Contents

1 Basic Test Results

```
=====================
===== EX4 TESTER =====
=====================


===== EXTRACTING =====


===== CHECKING FILES =====


===== COPYING NECESSARY DIRECTORIES =====


===== ANALYZE README =====
README issue:
Error: the README file does not exist





==== CHECKING DOCUMENTATION ====
Documentation issue
Documenation is missing in pepse/PepseGameManager.java:100

==== ERRORS ====
AUTO.MISSING_README
AUTO.INVALID_SUBMISSION {README issue}
AUTO.DOCUMENTATION_MISSING {pepse/PepseGameManager.java:100}
```

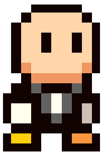
2 pepse/README

```
guytra2205, yuvaleyal
214987745, 326660610

we created the Trees package using the following classes:
1. the Flora class, that is responsible for creating all of the trees in the game.
2. the Tree class, representing a tree. a tree objects contains 3 fields:
    a trunk (instanse of Trunk)
    leaves (instanse of Leaves)
    fruits (instanse of Fruits)
3. Trunk: this class represents the trunk of a tree, and contains functions that add the trunk to the game objects, changes
4. Leaves and Fruits are classes conteining 1 (non-final) field each - a List of objects of Leaf and Fruit, respectively. th
5. Leaf: a class representing a single leaf. this class handles the things that affects each leaf in a different way, like t
6. Fruit: a class representing a single fruit. like the Leaf class, this class handles the things that affects each fruit ir

in short, the Flora class creates a List of Tree insatnces. a Tree contains a Trunk instace, a Leaves instace and a Fruits i
Leaves contains a List of Leaf instaces, and Fruits contains a List of Fruit instaces.

The design paterns we used are:
1. Facade: the Tree class wraps and handles all of the tree logic, and its the only one (except Flora) that is accessed from
2. Observer: it's, in fact, the PepseGameManager that is the Observer, but the Observer design pattern is used to affect the
```

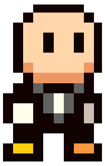
3 pepse/PepseGameManager.java

```java
package pepse;

import java.awt.Color;
import java.util.List;
import java.util.Random;

import danogl.GameManager;
import danogl.GameObject;
import danogl.collisions.Layer;
import danogl.gui.GameGUIComponent;
import danogl.gui.ImageReader;
import danogl.gui.SoundReader;
import danogl.gui.UserInputListener;
import danogl.gui.WindowController;
import danogl.util.Vector2;
import pepse.world.Sky;
import pepse.world.Avatar;
import pepse.world.Block;
import pepse.world.EnergyDisplay;
import pepse.world.JumpObserver;
import pepse.world.Terrain;
import pepse.world.daynight.Night;
import pepse.world.daynight.Sun;
import pepse.world.daynight.SunHalo;
import pepse.world.trees.Flora;
import pepse.world.trees.Fruit;
import pepse.world.trees.Tree;

/**
 * Represents the game manager for the Pepse game.
 */
public class PepseGameManager extends GameManager implements JumpObserver {
    private final int CYCLELENGTH = 30;
    private static final int MAX_COLOR = 256;
    private List<Tree> trees;

    /**
     * Constructs a PepseGameManager object.
     */
    public PepseGameManager() {
        super();
    }

    /**
     * Initializes the Pepse game.
     *
     * @param imageReader      The image reader for loading game images.
     * @param soundReader      The sound reader for loading game sounds.
     * @param inputListener    The user input listener for handling input events.
     * @param windowController The window controller for managing the game window.
     */
    @Override
    public void initializeGame(ImageReader imageReader, SoundReader soundReader,
            UserInputListener inputListener, WindowController windowController) {
        super.initializeGame(imageReader, soundReader, inputListener, windowController);
        Vector2 windowDimensions = windowController.getWindowDimensions();
        GameObject sky = Sky.create(windowDimensions);
        gameObjects().addGameObject(sky, Layer.BACKGROUND);
        int seed = (int) new Random().nextInt();
        Terrain terrain = new Terrain(windowDimensions, seed);
        List<Block> list_of_blocks = terrain.createInRange(0,
                (int) windowController.getWindowDimensions().x());
        for (Block block : list_of_blocks) {
            gameObjects().addGameObject(block);
        }
        GameObject night = Night.create(windowDimensions, CYCLELENGTH);
        gameObjects().addGameObject(night, Layer.UI);
        GameObject sun = Sun.create(windowDimensions, CYCLELENGTH * 2);
        GameObject sunHalo = SunHalo.create(sun);
        sunHalo.addComponent((float deltaTime) -> {
            sunHalo.setCenter(sun.getCenter());
        });
        gameObjects().addGameObject(sun, Layer.BACKGROUND);
        gameObjects().addGameObject(sunHalo, Layer.BACKGROUND);
        Vector2 pos = new Vector2(20, terrain.groundHeightAt(20) - Block.getSize());
        Avatar avater = new Avatar(pos, inputListener, imageReader);
        avater.registerObserver(this);
        gameObjects().addGameObject(avater);
        EnergyDisplay display = new EnergyDisplay(avater::getEnergy);
        gameObjects().addGameObject(display, Layer.UI);
        Flora flora = new Flora(x -> terrain.groundHeightAt(x));
        trees = flora.createInRange(0, (int) windowDimensions.x());
        addTreesToGame(trees);
        Fruit.setCycleLength(CYCLELENGTH);
    }

    public void onAvatarJump() {
        Color fruitColor = randomColor();
        for (Tree tree : trees) {
            tree.onAvatarJump(fruitColor);
        }
    }

    private void addTreesToGame(List<Tree> trees) {
        for (Tree tree : trees) {
            tree.addTree(gameObjects());
        }
    }

    public static Color randomColor() {
        Random random = new Random();
        return new Color(random.nextInt(0, MAX_COLOR), random.nextInt(0, MAX_COLOR),
                random.nextInt(0, MAX_COLOR));
    }

    /**
     * The main method to start the Pepse game.
     *
     * @param args Command-line arguments.
     */
    public static void main(String[] args) {
        PepseGameManager gameManager = new PepseGameManager();
        gameManager.run();
    }
}
```

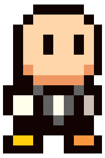
4 assets/assets/idle 0.png



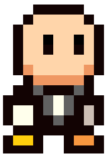
5 assets/assets/idle 1.png



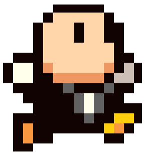
6 assets/assets/idle 2.png



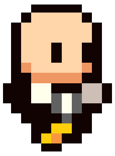
7 assets/assets/idle 3.png



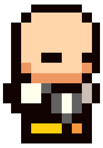
8 assets/assets/jump 0.png



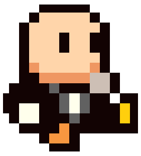
9 assets/assets/jump 1.png



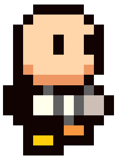
10 assets/assets/jump 2.png



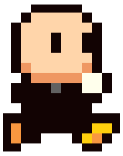
11 assets/assets/jump 3.png



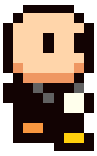
12 assets/assets/run 0.png



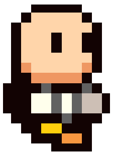
13 assets/assets/run 1.png



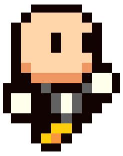
14 assets/assets/run 2.png



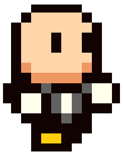
15 assets/assets/run 3.png



16 assets/assets/run 4.png



17 assets/assets/run 5.png



18 pepse/util/ColorSupplier.java

```java
package pepse.util;

import java.awt.*;
import java.util.Random;

/**
 * Provides procedurally-generated colors around a pivot.
 * @author Dan Nirel
 */
public final class ColorSupplier {
    private static final int DEFAULT_COLOR_DELTA = 10;
    private final static Random random = new Random();

    /**
     * Returns a color similar to baseColor, with a default delta.
     *
     * @param baseColor A color that we wish to approximate.
     * @return A color similar to baseColor.
     */
    public static Color approximateColor(Color baseColor) {
        return approximateColor(baseColor, DEFAULT_COLOR_DELTA);
    }

    /**
     * Returns a color similar to baseColor, with a difference of at most colorDelta.
     *
     * @param baseColor A color that we wish to approximate.
     * @param colorDelta The maximal difference (per channel) between the sampled color and the base color.
     * @return A color similar to baseColor.
     */
    public static Color approximateColor(Color baseColor, int colorDelta) {

        return new Color(
                randomChannelInRange(baseColor.getRed()-colorDelta, baseColor.getRed()+colorDelta),
                randomChannelInRange(baseColor.getGreen()-colorDelta, baseColor.getGreen()+colorDelta),
                randomChannelInRange(baseColor.getBlue()-colorDelta, baseColor.getBlue()+colorDelta));
    }

    /**
     * This method generates a random value for a color channel within the given range [min, max].
     *
     * @param min The lower bound of the given range.
     * @param max The upper bound of the given range.
     * @return A random number in the range [min, max], clipped to [0,255].
     */
    private static int randomChannelInRange(int min, int max) {
        int channel = random.nextInt(max-min+1) + min;
        return Math.min(255, Math.max(channel, 0));
    }
}
```

19 pepse/util/NoiseGenerator.java

```java
package pepse.util;

import java.util.Random;

public class NoiseGenerator {
    private double seed;
    private long default_size;
    private int[] p;
    private int[] permutation;
    private double startPoint;

    /**
     * The constructor of the NoiseGenerator class.
     *
     * @param seed can be anything you want (even 1234 or new Random().nextGaussian()).
     *             This seed is the basis of the random generator, which
     *             will draw upon it to generate pseudo-random noise.
     *
     * @param startPoint is a relative point that the noise will be generated from.
     *                   In our case it should be your ground height at X0 (specified in
     *                   ex4 when we talk about the terrain: 2.2.1).
     *
     */
    public NoiseGenerator(double seed, int startPoint) {
        this.seed = seed;
        this.startPoint = startPoint;
        init();
    }

    private void init() {
        // Initialize the permutation array.
        this.p = new int[512];
        this.permutation = new int[]{151, 160, 137, 91, 90, 15, 131, 13, 201,
                95, 96, 53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142, 8, 99,
                37, 240, 21, 10, 23, 190, 6, 148, 247, 120, 234, 75, 0, 26,
                197, 62, 94, 252, 219, 203, 117, 35, 11, 32, 57, 177, 33, 88,
                237, 149, 56, 87, 174, 20, 125, 136, 171, 168, 68, 175, 74,
                165, 71, 134, 139, 48, 27, 166, 77, 146, 158, 231, 83, 111,
                229, 122, 60, 211, 133, 230, 220, 105, 92, 41, 55, 46, 245, 40,
                244, 102, 143, 54, 65, 25, 63, 161, 1, 216, 80, 73, 209, 76,
                132, 187, 208, 89, 18, 169, 200, 196, 135, 130, 116, 188, 159,
                86, 164, 100, 109, 198, 173, 186, 3, 64, 52, 217, 226, 250,
                124, 123, 5, 202, 38, 147, 118, 126, 255, 82, 85, 212, 207,
                206, 59, 227, 47, 16, 58, 17, 182, 189, 28, 42, 223, 183, 170,
                213, 119, 248, 152, 2, 44, 154, 163, 70, 221, 153, 101, 155,
                167, 43, 172, 9, 129, 22, 39, 253, 19, 98, 108, 110, 79, 113,
                224, 232, 178, 185, 112, 104, 218, 246, 97, 228, 251, 34, 242,
                193, 238, 210, 144, 12, 191, 179, 162, 241, 81, 51, 145, 235,
                249, 14, 239, 107, 49, 192, 214, 31, 181, 199, 106, 157, 184,
                84, 204, 176, 115, 121, 50, 45, 127, 4, 150, 254, 138, 236,
                205, 93, 222, 114, 67, 29, 24, 72, 243, 141, 128, 195, 78, 66,
                215, 61, 156, 180};
        this.default_size = 35;

        // Populate it
        for (int i = 0; i < 256; i++) {
            p[256 + i] = p[i] = permutation[i];
        }

    }

    /**
     * Noise is responsible to generate pseudo random noise according to the seed given upon constructing the object.
     *
     * @param x the wanted x to receive noise for (in our case, the x coordinate of the terrain you'd want to create).
     * @param factor describes how large the noise should be (play with it, but BLOCK_SIZE *7 should be enough).
     * @return returns a noise you should *add* to the groundHeightAtX0 you have.
     *
     * example:
     * public float groundHeightAt(float x) {
     *           float noise = (float) noiseGenerator.noise(x, BLOCK_SIZE *7);
     *           return groundHeightAtX0 + noise;
     *       }
     *
     */
    public double noise(double x, double factor) {
        double value = 0.0;
        double currentPoint = startPoint;

        while (currentPoint >= 1) {
            value += smoothNoise((x / currentPoint), 0, 0) * currentPoint;
            currentPoint /= 2.0;
        }

        return value * factor / startPoint;
    }


    private double smoothNoise(double x, double y, double z) {
        // Offset each coordinate by the seed value
        x += this.seed;
        y += this.seed;
        x += this.seed;

        int X = (int) Math.floor(x) & 255; // FIND UNIT CUBE THAT
        int Y = (int) Math.floor(y) & 255; // CONTAINS POINT.
        int Z = (int) Math.floor(z) & 255;

        x -= Math.floor(x); // FIND RELATIVE X,Y,Z
        y -= Math.floor(y); // OF POINT IN CUBE.
        z -= Math.floor(z);

        double u = fade(x); // COMPUTE FADE CURVES
        double v = fade(y); // FOR EACH OF X,Y,Z.
        double w = fade(z);

        int A = p[X] + Y;
        int AA = p[A] + Z;
        int AB = p[A + 1] + Z; // HASH COORDINATES OF
        int B = p[X + 1] + Y;
        int BA = p[B] + Z;
        int BB = p[B + 1] + Z; // THE 8 CUBE CORNERS,

        return lerp(w, lerp(v, lerp(u, grad(p[AA], x, y, z),    // AND ADD
                                grad(p[BA], x - 1, y, z)), // BLENDED
                        lerp(u, grad(p[AB], x, y - 1, z),    // RESULTS
                                grad(p[BB], x - 1, y - 1, z))),// FROM 8
                lerp(v, lerp(u, grad(p[AA + 1], x, y, z - 1),    // CORNERS
                                grad(p[BA + 1], x - 1, y, z - 1)), // OF CUBE
                        lerp(u, grad(p[AB + 1], x, y - 1, z - 1),
                                grad(p[BB + 1], x - 1, y - 1, z - 1))));
    }

    private double fade(double t) {
        return t * t * t * (t * (t * 6 - 15) + 10);
    }

    private double lerp(double t, double a, double b) {
        return a + t * (b - a);
    }

    private double grad(int hash, double x, double y, double z) {
        int h = hash & 15; // CONVERT LO 4 BITS OF HASH CODE
        double u = h < 8 ? x : y, // INTO 12 GRADIENT DIRECTIONS.
               v = h < 4 ? y : h == 12 || h == 14 ? x : z;
        return ((h & 1) == 0 ? u : -u) + ((h & 2) == 0 ? v : -v);
    }
}
```

20 pepse/world/Avatar.java

```java
package pepse.world;

import danogl.GameObject;
import danogl.gui.ImageReader;
import danogl.gui.UserInputListener;
import danogl.gui.rendering.AnimationRenderable;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;
import java.awt.event.KeyEvent;
import java.util.ArrayList;
import java.util.List;

/**
 * Represents the avatar character in the game.
 */
public class Avatar extends GameObject implements JumpSubject{
    private final static float AVATARSIZE = 50f;
    private final static String AVATERORIGINALPIC = "Pepse\\src\\pepse\\assets\\assets\\idle_0.png";
    private static final float GRAVITY = 600;
    private static final float VELOCITY_X = 400;
    private static final float VELOCITY_Y = -300;
    private static final String AVATARTAG = "avatar";
    private final float RUNNINGCOST = 0.5f;
    private final float JUMPCOST = 10f;
    private final float ANIMATINGTIME = 0.2f;
    private final float MAXENERGY = 100;
    private List<JumpObserver> observers = new ArrayList<>();
    private final String[] IDLEPICS = new String[] {
            "Pepse\\src\\pepse\\assets\\assets\\idle_0.png",
            "Pepse\\src\\pepse\\assets\\assets\\idle_1.png",
            "Pepse\\src\\pepse\\assets\\assets\\idle_2.png",
            "Pepse\\src\\pepse\\assets\\assets\\idle_3.png" };
    private final String[] JUMPPICS = new String[] {
            "Pepse\\src\\pepse\\assets\\assets\\jump_0.png",
            "Pepse\\src\\pepse\\assets\\assets\\jump_1.png",
            "Pepse\\src\\pepse\\assets\\assets\\jump_2.png",
            "Pepse\\src\\pepse\\assets\\assets\\jump_3.png" };
    private final String[] RUNNINGPICS = new String[] {
            "Pepse\\src\\pepse\\assets\\assets\\run_0.png",
            "Pepse\\src\\pepse\\assets\\assets\\run_1.png",
            "Pepse\\src\\pepse\\assets\\assets\\run_2.png",
            "Pepse\\src\\pepse\\assets\\assets\\run_3.png",
            "Pepse\\src\\pepse\\assets\\assets\\run_4.png",
            "Pepse\\src\\pepse\\assets\\assets\\run_5.png" };
    private final Renderable[] IDLEFRAMES = new Renderable[4];
    private final Renderable[] JUMPFRAMES = new Renderable[4];
    private final Renderable[] RUNNINGFRAMES = new Renderable[6];
    private float energy = 100f;
    private UserInputListener inputListener;
    private AnimationRenderable idleAnimationRenderable;
    private AnimationRenderable jumpingAnimationRenderable;
    private AnimationRenderable runningAnimationRenderable;

    /**
     * Constructs an Avatar object with the specified position, input listener, and
     * image reader.
     *
     * @param pos           The position of the avatar.
     * @param inputListener The input listener for controlling the avatar.
     * @param imageReader   The image reader for reading avatar images.
     */
    public Avatar(Vector2 pos, UserInputListener inputListener, ImageReader imageReader) {
        super(pos, Vector2.ONES.mult(AVATARSIZE),
                imageReader.readImage(AVATERORIGINALPIC, true));
        physics().preventIntersectionsFromDirection(Vector2.ZERO);
        transform().setAccelerationY(GRAVITY);
        this.inputListener = inputListener;
        int i = 0;
        for (String pic : this.IDLEPICS) {
            this.IDLEFRAMES[i] = imageReader.readImage(pic, true);
            i++;
        }
        i = 0;
        for (String pic : this.JUMPPICS) {
            this.JUMPFRAMES[i] = imageReader.readImage(pic, true);
            i++;
        }
        i = 0;
        for (String pic : this.RUNNINGPICS) {
            this.RUNNINGFRAMES[i] = imageReader.readImage(pic, true);
            i++;
        }
        this.runningAnimationRenderable = new AnimationRenderable(RUNNINGFRAMES,
                ANIMATINGTIME);
        this.jumpingAnimationRenderable = new AnimationRenderable(JUMPFRAMES,
                ANIMATINGTIME);
        this.idleAnimationRenderable = new AnimationRenderable(IDLEFRAMES, ANIMATINGTIME);
        this.renderer().setRenderable(idleAnimationRenderable);
        this.setTag(AVATARTAG);
    }

    /**
     * Registers an observer to receive notifications about avatar jumps.
     *
     * @param observer The observer to be registered.
     */
    @Override
    public void registerObserver(JumpObserver observer){
        observers.add(observer);
    }

    /**
     * Removes an observer from the list of registered observers.
     *
     * @param observer The observer to be removed.
     */
    @Override
    public void removeObserver(JumpObserver observer) {
        observers.remove(observer);
    }

    /**
     * Notifies all registered observers when the avatar jumps.
     */
    @Override
    public void notifyObservers() {
        for (JumpObserver observer : observers) {
            observer.onAvatarJump();
        }
    }
    /**
     * Updates the avatar's state based on the input and time elapsed.
     *
     * @param deltaTime The time elapsed since the last update.
     */
    @Override
    public void update(float deltaTime) {
        super.update(deltaTime);
        float xVel = 0;
        int flag = 0;
        if (inputListener.isKeyPressed(KeyEvent.VK_LEFT)) {
            if (this.energy >= RUNNINGCOST) {
                xVel -= VELOCITY_X;
                this.energy = this.energy - RUNNINGCOST;
                flag = 1;
                this.renderer().setRenderable(runningAnimationRenderable);
                this.renderer().setIsFlippedHorizontally(true);
            }
        }
        if (inputListener.isKeyPressed(KeyEvent.VK_RIGHT)) {
            if (this.energy >= RUNNINGCOST) {
                xVel += VELOCITY_X;
                this.energy = this.energy - RUNNINGCOST;
                flag = 1;
                this.renderer().setRenderable(runningAnimationRenderable);
                this.renderer().setIsFlippedHorizontally(false);
            }
        }
        transform().setVelocityX(xVel);
        if (inputListener.isKeyPressed(KeyEvent.VK_SPACE) && getVelocity().y() == 0) {
            if (this.energy >= JUMPCOST) {
                transform().setVelocityY(VELOCITY_Y);
                this.energy = this.energy - JUMPCOST;
                flag = 1;
                this.renderer().setRenderable(jumpingAnimationRenderable);
                this.notifyObservers();
            }
        }
        if (getVelocity().y() != 0) {
            flag = 1;
        }
        if (flag == 0) {
            if (this.energy < MAXENERGY) {
                if (this.energy + 1 > MAXENERGY) {
                    this.energy = MAXENERGY;
                }
                else {
                    this.energy ++;
                }
            }
            this.renderer().setRenderable(idleAnimationRenderable);
        }
    }

    /**
     * Increases the energy of the avatar by the specified amount.
     *
     * @param energyToGain The amount of energy to increase.
     */
    public void energyGain(float energyToGain) {
        if (this.energy + energyToGain > MAXENERGY) {
            this.energy = MAXENERGY;
        }
        else {
            this.energy = this.energy + energyToGain;
        }
    }

    /**
     * Retrieves the energy of the avatar.
     *
     * @return The energy of the avatar.
     */
    public float getEnergy() {
        return this.energy;
```

```java
    }

    /**
     * Retrieves the tag of the avatar.
     *
     * @return The tag of the avatar.
     */
    public static String getAvatarTag() {
        return AVATARTAG;
    }
}
```

21 pepse/world/Block.java

```java
package pepse.world;

import danogl.GameObject;
import danogl.components.GameObjectPhysics;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;
/**
 * Represents a block in the game environment.
 */
public class Block extends GameObject{
    private static final int SIZE = 30;
    private static final String BLOCKTAG = "ground_block";

    /**
    * Constructs a Block object with the specified top-left corner position and renderable.
    * @param topLeftCorner The top-left corner position of the block.
    * @param renderable The renderable object to be displayed for the block.
    */
    public Block(Vector2 topLeftCorner, Renderable renderable) {
        super(topLeftCorner, Vector2.ONES.mult(SIZE), renderable);
        physics().preventIntersectionsFromDirection(Vector2.ZERO);
        physics().setMass(GameObjectPhysics.IMMOVABLE_MASS);
        setTag(BLOCKTAG);
    }

    /**
    * Retrieves the size of a block.
    * @return The size of a block.
    */
    public static int getSize(){
        return Block.SIZE;
    }
}
```

22 pepse/world/EnergyDisplay.java

```java
package pepse.world;

import java.util.function.Supplier;

import danogl.GameObject;
import danogl.gui.rendering.TextRenderable;
import danogl.util.Vector2;

/**
 * Represents an energy display object in the game.
 */
public class EnergyDisplay extends GameObject{
    private final Supplier<Float> getter;

    /**
     * Constructs an EnergyDisplay object with the specified getter.
     *
     * @param get The supplier function to get the energy value.
     */
    public EnergyDisplay(Supplier<Float> get){
        super(Vector2.ZERO,Vector2.ONES.mult(30) , new TextRenderable("100%"));
        getter = get;
    }

     /**
     * Updates the energy display based on the energy value obtained from the supplier.
     *
     * @param deltaTime The time elapsed since the last update.
     */
    @Override
    public void update(float deltaTime) {
        super.update(deltaTime);
        Float amount = this.getter.get();

        renderer().setRenderable(new TextRenderable(String.valueOf(amount) + "%"));
    }


}
```

23 pepse/world/JumpObserver.java

```java
package pepse.world;

/**
 * The JumpObserver interface defines the contract for classes that observe the jumping behavior of an avatar.
 */
public interface JumpObserver {
    /**
     * This method is called when the avatar jumps.
     */
    void onAvatarJump();
}
/**
 * The JumpSubject interface defines the contract for the subject (avatar) that other classes can observe.
 */
interface JumpSubject {
     /**
     * Registers an observer to receive notifications about avatar jumps.
     *
     * @param observer The observer to be registered.
     */
    void registerObserver(JumpObserver observer);
    /**
     * Removes an observer from the list of registered observers.
     *
     * @param observer The observer to be removed.
     */
    void removeObserver(JumpObserver observer);
    /**
     * Notifies all registered observers when the avatar jumps.
     */
    void notifyObservers();
}
```

24 pepse/world/Sky.java

```java
package pepse.world;

import danogl.GameObject;
import danogl.components.CoordinateSpace;
import danogl.gui.rendering.RectangleRenderable;
import danogl.util.Vector2;
import java.awt.Color;

/**
 * Represents the sky in the game.
 */
public class Sky {
    private static final Color BASIC_SKY_COLOR = Color.decode("#80c6e5");
    private static final String SKY_TAG = "sky";

    private Sky() {
    }

    /**
     * Creates a sky game object with the specified window dimensions.
     *
     * @param windowDimensions The dimensions of the game window.
     * @return The created sky game object.
     */
    public static GameObject create(Vector2 windowDimentions) {
        GameObject sky = new GameObject(Vector2.ZERO, windowDimentions,
                new RectangleRenderable(BASIC_SKY_COLOR));
        sky.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
        sky.setTag(SKY_TAG);
        return sky;
    }
}
```

25 pepse/world/Terrain.java

```java
package pepse.world;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.awt.Color;

import danogl.gui.rendering.RectangleRenderable;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;
import pepse.util.ColorSupplier;
import pepse.util.NoiseGenerator;

/**
 * Represents the terrain in the game environment.
 */
public class Terrain {
    private final float groundHeightAtX0;
    private static final Color BASE_GROUND_COLOR = new Color(212, 123,74);
    private static final int TERRAIN_DEPTH = 20;
    private final double GROUNDHEIGHTMULT = 2.0/3.0;
    private final int NOISEGENMULT = 7;
    private final NoiseGenerator noiseGenerator;
    private double seed;

    /**
     * Constructs a Terrain object with the specified window dimensions and seed.
     * @param windowDimensions The dimensions of the game window.
     * @param seed The seed used for generating pseudo-random noise.
     */
    public Terrain(Vector2 windowDimensions, int seed){
            this.groundHeightAtX0 = (float) (windowDimensions.y()*(GROUNDHEIGHTMULT));
            this.seed = seed;
            this.noiseGenerator = new NoiseGenerator(this.seed, (int)groundHeightAtX0);
        }

    /**
     * Gets the ground height at the specified x-coordinate.
     * @param x The x-coordinate.
     * @return The ground height at the specified x-coordinate.
     */
    public float groundHeightAt(float x) {
            float noise = (float) noiseGenerator.noise(x, Block.getSize() * NOISEGENMULT);
            return groundHeightAtX0 + noise;
        }

     /**
     * Creates a list of blocks within the specified range.
     * @param minX The minimum x-coordinate.
     * @param maxX The maximum x-coordinate.
     * @return The list of blocks within the specified range.
     */
    public List<Block> createInRange(int minX, int maxX) {
        List<Block> blocks = new ArrayList<>();
        for (int x = minX; x <= maxX; x+= Block.getSize()) {
            for (int i = 0; i < TERRAIN_DEPTH; i++) {
                float block_height = (float)Math.floor(groundHeightAt(x) / Block.getSize()) *
                Block.getSize() + i*Block.getSize();
                Vector2 block_position = new Vector2(x, block_height);
                Renderable renderable = new RectangleRenderable
                (ColorSupplier.approximateColor(BASE_GROUND_COLOR));
                Block block = new Block(block_position, renderable);
                blocks.add(block);
            }
        }
        return blocks;
    }
}
```

26 pepse/world/daynight/Night.java

```java
package pepse.world.daynight;

import java.awt.Color;

import danogl.GameObject;
import danogl.components.CoordinateSpace;
import danogl.components.Transition;
import danogl.components.Transition.TransitionType;
import danogl.gui.rendering.RectangleRenderable;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;
import pepse.util.ColorSupplier;

/**
 * Represents the night environment in the game.
 */
public class Night {
    private static final String NIGHT_TAG = "night";
    private static final Float MIDNIGHT_OPACITY = 0.5f;


    /**
     * Creates a GameObject representing the night environment.
     *
     * @param windowDimensions The dimensions of the game window.
     * @param cycleLength The length of the cycle for opacity transition.
     * @return A GameObject representing the night environment.
     */
    public static GameObject create(Vector2 windowDimensions,float cycleLength){
        Renderable renderable = new RectangleRenderable(Color.BLACK);
        GameObject night = new GameObject(Vector2.ZERO, windowDimensions, renderable);
        night.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
        night.setTag(NIGHT_TAG);
        new Transition<Float>(night, night.renderer()::setOpaqueness
        , 0.0f, MIDNIGHT_OPACITY, Transition.CUBIC_INTERPOLATOR_FLOAT, cycleLength,
         TransitionType.TRANSITION_BACK_AND_FORTH, null);
        return night;
    }

}
```

27 pepse/world/daynight/Sun.java

```java
package pepse.world.daynight;

import java.awt.Color;

import danogl.GameObject;
import danogl.components.CoordinateSpace;
import danogl.components.Transition;
import danogl.components.Transition.TransitionType;
import danogl.gui.rendering.OvalRenderable;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;

/**
 * Represents the Sun in the game environment.
 */
public class Sun {
    private static final int SUN_SIZE = 100;
    private static final String SUN_TAG = "sun";
    private static final Float FINALVAL = 360f;
    private static final int STARTINGPLACEPARAMX = 2;
    private static final int STARTINGPLACEPARAMY = 3;

    /**
     * Creates a sun object with the specified window dimensions and cycle length.
     * @param windowDimensions The dimensions of the window.
     * @param cycleLength The length of the cycle.
     * @return A GameObject representing the sun.
     */
    public static GameObject create(Vector2 windowDimensions, float cycleLength) {
        Renderable renderable = new OvalRenderable(Color.YELLOW);
        Vector2 starting_place = new Vector2(windowDimensions.x() / STARTINGPLACEPARAMX,
                windowDimensions.y() / STARTINGPLACEPARAMY);
        GameObject sun = new GameObject(starting_place, Vector2.ONES.mult(SUN_SIZE),
                renderable);
        sun.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
        sun.setTag(SUN_TAG);
        Vector2 initialSunCenter = sun.getCenter();
        Vector2 cycleCenter = new Vector2(windowDimensions.x() / 2,
                windowDimensions.y() * 2 / 3);
        new Transition<Float>(sun,
                (Float angle) -> sun.setCenter(initialSunCenter.subtract(cycleCenter)
                        .rotated(angle).add(cycleCenter)),
                0f, FINALVAL, Transition.LINEAR_INTERPOLATOR_FLOAT, cycleLength,
                TransitionType.TRANSITION_LOOP, null);
        return sun;
    }
}
```

28 pepse/world/daynight/SunHalo.java

```java
package pepse.world.daynight;

import java.awt.Color;

import danogl.GameObject;
import danogl.components.CoordinateSpace;
import danogl.gui.rendering.OvalRenderable;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;

/**
 * Represents the halo around the sun.
 */
public class SunHalo {
    private final static Color SUNHALO = new Color(255, 255, 0, 20);
    private final static int HALOSIZE = 175;
    private final static String SUNHALOTAG = "sunhalo";

    /**
     * Creates a GameObject representing the halo around the sun.
     *
     * @param sun The GameObject representing the sun.
     * @return A GameObject representing the halo around the sun.
     */
    public static GameObject create(GameObject sun){
        Renderable renderable = new OvalRenderable(SUNHALO);
        GameObject sunHalo = new GameObject(sun.getTopLeftCorner(), Vector2.ONES.mult(HALOSIZE), renderable);
        sunHalo.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
        sunHalo.setTag(SUNHALOTAG);
        return sunHalo;
    }
}
```

29 pepse/world/trees/Flora.java

```java
package pepse.world.trees;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.function.Function;
import java.util.function.Supplier;

import danogl.util.Vector2;
import pepse.world.Block;

/**
 * Represents flora in the game, including trees.
 */
public class Flora {
    private static final int MAX_HEIGHT = 4;
    private static final int NUM_OF_LEAVES = 7;
    private Function<Float, Float> terrainHeight;

    /**
     * Constructs a Flora object with the specified terrain height function.
     *
     * @param terrainHeight The function to determine terrain height.
     */
    public Flora(Function<Float, Float> terrainHeight) {
        this.terrainHeight = terrainHeight;
    }

    /**
     * Creates trees within a specified range.
     *
     * @param minX The minimum x-coordinate.
     * @param maxX The maximum x-coordinate.
     * @return A list of trees created within the specified range.
     */
    public List<Tree> createInRange(int minX, int maxX) {
        List<Tree> trees = new ArrayList<>();
        Random random = new Random();
        for (float location = minX; location < maxX; location += Block.getSize()) {
            if (random.nextDouble() < 0.1) {
                int height = random.nextInt(1, MAX_HEIGHT);
                trees.add(new Tree(height, NUM_OF_LEAVES,
                        new Vector2(location, terrainHeight.apply(location))));
            }
        }
        return trees;
    }

}
```

30 pepse/world/trees/Fruit.java

```java
package pepse.world.trees;

import java.awt.Color;
import java.util.Random;

import danogl.GameObject;
import danogl.collisions.Collision;
import danogl.components.ScheduledTask;
import danogl.gui.rendering.OvalRenderable;
import danogl.util.Vector2;
import pepse.world.Avatar;
import pepse.world.Block;

/**
 * Represents a fruit block in the game.
 */
public class Fruit extends Block {
    private static final int ENERGY_FROM_EATING = 10;
    private static float cycleLength;
    private boolean isInGame;

    /**
     * Constructs a Fruit object with the specified top-left corner position.
     *
     * @param topLeftCorner The top-left corner position of the fruit block.
     */
    public Fruit(Vector2 topLeftCorner, Color color) {
        super(topLeftCorner, new OvalRenderable(color));
        isInGame = true;
    }

    /**
     * Sets the cycle length for the game.
     *
     * @param length The length of the game cycle.
     */
    public static void setCycleLength(float length) {
        cycleLength = length;
    }

    /**
     * Determines whether the object should collide with the specified game object.
     *
     * @param other The other game object involved in the collision.
     * @return True if the object should collide with the specified game object, otherwise false.
     */
    @Override
    public boolean shouldCollideWith(GameObject other) {
        return other.getTag().equals(Avatar.getAvatarTag()) && isInGame;
    }

    /**
     * Handles the behavior when a collision occurs with another game object.
     *
     * @param other The other game object involved in the collision.
     * @param collision The collision information.
     */
    @Override
    public void onCollisionEnter(GameObject other, Collision collision) {
        if (shouldCollideWith(other)) {
            Avatar avatar = (Avatar) other;
            avatar.energyGain(ENERGY_FROM_EATING);
            removeFromGame();
            super.onCollisionEnter(other, collision);
        }
    }

    private void removeFromGame() {
        isInGame = false;
        this.renderer().setOpaqueness(0);
        new ScheduledTask(this, cycleLength, false, () -> addToGame());
    }

    private void addToGame() {
        isInGame = true;
        this.renderer().setOpaqueness(100f);
    }
}
```

31 pepse/world/trees/Fruits.java

```java
package pepse.world.trees;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import danogl.collisions.GameObjectCollection;
import danogl.gui.rendering.OvalRenderable;
import danogl.util.Vector2;
import pepse.PepseGameManager;

/**
 * Represents a collection of fruits in the game.
 */
public class Fruits {
    private static final Random random = new Random();
    private static final double SPAWNING_PROBABILITY = 0.3;
    private static final Color INITIAL_COLOR = new Color(200, 20, 144);
    private List<Fruit> fruits;

    /**
     * Constructs a collection of fruits with the specified number of fruits and
     * bottom-middle position.
     *
     * @param numOfFruits  The number of fruits to generate.
     * @param bottomMiddle The bottom-middle position for arranging the fruits.
     */
    public Fruits(int numOfFruits, Vector2 bottomMiddle) {
        int fruitSize = Leaf.getSize();
        fruits = new ArrayList<Fruit>(numOfFruits * numOfFruits);
        float initialX = bottomMiddle.x() - ((numOfFruits - 1) / 2 * fruitSize);
        float y = bottomMiddle.y() - (numOfFruits * fruitSize);
        float x;
        for (int i = 0; i < numOfFruits; i++) {
            x = initialX;
            for (int j = 0; j < numOfFruits; j++) {
                if (random.nextDouble() < SPAWNING_PROBABILITY) {
                    fruits.add(new Fruit(new Vector2(x, y), INITIAL_COLOR));
                }
                x += fruitSize;
            }
            y += fruitSize;
        }
    }

    /**
     * Adds the fruits to the specified game object collection.
     *
     * @param collection The game object collection to which the fruits will be
     *                   added.
     */
    public void addFruits(GameObjectCollection collection) {
        for (Fruit fruit : fruits) {
            collection.addGameObject(fruit);
        }
    }

    /**
```

```java
     * Changes the color of all the fruits.
     *
     * @param color The new color to apply to the fruits.
     */
    public void changeColor(Color color) {
        for (Fruit fruit : fruits) {
            fruit.renderer().setRenderable(new OvalRenderable(color));
        }
    }
}
```

32 pepse/world/trees/Leaf.java

```java
package pepse.world.trees;

import java.awt.Color;
import java.util.Random;

import danogl.components.ScheduledTask;
import danogl.components.Transition;
import danogl.components.Transition.TransitionType;
import danogl.gui.rendering.RectangleRenderable;
import danogl.util.Vector2;
import pepse.world.Block;

/**
 * Represents a leaf object in the game.
 */
public class Leaf extends Block {
    private static final Color BASIC_COLOR = new Color(50, 200, 30);
    private static final float FINAL_ROTATING_ANGLE = 45f;
    private static final float TRANSITION_LENGTH = 2;
    private static final float MIN_WIDTH = 10f;
    private static final float MAX_WIDTH = 60f;
    private static final Random random = new Random();

    /**
     * Constructs a leaf object with the specified top-left corner position.
     *
     * @param topLeftCorner The top-left corner position of the leaf.
     */
    public Leaf(Vector2 topLeftCorner) {
        super(topLeftCorner, new RectangleRenderable(BASIC_COLOR));
    }

    /**
     * Adds a transition effect to the leaf object.
     */
    public void addTransition() {
        new ScheduledTask(this, (float) random.nextDouble(), false,
                () -> setTransition());
    }

    private void setTransition() {
        float leafHeight = getSize();
        new Transition<Float>(this,
                (Float angle) -> this.renderer().setRenderableAngle(angle), 0f,
                FINAL_ROTATING_ANGLE, Transition.LINEAR_INTERPOLATOR_FLOAT,
                TRANSITION_LENGTH, TransitionType.TRANSITION_BACK_AND_FORTH, null);
        new Transition<Float>(this,
                (Float width) -> this.setDimensions(new Vector2(width, leafHeight)),
                MIN_WIDTH, MAX_WIDTH, Transition.LINEAR_INTERPOLATOR_FLOAT,
                TRANSITION_LENGTH, TransitionType.TRANSITION_BACK_AND_FORTH, null);
    }
}
```

33 pepse/world/trees/Leaves.java

```java
package pepse.world.trees;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import danogl.collisions.GameObjectCollection;
import danogl.collisions.Layer;
import danogl.components.ScheduledTask;
import danogl.components.Transition;
import danogl.components.Transition.TransitionType;
import danogl.gui.rendering.RectangleRenderable;
import danogl.util.Vector2;

/**
 * Represents a collection of leaves in the game.
 */
public class Leaves {
    private static final Random random = new Random();
    private static final double SPAWNING_PROBABILITY = 0.6;
    private static final int LEAVES_LAYER = -50;
    private List<Leaf> leaves;

    /**
     * Constructs a collection of leaves with the specified number of leaves and
     * bottom middle position.
     *
     * @param numOfLeaves  The number of leaves.
     * @param bottomMiddle The bottom middle position.
     */
    public Leaves(int numOfLeaves, Vector2 bottomMiddle) {
        int leafSize = Leaf.getSize();
        leaves = new ArrayList<Leaf>(numOfLeaves * numOfLeaves);
        float initialX = bottomMiddle.x() - ((numOfLeaves - 1) / 2 * leafSize);
        float y = bottomMiddle.y() - (numOfLeaves * leafSize);
        float x;
        for (int i = 0; i < numOfLeaves; i++) {
            x = initialX;
            for (int j = 0; j < numOfLeaves; j++) {
                if (random.nextDouble() < SPAWNING_PROBABILITY) {
                    leaves.add(new Leaf(new Vector2(x, y)));
                }
                x += leafSize;
            }
            y += leafSize;
        }
        setTransitions();
    }

    /**
     * Adds the leaves to the specified game object collection.
     *
     * @param collection The game object collection to add the leaves to.
     */
    public void addLeaves(GameObjectCollection collection) {
        for (Leaf leaf : leaves) {
            collection.addGameObject(leaf, LEAVES_LAYER);
        }
    }

    /**
     * Rotates all the leaves by 90 degrees.
     */
    public void rotate90Degrees() {
        for (Leaf leaf : leaves) {
            float curAngle = leaf.renderer().getRenderableAngle();
            leaf.renderer().setRenderableAngle(curAngle + 90);
        }
    }

    private void setTransitions() {
        for (Leaf leaf : leaves) {
            leaf.addTransition();
        }
    }

}
```

34 pepse/world/trees/Tree.java

```java
package pepse.world.trees;

import java.awt.Color;
import java.util.Random;

import danogl.collisions.GameObjectCollection;
import danogl.util.Vector2;

/**
 * Represents a tree object in the game.
 */
public class Tree {
    private final Trunk trunk;
    private final Leaves leaves;
    private final Fruits fruits;

    /**
     * Constructs a tree with the specified height, number of leaves, and position.
     *
     * @param height      The height of the tree trunk.
     * @param numOfLeaves The number of leaves on the tree.
     * @param placeToPut  The position to place the tree.
     */
    public Tree(int height, int numOfLeaves, Vector2 placeToPut) {
        trunk = new Trunk(placeToPut, height);
        Vector2 trunkTop = trunk.getTopLeftCorner();
        leaves = new Leaves(numOfLeaves, trunkTop);
        fruits = new Fruits(numOfLeaves, trunkTop);
    }

    /**
     * Adds the tree to the specified game object collection.
     *
     * @param collection The game object collection to which the tree will be added.
     */
    public void addTree(GameObjectCollection collection) {
        trunk.addTrunk(collection);
        leaves.addLeaves(collection);
        fruits.addFruits(collection);
    }

    /**
     * Notifies the observer that the avatar has jumped.
     *
     * @param newFruitColor The new color for the fruits.
     */
    public void onAvatarJump(Color newFruitColor) {
        trunk.changeColor();
        leaves.rotate90Degrees();
        fruits.changeColor(newFruitColor);
    }
}
```

35 pepse/world/trees/Trunk.java

```java
package pepse.world.trees;

import danogl.GameObject;
import danogl.collisions.GameObjectCollection;
import danogl.collisions.Layer;
import danogl.gui.rendering.RectangleRenderable;
import danogl.gui.rendering.Renderable;
import danogl.util.Vector2;
import pepse.util.ColorSupplier;
import pepse.world.Block;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

/**
 * Represents the trunk of a tree in the game.
 */
public class Trunk {
    private static final Color BASIC_COLOR = new Color(100, 50, 20);
    private final List<Block> trunk;
    private final Vector2 topLeftCorner;

    /**
     * Constructs a trunk with the specified bottom-left corner position and height.
     *
     * @param bottomLeftCorner The bottom-left corner position of the trunk.
     * @param height           The height of the trunk.
     */
    public Trunk(Vector2 bottomLeftCorner, int height) {
        trunk = new ArrayList<Block>(height);
        float x = bottomLeftCorner.x();
        float y = bottomLeftCorner.y();
        for (int blockNum = 0; blockNum < height; blockNum++) {
            y -= Block.getSize();
            trunk.add(new Block(new Vector2(x, y), new RectangleRenderable(BASIC_COLOR)));
        }
        topLeftCorner = new Vector2(x, y);
    }

    /**
     * Adds the trunk blocks to the specified game object collection.
     *
     * @param collection The game object collection to which the trunk blocks will
     *                   be added.
     */
    public void addTrunk(GameObjectCollection collection) {
        for (Block block : trunk) {
            collection.addGameObject(block, Layer.STATIC_OBJECTS);
        }
    }

    /**
     * Gets the top-left corner position of the trunk.
     *
     * @return The top-left corner position of the trunk.
     */
    public Vector2 getTopLeftCorner() {
        return new Vector2(topLeftCorner.x(), topLeftCorner.y());
    }

    /**
     * Changes the color of the trunk blocks to a new color.
     */
    public void changeColor() {
        Color newColor = ColorSupplier.approximateColor(BASIC_COLOR);
        for (Block block : trunk) {
            block.renderer().setRenderable(new RectangleRenderable(newColor));
        }
    }
}
```