CSCU9N6 Computer Games Development

Assignment Report

Student ID: 2118616

Date: Spring 2024

Table of Contents

Table of Figures

Overview

This report aims to critically evaluate the implementation of my game for CSCU9N6. It provides a brief overview of how to play the game and what to expect, then it addresses the game's successes and weaknesses.

Launching the Game

Firstly, I would like to highlight that the program starts from **Launch.java**. During testing, I have simply ran the game from this class using my IDE's run button, and compilation has been successful.

On successful compilation, the user is presented with a "Level Select" screen. From this screen, the user is able to choose which level they wish to play. Currently, there are two levels to choose from. It clearly looks slightly bland with a lot of empty space (see Figure 1), but this space is intended for future levels as this game is a prototype.

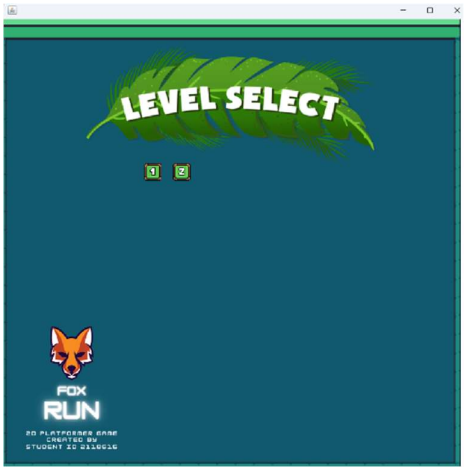

Figure 1 Level Select screen

The user must click on their choice of level. This is self-explanatory, but for sake of clarity: click the button with "1" to play level 1, and button "2" for level 2. This invokes the mouseClicked handler shown at Figure 2. The buttons are loaded in as Sprites, so to detect if a user has clicked one of these Sprites, I:

- take a note of the X and Y coordinate of the mouse click,
- convert the coordinates to a Point,
- convert each button to a Rectangle, then,
- flip a Boolean flag if the Rectangle contains the click (Point).

```
/**
 * Handler for the mouseClicked event.
 * Converts coordinates of user's click to a Point.
 * Converts bounds of each button to a Rectangle.
 * Evaluates if Point is within one of these Rectangles
 * and flips relevant boolean flag to true if it is.
 * Boolean flag used to control a level being constructed.
 *
 * @param e the mouse click.
 */
@Override
public void mouseClicked(MouseEvent e) {
    int mouseClickX = e.getX();
    int mouseClickY = e.getY();
    Point click = new Point(mouseClickX, mouseClickY);
    Rectangle button1Rect = getB1Rect();
    Rectangle button2Rect = getB2Rect();
    if(button1Rect.contains(click)) button1Clicked = true;
    else if (button2Rect.contains(click)) button2Clicked = true;
}//end method mouseClicked
```

User-defined methods

Figure 2 MainMenu.java method mouseClicked

I have included user defined methods to convert the two buttons to Rectangles. I have omitted these from this report to save space, but these can be found under the mouseClicked method of MainMenu.java. The Boolean flags button1Clicked and button2Clicked are used to control what Level is constructed (Figure 3). Overall, I think MainMenu.java is fairly well organized.

The game controls are kept simple in a bid to enhance the user experience. The arrow keys are used to move the player: left arrow to move left, right to move right, and up to move up.

```
/**
 * Method to override update method in GameCore.java.
 * Method will update state of the game and/or animations
 * based on the amount of elapsed time that has passed.
 * This method controls what specific Level is constructed
 * dependent on what button the user clicks on.
 */
public void update(long elapsed) {
    if(button1Clicked) {
        this.dispose();
        Level1 levelOne = new Level1();
    } else if (button2Clicked) {
        this.dispose();
        Level2 levelTwo = new Level2();
    }//end if else if
}//end method update
```

Figure 3 Boolean flags control level construction

Levels

One aspect of the game which I think works really well is how Levels are defined and implemented. We already have a very useful and necessary GameCore class, and as I progressed in developing the second level, I found myself repeating code. Therefore, I defined another abstract class Level.java. It is abstract because a Level by itself is not full and complete – it needs extra details before a Level can be constructed. This class extends GameCore as it contains necessary code to make any Level work. Regardless of what Level the user plays, the player Sprite remains the same, as do the controls to move the player. Also common to all levels are two Boolean flags: one to indicate if the user has failed (gameOver), and one to indicate if the level is complete (levelComplete).

The power of inheritance were clear when considering class variables representing game resources. If I implemented two separate classes representing Levels 1 and 2, the following 12 fields would appear in both classes:

```
//Game resources
protected Animation idle, run, hurt;
protected Sprite player = null;
protected LinkedList<Sprite> bgLayers = new LinkedList<Sprite>();
protected ArrayList<Sprite> pickups = new ArrayList<Sprite>(); //gems, cherries etc.
protected ArrayList<Enemy> enemies = new ArrayList<Enemy>();
protected LinkedList<Sprite> hearts = new LinkedList<Sprite>(); //level starts with 4 hearts. Enemy collision removes a heart.
protected ArrayList<Sprite> clouds = new ArrayList<Sprite>(); //David's clouds :)
protected ArrayList<Tile> collidedTiles = new ArrayList<Tile>(); //tiles a sprite has collided with.

protected TileMap tmap = new TileMap(); //the tilemap which the player Sprite interacts with
protected MIDI bgTrack; //MIDI track playing on loop

protected long total; //the points a player has gained.
protected long collisionCount = 0; //number of enemy collisions. If 4, gameOver flips to true.
```

Figure 4 Benefits of inheritance include reduction in code/increase in readability

By implementing Level.java that both Level1.java and Level2.java extend, we reduce 24 lines of code to 12. This increases the efficiency of the game as the program is more concise and maintainable, limiting the amount of redundant code. If I need to add in a new Animation for the player, I can simply add in another class variable to Level.java and this change will propagate through to all subclasses of Level. This means that even if I created another 100 levels, I would still only have to make one addition, and all 100 subclasses would still have this new Animation. Level.java contains various methods that makes the source code significantly more readable and understandable than previous, when I implemented each Level as their own class.

Sprite to Sprite Collisions

Some of these methods are responsible for collision detection and handling. I am pleased with how I have implemented Sprite to Sprite collision handling, as this felt daunting at the start of semester. Not only does Sprite to Sprite collision detection/handling work, but it is efficient as well. Though the game contains code to implement bounding box collisions, I have opted to use bounding circle collisions. I started by creating a method that checks if the circles around two Sprites intersect (Figure 5). I tested this method between my player Sprite and Enemy sprite and, though it worked fine and would have sufficed, I considered the end user's experience when trying to dodge enemies. I am on the user's side, and want the user to be able to complete the level successfully. Therefore, I implemented another method in Level.java, refinedBoundingCircleCollision (Figure 6). This is where efficiency is evidenced. The method first checks if an initial collision is suspected by calling boundingCircleCollision: if it returns false, refinedBoundingCircleCollision immediately returns false, and no further calculations are

```
/**
 * Method to check for Sprite to Sprite collisions using a bounding circle.
 * Used to detect an initial collision.
 *
 * @param s1 the Sprite to check.
 * @param s2 the other Sprite to check.
 * @return true if Sprites' bounding circles intersect; false otherwise.
 */
protected boolean boundingCircleCollision(Sprite s1, Sprite s2) {
    //calculation from centre of sprite; not top left x,y
    int dx, dy, minimum;
    dx = (int) (s1.getX() - s2.getX());
    dy = (int)(s1.getY() - s2.getY());
    minimum = (int)(s1.getRadius() + s2.getRadius());
    return (((dx * dx) + (dy * dy)) < (minimum * minimum));
}//end method boundingCircleCollision
```

Figure 5 Level.java - bounding circle collision

performed; however, if it returns true, it performs the test again on smaller circles through the following steps:

- Calculate smaller radius using radii of both Sprites:
    - to do this, we get the minimum radius of Sprite s1 and Sprite s2, then half this value (this could be adjusted by changing 0.5 to another value). I make use of Java's Math class to get the minimum value by invoking Math.min.
- Calculate distance between the center of each smaller circle:
    - subtract Sprite s2's X coordinate from Sprite s1's X coordinate,
    - subtract Sprite s2's Y coordinate from Sprite s1's Y coordinate,
    - then use Pythagorean theorem to compute the distance. I make use of Java's Math class again here to calculate the square root by invoking Math.sqrt.
- Evaluate if the distance value (dist) – which is the distance between the centers of each smaller circle – is less than the smaller radius value multiplied by 2:
    - if true, there is a refined collision between two Sprites, and the method returns true; otherwise, there is no refined collision and the method returns false, with no more checks being performed.

```
/**
 * Method to first check if there is an initial collision between two Sprites.
 * If initial collision suspected, test performed on smaller circles; otherwise
 * method returns false so resources not wasted on performing unnecessary
 * calculations.
 *
 * @param s1 the Sprite to check.
 * @param s2 the other Sprite to check.
 * @return true if a refined collision is detected; false otherwise.
 */
protected boolean refinedBoundingCircleCollision(Sprite s1, Sprite s2) {
    if(boundingCircleCollision(s1, s2)) {
        //calc new radius
        int smallerRadius = (int)(Math.min(s1.getRadius(), s2.getRadius()) * 0.5);
        //calc distance between centres
        int dx = (int)(s1.getX() - s2.getX());
        int dy = (int)(s1.getY() - s2.getY());
        int dist = (int)Math.sqrt(dx*dx + dy*dy);
        //refined collision detected?
        if(dist < smallerRadius *2) return true;
        //else no refined collision has been detected
        else return false;
        //no initial collision detected? return false
    } else return false;
}//end method boundingCircleCollision
```

If this returns false, the method returns false; resources not wasted on unnecessary computations.

Figure 6 Level.java bounding circle collision analysis

I deliberately do not use refinedBoundingCircleCollision for detecting collisions between the player Sprite and "pickup" Sprites (e.g. gems). This is because I do not want the user to get frustrated when trying to deliberately collide with these specific objects. I include an overloaded method to deal with these collisions (Figure 7), and I have defined this method so that it can deal with a **List** of Sprites (such as a List of gems). This contributes to making the game easy to extend, and is a more accurate way of dealing with player-to-"pickup" collisions because the "pickup" should be removed from the List if the player collides with it; the "pickup" should **not** simply be hidden (using the hide() method in Sprite.java), as I initially tested. If the "pickup" Sprite is just set to hide, the sound effect plays repeatedly because although it is not visible to the user, the player is in fact repeatedly colliding with the hidden "pickup" object.

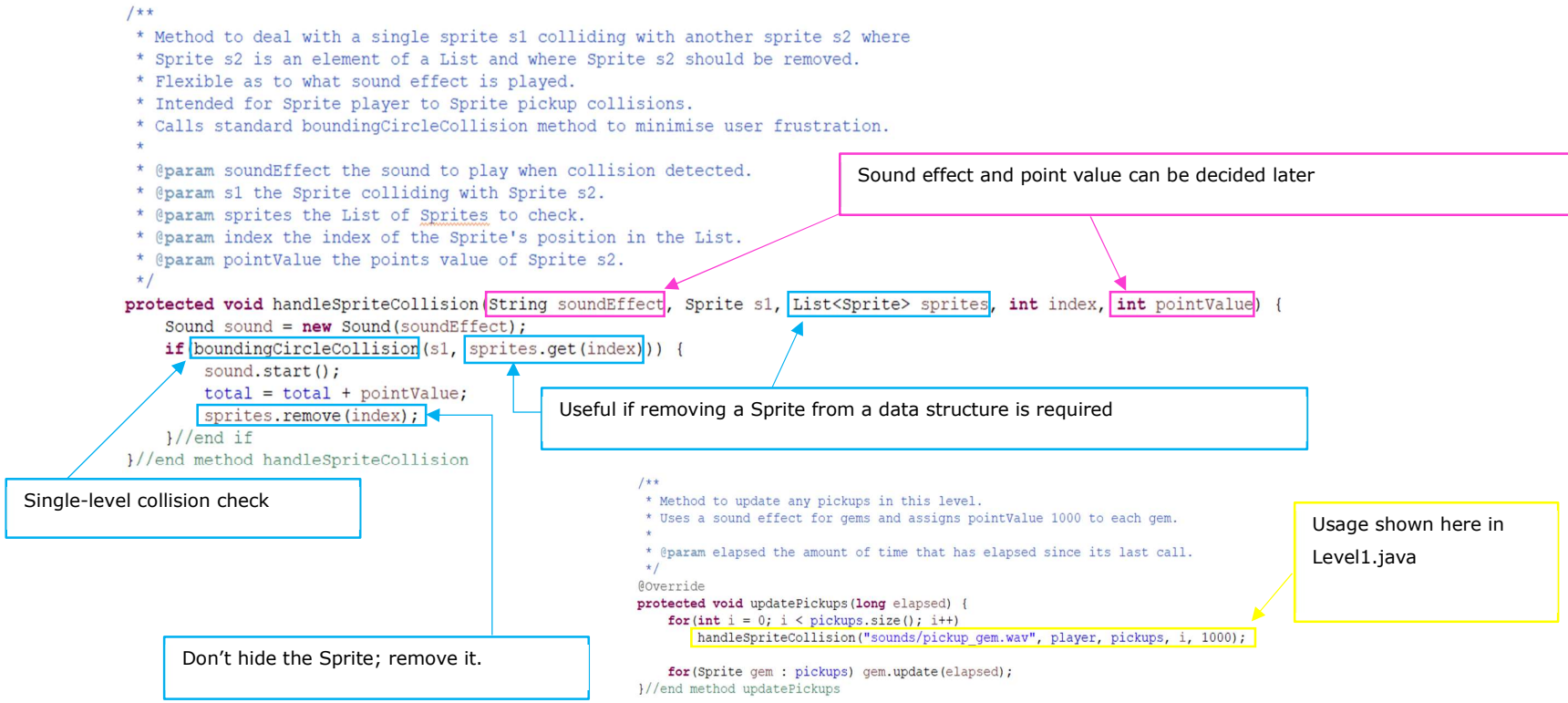

Figure 7 Level.java overloaded method for Sprite-to-Sprite collision where s2 is part of a List

Enemies

One common form of Sprite to Sprite collisions is a player colliding with an "enemy", another Sprite not controlled by the user. The implementation of Enemy Sprites is done to a fair extent, although this aspect of the game could be improved upon in terms of interacting with the environment.

Firstly, the benefits of inheritance were clear when considering whether to create an Enemy class. An Enemy is just another Sprite, but there's some additional behaviours that I would like only Enemy Sprites to perform and I wanted it to be very clear that these behaviours do not apply to all Sprites in the game. I believe this approach makes it easier to extend the game in future and easier to maintain also.

The class Enemy.java has three class variables used to determine the area an Enemy will patrol. They are fairly self-explanatory and represent the minimum (left most) and maximum (right most) position an Enemy will patrol. The value RANGE is used to determine when an Enemy will start to pursue the player. This is set to 250, so if the player comes within 250 pixels of an Enemy, that Enemy will chase the player at a quicker speed than its patrol speed. This is a constant variable and can be easily changed.

```
public class Enemy extends Sprite {

    /*================================CLASS VARIABLES================================*/

    private int patrolRangeMin;
    private int patrolRangeMax;
    /*
     * constant RANGE used to determine when
     * Enemy starts to chase player (RANGE number of pixels)
     */
    private static final int RANGE = 250;
```

Figure 8 Enemy.java class variables

The method patrol() is kept simple. It specifies that if this Enemy's X position is <= this Enemy's patrolRangeMin value, it should move to the right; if X postion is >= this Enemy's patrolRangeMax, its velocity is set to negative to change its movement direction to the left. In addition to accessor and mutator methods, Enemy.java also contains a method to calculate the Euclidian distance between it and a Sprite s (intended to be the player). It checks if the distance along the X-axis is within the constant RANGE value, and checks if the distance along the Y-axis is reasonable (between 0 and 20). If the check is true, the Enemy pursues the Sprite s (the player). To pursue the player, we must determine the direction which the Sprite s is approaching the Enemy. This is calculated by checking if Sprite s's X position is less than this Enemy's X position; if it is, the player is approaching from left to right, so this Enemy's velocity is set to negative in order to chase to the left. Otherwise, the Enemy chases to the right, thus a positive velocity value is set. Its velocity is also set to a higher than normal speed.

```
/*================================PATROLLING AND CHASING================================*/

/**
 * Method to move Enemy along a straight line within a range.
 */
public void patrol() {
    if(getX() <= getPatrolRangeMin()) setVelocityX(0.02f);//move right
    else if(getX() >= getPatrolRangeMax()) setVelocityX(-0.02f);//else move left
}//end method patrol

/**
 * Method to pursue a Sprite s.
 * If Sprite s is within RANGE (final int) number of pixels of this Enemy,
 * Enemy will follow Sprite s and move faster.
 *
 * @param s the Sprite to chase.
 */
public void chase(Sprite s) {
    float distanceX = Math.abs(s.getX() - getX());
    float distanceY = Math.abs(s.getY() - getY());
    //if player is within range, chase player
    if(distanceX <= RANGE && (distanceY >= 0 && distanceY <=20)) {
            //if player is to the enemy's left
            if(s.getX() < getX()) setVelocityX(-0.075f); //enemy chase to left
            else setVelocityX(0.075f); //otherwise enemy chase to right
    } else patrol(); //else the player is not in distance. Enemy should patrol.
}//end method chase
```

Figure 9 An Enemy will patrol and chase a Sprite

Improvements

Sound

Although a lot of time and effort has went into this assignment, there are many areas that need work. As someone who has never been a gamer, I have found it difficult to implement appropriate sound components to the game. Sound can make a big difference to the user's experience of a

game, and I think this is probably the poorest aspect of the submission. Sound effects are of course used, and the game successfully plays a background MIDI track. This track repeats until one of the Boolean flags gameOver or levelComplete are flipped to true, and so there is some evidence of implementing sound in the game. There are also appropriate sound effects played when a player collides with another Sprite, or if the player collides with a coin or end-of-level-flag tile. I struggled to come up with ideas as to what kind of sound filter could be used and what kind of event should trigger such a sound filter. I have attempted to implement two filters: a fade filter and an echo filter. The Sound class has been tweaked so that when a Sound object is constructed, the developer can easily specify what filter, if any, should be applied:

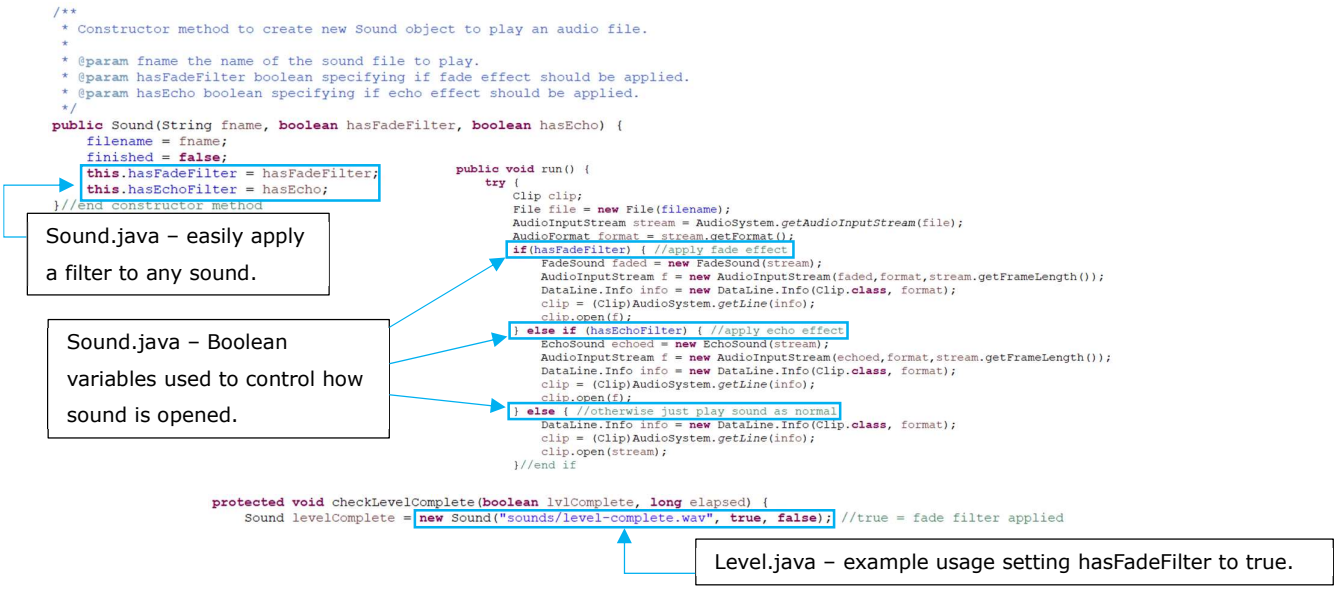

Figure 10 Sound.java and Level.java - applying a fade filter

The fade filter is more successful than the echo, and it is applied to the sound played when levelComplete is flipped to true. The volume starts close to silent when the user completes the level:

```
// Start volume at 0.05 (almost silent)
float volume = 0.05f;
```

Figure 11 FadeSound.java volume start value

Then the previously calculated change value is added to volume:

```
// Increase the volume
volume = volume + change;
```

Figure 12 FadeSound.java change value increases volume

Although the fade filter works successfully, I do not think it adds much to the game as a whole. The echo filter also works, but it created an effect that was more annoying for the user than adding to the game's atmosphere. To try and resolve this, I adjusted the fraction by which the original audio is divided by to create a less noticeable echo effect:

```
// Get the delayed value, add an echo to it
short original = getSample(sample, delayed);
// Apply echo - smaller fraction = less intense echo
echoed = (short)((original+amp)/4);
// Now put the new value back in the sample array.
setSample(sample,delayed,echoed);
```

Figure 13 EchoSound.java applying echo effect

I attempted to apply the echo filter to the player running sound, but this has been commented out to save frustration:

```
case KeyEvent.VK_RIGHT :
    //Sound runSound = new Sound("sounds/run_sound.wav", false, true); //true = echo applied
    //runSound.start();
    moveRight = true;
    moveLeft = false; //we can't move right and left at the same time
    leftReleased = false;
    break;
case KeyEvent.VK_LEFT :
    //Sound run = new Sound("sounds/run_sound.wav", false, true); //true = echo applied
    //run.start();
    moveLeft = true;
    moveRight = false; //we can't move left and right at the same time
    leftReleased = false;
    break;
```

Figure 14 Level.java keyPressed method

With this in mind, the sound component of the game needs more refinement.

Game World

In addition, the parallax scrolling of the background is not terrific and the game world in general needs some refinement. It does work, and I think I have done well to implement basic parallax scrolling: Level.java contains a class variable which is a List of Sprites (bgLayers) and a method drawBackground which can handle drawing a background of an unknown number of layers. This means that subclasses can have differing numbers of background layers (Level1 here has 5 and Level2 has 7), but this single method will achieve the task of drawing these to screen and repeating each layer as the view moves.

```
protected LinkedList<Sprite> bgLayers = new LinkedList<Sprite>();
/**
 * Method to draw background to screen.
 * This method can handle varying numbers of background layers.
 * Each background layer is stored in class member bgLayers (LinkedList).
 * While the bgX value is less than the width of the window, it will
 * set the offsets for the current background layer, draw it to screen,
 * then update bgX so that it moves to the next position to draw the current
 * background at. The bgX value is reset on exiting the while loop in
 * readiness for the next layer.
 *
 * @param xo the x-offset value to shift the view by.
 * @param yo the y-offset value to shift the view by.
 * @param g the Graphics2D object to draw with.
 */
protected void drawBackground(int xo, int yo, Graphics2D g) {
    int bgX = xo;
    for(int i = 0; i<bgLayers.size(); i++) {
        while(bgX < getWidth()) {
            bgLayers.get(i).setOffsets(bgX, yo);
            bgLayers.get(i).draw(g);
            bgX += bgLayers.get(i).getWidth();
        }//end while
        bgX = xo;
    }//end for
}//end method drawBackground
```

Figure 15 Level.java method to draw a List of background layers (Sprites)

However, the game would appear much better if there was more control around when the world moves. For example, when the player approach the right edge of the screen, although the player is prevented from running past the screen edge, the whitespace is still visible:

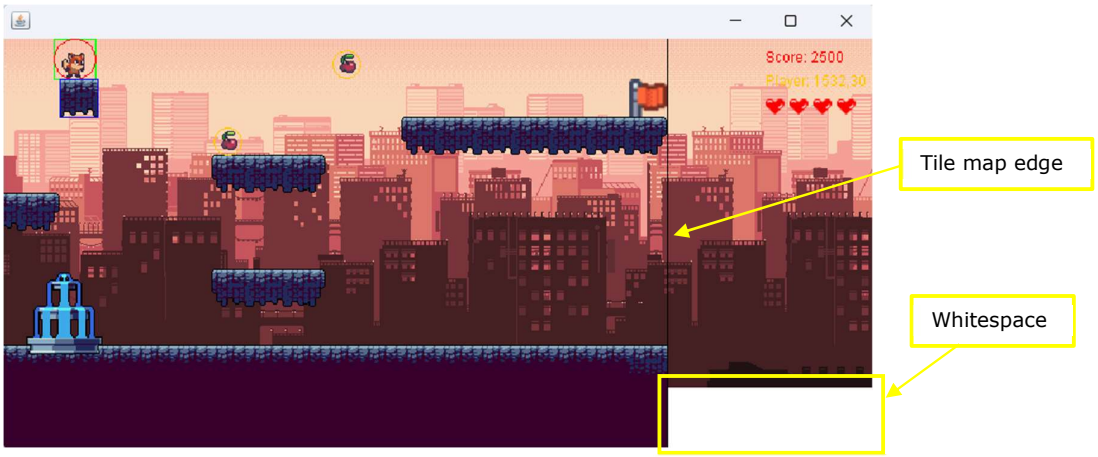

Figure 16 Game world needs improvement

Another aspect I would really like to have improved if time permitted is more accurate tile collision handling. I made an attempt to implement tile collision detection and handling for all Sprites, not just the player, but it did not work quite as successfully for non-player controlled sprites. See, for example, the method checkMidLeft in Level1.java:

```
/**
 * Method to check the tile positioned at Sprite's mid-left.
 * Method checks for special tiles (e.g. coins and spikes).
 * If no "bad" tile collisions, then Sprite s is repositioned.
 *
 * @param sx the Sprite's X coordinate.
 * @param sy the Sprite's Y coordinate.
 * @param tileWidth the width of a tile used to calculate number of
 * tiles across x-axis that sprite is positioned at.
 * @param tileHeight the height of a tile used to calculate number of
 * tiles down y-axis the sprite is positioned at.
 * @param coin the sound effect to be played if Sprite collides with a coin tile.
 * @param s the Sprite to check.
 */
public void checkMidLeft(float sx, float sy, float tileWidth, float tileHeight, Sound coin, Sprite s) {
    //MID-LEFT
    int xtile = (int)((sx / tileWidth));
    int ytile = (int)(((sy + s.getHeight()/2) / tileHeight));
    Tile ml = tmap.getTile(xtile, ytile);
    if (ml != null && ml.getCharacter() != '.'){ // If it's not a dot (empty space), handle it
        //s.stop();
        s.setX((xtile+1)*tileWidth);
        collidedTiles.add(ml);
        if(s==player) {
            if(isFlag(ml)) levelComplete = true;
            if(isSpike(ml)) gameOver = true;
            if(isWater(ml)) gameOver = true;
            if(isCoin(ml)) {
                increaseScore(100);
                replaceCoin(coin, xtile, ytile);
            }
        }//end if s==player
    }//endif
}//end method checkMidLeft
```

Figure 17 Level1.java method checkMidLeft

I include an if statement to check if the Sprite s is the player Sprite, as I do not want any other Sprite to have the ability to flip the game state flags levelComplete and gameOver. In Level.java where Enemy Sprites are updated, I have commented out a call to checkTileCollisions using each Enemy Sprite:

```
/**
 * Method to update Enemy Sprites.
 * Iterates through enhanced for loop so all Enemies
 * continually patrol and pursue the player Sprite if
 * player comes within certain range of Enemy.
 *
 * @param elapsed the amount of time elapsed since this method was last called.
 */
protected void updateEnemies(long elapsed) {
    for(Enemy e : enemies) {
        //checkTileCollision(e, tmap, elapsed);
        e.chase(player);
        handleSpriteCollision(player, e);
        e.update(elapsed);
    }//end for
}//end method updateEnemies
```

Figure 18 Level.java tile collisions with Enemies

I was trying to implement tile collision handling for Enemy Sprites as well as the player Sprite. However, this had undesired effects such as an Enemy only moving left and not appearing to patrol as normal. I should note that there are two additional classes in my game that are not in use – SpriteCollisionHandler.java and TileCollisionHandler.java. I found my game had less issues where I implemented collision detection/handling within Level.java and within the subclasses themselves. My intention was to construct a SpriteCollisionHandler and TileCollisionHandler object in Level.java and any relevant subclasses, so the code for collision detection/handling is separated and organized accordingly.

Other Bugs

The other notable bug to address is in Level 1. From the starting position, if the player turns and moves to the left, a "bad" tile collision is registered. When debugging, I discovered that the gameOver Boolean flag was set to true because the player collided with either a spike or water. However, it is clear that there is no spike or water to the player's left. In fact, the player is simply at the left edge of the screen. If the player moves upwards and runs to the left screen edge, the method handleLeftEdge in Level.java is invoked correctly. I am yet to figure out what is causing this.

Figure 19 Level 1 bug

Conclusion

Even though there are several areas that need refinement and improvement, I have applied a significant amount of time and effort to my game. This assignment has presented many challenges, but I have enjoyed tackling each component of the game and it has been an excellent opportunity to exercise problem-solving skills. Overall, there are several ways in which my game can be easily expanded – it would not require significant changes to add in another 10 levels, for example; the foundations are already there for that kind of expansion. I make an effort to use Lists of Sprites rather than individually named variables – the only individual named Sprite, apart from background layer Sprites, is the player. Where the game could use more refinement is in computer controlled sprites interacting with the environment and in utilizing sound more effectively. I now have a new-found respect for game developers.

Sources

Game assets have been taken from various websites.

The player, possum, gem, and cherry sprites were sourced from: https://ansimuz.itch.io/sunny-land-pixel-game-art

Tiles in Level Select screen and "green thing" Enemies were sourced from: https://craftpix.net/freebies/free-simple-platformer-game-kit-pixel-art/

"Level Select" header sourced from: https://craftpix.net/freebies/free-jungle-cartoon-2d-game-ui/

Logo in Level Select screen created using Canva: https://www.canva.com/

Tiles in Level 1 were sourced from: https://kenney.nl/assets/pixel-platformer

Tiles in Level 2 were sourced from: https://craftpix.net/freebies/free-green-zone-tileset-pixel-art/

Background in Level 1 sourced from: https://craftpix.net/freebies/free-swamp-game-tileset-pixel-art/

Background in Level 2 sourced from: https://craftpix.net/freebies/free-city-backgrounds-pixel-art/

Sound effects sourced from: https://pixabay.com/

Background MIDI track created using Anvil Studio: https://www.anvilstudio.com/