

Contents

1	Basic Test Results	2
2	README	3
3	pepse/PepseGameManager.java	4
4	assets/assets/idle 0.png	6
5	assets/assets/idle 1.png	7
6	assets/assets/idle 2.png	8
7	assets/assets/idle 3.png	9
8	assets/assets/jump 0.png	10
9	assets/assets/jump 1.png	11
10	assets/assets/jump 2.png	12
11	assets/assets/jump 3.png	13
12	assets/assets/run 0.png	14
13	assets/assets/run 1.png	15
14	assets/assets/run 2.png	16
15	assets/assets/run 3.png	17
16	assets/assets/run 4.png	18
17	assets/assets/run 5.png	19
18	pepse/util/ColorSupplier.java	20
19	pepse/util/NoiseGenerator.java	21
20	pepse/world/Avatar.java	24
21	pepse/world/Block.java	28
22	pepse/world/EnergyDisplay.java	29
23	pepse/world/JumpObserver.java	30

24	pepse/world/Sky.java	31
25	pepse/world/Terrain.java	32
26	pepse/world/daynight/Night.java	34
27	pepse/world/daynight/Sun.java	35
28	pepse/world/daynight/SunHalo.java	36
29	pepse/world/trees/Flora.java	37
30	pepse/world/trees/Fruit.java	38
31	pepse/world/trees/Fruits.java	40
32	pepse/world/trees/Leaf.java	42
33	pepse/world/trees/Leaves.java	43
34	pepse/world/trees/Tree.java	45
35	pepse/world/trees/Trunk.java	46

1 Basic Test Results

```
1  =====
2  ===== EX4 TESTER =====
3  =====
4
5
6  ===== EXTRACTING =====
7
8
9  ===== CHECKING FILES =====
10
11
12  ===== COPYING NECESSARY DIRECTORIES =====
13
14
15  ===== ANALYZE README =====
16  Whats up guytra2205, yuvaleyal?
17
18
19  ==== CHECKING LINE LENGTHS ====
20  Lines lengths issue
21  Line 10 in file ./README exceeds the allowed line length (described by the coding style) (length is 183)
22  Line 11 in file ./README exceeds the allowed line length (described by the coding style) (length is 289)
23  Line 12 in file ./README exceeds the allowed line length (described by the coding style) (length is 154)
24  Line 13 in file ./README exceeds the allowed line length (described by the coding style) (length is 193)
25  Line 15 in file ./README exceeds the allowed line length (described by the coding style) (length is 132)
26  Line 19 in file ./README exceeds the allowed line length (described by the coding style) (length is 153)
27  Line 20 in file ./README exceeds the allowed line length (described by the coding style) (length is 160)
28
29
30  ==== CHECKING DOCUMENTATION ====
31
32
33
34  ===== ERRORS =====
35  AUTO.LINE_TOO_LONG
36  AUTO.LINE_TOO_LONG
37  AUTO.LINE_TOO_LONG
38  AUTO.LINE_TOO_LONG
39  AUTO.LINE_TOO_LONG
```

2 README

```
1  guytra2205, yuvaleyai
2  214987745, 326660610
3
4  we created the Trees package using the following classes:
5  1. the Flora class, that is responsible for creating all of the trees in the game.
6  2. the Tree class, representing a tree. a tree objects contains 3 fields:
7      a trunk (instanse of Trunk)
8      leaves (instanse of Leaves)
9      fruits (instanse of Fruits)
10 3. Trunk: this class represents the trunk of a tree, and contains functions that add the trunk to the game objects, changes
11 4. Leaves and Fruits are classes contening 1 (non-final) field each - a List of objects of Leaf and Fruit, respectively. th
12 5. Leaf: a class representing a single leaf. this class handles the things that affects each leaf in a different way, like t
13 6. Fruit: a class representing a single fruit. like the Leaf class, this class handles the things that affects each fruit in
14
15 in short, the Flora class creates a List of Tree insatnces. a Tree contains a Trunk instace, a Leaves instace and a Fruits i
16 Leaves contains a List of Leaf instaces, and Fruits contains a List of Fruit instaces.
17
18 The design paterns we used are:
19 1. Facade: the Tree class wraps and handles all of the tree logic, and its the only one (except Flora) that is accessed from
20 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

```
1  package pepse;
2
3  import java.awt.Color;
4  import java.util.List;
5  import java.util.Random;
6
7  import danogl.GameManager;
8  import danogl.GameObject;
9  import danogl.collisions.Layer;
10 import danogl.gui.GameGUIComponent;
11 import danogl.gui.ImageReader;
12 import danogl.gui.SoundReader;
13 import danogl.gui.UserInputListener;
14 import danogl.gui.WindowController;
15 import danogl.util.Vector2;
16 import pepse.world.Sky;
17 import pepse.world.Avatar;
18 import pepse.world.Block;
19 import pepse.world.EnergyDisplay;
20 import pepse.world.JumpObserver;
21 import pepse.world.Terrain;
22 import pepse.world.daynight.Night;
23 import pepse.world.daynight.Sun;
24 import pepse.world.daynight.SunHalo;
25 import pepse.world.trees.Flora;
26 import pepse.world.trees.Fruit;
27 import pepse.world.trees.Tree;
28
29 /**
30  * Represents the game manager for the Pepse game.
31  */
32 public class PepseGameManager extends GameManager implements JumpObserver {
33     private final int CYCLELENGTH = 30;
34     private static final int MAX_COLOR = 256;
35     private List<Tree> trees;
36
37     /**
38      * Constructs a PepseGameManager object.
39      */
40     public PepseGameManager() {
41         super();
42     }
43
44     /**
45      * Initializes the Pepse game.
46      *
47      * @param imageReader The image reader for loading game images.
48      * @param soundReader The sound reader for loading game sounds.
49      * @param inputListener The user input listener for handling input events.
50      * @param windowController The window controller for managing the game window.
51      */
52     @Override
53     public void initializeGame(ImageReader imageReader, SoundReader soundReader,
54                               UserInputListener inputListener, WindowController windowController) {
55         super.initializeGame(imageReader, soundReader, inputListener, windowController);
56         Vector2 windowDimensions = windowController.getWindowDimensions();
57         GameObject sky = Sky.create(windowDimensions);
58         gameObjects().addGameObject(sky, Layer.BACKGROUND);
59         int seed = (int) new Random().nextInt();
```

```

60     Terrain terrain = new Terrain(windowDimensions, seed);
61     List<Block> list_of_blocks = terrain.createInRange(0,
62         (int) windowController.getWindowDimensions().x());
63     for (Block block : list_of_blocks) {
64         gameObjects().addGameObject(block);
65     }
66     GameObject night = Night.create(windowDimensions, CYCLELENGTH);
67     gameObjects().addGameObject(night, Layer.UI);
68     GameObject sun = Sun.create(windowDimensions, CYCLELENGTH * 2);
69     GameObject sunHalo = SunHalo.create(sun);
70     sunHalo.addComponent((float deltaTime) -> {
71         sunHalo.setCenter(sun.getCenter());
72     });
73     gameObjects().addGameObject(sun, Layer.BACKGROUND);
74     gameObjects().addGameObject(sunHalo, Layer.BACKGROUND);
75     Vector2 pos = new Vector2(20, terrain.groundHeightAt(20) - Block.getSize());
76     Avatar avater = new Avatar(pos, inputListener, imageReader);
77     avater.registerObserver(this);
78     gameObjects().addGameObject(avater);
79     EnergyDisplay display = new EnergyDisplay(avater::getEnergy);
80     gameObjects().addGameObject(display, Layer.UI);
81     Flora flora = new Flora(x -> terrain.groundHeightAt(x));
82     trees = flora.createInRange(0, (int) windowDimensions.x());
83     addTreesToGame(trees);
84     Fruit.setCycleLength(CYCLELENGTH);
85 }
86
87 public void onAvatarJump() {
88     Color fruitColor = randomColor();
89     for (Tree tree : trees) {
90         tree.onAvatarJump(fruitColor);
91     }
92 }
93
94 private void addTreesToGame(List<Tree> trees) {
95     for (Tree tree : trees) {
96         tree.addTree(gameObjects());
97     }
98 }
99
100 private static Color randomColor() {
101     Random random = new Random();
102     return new Color(random.nextInt(0, MAX_COLOR), random.nextInt(0, MAX_COLOR),
103         random.nextInt(0, MAX_COLOR));
104 }
105
106 /**
107  * The main method to start the Pepse game.
108  *
109  * @param args Command-line arguments.
110  */
111 public static void main(String[] args) {
112     PepseGameManager gameManager = new PepseGameManager();
113     gameManager.run();
114 }
115 }

```

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

```
1  package pepse.util;
2
3  import java.awt.*;
4  import java.util.Random;
5
6  /**
7   * Provides procedurally-generated colors around a pivot.
8   * @author Dan Nirel
9   */
10 public final class ColorSupplier {
11     private static final int DEFAULT_COLOR_DELTA = 10;
12     private final static Random random = new Random();
13
14     /**
15      * Returns a color similar to baseColor, with a default delta.
16      *
17      * @param baseColor A color that we wish to approximate.
18      * @return A color similar to baseColor.
19      */
20     public static Color approximateColor(Color baseColor) {
21         return approximateColor(baseColor, DEFAULT_COLOR_DELTA);
22     }
23
24     /**
25      * Returns a color similar to baseColor, with a difference of at most colorDelta.
26      *
27      * @param baseColor A color that we wish to approximate.
28      * @param colorDelta The maximal difference (per channel) between the sampled color and the base color.
29      * @return A color similar to baseColor.
30      */
31     public static Color approximateColor(Color baseColor, int colorDelta) {
32
33         return new Color(
34             randomChannelInRange(baseColor.getRed()-colorDelta, baseColor.getRed()+colorDelta),
35             randomChannelInRange(baseColor.getGreen()-colorDelta, baseColor.getGreen()+colorDelta),
36             randomChannelInRange(baseColor.getBlue()-colorDelta, baseColor.getBlue()+colorDelta));
37     }
38
39     /**
40      * This method generates a random value for a color channel within the given range [min, max].
41      *
42      * @param min The lower bound of the given range.
43      * @param max The upper bound of the given range.
44      * @return A random number in the range [min, max], clipped to [0,255].
45      */
46     private static int randomChannelInRange(int min, int max) {
47         int channel = random.nextInt(max-min+1) + min;
48         return Math.min(255, Math.max(channel, 0));
49     }
50 }
```

19 pepse/util/NoiseGenerator.java

```
1  package pepse.util;
2
3  import java.util.Random;
4
5  public class NoiseGenerator {
6      private double seed;
7      private long default_size;
8      private int[] p;
9      private int[] permutation;
10     private double startPoint;
11
12     /**
13      * The constructor of the NoiseGenerator class.
14      *
15      * @param seed can be anything you want (even 1234 or new Random().nextGaussian()).
16      *           This seed is the basis of the random generator, which
17      *           will draw upon it to generate pseudo-random noise.
18      *
19      * @param startPoint is a relative point that the noise will be generated from.
20      *           In our case it should be your ground height at X0 (specified in
21      *           ex4 when we talk about the terrain: 2.2.1).
22      *
23      */
24     public NoiseGenerator(double seed, int startPoint) {
25         this.seed = seed;
26         this.startPoint = startPoint;
27         init();
28     }
29
30     private void init() {
31         // Initialize the permutation array.
32         this.p = new int[512];
33         this.permutation = new int[]{151, 160, 137, 91, 90, 15, 131, 13, 201,
34             95, 96, 53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142, 8, 99,
35             37, 240, 21, 10, 23, 190, 6, 148, 247, 120, 234, 75, 0, 26,
36             197, 62, 94, 252, 219, 203, 117, 35, 11, 32, 57, 177, 33, 88,
37             237, 149, 56, 87, 174, 20, 125, 136, 171, 168, 68, 175, 74,
38             165, 71, 134, 139, 48, 27, 166, 77, 146, 158, 231, 83, 111,
39             229, 122, 60, 211, 133, 230, 220, 105, 92, 41, 55, 46, 245, 40,
40             244, 102, 143, 54, 65, 25, 63, 161, 1, 216, 80, 73, 209, 76,
41             132, 187, 208, 89, 18, 169, 200, 196, 135, 130, 116, 188, 159,
42             86, 164, 100, 109, 198, 173, 186, 3, 64, 52, 217, 226, 250,
43             124, 123, 5, 202, 38, 147, 118, 126, 255, 82, 85, 212, 207,
44             206, 59, 227, 47, 16, 58, 17, 182, 189, 28, 42, 223, 183, 170,
45             213, 119, 248, 152, 2, 44, 154, 163, 70, 221, 153, 101, 155,
46             167, 43, 172, 9, 129, 22, 39, 253, 19, 98, 108, 110, 79, 113,
47             224, 232, 178, 185, 112, 104, 218, 246, 97, 228, 251, 34, 242,
48             193, 238, 210, 144, 12, 191, 179, 162, 241, 81, 51, 145, 235,
49             249, 14, 239, 107, 49, 192, 214, 31, 181, 199, 106, 157, 184,
50             84, 204, 176, 115, 121, 50, 45, 127, 4, 150, 254, 138, 236,
51             205, 93, 222, 114, 67, 29, 24, 72, 243, 141, 128, 195, 78, 66,
52             215, 61, 156, 180};
53         this.default_size = 35;
54
55         // Populate it
56         for (int i = 0; i < 256; i++) {
57             p[256 + i] = p[i] = permutation[i];
58         }
59     }
```

```

60     }
61
62     /**
63      * Noise is responsible to generate pseudo random noise according to the seed given upon constructing the object.
64      *
65      * @param x the wanted x to receive noise for (in our case, the x coordinate of the terrain you'd want to create).
66      * @param factor describes how large the noise should be (play with it, but BLOCK_SIZE *7 should be enough).
67      * @return returns a noise you should *add* to the groundHeightAtX0 you have.
68      *
69      * example:
70      * public float groundHeightAt(float x) {
71      *     float noise = (float) noiseGenerator.noise(x, BLOCK_SIZE *7);
72      *     return groundHeightAtX0 + noise;
73      * }
74      *
75     */
76     public double noise(double x, double factor) {
77         double value = 0.0;
78         double currentPoint = startPoint;
79
80         while (currentPoint >= 1) {
81             value += smoothNoise((x / currentPoint), 0, 0) * currentPoint;
82             currentPoint /= 2.0;
83         }
84
85         return value * factor / startPoint;
86     }
87
88
89     private double smoothNoise(double x, double y, double z) {
90         // Offset each coordinate by the seed value
91         x += this.seed;
92         y += this.seed;
93         z += this.seed;
94
95         int X = (int) Math.floor(x) & 255; // FIND UNIT CUBE THAT
96         int Y = (int) Math.floor(y) & 255; // CONTAINS POINT.
97         int Z = (int) Math.floor(z) & 255;
98
99         x -= Math.floor(x); // FIND RELATIVE X,Y,Z
100        y -= Math.floor(y); // OF POINT IN CUBE.
101        z -= Math.floor(z);
102
103        double u = fade(x); // COMPUTE FADE CURVES
104        double v = fade(y); // FOR EACH OF X,Y,Z.
105        double w = fade(z);
106
107        int A = p[X] + Y;
108        int AA = p[A] + Z;
109        int AB = p[A + 1] + Z; // HASH COORDINATES OF
110        int B = p[X + 1] + Y;
111        int BA = p[B] + Z;
112        int BB = p[B + 1] + Z; // THE 8 CUBE CORNERS,
113
114        return lerp(w, lerp(v, lerp(u, grad(p[AA], x, y, z), // AND ADD
115                                     grad(p[BA], x - 1, y, z)), // BLENDED
116                lerp(u, grad(p[AB], x, y - 1, z), // RESULTS
117                     grad(p[BB], x - 1, y - 1, z))), // FROM 8
118                lerp(v, lerp(u, grad(p[AA + 1], x, y, z - 1), // CORNERS
119                             grad(p[BA + 1], x - 1, y, z - 1)), // OF CUBE
120                lerp(u, grad(p[AB + 1], x, y - 1, z - 1),
121                     grad(p[BB + 1], x - 1, y - 1, z - 1))));
122    }
123
124    private double fade(double t) {
125        return t * t * t * (t * (t * 6 - 15) + 10);
126    }
127

```

```

128     private double lerp(double t, double a, double b) {
129         return a + t * (b - a);
130     }
131
132     private double grad(int hash, double x, double y, double z) {
133         int h = hash & 15; // CONVERT LO 4 BITS OF HASH CODE
134         double u = h < 8 ? x : y, // INTO 12 GRADIENT DIRECTIONS.
135             v = h < 4 ? y : h == 12 || h == 14 ? x : z;
136         return ((h & 1) == 0 ? u : -u) + ((h & 2) == 0 ? v : -v);
137     }
138 }

```


20 pepse/world/Avatar.java

```
1 package pepse.world;
2
3 import danogl.GameObject;
4 import danogl.gui.ImageReader;
5 import danogl.gui.UserInputListener;
6 import danogl.gui.rendering.AnimationRenderable;
7 import danogl.gui.rendering.Renderable;
8 import danogl.util.Vector2;
9 import java.awt.event.KeyEvent;
10 import java.util.ArrayList;
11 import java.util.List;
12
13 /**
14  * Represents the avatar character in the game.
15  */
16 public class Avatar extends GameObject implements JumpSubject{
17     private final static float AVATARSIZE = 50f;
18     private final static String AVATERORIGINALPIC = "Pepse\\src\\pepse\\assets\\assets\\idle_0.png";
19     private static final float GRAVITY = 600;
20     private static final float VELOCITY_X = 400;
21     private static final float VELOCITY_Y = -300;
22     private static final String AVATARTAG = "avatar";
23     private final float RUNNINGCOST = 0.5f;
24     private final float JUMPCOST = 10f;
25     private final float ANIMATINGTIME = 0.2f;
26     private final float MAXENERGY = 100;
27     private List<JumpObserver> observers = new ArrayList<>();
28     private final String[] IDLEPICS = new String[] {
29         "Pepse\\src\\pepse\\assets\\assets\\idle_0.png",
30         "Pepse\\src\\pepse\\assets\\assets\\idle_1.png",
31         "Pepse\\src\\pepse\\assets\\assets\\idle_2.png",
32         "Pepse\\src\\pepse\\assets\\assets\\idle_3.png" };
33     private final String[] JUMPPICS = new String[] {
34         "Pepse\\src\\pepse\\assets\\assets\\jump_0.png",
35         "Pepse\\src\\pepse\\assets\\assets\\jump_1.png",
36         "Pepse\\src\\pepse\\assets\\assets\\jump_2.png",
37         "Pepse\\src\\pepse\\assets\\assets\\jump_3.png" };
38     private final String[] RUNNINGPICS = new String[] {
39         "Pepse\\src\\pepse\\assets\\assets\\run_0.png",
40         "Pepse\\src\\pepse\\assets\\assets\\run_1.png",
41         "Pepse\\src\\pepse\\assets\\assets\\run_2.png",
42         "Pepse\\src\\pepse\\assets\\assets\\run_3.png",
43         "Pepse\\src\\pepse\\assets\\assets\\run_4.png",
44         "Pepse\\src\\pepse\\assets\\assets\\run_5.png" };
45     private final Renderable[] IDLEFRAMES = new Renderable[4];
46     private final Renderable[] JUMPFRAMES = new Renderable[4];
47     private final Renderable[] RUNNINGFRAMES = new Renderable[6];
48     private float energy = 100f;
49     private UserInputListener inputListener;
50     private AnimationRenderable idleAnimationRenderable;
51     private AnimationRenderable jumpingAnimationRenderable;
52     private AnimationRenderable runningAnimationRenderable;
53
54     /**
55      * Constructs an Avatar object with the specified position, input listener, and
56      * image reader.
57      *
58      * @param pos The position of the avatar.
59      * @param inputListener The input listener for controlling the avatar.
```

```

60     * @param imageReader The image reader for reading avatar images.
61     */
62     public Avatar(Vector2 pos, UserInputListener inputListener, ImageReader imageReader) {
63         super(pos, Vector2.ONES.mult(AVATARSIZE),
64             imageReader.readImage(AVATERORIGINALPIC, true));
65         physics().preventIntersectionsFromDirection(Vector2.ZERO);
66         transform().setAccelerationY(GRAVITY);
67         this.inputListener = inputListener;
68         int i = 0;
69         for (String pic : this.IDLEPICS) {
70             this.IDLEFRAMES[i] = imageReader.readImage(pic, true);
71             i++;
72         }
73         i = 0;
74         for (String pic : this.JUMPPICS) {
75             this.JUMPPFRAMES[i] = imageReader.readImage(pic, true);
76             i++;
77         }
78         i = 0;
79         for (String pic : this.RUNNINGPICS) {
80             this.RUNNINGFRAMES[i] = imageReader.readImage(pic, true);
81             i++;
82         }
83         this.runningAnimationRenderable = new AnimationRenderable(RUNNINGFRAMES,
84             ANIMATINGTIME);
85         this.jumpingAnimationRenderable = new AnimationRenderable(JUMPPFRAMES,
86             ANIMATINGTIME);
87         this.idleAnimationRenderable = new AnimationRenderable(IDLEFRAMES, ANIMATINGTIME);
88         this.renderer().setRenderable(idleAnimationRenderable);
89         this.setTag(AVATARTAG);
90     }
91
92     /**
93      * Registers an observer to receive notifications about avatar jumps.
94      *
95      * @param observer The observer to be registered.
96      */
97     @Override
98     public void registerObserver(JumpObserver observer){
99         observers.add(observer);
100     }
101
102     /**
103      * Removes an observer from the list of registered observers.
104      *
105      * @param observer The observer to be removed.
106      */
107     @Override
108     public void removeObserver(JumpObserver observer) {
109         observers.remove(observer);
110     }
111
112     /**
113      * Notifies all registered observers when the avatar jumps.
114      */
115     @Override
116     public void notifyObservers() {
117         for (JumpObserver observer : observers) {
118             observer.onAvatarJump();
119         }
120     }
121
122     /**
123      * Updates the avatar's state based on the input and time elapsed.
124      *
125      * @param deltaTime The time elapsed since the last update.
126      */
127     @Override
128     public void update(float deltaTime) {

```

```

128     super.update(deltaTime);
129     float xVel = 0;
130     int flag = 0;
131     if (inputListener.isKeyPressed(KeyEvent.VK_LEFT)) {
132         if (this.energy >= RUNNINGCOST) {
133             xVel -= VELOCITY_X;
134             this.energy = this.energy - RUNNINGCOST;
135             flag = 1;
136             this.renderer().setRenderable(runningAnimationRenderable);
137             this.renderer().setIsFlippedHorizontally(true);
138         }
139     }
140     if (inputListener.isKeyPressed(KeyEvent.VK_RIGHT)) {
141         if (this.energy >= RUNNINGCOST) {
142             xVel += VELOCITY_X;
143             this.energy = this.energy - RUNNINGCOST;
144             flag = 1;
145             this.renderer().setRenderable(runningAnimationRenderable);
146             this.renderer().setIsFlippedHorizontally(false);
147         }
148     }
149     transform().setVelocityX(xVel);
150     if (inputListener.isKeyPressed(KeyEvent.VK_SPACE) && getVelocity().y() == 0) {
151         if (this.energy >= JUMPCOST) {
152             transform().setVelocityY(VELOCITY_Y);
153             this.energy = this.energy - JUMPCOST;
154             flag = 1;
155             this.renderer().setRenderable(jumpingAnimationRenderable);
156             this.notifyObservers();
157         }
158     }
159     if (getVelocity().y() != 0) {
160         flag = 1;
161     }
162     if (flag == 0) {
163         if (this.energy < MAXENERGY) {
164             if (this.energy + 1 > MAXENERGY) {
165                 this.energy = MAXENERGY;
166             }
167             else {
168                 this.energy ++;
169             }
170         }
171         this.renderer().setRenderable(idleAnimationRenderable);
172     }
173 }
174
175 /**
176  * Increases the energy of the avatar by the specified amount.
177  *
178  * @param energyToGain The amount of energy to increase.
179  */
180 public void energyGain(float energyToGain) {
181     if (this.energy + energyToGain > MAXENERGY) {
182         this.energy = MAXENERGY;
183     }
184     else {
185         this.energy = this.energy + energyToGain;
186     }
187 }
188
189 /**
190  * Retrieves the energy of the avatar.
191  *
192  * @return The energy of the avatar.
193  */
194 public float getEnergy() {
195     return this.energy;

```

```
196     }
197
198     /**
199      * Retrieves the tag of the avatar.
200      *
201      * @return The tag of the avatar.
202      */
203     public static String getAvatarTag() {
204         return AVATARTAG;
205     }
206 }
```

21 pepse/world/Block.java

```
1  package pepse.world;
2
3  import danogl.GameObject;
4  import danogl.components.GameObjectPhysics;
5  import danogl.gui.rendering.Renderable;
6  import danogl.util.Vector2;
7  /**
8   * Represents a block in the game environment.
9   */
10 public class Block extends GameObject{
11     private static final int SIZE = 30;
12     private static final String BLOCKTAG = "ground_block";
13
14     /**
15      * Constructs a Block object with the specified top-left corner position and renderable.
16      * @param topLeftCorner The top-left corner position of the block.
17      * @param renderable The renderable object to be displayed for the block.
18      */
19     public Block(Vector2 topLeftCorner, Renderable renderable) {
20         super(topLeftCorner, Vector2.ONES.mult(SIZE), renderable);
21         physics().preventIntersectionsFromDirection(Vector2.ZERO);
22         physics().setMass(GameObjectPhysics.IMMOVABLE_MASS);
23         setTag(BLOCKTAG);
24     }
25
26     /**
27      * Retrieves the size of a block.
28      * @return The size of a block.
29      */
30     public static int getSize(){
31         return Block.SIZE;
32     }
33 }
```

22 pepse/world/EnergyDisplay.java

```
1  package pepse.world;
2
3  import java.util.function.Supplier;
4
5  import danogl.GameObject;
6  import danogl.gui.rendering.TextRenderable;
7  import danogl.util.Vector2;
8
9  /**
10   * Represents an energy display object in the game.
11   */
12  public class EnergyDisplay extends GameObject{
13      private final Supplier<Float> getter;
14
15      /**
16       * Constructs an EnergyDisplay object with the specified getter.
17       *
18       * @param get The supplier function to get the energy value.
19       */
20      public EnergyDisplay(Supplier<Float> get){
21          super(Vector2.ZERO,Vector2.ONES.mult(30) , new TextRenderable("100%"));
22          getter = get;
23      }
24
25      /**
26       * Updates the energy display based on the energy value obtained from the supplier.
27       *
28       * @param deltaTime The time elapsed since the last update.
29       */
30      @Override
31      public void update(float deltaTime) {
32          super.update(deltaTime);
33          Float amount = this.getter.get();
34
35          renderer().setRenderable(new TextRenderable(String.valueOf(amount) + "%"));
36      }
37
38
39  }
```

23 pepse/world/JumpObserver.java

```
1  package pepse.world;
2
3  /**
4   * The JumpObserver interface defines the contract for classes that observe the jumping behavior of an avatar.
5   */
6  public interface JumpObserver {
7      /**
8       * This method is called when the avatar jumps.
9       */
10     void onAvatarJump();
11 }
12 /**
13  * The JumpSubject interface defines the contract for the subject (avatar) that other classes can observe.
14  */
15 interface JumpSubject {
16     /**
17      * Registers an observer to receive notifications about avatar jumps.
18      *
19      * @param observer The observer to be registered.
20      */
21     void registerObserver(JumpObserver observer);
22     /**
23      * Removes an observer from the list of registered observers.
24      *
25      * @param observer The observer to be removed.
26      */
27     void removeObserver(JumpObserver observer);
28     /**
29      * Notifies all registered observers when the avatar jumps.
30      */
31     void notifyObservers();
32 }
```

24 pepse/world/Sky.java

```
1  package pepse.world;
2
3  import danogl.GameObject;
4  import danogl.components.CoordinateSpace;
5  import danogl.gui.rendering.RectangleRenderable;
6  import danogl.util.Vector2;
7  import java.awt.Color;
8
9  /**
10   * Represents the sky in the game.
11   */
12  public class Sky {
13      private static final Color BASIC_SKY_COLOR = Color.decode("#80c6e5");
14      private static final String SKY_TAG = "sky";
15
16      private Sky() {
17      }
18
19      /**
20       * Creates a sky game object with the specified window dimensions.
21       *
22       * @param windowDimensions The dimensions of the game window.
23       * @return The created sky game object.
24       */
25      public static GameObject create(Vector2 windowDimensions) {
26          GameObject sky = new GameObject(Vector2.ZERO, windowDimensions,
27              new RectangleRenderable(BASIC_SKY_COLOR));
28          sky.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
29          sky.setTag(SKY_TAG);
30          return sky;
31      }
32  }
```


25 pepse/world/Terrain.java

```
1  package pepse.world;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Random;
6  import java.awt.Color;
7
8  import danogl.gui.rendering.RectangleRenderable;
9  import danogl.gui.rendering.Renderable;
10 import danogl.util.Vector2;
11 import pepse.util.ColorSupplier;
12 import pepse.util.NoiseGenerator;
13
14 /**
15  * Represents the terrain in the game environment.
16  */
17 public class Terrain {
18     private final float groundHeightAtX0;
19     private static final Color BASE_GROUND_COLOR = new Color(212, 123,74);
20     private static final int TERRAIN_DEPTH = 20;
21     private final double GROUNDHEIGHTMULT = 2.0/3.0;
22     private final int NOISEGENMULT = 7;
23     private final NoiseGenerator noiseGenerator;
24     private double seed;
25
26     /**
27      * Constructs a Terrain object with the specified window dimensions and seed.
28      * @param windowDimensions The dimensions of the game window.
29      * @param seed The seed used for generating pseudo-random noise.
30      */
31     public Terrain(Vector2 windowDimensions, int seed){
32         this.groundHeightAtX0 = (float) (windowDimensions.y()*(GROUNDHEIGHTMULT));
33         this.seed = seed;
34         this.noiseGenerator = new NoiseGenerator(this.seed, (int)groundHeightAtX0);
35     }
36
37     /**
38      * Gets the ground height at the specified x-coordinate.
39      * @param x The x-coordinate.
40      * @return The ground height at the specified x-coordinate.
41      */
42     public float groundHeightAt(float x) {
43         float noise = (float) noiseGenerator.noise(x, Block.getSize() * NOISEGENMULT);
44         return groundHeightAtX0 + noise;
45     }
46
47     /**
48      * Creates a list of blocks within the specified range.
49      * @param minX The minimum x-coordinate.
50      * @param maxX The maximum x-coordinate.
51      * @return The list of blocks within the specified range.
52      */
53     public List<Block> createInRange(int minX, int maxX) {
54         List<Block> blocks = new ArrayList<>();
55         for (int x = minX; x <= maxX; x+= Block.getSize()) {
56             for (int i = 0; i < TERRAIN_DEPTH; i++) {
57                 float block_height = (float)Math.floor(groundHeightAt(x) / Block.getSize()) *
58                     Block.getSize() + i*Block.getSize();
59                 Vector2 block_position = new Vector2(x, block_height);
```

```
60         Renderable renderable = new RectangleRenderable
61             (ColorSupplier.approximateColor(BASE_GROUND_COLOR));
62         Block block = new Block(block_position, renderable);
63         blocks.add(block);
64     }
65 }
66 return blocks;
67 }
68 }
```

26 pepse/world/daynight/Night.java

```
1  package pepse.world.daynight;
2
3  import java.awt.Color;
4
5  import danogl.GameObject;
6  import danogl.components.CoordinateSpace;
7  import danogl.components.Transition;
8  import danogl.components.Transition.TransitionType;
9  import danogl.gui.rendering.RectangleRenderable;
10 import danogl.gui.rendering.Renderable;
11 import danogl.util.Vector2;
12 import pepse.util.ColorSupplier;
13
14 /**
15  * Represents the night environment in the game.
16  */
17 public class Night {
18     private static final String NIGHT_TAG = "night";
19     private static final Float MIDNIGHT_OPACITY = 0.5f;
20
21
22     /**
23      * Creates a GameObject representing the night environment.
24      *
25      * @param windowDimensions The dimensions of the game window.
26      * @param cycleLength The length of the cycle for opacity transition.
27      * @return A GameObject representing the night environment.
28      */
29     public static GameObject create(Vector2 windowDimensions, float cycleLength){
30         Renderable renderable = new RectangleRenderable(Color.BLACK);
31         GameObject night = new GameObject(Vector2.ZERO, windowDimensions, renderable);
32         night.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
33         night.setTag(NIGHT_TAG);
34         new Transition<Float>(night, night.renderer()::setOpacity,
35             0.0f, MIDNIGHT_OPACITY, Transition.CUBIC_INTERPOLATOR_FLOAT, cycleLength,
36             TransitionType.TRANSITION_BACK_AND_FORTH, null);
37         return night;
38     }
39 }
40 }
```

27 pepse/world/daynight/Sun.java

```
1  package pepse.world.daynight;
2
3  import java.awt.Color;
4
5  import danogl.GameObject;
6  import danogl.components.CoordinateSpace;
7  import danogl.components.Transition;
8  import danogl.components.Transition.TransitionType;
9  import danogl.gui.rendering.OvalRenderable;
10 import danogl.gui.rendering.Renderable;
11 import danogl.util.Vector2;
12
13 /**
14  * Represents the Sun in the game environment.
15  */
16 public class Sun {
17     private static final int SUN_SIZE = 100;
18     private static final String SUN_TAG = "sun";
19     private static final Float FINALVAL = 360f;
20     private static final int STARTINGPLACEPARAMX = 2;
21     private static final int STARTINGPLACEPARAMY = 3;
22
23     /**
24      * Creates a sun object with the specified window dimensions and cycle length.
25      * @param windowDimensions The dimensions of the window.
26      * @param cycleLength The length of the cycle.
27      * @return A GameObject representing the sun.
28      */
29     public static GameObject create(Vector2 windowDimensions, float cycleLength) {
30         Renderable renderable = new OvalRenderable(Color.YELLOW);
31         Vector2 starting_place = new Vector2(windowDimensions.x() / STARTINGPLACEPARAMX,
32             windowDimensions.y() / STARTINGPLACEPARAMY);
33         GameObject sun = new GameObject(starting_place, Vector2.ONES.mult(SUN_SIZE),
34             renderable);
35         sun.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
36         sun.setTag(SUN_TAG);
37         Vector2 initialSunCenter = sun.getCenter();
38         Vector2 cycleCenter = new Vector2(windowDimensions.x() / 2,
39             windowDimensions.y() * 2 / 3);
40         new Transition<Float>(sun,
41             (Float angle) -> sun.setCenter(initialSunCenter.subtract(cycleCenter)
42                 .rotated(angle).add(cycleCenter)),
43             Of, FINALVAL, Transition.LINEAR_INTERPOLATOR_FLOAT, cycleLength,
44             TransitionType.TRANSITION_LOOP, null);
45         return sun;
46     }
47 }
```

28 pepse/world/daynight/SunHalo.java

```
1  package pepse.world.daynight;
2
3  import java.awt.Color;
4
5  import danogl.GameObject;
6  import danogl.components.CoordinateSpace;
7  import danogl.gui.rendering.OvalRenderable;
8  import danogl.gui.rendering.Renderable;
9  import danogl.util.Vector2;
10
11  /**
12   * Represents the halo around the sun.
13   */
14  public class SunHalo {
15      private final static Color SUNHALO = new Color(255, 255, 0, 20);
16      private final static int HALOSIZE = 175;
17      private final static String SUNHALOTAG = "sunhalo";
18
19      /**
20       * Creates a GameObject representing the halo around the sun.
21       *
22       * @param sun The GameObject representing the sun.
23       * @return A GameObject representing the halo around the sun.
24       */
25      public static GameObject create(GameObject sun){
26          Renderable renderable = new OvalRenderable(SUNHALO);
27          GameObject sunHalo = new GameObject(sun.getTopLeftCorner(), Vector2.ONES.mult(HALOSIZE), renderable);
28          sunHalo.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
29          sunHalo.setTag(SUNHALOTAG);
30          return sunHalo;
31      }
32  }
```

29 pepse/world/trees/Flora.java

```
1  package pepse.world.trees;
2
3  import java.awt.Color;
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.Random;
7  import java.util.function.Function;
8  import java.util.function.Supplier;
9
10 import danogl.util.Vector2;
11 import pepse.world.Block;
12
13 /**
14  * Represents flora in the game, including trees.
15  */
16 public class Flora {
17     private static final int MAX_HEIGHT = 4;
18     private static final int NUM_OF_LEAVES = 7;
19     private Function<Float, Float> terrainHeight;
20
21     /**
22      * Constructs a Flora object with the specified terrain height function.
23      *
24      * @param terrainHeight The function to determine terrain height.
25      */
26     public Flora(Function<Float, Float> terrainHeight) {
27         this.terrainHeight = terrainHeight;
28     }
29
30     /**
31      * Creates trees within a specified range.
32      *
33      * @param minX The minimum x-coordinate.
34      * @param maxX The maximum x-coordinate.
35      * @return A list of trees created within the specified range.
36      */
37     public List<Tree> createInRange(int minX, int maxX) {
38         List<Tree> trees = new ArrayList<>();
39         Random random = new Random();
40         for (float location = minX; location < maxX; location += Block.getSize()) {
41             if (random.nextDouble() < 0.1) {
42                 int height = random.nextInt(1, MAX_HEIGHT);
43                 trees.add(new Tree(height, NUM_OF_LEAVES,
44                     new Vector2(location, terrainHeight.apply(location))));
45             }
46         }
47         return trees;
48     }
49 }
50 }
```

30 pepse/world/trees/Fruit.java

```
1  package pepse.world.trees;
2
3  import java.awt.Color;
4  import java.util.Random;
5
6  import danogl.GameObject;
7  import danogl.collisions.Collision;
8  import danogl.components.ScheduledTask;
9  import danogl.gui.rendering.OvalRenderable;
10 import danogl.util.Vector2;
11 import pepse.world.Avatar;
12 import pepse.world.Block;
13
14 /**
15  * Represents a fruit block in the game.
16  */
17 public class Fruit extends Block {
18     private static final int ENERGY_FROM_EATING = 10;
19     private static float cycleLength;
20     private boolean isInGame;
21
22     /**
23      * Constructs a Fruit object with the specified top-left corner position.
24      *
25      * @param topLeftCorner The top-left corner position of the fruit block.
26      */
27     public Fruit(Vector2 topLeftCorner, Color color) {
28         super(topLeftCorner, new OvalRenderable(color));
29         isInGame = true;
30     }
31
32     /**
33      * Sets the cycle length for the game.
34      *
35      * @param length The length of the game cycle.
36      */
37     public static void setCycleLength(float length) {
38         cycleLength = length;
39     }
40
41     /**
42      * Determines whether the object should collide with the specified game object.
43      *
44      * @param other The other game object involved in the collision.
45      * @return True if the object should collide with the specified game object, otherwise false.
46      */
47     @Override
48     public boolean shouldCollideWith(GameObject other) {
49         return other.getTag().equals(Avatar.getAvatarTag()) && isInGame;
50     }
51
52     /**
53      * Handles the behavior when a collision occurs with another game object.
54      *
55      * @param other The other game object involved in the collision.
56      * @param collision The collision information.
57      */
58     @Override
59     public void onCollisionEnter(GameObject other, Collision collision) {
```

```

60         if (shouldCollideWith(other)) {
61             Avatar avatar = (Avatar) other;
62             avatar.energyGain(ENERGY_FROM_EATING);
63             removeFromGame();
64             super.onCollisionEnter(other, collision);
65         }
66     }
67
68     private void removeFromGame() {
69         isInGame = false;
70         this.renderer().setOpaqueness(0);
71         new ScheduledTask(this, cycleLength, false, () -> addToGame());
72     }
73
74     private void addToGame() {
75         isInGame = true;
76         this.renderer().setOpaqueness(100f);
77     }
78 }

```


31 pepse/world/trees/Fruits.java

```
1  package pepse.world.trees;
2
3  import java.awt.Color;
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.Random;
7
8  import danogl.collisions.GameObjectCollection;
9  import danogl.gui.rendering.OvalRenderable;
10 import danogl.util.Vector2;
11 import pepse.PepseGameManager;
12
13 /**
14  * Represents a collection of fruits in the game.
15  */
16 public class Fruits {
17     private static final Random random = new Random();
18     private static final double SPAWNING_PROBABILITY = 0.3;
19     private static final Color INITIAL_COLOR = new Color(200, 20, 144);
20     private List<Fruit> fruits;
21
22     /**
23      * Constructs a collection of fruits with the specified number of fruits and
24      * bottom-middle position.
25      *
26      * @param numOfFruits The number of fruits to generate.
27      * @param bottomMiddle The bottom-middle position for arranging the fruits.
28      */
29     public Fruits(int numOfFruits, Vector2 bottomMiddle) {
30         int fruitSize = Leaf.getSize();
31         fruits = new ArrayList<Fruit>(numOfFruits * numOfFruits);
32         float initialX = bottomMiddle.x() - ((numOfFruits - 1) / 2 * fruitSize);
33         float y = bottomMiddle.y() - (numOfFruits * fruitSize);
34         float x;
35         for (int i = 0; i < numOfFruits; i++) {
36             x = initialX;
37             for (int j = 0; j < numOfFruits; j++) {
38                 if (random.nextDouble() < SPAWNING_PROBABILITY) {
39                     fruits.add(new Fruit(new Vector2(x, y), INITIAL_COLOR));
40                 }
41                 x += fruitSize;
42             }
43             y += fruitSize;
44         }
45     }
46
47     /**
48      * Adds the fruits to the specified game object collection.
49      *
50      * @param collection The game object collection to which the fruits will be
51      *                    added.
52      */
53     public void addFruits(GameObjectCollection collection) {
54         for (Fruit fruit : fruits) {
55             collection.addGameObject(fruit);
56         }
57     }
58
59     /**
```

```
60      * Changes the color of all the fruits.
61      *
62      * @param color The new color to apply to the fruits.
63      */
64      public void changeColor(Color color) {
65          for (Fruit fruit : fruits) {
66              fruit.renderer().setRenderable(new OvalRenderable(color));
67          }
68      }
69  }
```

32 pepse/world/trees/Leaf.java

```
1  package pepse.world.trees;
2
3  import java.awt.Color;
4  import java.util.Random;
5
6  import danogl.components.ScheduledTask;
7  import danogl.components.Transition;
8  import danogl.components.Transition.TransitionType;
9  import danogl.gui.rendering.RectangleRenderable;
10 import danogl.util.Vector2;
11 import pepse.world.Block;
12
13 /**
14  * Represents a leaf object in the game.
15  */
16 public class Leaf extends Block {
17     private static final Color BASIC_COLOR = new Color(50, 200, 30);
18     private static final float FINAL_ROTATING_ANGLE = 45f;
19     private static final float TRANSITION_LENGTH = 2;
20     private static final float MIN_WIDTH = 10f;
21     private static final float MAX_WIDTH = 60f;
22     private static final Random random = new Random();
23
24     /**
25      * Constructs a leaf object with the specified top-left corner position.
26      *
27      * @param topLeftCorner The top-left corner position of the leaf.
28      */
29     public Leaf(Vector2 topLeftCorner) {
30         super(topLeftCorner, new RectangleRenderable(BASIC_COLOR));
31     }
32
33     /**
34      * Adds a transition effect to the leaf object.
35      */
36     public void addTransition() {
37         new ScheduledTask(this, (float) random.nextDouble(), false,
38             () -> setTransition());
39     }
40
41     private void setTransition() {
42         float leafHeight = getSize();
43         new Transition<Float>(this,
44             (Float angle) -> this.renderer().setRenderableAngle(angle), 0f,
45             FINAL_ROTATING_ANGLE, Transition.LINEAR_INTERPOLATOR_FLOAT,
46             TRANSITION_LENGTH, TransitionType.TRANSITION_BACK_AND_FORTH, null);
47         new Transition<Float>(this,
48             (Float width) -> this.setDimensions(new Vector2(width, leafHeight)),
49             MIN_WIDTH, MAX_WIDTH, Transition.LINEAR_INTERPOLATOR_FLOAT,
50             TRANSITION_LENGTH, TransitionType.TRANSITION_BACK_AND_FORTH, null);
51     }
52 }
```

33 pepse/world/trees/Leaves.java

```
1  package pepse.world.trees;
2
3  import java.awt.Color;
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.Random;
7
8  import danogl.collisions.GameObjectCollection;
9  import danogl.collisions.Layer;
10 import danogl.components.ScheduledTask;
11 import danogl.components.Transition;
12 import danogl.components.Transition.TransitionType;
13 import danogl.gui.rendering.RectangleRenderable;
14 import danogl.util.Vector2;
15
16 /**
17  * Represents a collection of leaves in the game.
18  */
19 public class Leaves {
20     private static final Random random = new Random();
21     private static final double SPAWNING_PROBABILITY = 0.6;
22     private static final int LEAVES_LAYER = -50;
23     private List<Leaf> leaves;
24
25     /**
26      * Constructs a collection of leaves with the specified number of leaves and
27      * bottom middle position.
28      *
29      * @param numOfLeaves The number of leaves.
30      * @param bottomMiddle The bottom middle position.
31      */
32     public Leaves(int numOfLeaves, Vector2 bottomMiddle) {
33         int leafSize = Leaf.getSize();
34         leaves = new ArrayList<Leaf>(numOfLeaves * numOfLeaves);
35         float initialX = bottomMiddle.x() - ((numOfLeaves - 1) / 2 * leafSize);
36         float y = bottomMiddle.y() - (numOfLeaves * leafSize);
37         float x;
38         for (int i = 0; i < numOfLeaves; i++) {
39             x = initialX;
40             for (int j = 0; j < numOfLeaves; j++) {
41                 if (random.nextDouble() < SPAWNING_PROBABILITY) {
42                     leaves.add(new Leaf(new Vector2(x, y)));
43                 }
44                 x += leafSize;
45             }
46             y += leafSize;
47         }
48         setTransitions();
49     }
50
51     /**
52      * Adds the leaves to the specified game object collection.
53      *
54      * @param collection The game object collection to add the leaves to.
55      */
56     public void addLeaves(GameObjectCollection collection) {
57         for (Leaf leaf : leaves) {
58             collection.addObject(leaf, LEAVES_LAYER);
59         }
60     }
61 }
```

```

60     }
61
62     /**
63      * Rotates all the leaves by 90 degrees.
64      */
65     public void rotate90Degrees() {
66         for (Leaf leaf : leaves) {
67             float curAngle = leaf.renderer().getRenderableAngle();
68             leaf.renderer().setRenderableAngle(curAngle + 90);
69         }
70     }
71
72     private void setTransitions() {
73         for (Leaf leaf : leaves) {
74             leaf.addTransition();
75         }
76     }
77
78 }

```

34 pepse/world/trees/Tree.java

```
1  package pepse.world.trees;
2
3  import java.awt.Color;
4  import java.util.Random;
5
6  import danogl.collisions.GameObjectCollection;
7  import danogl.util.Vector2;
8
9  /**
10   * Represents a tree object in the game.
11   */
12  public class Tree {
13      private final Trunk trunk;
14      private final Leaves leaves;
15      private final Fruits fruits;
16
17      /**
18       * Constructs a tree with the specified height, number of leaves, and position.
19       *
20       * @param height      The height of the tree trunk.
21       * @param numOfLeaves The number of leaves on the tree.
22       * @param placeToPut  The position to place the tree.
23       */
24      public Tree(int height, int numOfLeaves, Vector2 placeToPut) {
25          trunk = new Trunk(placeToPut, height);
26          Vector2 trunkTop = trunk.getTopLeftCorner();
27          leaves = new Leaves(numOfLeaves, trunkTop);
28          fruits = new Fruits(numOfLeaves, trunkTop);
29      }
30
31      /**
32       * Adds the tree to the specified game object collection.
33       *
34       * @param collection The game object collection to which the tree will be added.
35       */
36      public void addTree(GameObjectCollection collection) {
37          trunk.addTrunk(collection);
38          leaves.addLeaves(collection);
39          fruits.addFruits(collection);
40      }
41
42      /**
43       * Notifies the observer that the avatar has jumped.
44       *
45       * @param newFruitColor The new color for the fruits.
46       */
47      public void onAvatarJump(Color newFruitColor) {
48          trunk.changeColor();
49          leaves.rotate90Degrees();
50          fruits.changeColor(newFruitColor);
51      }
52  }
```

35 pepse/world/trees/Trunk.java

```
1  package pepse.world.trees;
2
3  import danogl.GameObject;
4  import danogl.collisions.GameObjectCollection;
5  import danogl.collisions.Layer;
6  import danogl.gui.rendering.RectangleRenderable;
7  import danogl.gui.rendering.Renderable;
8  import danogl.util.Vector2;
9  import pepse.util.ColorSupplier;
10 import pepse.world.Block;
11
12 import java.awt.Color;
13 import java.util.ArrayList;
14 import java.util.List;
15
16 /**
17  * Represents the trunk of a tree in the game.
18  */
19 public class Trunk {
20     private static final Color BASIC_COLOR = new Color(100, 50, 20);
21     private final List<Block> trunk;
22     private final Vector2 topLeftCorner;
23
24     /**
25      * Constructs a trunk with the specified bottom-left corner position and height.
26      *
27      * @param bottomLeftCorner The bottom-left corner position of the trunk.
28      * @param height           The height of the trunk.
29      */
30     public Trunk(Vector2 bottomLeftCorner, int height) {
31         trunk = new ArrayList<Block>(height);
32         float x = bottomLeftCorner.x();
33         float y = bottomLeftCorner.y();
34         for (int blockNum = 0; blockNum < height; blockNum++) {
35             y -= Block.getSize();
36             trunk.add(new Block(new Vector2(x, y), new RectangleRenderable(BASIC_COLOR)));
37         }
38         topLeftCorner = new Vector2(x, y);
39     }
40
41     /**
42      * Adds the trunk blocks to the specified game object collection.
43      *
44      * @param collection The game object collection to which the trunk blocks will
45      *                   be added.
46      */
47     public void addTrunk(GameObjectCollection collection) {
48         for (Block block : trunk) {
49             collection.addGameObject(block, Layer.STATIC_OBJECTS);
50         }
51     }
52
53     /**
54      * Gets the top-left corner position of the trunk.
55      *
56      * @return The top-left corner position of the trunk.
57      */
58     public Vector2 getTopLeftCorner() {
59         return new Vector2(topLeftCorner.x(), topLeftCorner.y());
60     }
61 }
```

```
60     }
61
62     /**
63      * Changes the color of the trunk blocks to a new color.
64      */
65     public void changeColor() {
66         Color newColor = ColorSupplier.approximateColor(BASIC_COLOR);
67         for (Block block : trunk) {
68             block.renderer().setRenderable(new RectangleRenderable(newColor));
69         }
70     }
71 }
```