SWENG452W: Real-Time Embedded Systems

Final Project Report

MissileCommand Pi

Anthony Vallin, aav5195@psu.edu

ABSTRACT
This project is a recreation of the classic “Missile Command” game from Atari for the Raspberry Pi 4B. The goal of the game is for the player to defend their cities from an incoming ballistic missile attack, using only the blast radius of an anti-ballistic missile for defense. The player controls the anti-ballistic missile path using the x & y coordinates provided by their mouse position. To defend, the player launches a missile from a missile battery using their left mouse button, which starts an explosive blast. An explosive blast destroys any enemy missiles caught in the radius. The visual display shows player cities, player missiles, enemy missiles, and the player missile cursor.

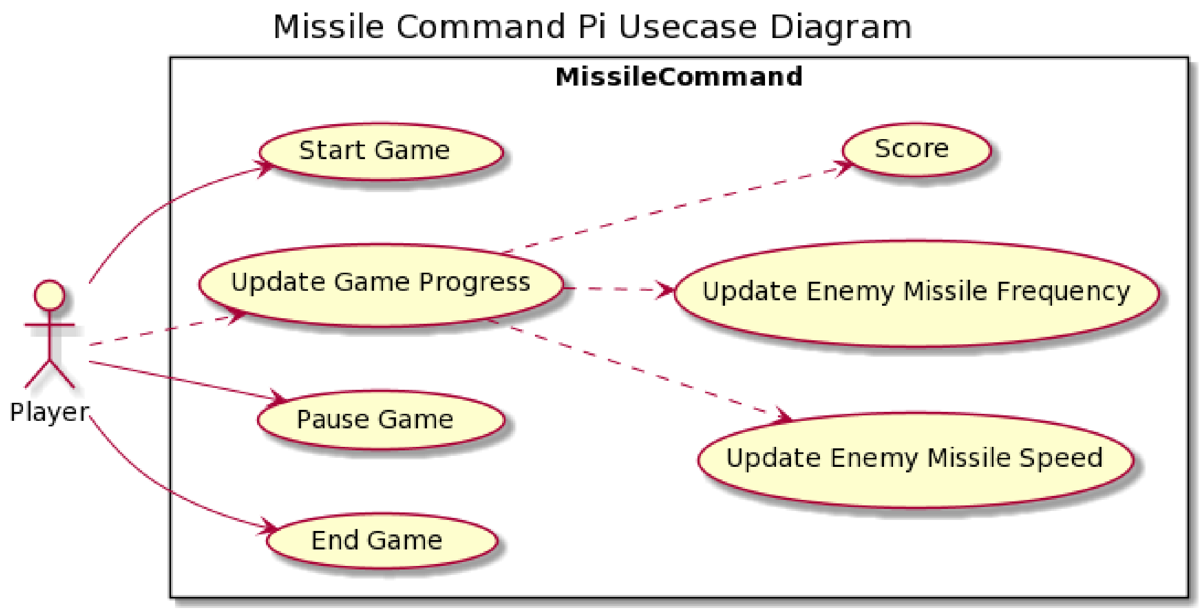
1.0 INTRODUCTION

The initial motivation behind the Missile Command Pi project was to answer two questions: Is it possible to run a Java Virtual Machine based game on the Raspberry Pi and could said game also be developed on the Raspberry Pi? Some may see the notion that Java could be a viable alternative to C/C++ in a low-powered embedded programming environment with disbelief. However, the Java concept of ‘write once, run everywhere”, i.e., portability, was worth exploring for embedded devices, particularly the Raspberry Pi.

My Missile Command Pi project required an OS change and a hardware addition to get the development environment working on the Raspberry Pi. The Raspberry Pi 32-bit OS did not support an IDE with the required features needed to handle the scope of the project. Manjaro plasma on ARM 64-bit OS replaced the original PI OS, proving to be in my opinion a superior alternative. Additionally, I replaced the 32GB flash drive with a 128 GB SSD (a 128GB flash drive would have worked too). The memory upgrade sped up IDE compile/build times and prevented intermittent low memory IDE crashes.

2.0 FUNCTIONAL SPECIFICATION

The use case diagram shows three use cases available to the player and a system interaction use case based on player progress. The player can start a new game, pause the game in progress, or end the game. Also, the game updates the enemy missile speed, enemy missile frequency, and the score as the player progresses in the game. The score increases for every successful enemy missile destroyed and decreases for every enemy missile that reaches a city.


Missile Command Pi Usecase Diagram
MissileCommand
Start Game
Score
Update Game Progress
Update Enemy Missile Frequency
Player
Pause Game
Update Enemy Missile Speed
End Game

3.0 SOFTWARE DESIGN

Missile Command Pi requires two inputs from the player: a left mouse button click and the x & y coordinates of current mouse position. Both of which are processed by overriding the MouseListener's mouseClicked method. The x & y coordinates of the left button mouse click act as the terminus for the player missile path *and* center reference points for the explosive collision detection radius.[1]

The GameGraphics class is responsible for drawing the output on the JFrame, which was originally created by the GameFrame class. Display output includes player missiles, enemy missiles, cities, turret, and explosions.

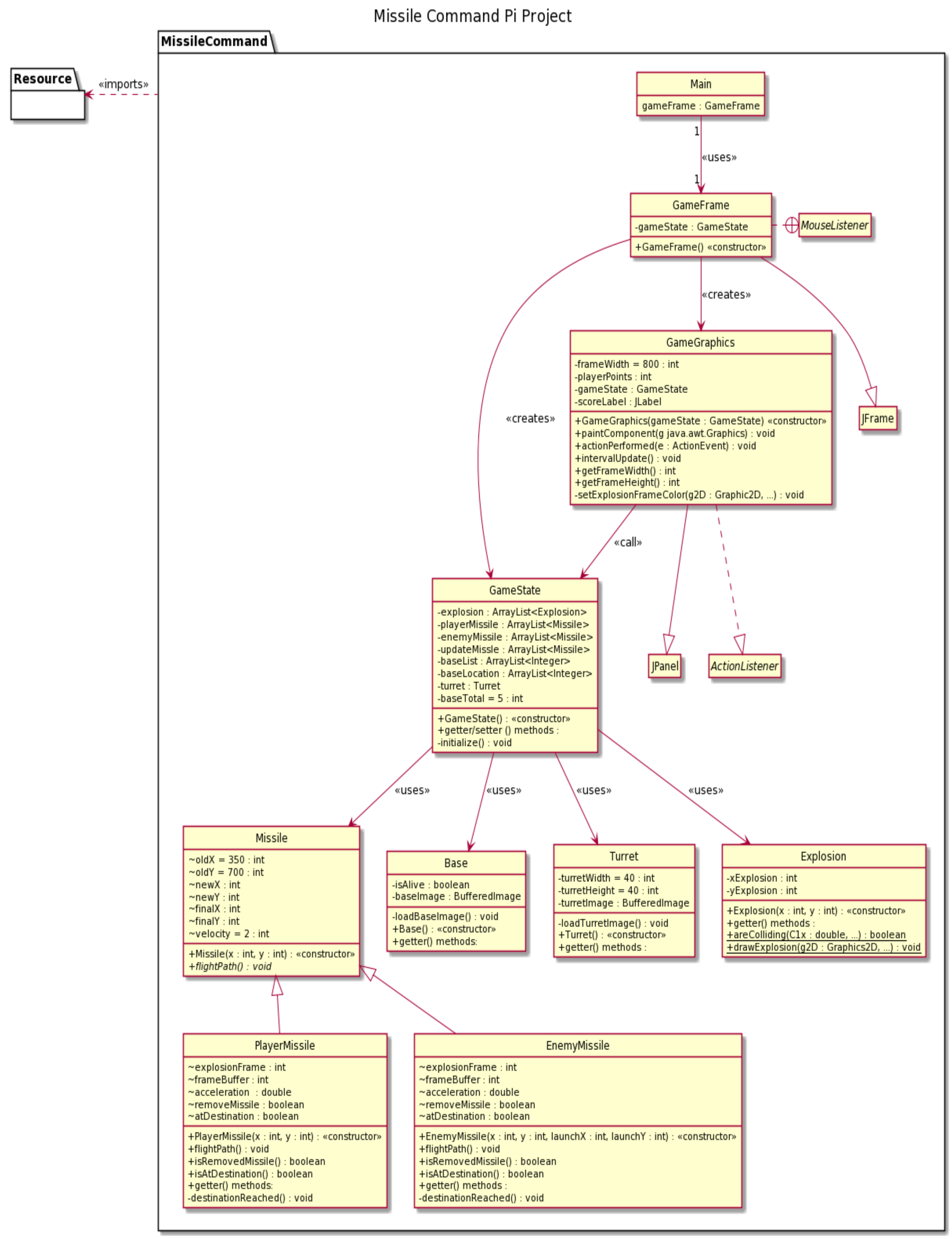
Player and enemy missiles are redrawn "frame" by "frame" on the display via the intervalUpdate() method found in the GameGraphics class. The implementation is simple. A new ArrayList is populated with updated x & y coordinates, calculated via the flightpath methods found in the enemyMissile and playerMissile classes. Flight path data is required to continuously paint the missile along its path. Last, the "old" coordinates in the original player and enemy missile lists are replaced by the updated ArrayList.[2]

Player and enemy missile collisions are calculated in the explosion class via the areColliding function. To see if a player and enemy missile collided, the function calculates the distance from the centers of the two missiles and checks if it's greater than the sum of the two radii.[3]

A timing schedule inside of the GameGraphics class determines frequency of enemy missile launches, i.e., missile frequency increase after a predetermine time block passes. The current implementation has a relatively short release timing schedule, which was done for testing/demonstration purposes.[4]

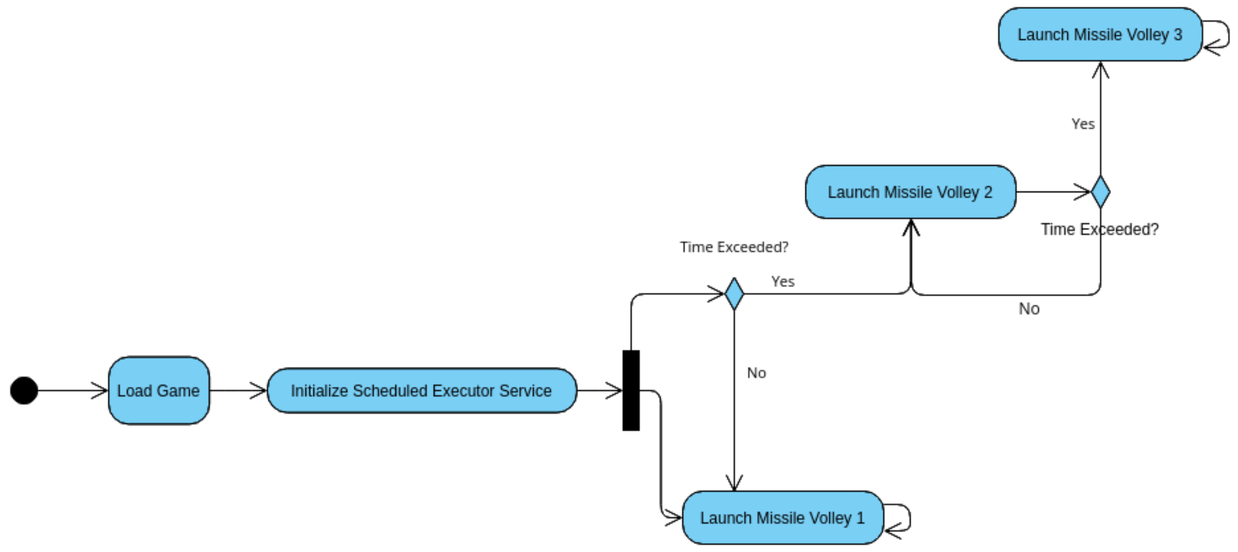
Finally, the GameState class acts as the central repository for game assets, e.g., missile lists, base lists, turrets. Additionally, the GameState class contains the base locations.[5]

UML Class Diagram:



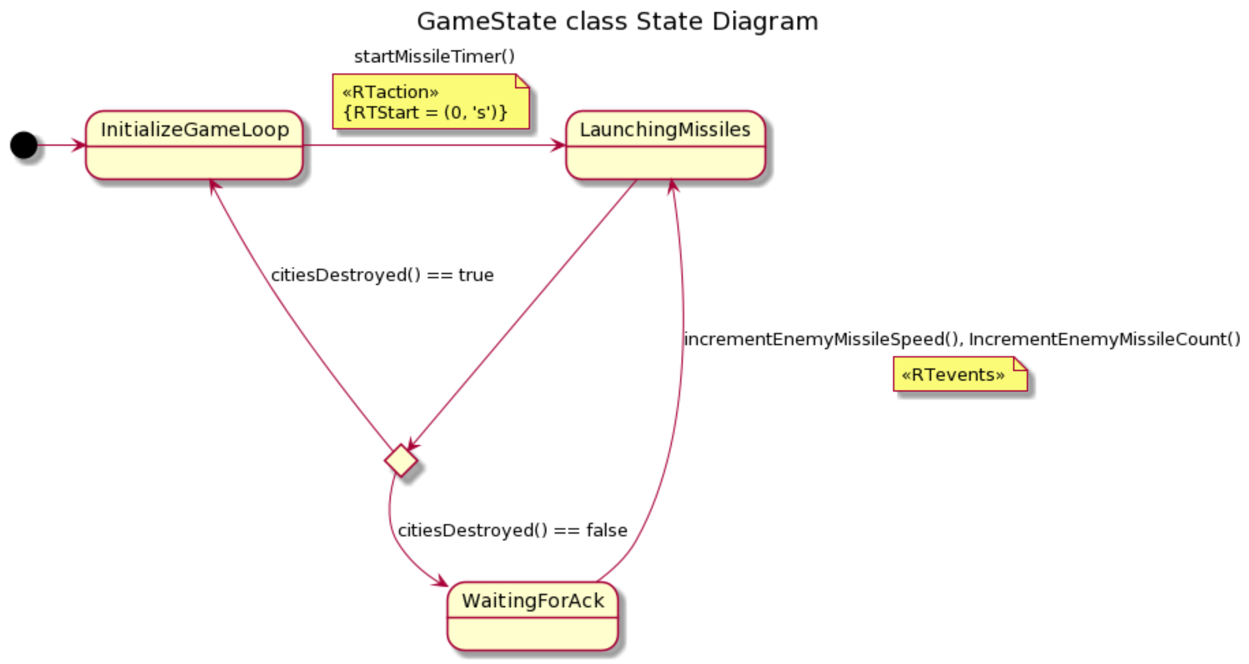
Activity Diagram

The activity diagram describes how enemy missile launches are handled in game



State Diagram

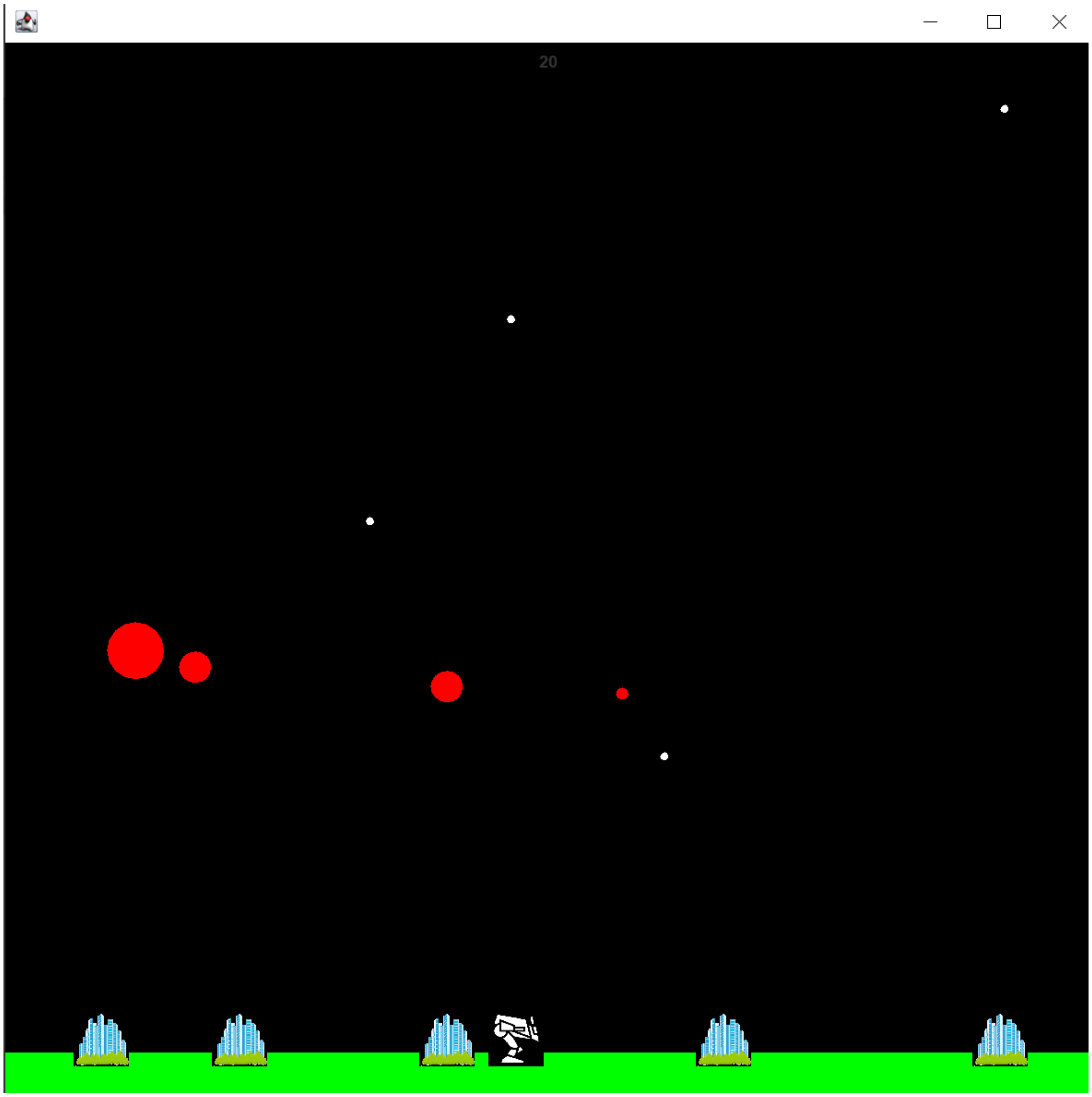
The state diagram describes the behavior of states dealing with the missile timing update events.


GameState class State Diagram
startMissileTimer()
«RTaction»
{RTStart = (0, 's')}
InitializeGameLoop
LaunchingMissiles
citiesDestroyed() == true
incrementEnemyMissileSpeed(), IncrementEnemyMissileCount()
«RTevents»
citiesDestroyed() == false
WaitingForAck

4.0 RESULTS

4.0.1 Functional Features

The GUI displays all required output: player missiles, enemy missiles, static images (turret and cities), player score, and explosions. Additionally, the core game functions are implemented: missile collision detection, player & enemy missile trajectories, score keeping, and a basic timing scheme for enemy missile launches.



4.0.2 Unresolved Issues

There are two main bugs in the game: First, the collision detection system intermittently produces inaccurate collisions between enemy and player missiles. For example, an enemy missile will register as destroyed even though it is beyond the radius of a player missile explosion. Second, the explosion animation does not trigger for enemy missiles when a city is hit or when destroyed by a player missile.

4.0.3 Non-implemented Features

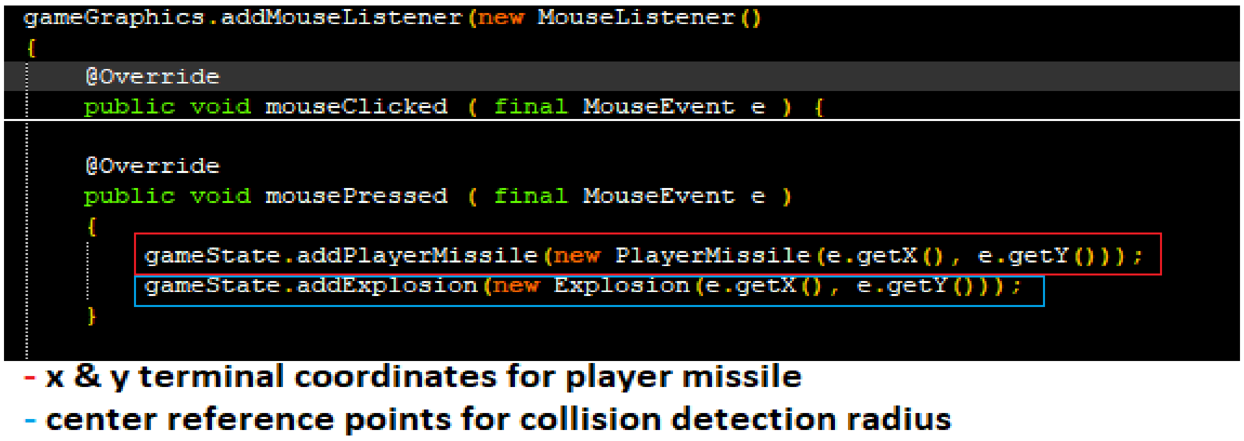
The menu system (start, pause, quit) did not make the final cut. Unfortunately, core game mechanics required more time than anticipated, which left little to no time for the menu system (see use case diagram). Additionally, the difficulty mechanic of increasing enemy missile speed and count when all cities are destroyed was removed from the implementation (see state diagram). The primary reason for its removal had to do with the framerate stutter of the enemy missile as it flew across the frame. A specific acceleration value was required to keep the enemy missile "fluid" as it flew across the display.

6.0 CONCLUSIONS

I set out to answer two questions in my project: Can a JVM based game run on a Raspberry Pi and can said game be developed directly on the Pi? The answer is yes, but you're not going to like it and there are better options available. A big drag on the project was the lack of a non-resource intensive ide for java on the 32-bit Raspberry Pi OS. An OS change and a storage increase were required to develop the program on the Pi (for this project a remote host was purposely not used). Finally, the Pi did not run the developed program smoothly, stuttering as the missiles, player and enemy, flew across the screen.

APPENDIX/ATTACHMENTS

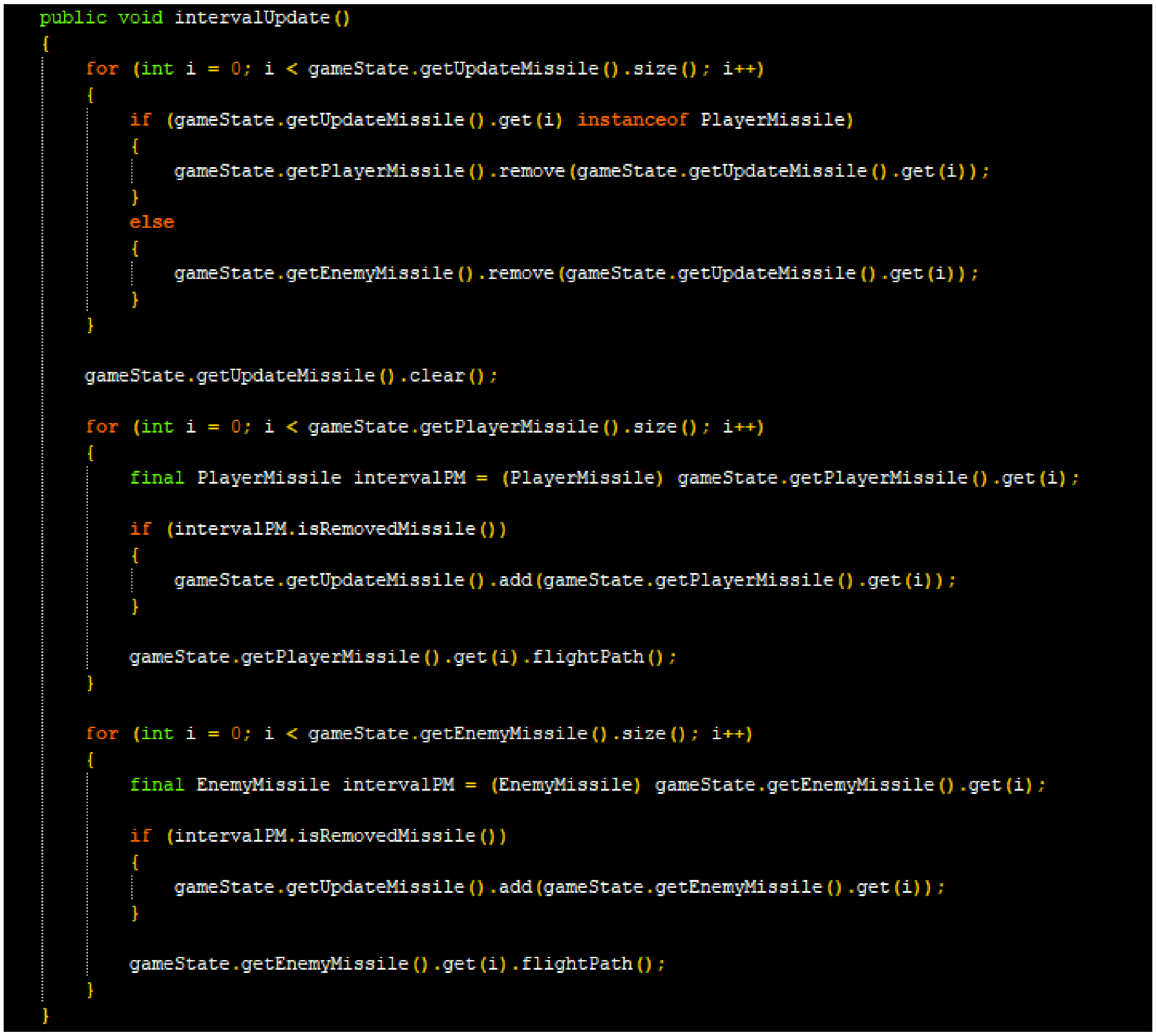
1



```
gameGraphics.addMouseListener(new MouseListener()
{
    @Override
    public void mouseClicked ( final MouseEvent e ) {

    @Override
    public void mousePressed ( final MouseEvent e )
    {
        gameState.addPlayerMissile(new PlayerMissile(e.getX(), e.getY()));
        gameState.addExplosion(new Explosion(e.getX(), e.getY()));
    }
```

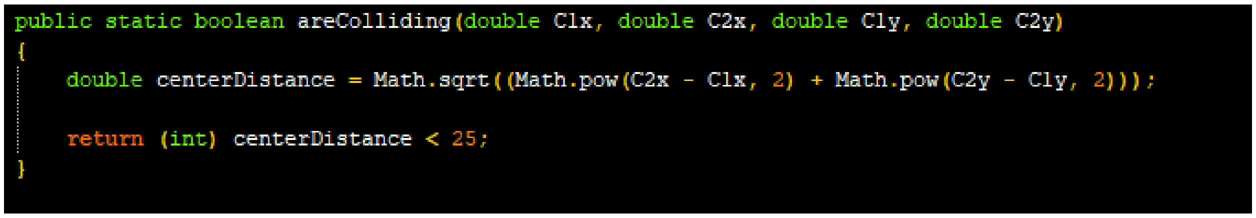
- x & y terminal coordinates for player missile
- center reference points for collision detection radius

2



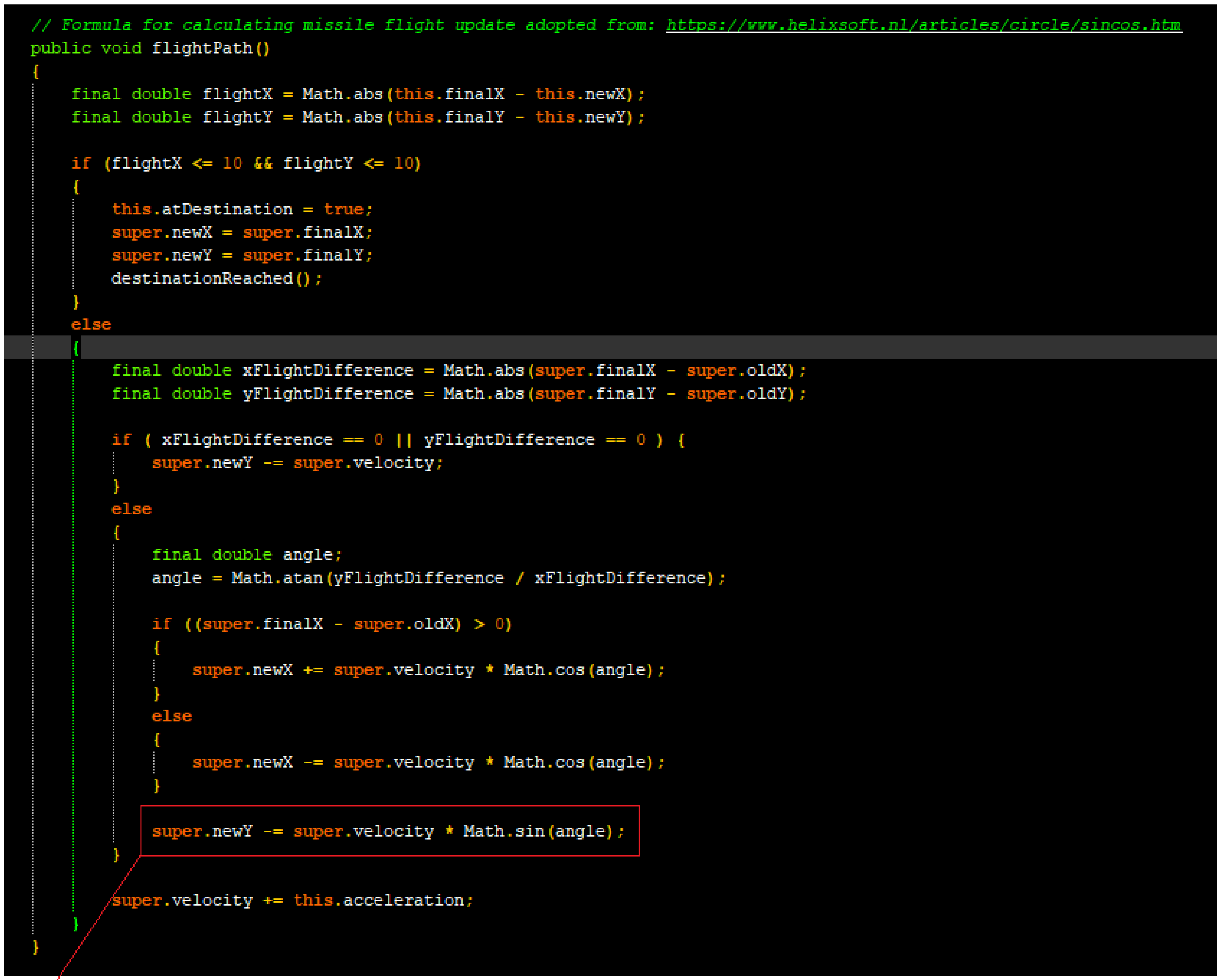
```java
public void intervalUpdate()
{
    for (int i = 0; i < gameState.getUpdateMissile().size(); i++)
    {
        if (gameState.getUpdateMissile().get(i) instanceof PlayerMissile)
        {
            gameState.getPlayerMissile().remove(gameState.getUpdateMissile().get(i));
        }
        else
        {
            gameState.getEnemyMissile().remove(gameState.getUpdateMissile().get(i));
        }
    }

    gameState.getUpdateMissile().clear();

    for (int i = 0; i < gameState.getPlayerMissile().size(); i++)
    {
        final PlayerMissile intervalPM = (PlayerMissile) gameState.getPlayerMissile().get(i);

        if (intervalPM.isRemovedMissile())
        {
            gameState.getUpdateMissile().add(gameState.getPlayerMissile().get(i));
        }

        gameState.getPlayerMissile().get(i).flightPath();
    }

    for (int i = 0; i < gameState.getEnemyMissile().size(); i++)
    {
        final EnemyMissile intervalPM = (EnemyMissile) gameState.getEnemyMissile().get(i);

        if (intervalPM.isRemovedMissile())
        {
            gameState.getUpdateMissile().add(gameState.getEnemyMissile().get(i));
        }

        gameState.getEnemyMissile().get(i).flightPath();
    }
}
```

3



```java
public static boolean areColliding(double C1x, double C2x, double C1y, double C2y)
{
    double centerDistance = Math.sqrt((Math.pow(C2x - C1x, 2) + Math.pow(C2y - C1y, 2)));

    return (int) centerDistance < 25;
}
```

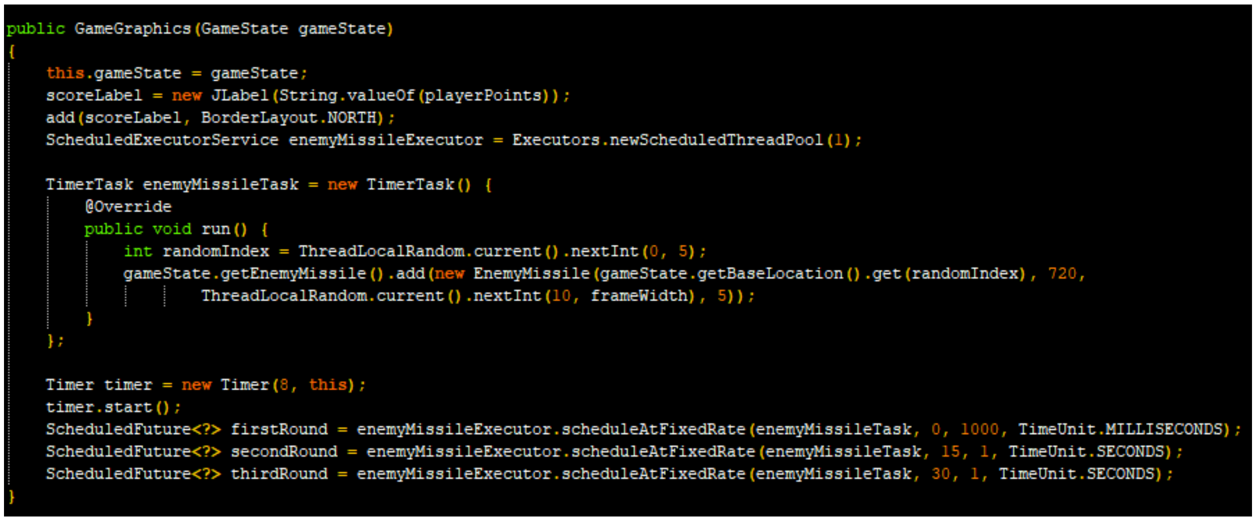
3



```java
// Formula for calculating missile flight update adopted from: https://www.helixsoft.nl/articles/circle/sincos.htm
public void flightPath()
{
    final double flightX = Math.abs(this.finalX - this.newX);
    final double flightY = Math.abs(this.finalY - this.newY);

    if (flightX <= 10 && flightY <= 10)
    {
        this.atDestination = true;
        super.newX = super.finalX;
        super.newY = super.finalY;
        destinationReached();
    }
    else
    {
        final double xFlightDifference = Math.abs(super.finalX - super.oldX);
        final double yFlightDifference = Math.abs(super.finalY - super.oldY);

        if ( xFlightDifference == 0 || yFlightDifference == 0 ) {
            super.newY -= super.velocity;
        }
        else
        {
            final double angle;
            angle = Math.atan(yFlightDifference / xFlightDifference);

            if ((super.finalX - super.oldX) > 0)
            {
                super.newX += super.velocity * Math.cos(angle);
            }
            else
            {
                super.newX -= super.velocity * Math.cos(angle);
            }

            super.newY -= super.velocity * Math.sin(angle);
        }

        super.velocity += this.acceleration;
    }
}
```

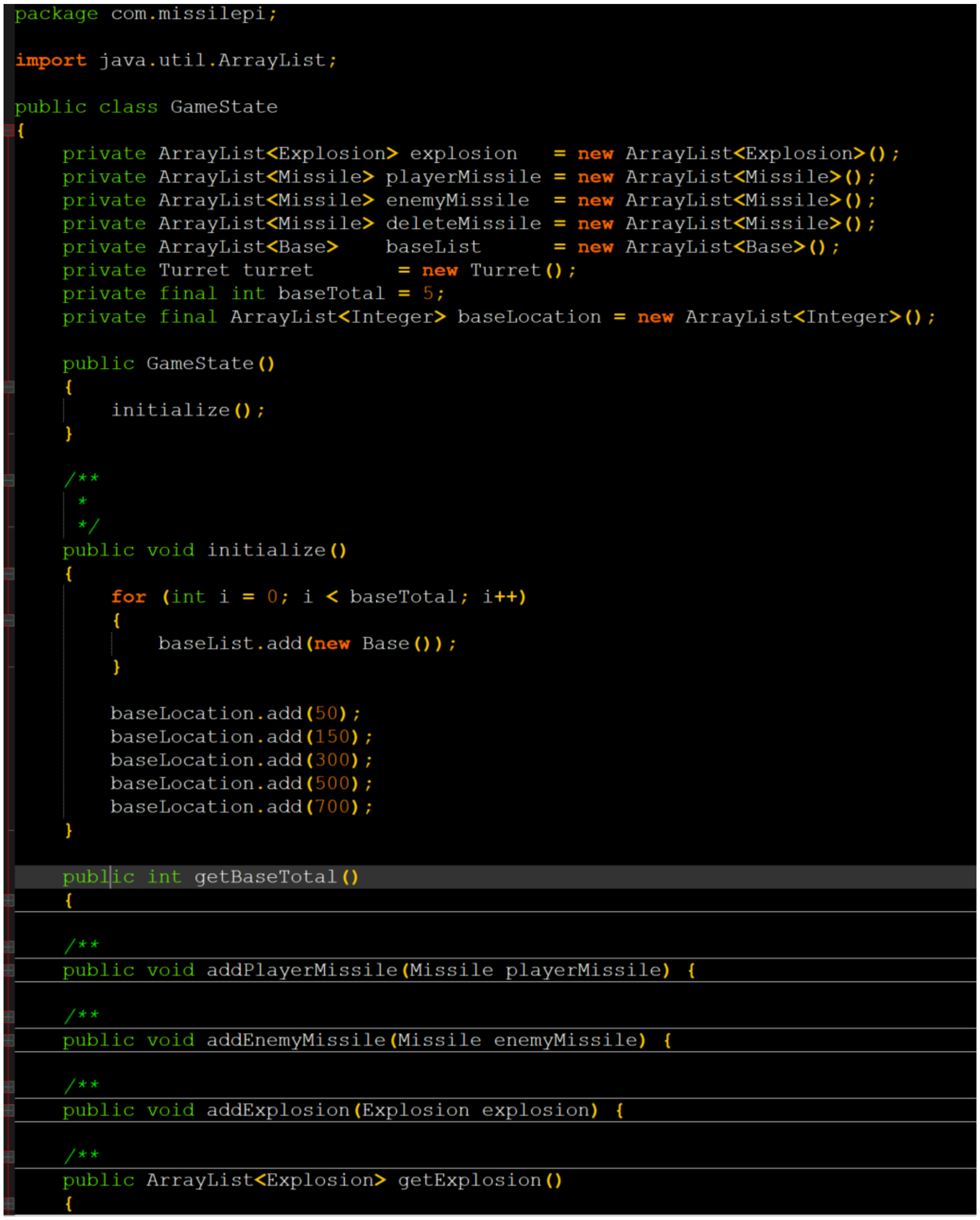
- EnemyMissile code differs by adding newY to result of velocity * sin(angle)

4



```java
public GameGraphics(GameState gameState)
{
    this.gameState = gameState;
    scoreLabel = new JLabel(String.valueOf(playerPoints));
    add(scoreLabel, BorderLayout.NORTH);
    ScheduledExecutorService enemyMissileExecutor = Executors.newScheduledThreadPool(1);

    TimerTask enemyMissileTask = new TimerTask() {
        @Override
        public void run() {
            int randomIndex = ThreadLocalRandom.current().nextInt(0, 5);
            gameState.getEnemyMissile().add(new EnemyMissile(gameState.getBaseLocation().get(randomIndex), 720,
                    ThreadLocalRandom.current().nextInt(10, frameWidth), 5));
        }
    };

    Timer timer = new Timer(8, this);
    timer.start();
    ScheduledFuture<?> firstRound = enemyMissileExecutor.scheduleAtFixedRate(enemyMissileTask, 0, 1000, TimeUnit.MILLISECONDS);
    ScheduledFuture<?> secondRound = enemyMissileExecutor.scheduleAtFixedRate(enemyMissileTask, 15, 1, TimeUnit.SECONDS);
    ScheduledFuture<?> thirdRound = enemyMissileExecutor.scheduleAtFixedRate(enemyMissileTask, 30, 1, TimeUnit.SECONDS);
}
```

5



```
package com.missilepi;

import java.util.ArrayList;

public class GameState
{
    private ArrayList<Explosion> explosion    = new ArrayList<Explosion>();
    private ArrayList<Missile> playerMissile = new ArrayList<Missile>();
    private ArrayList<Missile> enemyMissile  = new ArrayList<Missile>();
    private ArrayList<Missile> deleteMissile = new ArrayList<Missile>();
    private ArrayList<Base>    baseList      = new ArrayList<Base>();
    private Turret turret         = new Turret();
    private final int baseTotal = 5;
    private final ArrayList<Integer> baseLocation = new ArrayList<Integer>();

    public GameState()
    {
        initialize();
    }

    /**
     *
     */
    public void initialize()
    {
        for (int i = 0; i < baseTotal; i++)
        {
            baseList.add(new Base());
        }

        baseLocation.add(50);
        baseLocation.add(150);
        baseLocation.add(300);
        baseLocation.add(500);
        baseLocation.add(700);
    }

    public int getBaseTotal()
    {

    /**
    public void addPlayerMissile(Missile playerMissile) {

    /**
    public void addEnemyMissile(Missile enemyMissile) {

    /**
    public void addExplosion(Explosion explosion) {

    /**
    public ArrayList<Explosion> getExplosion()
    {
```

REFERENCES

"*Circle Collision.*" Flat Red Ball. https://flatredball.com/documentation/tutorials/math/circle-collision/ (accessed Nov, 2021).

Amarillion. "*Sin & Cos: The Programmer's Pals!.*" HelixSoft. https://www.helixsoft.nl/articles/circle/sincos.htm (accessed Nov, 2021).