INTERNATIONAL UNIVERSITY – VIETNAM NATIONAL UNIVERSITY HCMC SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

MEGAMAN PROJECT GAME

COURSE: OBJECT-ORIENTED PROGRAMMING

Link project: https://github.com/JerrySam3912/demo_megaman
GROUP: Billie English

Game Developer:

Hoàng Xuân Dũng	ITITWE22009
Nguyễn Quang Thái	ITCSIU22227
Nguyễn Nhật Nam	ITITWE22419
Nguyễn Trung Anh	ITITWE22122

Table of Contents

I. Introduction

The MegaMan game project is a modularly designed application inspired by the classic MegaMan series. This report covers the development methodology, tools, UML design, and future improvements for a 2D platformer game created using object-oriented programming principles

II. Project Objectives

The MegaMan game project aims to showcase the practical application of Object-Oriented Programming (OOP) principles in game development. A central goal is to create a fully functional 2D platformer game where players control the protagonist, Megaman, navigating through various levels filled with challenges. The game mechanics allow the player to jump, attack, and interact with the environment, including defeating enemies, collecting items, and achieving level objectives. These features are designed to demonstrate the versatility and effectiveness of OOP techniques.

This framework enables the game to run seamlessly on multiple platforms, including desktop and web environments, by leveraging its core and desktop modules. The project's architecture is intentionally modular, with well-defined packages such as Control, GameEffect, GameObject, and UserInterface, ensuring scalability and maintainability. This structure supports the addition of new features, like additional levels, enemies, and characters, with minimal disruption to the existing codebase.

The project emphasizes the application of OOP concepts, including inheritance, polymorphism, and encapsulation, to foster code reusability and clarity. Advanced design patterns, such as Bridge, Command, and Singleton, are implemented to streamline the development process and enhance code organization. Furthermore, the game includes a user-friendly interface with menus, heads-up displays (HUD), and visually engaging transitions, supported by sound effects and animations to provide an immersive player experience.

III. Tools and Frameworks Used

The MegaMan project leverages a robust set of tools and frameworks to ensure a smooth and scalable development process. The primary programming language used is **Java**, chosen for its versatility and extensive library support, making it ideal for game development. The Java Swing framework plays a vital part by enabling cross-platform functionality, allowing the game to run

seamlessly on both desktop and web environments. This framework facilitates the development of a high-performance 2D game engine, simplifying tasks like rendering, animation, and input handling.

For development, tools like IntelliJ IDEA and NetBeans IDE provide an integrated environment for writing, debugging, and testing code. These IDEs enhance productivity with features like syntax highlighting, version control integration, and efficient refactoring options. Version control is managed using Git, ensuring collaborative work and codebase tracking across team members, while also enabling the rollback of changes if needed.

The build process relies on Apache Ant, a flexible build automation tool that streamlines compiling, packaging, and deploying the project. This system ensures consistent builds across various platforms. Game assets, including sprites, sound effects, and level data, are organized within a structured **data folder**, making it easy to manage and update resources without modifying core code. By combining these tools and frameworks, the MegaMan project achieves both technical sophistication and maintainability.

IV. Package and Systems Components:

This project is structured into several key packages: Control, GameEffect, GameObject, and UserInterface, along with an independent MegaMan class. These components work together to provide a modular and scalable game design.

The Control package handles the overall game flow, managing initialization, state transitions, and input handling. Key classes include Button.java, which defines interactive buttons, and RectangleButton.java, which builds on this functionality by adding position and size attributes to buttons. These classes ensure seamless user interaction and flexible interface management.

The GameEffect package enhances the visual and auditory experience of the game. It includes classes such as Animation.java for managing sprite animations and CacheDataLoader.java for efficient asset loading and caching. These components ensure that the game runs smoothly with immersive effects.

The GameObject package focuses on the interactive entities in the game. It contains classes like Megaman.java, which defines the main character's behavior and actions, FinalBoss.java for the final enemy logic, and BulletManager.java to handle projectile interactions. These classes encapsulate the core mechanics of gameplay, from character movement to enemy AI.

The UserInterface package manages the visual and interactive elements displayed to the player. Classes such as GameFrame.java, GamePanel.java, and InputManager.java ensure a cohesive and responsive interface, providing players with essential HUD elements and seamless menu navigation.

Finally, the MegaMan class acts as the entry point for the game. it initializes the virtual screen size, defines bit configurations for game objects, and loads all assets required for gameplay. This class is also responsible for setting the initial screen, ensuring a smooth start to the player's experience.

Together, these packages and classes form the foundation of the MegaMan game, ensuring a structured, maintainable, and engaging gameplay experience.

1. Control

The Control package is an essential part of the MegaMan game, managing core logic such as initialization, state transitions, and user interactions. It provides the foundational classes for building flexible and interactive game components.

For example, Button.java is an abstract class that establishes the basic structure for interactive buttons. It includes core properties like position, dimensions, and states (e.g., NONE, PRESSED, HOVER) that allow for dynamic interactions. A key method is isInButton, which determines if a mouse click is within the button's bounds:

```
public abstract boolean isInButton(int x, int y);
```

Building on this, RectangleButton.java extends Button to provide specific logic for rectangular buttons. It implements the isInButton method to detect hover and click events based on the button's position and size:

```
@Override
public boolean isInButton(int x, int y) {
    return (enabled && x >= posX && x <= posX + width && y >= posY && y <= posY + height);
}
```

It also renders buttons dynamically with the draw method, changing their appearance based on states like NONE, PRESSED, or HOVER:

```java
@Override
public void draw(Graphics g) {
    if(enabled) {
        switch (state) {
            case NONE: g.setColor(bgColor); break;
            case PRESSED: g.setColor(pressedBgColor); break;
            case HOVER: g.setColor(hoverBgColor); break;
        }
    } else {
        g.setColor(Color.GRAY);
    }
    g.fillRect(posX, posY, width, height);

    g.setColor(Color.PINK);
    g.drawRect(posX, posY, width, height);
    g.drawRect(posX + 1, posY + 1, width - 2, height - 2);

    g.setColor(Color.WHITE);
    g.setFont(new Font(name:"TimesRoman", Font.PLAIN, size:14));
    g.drawString(text, posX + paddingTextX, posY + paddingTextY);
}
```

2. GameEffect

The GameEffect package is responsible for managing the visual and auditory aspects of the game, enhancing the player experience through animations and sound effects. This package ensures smooth sprite transitions and immersive audio feedback, making the game more dynamic and engaging

The Animation class provides frame-based animations for game objects. It allows objects such as the player and enemies to transition seamlessly between states like idle, running, and attacking, ensuring a polished visual representation during gameplay. Key methods include adding frames to an animation and updating the current frame:

The CacheDataLoader class handles the efficient loading and caching of game assets such as images, sprite sheets, and sound effects. This reduces loading times and ensures resources are reused efficiently. For example, the class uses a hashtable to store assets for quick access. It imports necessary libraries for managing files, images, and audio clips, as seen in the snippet below:

src > GameEffect > CacheDataLoader.java > CacheDataLoader > LoadSounds()

```java

package GameEffect;

import java.awt.image.BufferedImage;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Hashtable;
import javax.imageio.ImageIO;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;
/**
 *
 * @author ADMIN
 */
public class CacheDataLoader {

    private static CacheDataLoader instance;

    private String framefile = "data/frame.txt";
    private String animationfile = "data/animation.txt";
    private String physmapfile = "data/phys_map.txt";
    private String backgroundmapfile = "data/background_map.txt";
    private String soundfile = "data/sounds.txt";


    private int[][] phys_map;
    private int[][] background_map;

    // tạo ra constructor private để cho chương trình chỉ sử dụng mỗi instance
    //mà ta khởi tạo

    // <key, value>

    private Hashtable<String, FrameImage> frameImages;
    private Hashtable<String, Animation> animations;
    private Hashtable<String, Clip> sounds;

    private CacheDataLoader(){

//        frameImage = new Hashtable<String, FrameImage>();
//        animation = new Hashtable<String, Animation>();

    }

    // kiểm tra xem instance được tạo ra chưa? chưa thì tạo mới

```

Java: Ready

The **FrameImage** class supports animations by encapsulating individual frames of a sprite sheet. It provides methods for retrieving and managing frame images, ensuring consistent transitions between frames:

```java
public class FrameImage{
    private String name;
    private BufferedImage image;


    public FrameImage(){
        this.name = null;
        this.image = null;
    }

    public FrameImage(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public FrameImage(FrameImage frameImage){
        this.image = new BufferedImage(frameImage.getImageWidth(), frameImage.getImageHeight(), frameImage

        Graphics g = image.getGraphics();
        g.drawImage(frameImage.getImage(), x:0, y:0, observer:null);

    }
```

By integrating with other packages such as GameObject and UserInterface, the GameEffect package provides essential support for animations like running, jumping, and attacking, as well as sound effects triggered during collisions, attacks, and item pickups. This cohesive functionality ensures the game delivers a visually appealing and engaging experience for players.

3. GameObject

The GameObject package is the central backbone of the game, responsible for managing all the entities and their behaviors within the game world. It encapsulates key game mechanics such as movement, interactions, and environmental management. Below is a detailed overview of the primary classes in this package:

Megaman.java: This class represents the main protagonist, MegaMan. It is responsible for implementing the character's core mechanics, such as movement, health management, and attack systems. For instance, the jump() method ensures that MegaMan can perform jumps while keeping his state consistent:

```java
@Override
public void jump() {

    if(!getIsJumping()){
        setIsJumping(isJumping:true);
        setSpeedY(-5.0f);
        flyBackAnim.reset();
        flyForwardAnim.reset();
    }
    // for clim wall
    else{
        Rectangle rectRightWall = getBoundForCollisionWithMap();
        rectRightWall.x += 1;
        Rectangle rectLeftWall = getBoundForCollisionWithMap();
        rectLeftWall.x -= 1;
```

FinalBoss.java: The FinalBoss class inherits from the Human class. It represents the ultimate challenge in the game, featuring predefined attack and movement patterns rather than dynamic tracking. The boss interacts with MegaMan based on proximity or scripted behaviors, creating strategic gameplay challenges

```java
public class FinalBoss extends Human {

    private Animation idleforward, idleback;
    private Animation shootingforward, shootingback;
    private Animation slideforward, slideback;

    private long startTimeForAttacked;

    private Hashtable<String, Long> timeAttack = new Hashtable<String, Long>();
    private String[] attackType = new String[4];
    private int attackIndex = 0;
    private long lastAttackTime;

    public FinalBoss(float x, float y, GameWorld gameWorld) {
        super(x, y, width:110, height:150, mass:0.1f, blood:100, gameWorld);
        idleback = CacheDataLoader.getInstance().getAnimation(name:"boss_idle");
        idleforward = CacheDataLoader.getInstance().getAnimation(name:"boss_idle");
        idleforward.flipAllImage();

        shootingback = CacheDataLoader.getInstance().getAnimation(name:"boss_shooting");
        shootingforward = CacheDataLoader.getInstance().getAnimation(name:"boss_shooting");
        shootingforward.flipAllImage();

        slideback = CacheDataLoader.getInstance().getAnimation(name:"boss_slide");
        slideforward = CacheDataLoader.getInstance().getAnimation(name:"boss_slide");
        slideforward.flipAllImage();
```

Human.java: The Human class serves as a base class for human-like characters in the game, including both the player (Megaman) and certain enemies like FinalBoss. It encapsulates shared functionality such as movement, health, and animations

```
public abstract class Human extends ParticularObject{

    private boolean isJumping;
    private boolean isDicking;

    private boolean isLanding;

    public Human(float x, float y, float width, float height, float mass, int blood, GameWorld gameWorld)
        super(x, y, width, height, mass, blood, gameWorld);
        setState(ALIVE);
    }

    public abstract void run();

    public abstract void jump();

    public abstract void dick();

    public abstract void standUp();

    public abstract void stopRun();

    public boolean getIsJumping() {
        return isJumping;
    }
```

DarkRaise.java**:** The DarkRaise class represents a mid-level enemy with unique attack patterns. It tracks the player and shoots projectiles (defined in DarkRaiseBullet.java) to challenge the player.

```java
public class DarkRaise extends ParticularObject{

    private Animation forwardAnim, backAnim;

    private long startTimeToShoot;
    private float x1, x2;

    public DarkRaise(float x, float y, GameWorld gameWorld) {
        super(x, y, width:127, height:89, mass:0, blood:100, gameWorld);
        backAnim = CacheDataLoader.getInstance().getAnimation(name:"darkraise");
        forwardAnim = CacheDataLoader.getInstance().getAnimation(name:"darkraise");
        forwardAnim.flipAllImage();
        startTimeToShoot = 0;
        setTimeForNoBehurt(time:300000000);

        x1 = x - 100;
        x2 = x + 100;
        setSpeedX(speedX:1);

        setDamage(damage:10);
    }
```

ParticularObject.java: The ParticularObject class is the base class for all specific game objects, including both characters and interactive elements. It defines shared properties such as position, velocity, and state

```java
    public ParticularObject(float x, float y, float width, float height, float mass, int blood, GameWorld

        // posX and posY are the middle coordinate of the object
        super(x, y, gameWorld);
        setWidth(width);
        setHeight(height);
        setMass(mass);
        setBlood(blood);

        direction = RIGHT_DIR;

    }

    public void setTimeForNoBehurt(long time){
        timeForNoBeHurt = time;
    }

    public long getTimeForNoBeHurt(){
        return timeForNoBeHurt;
    }

    public void setState(int state){
        this.state = state;
    }

    public int getState(){
        return state;
```

ParticularObjectManager.java: The ParticularObjectManager class handles collections of ParticularObject instances. It ensures efficient updates, collision checks, and rendering of all game objects.

```
public void UpdateObjects(){

    synchronized(particularObjects){
        for(int id = 0; id < particularObjects.size(); id++){

            ParticularObject object = particularObjects.get(id);


            if(!object.isObjectOutOfCameraView()) object.Update();

            if(object.getState() == ParticularObject.DEATH){
                particularObjects.remove(id);
            }
        }
    }

    //System.out.println("Camerawidth  = "+camera.getWidth());

}
```

BlueFire.java: The FireBall class defines the projectiles used by MegaMan. These projectiles damage enemies and disappear after a set duration. The update() method ensures fireballs are updated and destroyed appropriately

```java
public class BlueFire extends Bullet{

    private Animation forwardBulletAnim, backBulletAnim;

    public BlueFire(float x, float y, GameWorld gameWorld) {
        super(x, y, width:60, height:30, mass:1.0f, damage:10, gameWorld);
        forwardBulletAnim = CacheDataLoader.getInstance().getAnimation(name:"bluefire");
        backBulletAnim = CacheDataLoader.getInstance().getAnimation(name:"bluefire");
        backBulletAnim.flipAllImage();
    }



    @Override
    public Rectangle getBoundForCollisionWithEnemy() {
        return getBoundForCollisionWithMap();
    }

    @Override
    public void draw(Graphics2D g2) {

        if(getSpeedX() > 0){
            if(!forwardBulletAnim.isIgnoreFrame(id:0) && forwardBulletAnim.getCurrentFrame() == 3){
                forwardBulletAnim.setIgnoreFrame(id:0);
                forwardBulletAnim.setIgnoreFrame(id:1);
                forwardBulletAnim.setIgnoreFrame(id:2);
            }
```

RedEyeDevil.java: The RedEyeDevil class represents another enemy type with distinct behaviors. It shoots RedEyeBullet projectiles at the player.

```java
public class RedEyeDevil extends ParticularObject {

    private Animation forwardAnim, backAnim;

    private long startTimeToShoot;

    private AudioClip shooting;

    public RedEyeDevil(float x, float y, GameWorld gameWorld) {
        super(x, y, width:127, height:89, mass:0, blood:100, gameWorld);
        backAnim = CacheDataLoader.getInstance().getAnimation(name:"redeye");
        forwardAnim = CacheDataLoader.getInstance().getAnimation(name:"redeye");
        forwardAnim.flipAllImage();
        startTimeToShoot = 0;
        setDamage(damage:10);
        setTimeForNoBehurt(time:300000000);
        shooting = CacheDataLoader.getInstance().getSound(name:"redeyeshooting");
    }

    @Override
    public void attack() {

        shooting.play();
        Bullet bullet = new RedEyeBullet(getPosX(), getPosY(), getGameWorld());
        if(getDirection() == LEFT_DIR) bullet.setSpeedX(-8);
        else bullet.setSpeedX(speedX:8);
        bullet.setTeamType(getTeamType());
        getGameWorld().bulletManager.addObject(bullet);
```

BackgroundMap.java: The BackgroundMap class is responsible for rendering the static environment of the game, ensuring the background remains consistent and visually appealing.

```java
public class BackgroundMap extends GameObject {

    public int[][] map;
    private int tileSize;

    public BackgroundMap(float x, float y, GameWorld gameWorld) {
        super(x, y, gameWorld);
        map = CacheDataLoader.getInstance().getBackgroundMap();
        tileSize = 30;
    }

    @Override
    public void Update() {}

    public void draw(Graphics2D g2){

        Camera camera = getGameWorld().camera;

        g2.setColor(Color.RED);
        for(int i = 0;i< map.length;i++)
            for(int j = 0;j<map[0].length;j++)
                if(map[i][j]!=0 && j*tileSize - camera.getPosX() > -30 && j*tileSize - camera.getPosX() <
                        && i*tileSize - camera.getPosY() > -30 && i*tileSize - camera.getPosY() < GameFram
                    g2.drawImage(CacheDataLoader.getInstance().getFrameImage("tiled"+map[i][j]).getImage()
                        (int) getPosY() + i*tileSize - (int) camera.getPosY(), observer:null);
                }
```

PhysicalMap.java: The PhysicalMap class defines the physical boundaries and collision logic for the game world. Using Box2D, it creates static objects like walls and platforms

```java
public class PhysicalMap extends GameObject {
    public int[][] phys_map;
    private int tileSize;

    //Testing

    public PhysicalMap(float x, float y, GameWorld gameWorld) {
        super(x,y,gameWorld);
        this.tileSize = 30;
        phys_map = CacheDataLoader.getInstance().getPhysicalMap();

    }

    public int getTileSize(){
        return tileSize;
    }

    public Rectangle haveCollisionWithTop(Rectangle rect){

        int posX1 = rect.x/tileSize;
        posX1 -= 2;
        int posX2 = (rect.x + rect.width)/tileSize;
        posX2 += 2;

        //int posY = (rect.y + rect.height)/tileSize;
        int posY = rect.y/tileSize;

        if(posX1 < 0) posX1 = 0;
```

GameWorld.java: The GameWorld class serves as the central hub for managing all game objects. It updates and renders entities every frame, ensuring smooth gameplay and interactions.

```java
public class GameWorld extends State{

    private BufferedImage bufferedImage;
    private int lastState;

    public ParticularObjectManager particularObjectManager;
    public BulletManager bulletManager;

    public Megaman megaMan;

    public PhysicalMap physicalMap;
    public BackgroundMap backgroundMap;
    public Camera camera;

    public static final int finalBossX = 3600;

    public static final int INIT_GAME = 0;
    public static final int TUTORIAL = 1;
    public static final int GAMEPLAY = 2;
    public static final int GAMEOVER = 3;
    public static final int GAMEWIN = 4;
    public static final int PAUSEGAME = 5;

    public static final int INTROGAME = 0;
    public static final int MEETFINALBOSS = 1;

    public int openIntroGameY = 0;
    public int state = INIT_GAME;
```

State.java: The State class manages different game states such as "RUNNING," "PAUSED," and "GAME_OVER." This allows the game to handle transitions and control flow effectively.

```java
package GameObject;

import UserInterface.GamePanel;
import java.awt.image.BufferedImage;

/**
 *
 * @author ADMIN
 */
public abstract class State {

    protected GamePanel gamePanel;

    public State(GamePanel gamePanel) {
        this.gamePanel = gamePanel;
    }

    public abstract void Update();
    public abstract void Render();
    public abstract BufferedImage getBufferedImage();

    public abstract void setPressedButton(int code);
    public abstract void setReleasedButton(int code);
}
```

Bullet.java: Represents a generic projectile in the game. It includes movement logic and interactions with game objects.

```java
package GameObject;

import java.awt.Graphics2D;

/**
 *
 * @author ADMIN
 */
public abstract class Bullet extends ParticularObject{

    public Bullet(float x, float y, float width, float height, float mass, int damage, GameWorld gameWorld
            super(x, y, width, height, mass, blood:1, gameWorld);
            setDamage(damage);
    }

    public abstract void draw(Graphics2D g2d);

    public void Update(){
        super.Update();
        setPosX(getPosX() + getSpeedX());
        setPosY(getPosY() + getSpeedY());
        ParticularObject object = getGameWorld().particularObjectManager.getCollisionWidthEnemyObject(this
        if(object!=null && object.getState() == ALIVE){
            setBlood(blood:0);
            object.beHurt(getDamage());
            System.out.println(x:"Bullet set behurt for enemy");
        }
```

BulletManager.java: Manages all active bullets in the game, ensuring efficient updates and rendering.

```java
public class BulletManager extends ParticularObjectManager {

    public BulletManager(GameWorld gameWorld) {
        super(gameWorld);
    }

    @Override
    public void UpdateObjects() {
        super.UpdateObjects();
        synchronized(particularObjects){
            for(int id = 0; id < particularObjects.size(); id++){

                ParticularObject object = particularObjects.get(id);

                if(object.isObjectOutOfCameraView() || object.getState() == ParticularObject.DEATH){
                    particularObjects.remove(id);
                    //System.out.println("Remove");
                }
            }
        }
    }
```

Camera.java: Controls the in-game camera, ensuring it follows the player or focuses on specific objects when necessary.

```java
public class Camera extends GameObject {

    private float widthView;
    private float heightView;

    private boolean isLocked = false;

    public Camera(float x, float y, float widthView, float heightView, GameWorld gameWorld) {
        super(x, y, gameWorld);
        this.widthView = widthView;
        this.heightView = heightView;
    }

    public void lock(){
        isLocked = true;
    }

    public void unlock(){
        isLocked = false;
    }

    @Override
    public void Update() {

        // NOTE: WHEN SEE FINAL BOSS, THE CAMERA WON'T CHANGE THE POSITION,
        // AFTER THE TUTORIAL, CAMERA WILL SET THE NEW POS
```

RobotR.java: Represents a robotic enemy with specific attack patterns and behaviors.

```java
public class RobotR extends ParticularObject {

    private Animation forwardAnim, backAnim;

    private long startTimeToShoot;
    private float x1, x2, y1, y2;

    private AudioClip shooting;

    public RobotR(float x, float y, GameWorld gameWorld) {
        super(x, y, width:127, height:89, mass:0, blood:100, gameWorld);
        backAnim = CacheDataLoader.getInstance().getAnimation(name:"robotR");
        forwardAnim = CacheDataLoader.getInstance().getAnimation(name:"robotR");
        forwardAnim.flipAllImage();
        startTimeToShoot = 0;
        setTimeForNoBehurt(time:300000000);
        setDamage(damage:10);

        x1 = x - 100;
        x2 = x + 100;
        y1 = y - 50;
        y2 = y + 50;

        setSpeedX(speedX:1);
        setSpeedY(speedY:1);

        shooting = CacheDataLoader.getInstance().getSound(name:"robotRshooting");
    }
```

SmallRedGun.java: A small enemy type that fires projectiles at the player, providing a basic challenge.

```java
public class SmallRedGun extends ParticularObject{

    private Animation forwardAnim, backAnim;

    private long startTimeToShoot;

    public SmallRedGun(float x, float y, GameWorld gameWorld) {
        super(x, y, width:127, height:89, mass:0, blood:100, gameWorld);
        backAnim = CacheDataLoader.getInstance().getAnimation(name:"smallredgun");
        forwardAnim = CacheDataLoader.getInstance().getAnimation(name:"smallredgun");
        forwardAnim.flipAllImage();
        startTimeToShoot = 0;
        setTimeForNoBehurt(time:300000000);
    }

    @Override
    public void attack() {

        Bullet bullet = new YellowFlowerBullet(getPosX(), getPosY(), getGameWorld());
        bullet.setSpeedX(-3);
        bullet.setSpeedY(speedY:3);
        bullet.setTeamType(getTeamType());
        getGameWorld().bulletManager.addObject(bullet);

        bullet = new YellowFlowerBullet(getPosX(), getPosY(), getGameWorld());
        bullet.setSpeedX(speedX:3);
        bullet.setSpeedY(speedY:3);
```

YellowFlowerBullet.java: Defines specialized projectiles fired by certain enemies.

```java
public class YellowFlowerBullet extends Bullet{

    private Animation forwardBulletAnim, backBulletAnim;

    public YellowFlowerBullet(float x, float y, GameWorld gameWorld) {
            super(x, y, width:30, height:30, mass:1.0f, damage:10, gameWorld);
            forwardBulletAnim = CacheDataLoader.getInstance().getAnimation(name:"yellow_flower_bullet");
            backBulletAnim = CacheDataLoader.getInstance().getAnimation(name:"yellow_flower_bullet");
            backBulletAnim.flipAllImage();
    }



    @Override
    public Rectangle getBoundForCollisionWithEnemy() {
            // TODO Auto-generated method stub
            return getBoundForCollisionWithMap();
    }

    @Override
    public void draw(Graphics2D g2) {
            // TODO Auto-generated method stub
        if(getSpeedX() > 0){
            forwardBulletAnim.Update(System.nanoTime());
            forwardBulletAnim.draw((int) (getPosX() - getGameWorld().camera.getPosX()), (int) getPosY() -
        }else{
            backBulletAnim.Update(System.nanoTime());
            backBulletAnim.draw((int) (getPosX() - getGameWorld().camera.getPosX()), (int) getPosY() - (in
```

MenuState.java: The MenuState class manages the game's menu functionality, including navigation and user interaction. It allows players to start the game, view instructions, or exit. For example, its render() method handles menu rendering

```java
@Override
public void Render() {
    if(bufferedImage == null) {
        bufferedImage = new BufferedImage(GameFrame.SCREEN_WIDTH, GameFrame.SCREEN_HEIGHT, BufferedIma
        return;
    }
    graphicsPaint = bufferedImage.getGraphics();
    if(graphicsPaint == null) {
        graphicsPaint = bufferedImage.getGraphics();
        return;
    }
    graphicsPaint.setColor(Color.CYAN);
    graphicsPaint.fillRect(x:0, y:0, bufferedImage.getWidth(), bufferedImage.getHeight());
    for (Button bt : buttons) {
        bt.draw(graphicsPaint);
    }
```

4. UserInterface

The UserInterface package is responsible for managing the game's graphical interface and input handling. Below are the key components:

GameFrame.java: This class serves as the central hub for initializing and managing the game window. It provides methods to configure the frame's properties and integrates the GamePanel.

```
public class GameFrame extends JFrame{
    public static final int SCREEN_WIDTH = 1000;
    public static final int SCREEN_HEIGHT = 600;

    GamePanel gamePanel;

    public GameFrame() {
        Toolkit toolkit = this.getToolkit();
        Dimension dimension = toolkit.getScreenSize();

        this.setBounds((dimension.width - SCREEN_WIDTH)/2, (dimension.height - SCREEN_HEIGHT)/2, SCREEN_WI

        try {
            CacheDataLoader.getInstance().LoadData();
        } catch (IOException ex) {
           ex.printStackTrace();
        }

        gamePanel = new GamePanel();
        add(gamePanel);
        this.addKeyListener(gamePanel);


    }
```

InputManager.java: The InputManager class handles all user input from the keyboard and maps these inputs to game actions. It ensures the game responds dynamically to player commands.

```java
public class InputManager {

    private State gameState;

    public InputManager(State state){
        this.gameState = state;
    }

    public void setState(State state) {
        gameState = state;
    }

    public void setPressedButton(int code){
        gameState.setPressedButton(code);
    }

    public void setReleasedButton(int code){
        gameState.setReleasedButton(code);
    }

}
```

V. **UML**

UML Analysis: The UML diagram provides a comprehensive view of the overall structure and relationships between classes in the MegaMan game project. It highlights the implementation of core Object-Oriented Programming (OOP) principles, such as inheritance, encapsulation, and polymorphism. The following are detailed observations:

Class Hierarchy: At the core of the class structure are foundational classes like GameObject, ParticularObject, and Human, which serve as base classes for more specialized components. For instance, Megaman inherits from Human, while FinalBoss and Enemy extend ParticularObject. This hierarchy ensures shared properties and methods, such as movement and health, are reused and extended, adhering to the principle of inheritance.

Relationships:

Inheritance: Derived classes such as Megaman and FinalBoss leverage the functionality of their parent classes, allowing for code reuse and specialization. For example, FinalBoss overrides attack behaviors defined in Human to create unique combat mechanics.

Composition: Classes like GamePanel and GameWorld integrate multiple objects, such as InputManager and Animation, to perform specific tasks. This modular design enables flexibility and scalability in managing game states and rendering.

Subsystems: The system is divided into distinct subsystems to streamline functionality and maintain separation of concerns:

User Interface (UI): This subsystem includes classes like GameFrame, GamePanel, and InputManager. Together, they manage user inputs and graphical rendering, ensuring seamless interaction between the player and the game.

Game Entities: Classes such as Megaman, Enemy, and Bullet represent in-game objects. These classes encapsulate properties and behaviors specific to their roles, such as movement, attacking, and interaction with other objects.

Core Systems: Central components like GameWorld, PhysicalMap, and CacheDataLoader handle the game's state, physics, and resource management. For instance, GameWorld updates the state of all entities and ensures smooth gameplay.

Key Features:

Encapsulation: Shared behaviors and properties, such as position and state, are encapsulated in base classes like ParticularObject. This ensures consistency and reduces redundancy across the codebase.

Polymorphism: Derived classes override methods from their parent classes to implement specific behaviors. For example, FinalBoss redefines attack patterns, making it distinct from other Enemy objects while maintaining compatibility with the ParticularObject interface.

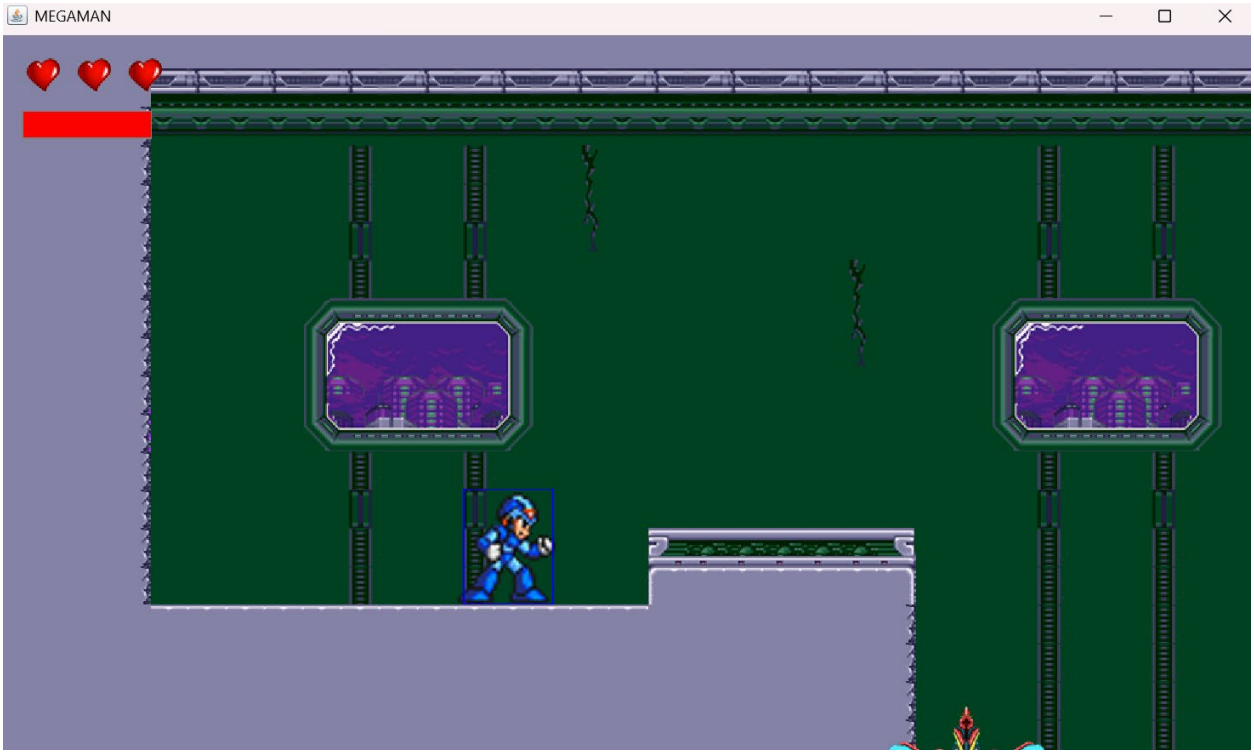
Overall, the UML diagram underscores a well-structured and modular design, enabling easy maintenance and scalability of the MegaMan game project.

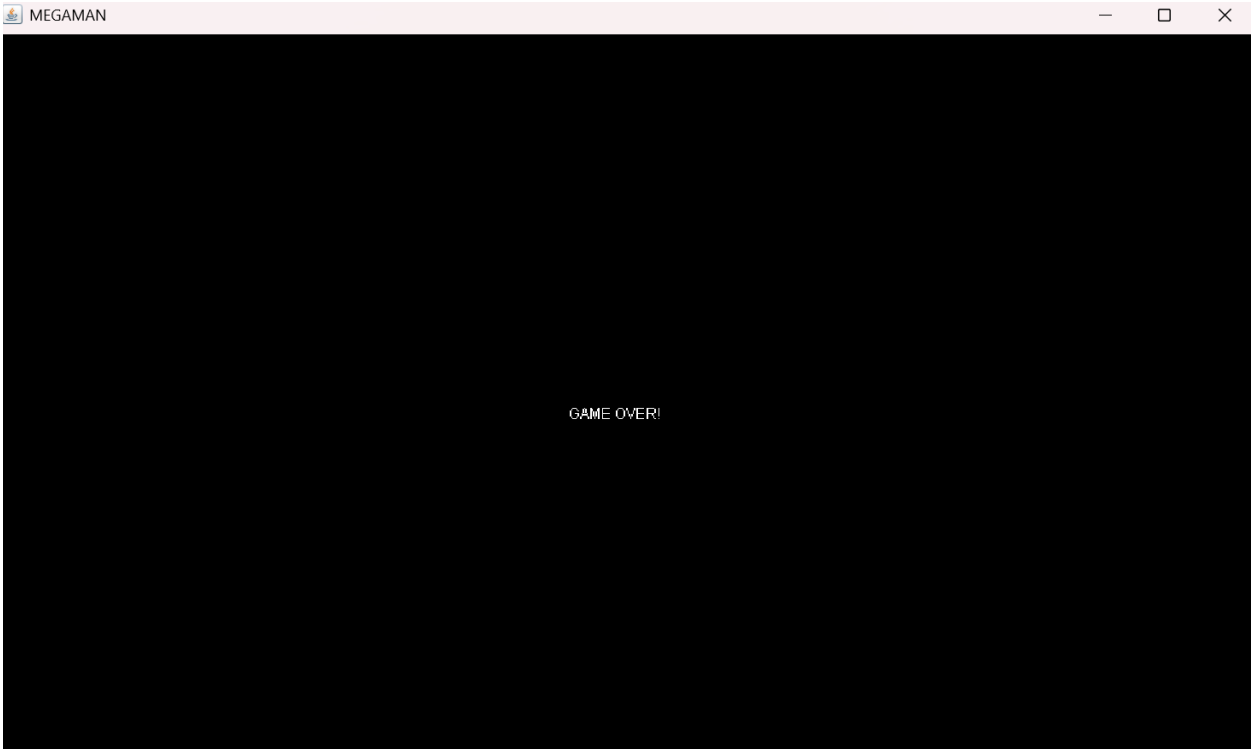
VI. Results, Limitations, and Future Development

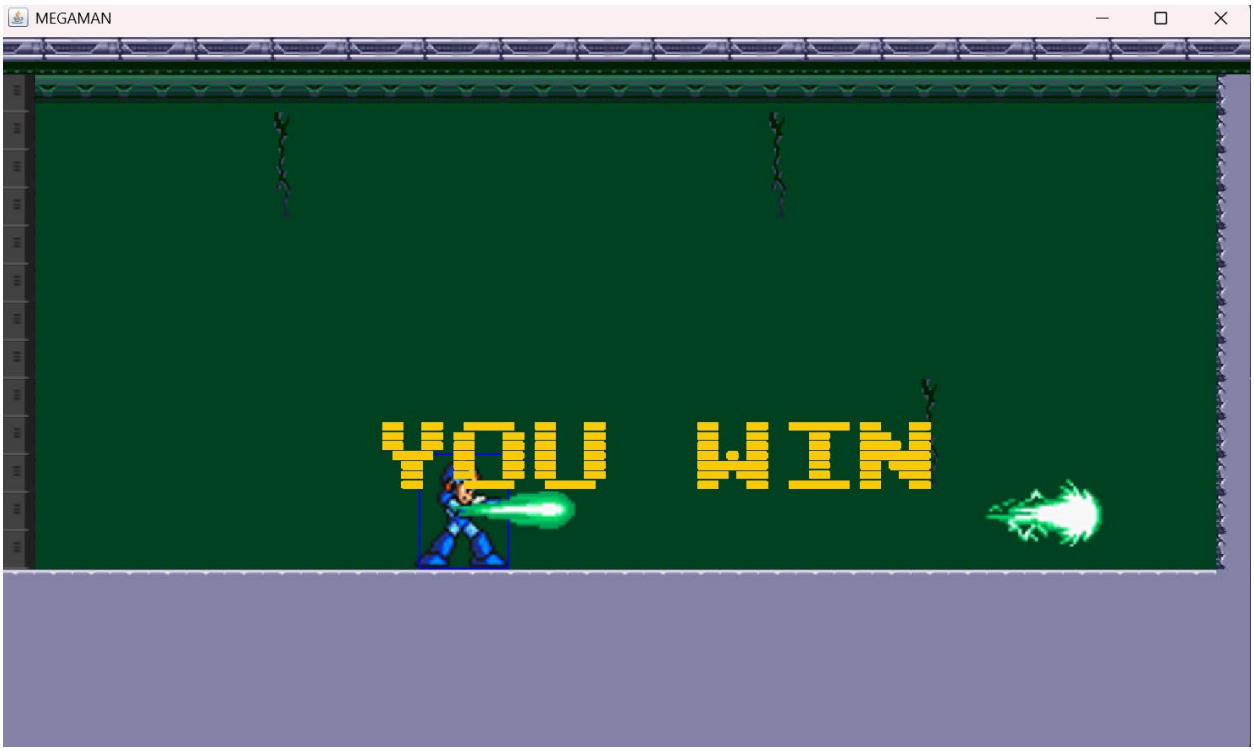
Based on our understanding of the Java programming language, OOP principles, and the libGDX library, we have completed our game with basic rules and logic. All of the classes in our projects are logically presented and successfully output to a user interface where the user can move the main character using the keyboard.

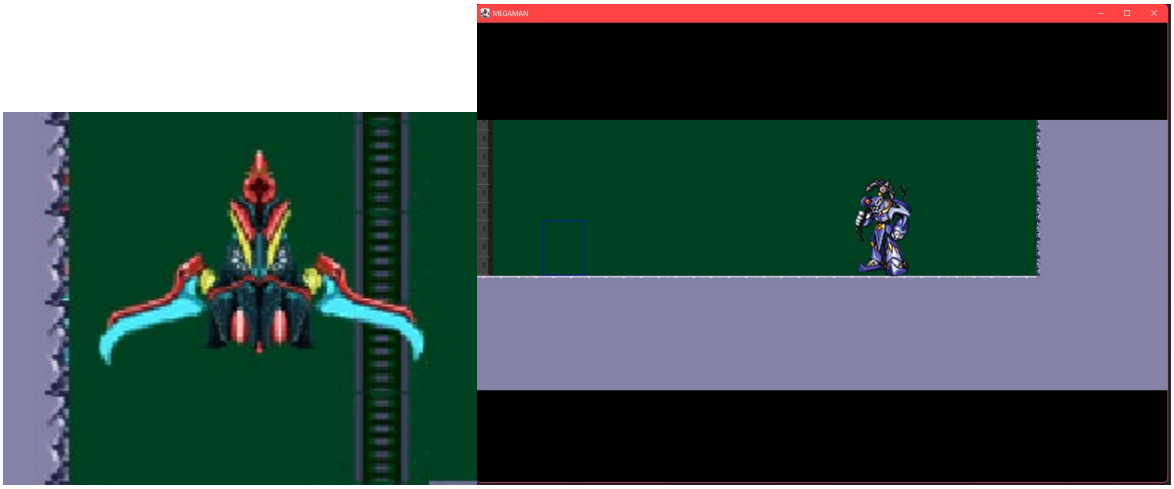
This is the screen when we run the game


MEGAMAN
We are heros, and our mission is protecting our Home Earth..


MEGAMAN


MEGAMAN
GAME OVER!


MEGAMAN
YOU WIN



Limitations:

Despite its successes, the project faced several limitations that hindered its full potential. Firstly, the game is restricted to a single level, which significantly impacts replayability and fails to provide players with a variety of challenges. This limitation reduces engagement and the overall gaming experience. Secondly, technical issues such as memory crashes during the initial launches and unexpected resets were observed. These problems disrupt gameplay and highlight areas where optimization is required. Addressing these issues would ensure a smoother and more immersive experience for players.

Future Directions:

To enhance the game and address its current limitations, several improvements have been proposed to ensure a richer and more engaging experience. First, the introduction of multiple levels with diverse environments, enemies, and challenges will significantly increase gameplay depth and replayability. Expanding the game content will provide players with a sense of progression and variety. Next, enhancing character abilities by adding features such as wall-climbing, dashing, and new attack modes will make the gameplay more dynamic and enjoyable. Furthermore,

developing more sophisticated AI behaviors for enemies, including adaptive strategies that respond to player actions, will create a more challenging and immersive experience.

VII. Conclusion

In conclusion, by building the Megaman game based on Object-Oriented Programming techniques, the process is much easier, and the code is arranged in a very logical way. The project has shown many properties of Object-Oriented Programming, such as polymorphism, inheritance, encapsulation, data abstraction, etc. Besides practicing all the OOP techniques, learning more knowledge outside the course limit, such as working with a framework, getting used to Git is one of the most important things to do while working on this project.

VIII. References

https://www.youtube.com/watch?v=m2JRPbqP_fo&ab_channel=DũngHoàngXuân