressed(java.awt.event.KeyEvent evt) {

pacmanBoard.handleKeyPress(evt);

}

});

pacmanBoard.requestFocusInWindow();

setFocusable(true);

setVisible(true);

revalidate();

repaint();

}

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

PacmanGUI pacmanGame = new PacmanGUI();

pacmanGame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE); // Add this line

pacmanGame.setVisible(true);

});

}

class MenuPanel extends JPanel {

private JButton playButton;

private JButton quitButton;

public MenuPanel() {

setLayout(new BorderLayout());

JLabel pacmanLabel = new JLabel("Pacman", SwingConstants.CENTER);

pacmanLabel.setFont(new Font("Arial", Font.BOLD, 40));

add(pacmanLabel, BorderLayout.NORTH);

JPanel buttonPanel = new JPanel(new GridLayout(2, 1, 0, 20));

playButton = new JButton("Play");

quitButton = new JButton("Quit");

buttonPanel.add(playButton);

buttonPanel.add(quitButton);

add(buttonPanel, BorderLayout.CENTER);

}

public JButton getPlayButton() {

return playButton;

}

public JButton getQuitButton() {

return quitButton;

}

}

}

**PacmanPlayer class**:

import java.awt.Color;

import java.awt.Graphics;

import java.awt.\*;

import java.util.List;

public class PacmanPlayer {

public int x, y;

private char currDirection;

private char direction;

private char desiredDirection;

private int score;

public PacmanPlayer(int startX, int startY) {

x = startX;

y = startY;

currDirection = 'L';

direction = 'L';

desiredDirection = 'L';

score = 0;

}

public void collectCoins(List<Coin> coins) {

for (Coin coin : coins) {

if (!coin.isCollected() && x == coin.getX() && y == coin.getY()) {

coin.setCollected(true);

score += 10; // adjust score

}

}

}

public int getScore() {

return score;

}

public boolean collectsCoin(Coin coin) {

int playerRadius = 20;

int coinRadius = 5; // Adjust the size of the coins on the map

int playerCenterX = x + playerRadius;

int playerCenterY = y + playerRadius;

int coinCenterX = coin.getX() + coinRadius;

int coinCenterY = coin.getY() + coinRadius;

double distance = Math.sqrt(Math.pow(playerCenterX - coinCenterX, 2) + Math.pow(playerCenterY - coinCenterY, 2));

return distance < playerRadius + coinRadius;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public void setDesiredDirection(char newDirection) {

desiredDirection = newDirection;

}

public void reset() {

// Reset the player's position or any other necessary attributes

x = 50;

y = 50;

currDirection = 'L';

direction = 'L';

desiredDirection = 'L';

}

public void move(int[][] walls) {

int newX = x;

int newY = y;

// Updates the current direction before moving

currDirection = desiredDirection;

switch (desiredDirection) {

case 'L':

newX -= 5;

break;

case 'R':

newX += 5;

break;

case 'U':

newY -= 5;

break;

case 'D':

newY += 5;

break;

}

// Checks for collisions or boundaries before updating the position of the player

if (isValidMove(newX, newY, walls)) {

x = newX;

y = newY;

// Updates the current direction to the desired direction

direction = desiredDirection;

}

}

public void undoMove() {

// Reverts the player's position based on the last move

switch (direction) {

case 'L':

x += 5;

break;

case 'R':

x -= 5;

break;

case 'U':

y += 5;

break;

case 'D':

y -= 5;

break;

}

}

public char getCurrDirection() {

return currDirection;

}

private boolean isValidMove(int newX, int newY, int[][] walls) {

// Checks if the new position is within the allowed bounds and doesn't collide with walls

int playerRadius = 20; // Adjust the value based on the playersize

int playerLeft = newX;

int playerRight = newX + 2 \* playerRadius;

int playerTop = newY;

int playerBottom = newY + 2 \* playerRadius;

// Checks if the new position collides with walls

for (int[] wall : walls) {

int wallLeft = wall[0];

int wallRight = wall[0] + wall[2];

int wallTop = wall[1];

int wallBottom = wall[1] + wall[3];

if (playerRight > wallLeft && playerLeft < wallRight && playerBottom > wallTop && playerTop < wallBottom) {

return false; // wall collision

}

}

// Check if the new position is within the allowed bounds

return newX >= 0 && newX + 2 \* playerRadius <= 600 && newY >= 0 && newY + 2 \* playerRadius <= 600;

}

public void draw(Graphics g) {

// controls the playermodel, what way the player is looking via wasd

g.setColor(Color.YELLOW);

int startAngle = 0;

int arcAngle = 0;

switch (currDirection) {

case 'L':

startAngle = 225;

arcAngle = 270;

break;

case 'R':

startAngle = 45;

arcAngle = 270;

break;

case 'U':

startAngle = 135;

arcAngle = 270;

break;

case 'D':

startAngle = -45;

arcAngle = 270;

break;

}

g.fillArc(x, y, 40, 40, startAngle, arcAngle);

}

}

**PacmanBoard class:**

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

import java.util.List;

import javax.swing.\*;

public class PacmanBoard extends JPanel {

private PacmanPlayer pacmanPlayer;

private List<Ghost> ghosts;

private Timer timer;

private List<Coin> coins;

private int score;

private boolean paused;

private int[][] walls;

public PacmanBoard() {

pacmanPlayer = new PacmanPlayer(50, 50);

ghosts = new ArrayList<>();

ghosts.add(new Ghost(300, 300)); // Initial ghost at (300, 300)

paused = false;

// walls in map (xywh adjusting

walls = new int[][] {

{0, -16, 600, 20}, // Top border

{-16, 0, 20, 600}, // Left border

{580, 0, 20, 600}, // Right border

{0, 557, 600, 20}, // Bottom border

// x, y, width, height

// Vertical walls

{50, 50, 7, 200},

{50, 310, 7, 200},

{160, 50, 7, 200},

{175, 310, 7, 200},

{290, 150, 7, 200},

{400, 50, 7, 200},

{400, 354, 7, 150},

{525, 50, 7, 200},

{525, 354, 7, 155},

// Horizontal walls

{100, 50, 150, 7},

{100, 200, 150, 7},

{360, 300, 200, 7},

{100, 400, 150, 7},

{300, 500, 200, 7},

{250, 100, 100, 7},

{250, 250, 100, 7},

{300, 400, 100, 7},

};

setPreferredSize(new Dimension(600, 600));

setBackground(Color.BLACK);

setFocusable(true);

addKeyListener(new KeyAdapter() {

@Override

public void keyPressed(KeyEvent e) {

handleKeyPress(e);

}

});

coins = new ArrayList<>();

// 30 more coins

for (int i = 0; i < 30; i++) {

int randomX, randomY;

do {

randomX = (int) (Math.random() \* 600);

randomY = (int) (Math.random() \* 600);

} while (coinOnWall(randomX, randomY));

coins.add(new Coin(randomX, randomY));

}

score = 0;

timer = new Timer(30, e -> {

if (!paused) {

handleGameLogic();

repaint();

}

});

timer.start();

}

public void handleKeyPress(KeyEvent evt) {

char newDirection = 'L';

switch (evt.getKeyChar()) {

case 'a':

newDirection = 'L';

break;

case 'd':

newDirection = 'R';

break;

case 'w':

newDirection = 'U';

break;

case 's':

newDirection = 'D';

break;

case KeyEvent.VK\_ESCAPE:

togglePause();

break;

}

pacmanPlayer.setDesiredDirection(newDirection);

}

private void togglePause() {

paused = !paused;

if (paused) {

timer.stop();

} else {

timer.start();

}

}

public boolean collidesWithGhost(Ghost ghost) {

int playerRadius = 20;

int ghostRadius = 20;

int playerCenterX = pacmanPlayer.getX() + playerRadius;

int playerCenterY = pacmanPlayer.getY() + playerRadius;

int ghostCenterX = ghost.getX() + ghostRadius;

int ghostCenterY = ghost.getY() + ghostRadius;

double distance = Math.sqrt(Math.pow(playerCenterX - ghostCenterX, 2) + Math.pow(playerCenterY - ghostCenterY, 2));

return distance < playerRadius + ghostRadius;

}

private int collectedCoins = 0;

private void handleGameLogic() {

pacmanPlayer.move(walls);

for (int[] wall : walls) {

int wallX = wall[0];

int wallY = wall[1];

int wallWidth = wall[2];

int wallHeight = wall[3];

if (pacmanPlayer.getX() < wallX + wallWidth && pacmanPlayer.getX() + 40 > wallX && pacmanPlayer.getY() < wallY + wallHeight

&& pacmanPlayer.getY() + 40 > wallY) {

pacmanPlayer.undoMove();

}

}

for (Ghost ghost : ghosts) {

ghost.chasePlayer(pacmanPlayer, walls);

if (collidesWithGhost(ghost)) {

handleGameOver();

return;

}

}

// coin collection handling

for (Coin coin : coins) {

if (pacmanPlayer.collectsCoin(coin)) {

score++;

coin.reset(walls); // Resets the coin's position

// Spawn ghosts based on the player's score

spawnGhostsBasedOnScore();

}

}

}

private void spawnGhostsBasedOnScore() {

// One ghost is already pre-spawned (hence the immediate second, rather than first.

// Spawns a second ghost when the player gets 10 coins

if (score == 10 && ghosts.size() < 2) {

spawnGhost();

}

// Spawns a third ghost when the player gets 30 coins

if (score == 30 && ghosts.size() < 3) {

spawnGhost();

}

// Spawns a fourth ghost when the player gets 50 coins

if (score == 50 && ghosts.size() < 4) {

spawnGhost();

}

// Spawns a fifth ghost when the player gets 75 coins

if (score == 75 && ghosts.size() < 5) {

spawnGhost();

}

}

private void spawnGhost() {

int ghostRadius = 20;

int maxAttempts = 100; // if the ghost cant spawn in a safe location after 100 attempts, the ghost doesn't spawn (in the walls/border avoids)

for (int attempt = 0; attempt < maxAttempts; attempt++) {

int randomX = (int) (Math.random() \* (600 - 2 \* ghostRadius));

int randomY = (int) (Math.random() \* (600 - 2 \* ghostRadius));

// Check if the randomly generated position is actually valid

if (isValidGhostSpawn(randomX, randomY)) {

Ghost newGhost = new Ghost(randomX, randomY);

ghosts.add(newGhost);

return; // Exits the loop once a valid position is found

}

}

}

private boolean isValidGhostSpawn(int x, int y) {

int ghostRadius = 20;

int ghostLeft = x;

int ghostRight = x + 2 \* ghostRadius;

int ghostTop = y;

int ghostBottom = y + 2 \* ghostRadius;

for (int[] wall : walls) {

int wallLeft = wall[0];

int wallRight = wall[0] + wall[2];

int wallTop = wall[1];

int wallBottom = wall[1] + wall[3];

if (ghostRight > wallLeft && ghostLeft < wallRight && ghostBottom > wallTop && ghostTop < wallBottom) {

return false; // Avoiding collision with wall

}

}

return true;

}

private void handleGameOver() {

JOptionPane.showMessageDialog(this, "Game Over - You Lost!", "Game Over", JOptionPane.INFORMATION\_MESSAGE);

pacmanPlayer.reset();

ghosts.clear();

resetCoins();

score = 0;

}

private void resetCoins() {

for (Coin coin : coins) {

coin.reset(walls);

}

}

private boolean coinOnWall(int x, int y) {

for (int[] wall : walls) {

int wallX = wall[0];

int wallY = wall[1];

int wallWidth = wall[2];

int wallHeight = wall[3];

if (x >= wallX && x < wallX + wallWidth && y >= wallY && y < wallY + wallHeight) {

return true;

}

}

return false; // This boolean is attempting to stop coins spawning in points where the player can't grab them (on walls/map border etc)

}

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g);

// Draws background (black)

g.setColor(Color.BLACK);

g.fillRect(0, 0, getWidth(), getHeight());

// Draws the walls

g.setColor(Color.BLUE);

for (int[] wall : walls) {

g.fillRect(wall[0], wall[1], wall[2], wall[3]);

}

// Draws the pacman player

g.setColor(Color.YELLOW);

int startAngle = 0;

int arcAngle = 0;

switch (pacmanPlayer.getCurrDirection()) {

case 'L':

startAngle = 225;

arcAngle = 270;

break;

case 'R':

startAngle = 45;

arcAngle = 270;

break;

case 'U':

startAngle = 135;

arcAngle = 270;

break;

case 'D':

startAngle = -45;

arcAngle = 270;

break;

}

g.fillArc(pacmanPlayer.getX(), pacmanPlayer.getY(), 40, 40, startAngle, arcAngle);

// Draws ghosts

g.setColor(Color.RED);

for (Ghost ghost : ghosts) {

g.fillOval(ghost.getX(), ghost.getY(), 40, 40);

}

// Draws coins

g.setColor(Color.YELLOW);

for (Coin coin : coins) {

g.fillOval(coin.getX(), coin.getY(), 10, 10); // Adjust the size as needed

}

// Displays score

g.setColor(Color.WHITE);

g.drawString("Score: " + score, 10, 20);

}

}

**Ghost class:**

import java.awt.Graphics;

import java.util.\*;

public class Ghost {

private int x, y;

private int startX, startY;

private int speedMultiplier = 2; // Adjusts the ghosts speed (multiplier)

public Ghost(int startX, int startY) {

x = this.startX = startX;

y = this.startY = startY;

}

public void chasePlayer(PacmanPlayer player, int[][] walls) {

int playerX = player.getX();

int playerY = player.getY();

// Creates nodes for the ghost and player positions

Node ghostNode = new Node(x, y);

Node playerNode = new Node(playerX, playerY);

// Get the path from the ghost to the player using A\*

List<Node> path = findPath(ghostNode, playerNode, walls);

// Moves the ghost along the path

if (path.size() > 1) {

Node nextNode = path.get(1);

// this adjusts the speed by moving twice as fast (the multiplier)

x += (nextNode.getX() - x) \* speedMultiplier;

y += (nextNode.getY() - y) \* speedMultiplier;

}

}

private List<Node> findPath(Node start, Node goal, int[][] walls) {

PriorityQueue<Node> openSet = new PriorityQueue<>();

Set<Node> closedSet = new HashSet<>();

openSet.add(start);

while (!openSet.isEmpty()) {

Node current = openSet.poll();

if (current.equals(goal)) {

List<Node> path = new ArrayList<>();

while (current != null) {

path.add(current);

current = current.getParent();

}

Collections.reverse(path);

return path;

}

closedSet.add(current);

for (Node neighbor : getNeighbors(current, walls)) {

if (closedSet.contains(neighbor)) {

continue;

}

int tentativeGScore = current.getG() + 1;

if (!openSet.contains(neighbor) || tentativeGScore < neighbor.getG()) {

neighbor.setG(tentativeGScore);

neighbor.setH(heuristic(neighbor, goal));

neighbor.setParent(current);

if (!openSet.contains(neighbor)) {

openSet.add(neighbor);

}

}

}

}

return new ArrayList<>();

}

private List<Node> getNeighbors(Node node, int[][] walls) {

List<Node> neighbors = new ArrayList<>();

int[][] directions = { { 0, 1 }, { 0, -1 }, { 1, 0 }, { -1, 0 } };

for (int[] dir : directions) {

int newX = node.getX() + dir[0];

int newY = node.getY() + dir[1];

if (isValidMove(newX, newY, walls)) {

neighbors.add(new Node(newX, newY));

}

}

return neighbors;

}

private boolean isValidMove(int newX, int newY, int[][] walls) {

// Checks if the new position is within the allowed bounds and doesn't collide with walls

int ghostRadius = 20; // Adjusts the value based on the size of the ghost

int ghostLeft = newX;

int ghostRight = newX + 2 \* ghostRadius;

int ghostTop = newY;

int ghostBottom = newY + 2 \* ghostRadius;

// Checks if the new position collides with any walls

for (int[] wall : walls) {

int wallLeft = wall[0];

int wallRight = wall[0] + wall[2];

int wallTop = wall[1];

int wallBottom = wall[1] + wall[3];

if (ghostRight > wallLeft && ghostLeft < wallRight && ghostBottom > wallTop && ghostTop < wallBottom) {

return false; // had a collision with wall

}

}

// Checks if the new position is within the allowed bounds

return newX >= 0 && newX + 2 \* ghostRadius <= 600 && newY >= 0 && newY + 2 \* ghostRadius <= 600;

}

private int heuristic(Node a, Node b) {

return Math.abs(a.getX() - b.getX()) + Math.abs(a.getY() - b.getY());

}

void draw(Graphics g) {

throw new UnsupportedOperationException("Not supported yet."); // Generated from nbfs://nbhost/SystemFileSystem/Templates/Classes/Code/GeneratedMethodBody

}

private static class Node implements Comparable<Node> {

private int x, y;

private int g, h;

private Node parent;

public Node(int x, int y) {

this.x = x;

this.y = y;

this.g = 0;

this.h = 0;

this.parent = null;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public int getG() {

return g;

}

public void setG(int g) {

this.g = g;

}

public int getH() {

return h;

}

public void setH(int h) {

this.h = h;

}

public Node getParent() {

return parent;

}

public void setParent(Node parent) {

this.parent = parent;

}

@Override

public int compareTo(Node other) {

return Integer.compare(g + h, other.g + other.h);

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

Node other = (Node) obj;

return x == other.x && y == other.y;

}

@Override

public int hashCode() {

return Objects.hash(x, y);

}

}

public void reset() {

// Resets the ghost's position

x = startX;

y = startY;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

}

**Coin class:**

import java.awt.Color;

import java.awt.Graphics;

public class Coin {

private int x, y;

private boolean collected;

public Coin(int x, int y) {

this.x = x;

this.y = y;

this.collected = false;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public boolean isCollected() {

return collected;

}

public void setCollected(boolean collected) {

this.collected = collected;

}

public void reset(int[][] walls) {

// Resets the coin's position

// Ensures that the new position is not in the border or within walls

do {

x = generateRandomPosition(50, 550); // Adjust the range of where the coins can spawn at

y = generateRandomPosition(50, 550);

} while (isPositionInvalid(walls));

collected = false;

}

private int generateRandomPosition(int min, int max) {

return min + (int) (Math.random() \* ((max - min) + 1));

}

private boolean isPositionInvalid(int[][] walls) {

// Checks if the coin's position is within the border or collides with walls

for (int[] wall : walls) {

int wallX = wall[0];

int wallY = wall[1];

int wallWidth = wall[2];

int wallHeight = wall[3];

if (x >= wallX && x <= wallX + wallWidth && y >= wallY && y <= wallY + wallHeight) {

return true; // Coin spawned in a wall

}

}

if (x <= 20 || x >= 580 || y <= 20 || y >= 580) {

return true; // Coin spawned in the border

}

return false; // Valid position

}

public void draw(Graphics g) {

if (!collected) {

g.setColor(Color.YELLOW);

g.fillOval(x, y, 10, 10); // Adjusts the size of the coins

}

}

}