INFO 3225 Team Project Report

MATTHEW ALAIN	LEONAS VON MEDEM	PEDRO SANCHEZ
100349282	100464452	100464450

ABSTRACT

This program combines three previously submitted individual projects into a single more complete program, having taken elements from each to fulfill the specified technical requirements and create a game that can be continually developed and improved upon. This report will describe how the program incorporates loops, conditional statements, arrays, 2D transformation, inheritance, user interactions, and custom user interfaces, while showcasing the project's finite state machine diagram, gameplay screenshots, code segments, and inspirational references.

KEYWORDS

Game; Simulation; Micromanagement; Multitasking; Optimization; Environment.

INTRODUCTION

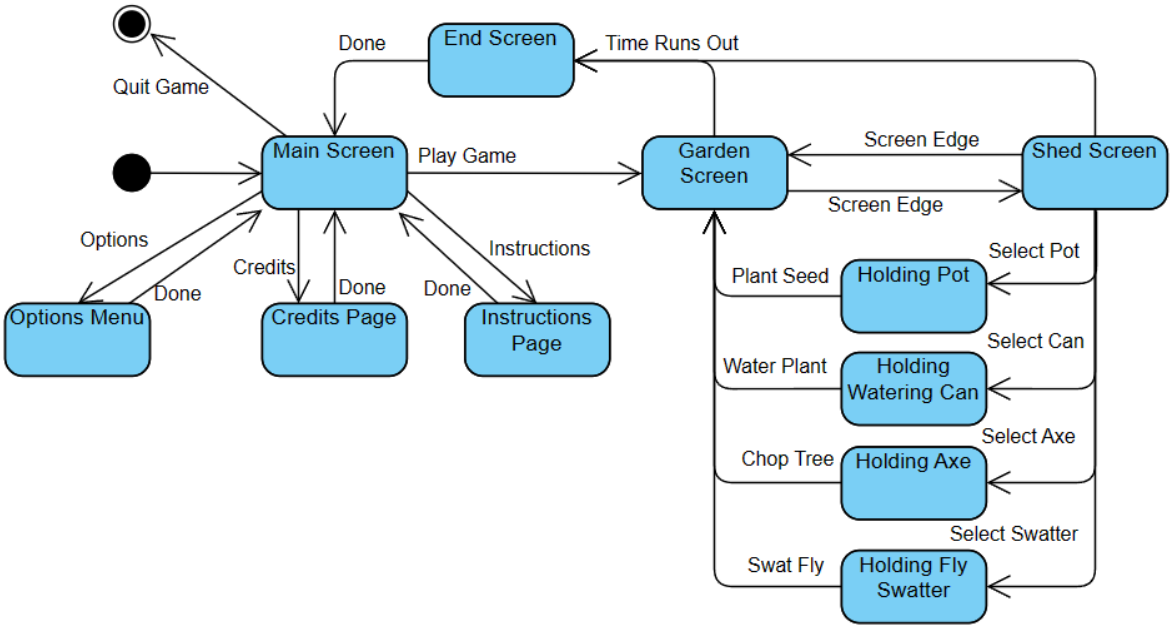
This project is a timed game made in Processing, in which the player must plant, grow, and cut down trees while protecting them from flies. The gameplay loop requires the player to move between game screens to bring the appropriate tool to a specific location as quickly as possible to maximize the number of trees grown before the timer runs out and the game ends. The player's score increases for each tree that completely grows and gets cut down, and decreases for each plant that is not protected from the attacking flies that will destroy the plants if not interrupted by the player.

The project uses multi-level inheritance (Human > Belly > Chest > Arm > Hand) in conjunction with push and pop matrix to allow the character and held items to move in unison between game screens. Class abstraction is used (Seed, Sapling, and Tree extend Plant) to minimize duplicate code, and allows for the opportunity to scale up the project by creating additional plant stages with different requirements. Class interfaces are used (Cloud and Human implement MovableObj) to enforce program standards and integrity.

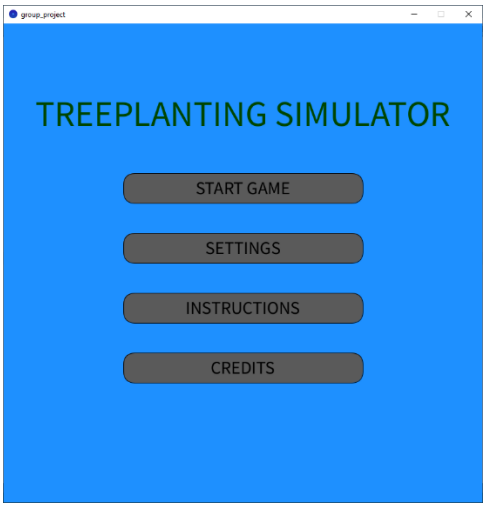
The game accepts the user's keyboard interaction to move the character, and mouse interactions to direct the character's eye movement and select and use tools. Custom created UI elements include interactable main screen directory buttons, animated progress indicators for sunlight, water, and plant health, and a counting down game timer.

We utilize search and sort algorithms to analyze the randomization of moving objects within the program. We utilized bubble sort to organize the cloud objects in descending order, so the larger clouds appeared higher on the screen to seem closer to the player's viewpoint, and the smaller clouds were placed lower on the screen to appear further away from the player. We hoped that through this we would be able to create a realistic perspective of the sky. We used a search algorithm to find the X and Y coordinates of the randomly moving fly objects and outputted those coordinates to an external file for further analysis on what movement trends the objects tended to follow.

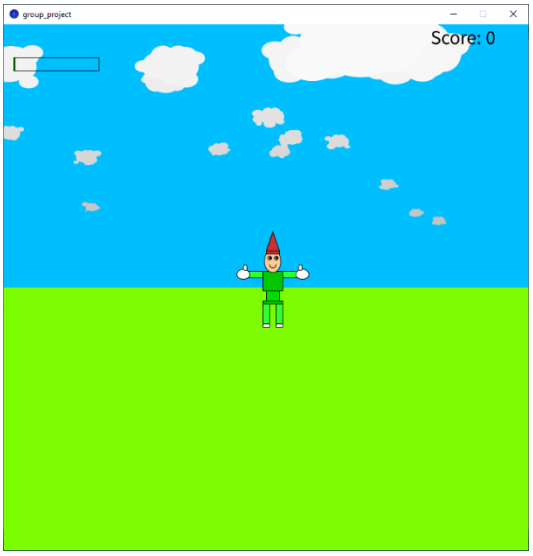
FINITE STATE MACHINE DIAGRAM



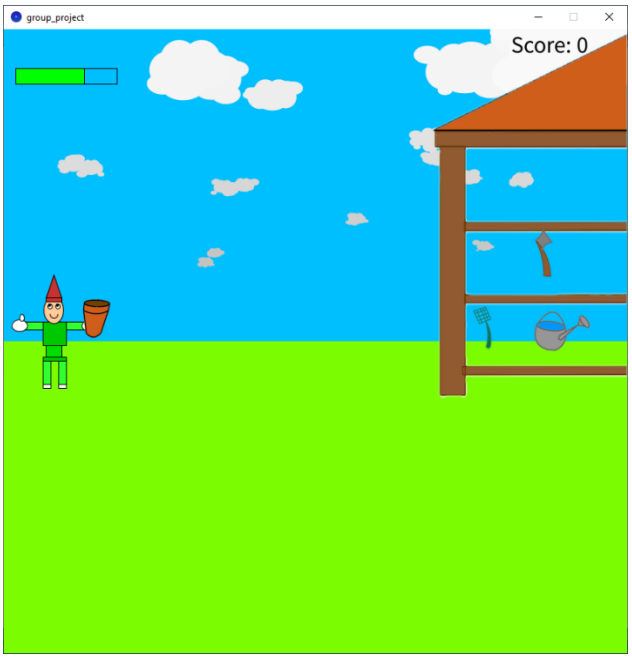
SCREENSHOTS



This showcases the default main screen when the program is run. On this screen, users can navigate to the different menus to change the settings, view the game instructions, and see the credits and references. When “start game” is selected, the game begins and sends the user to the main game screen.

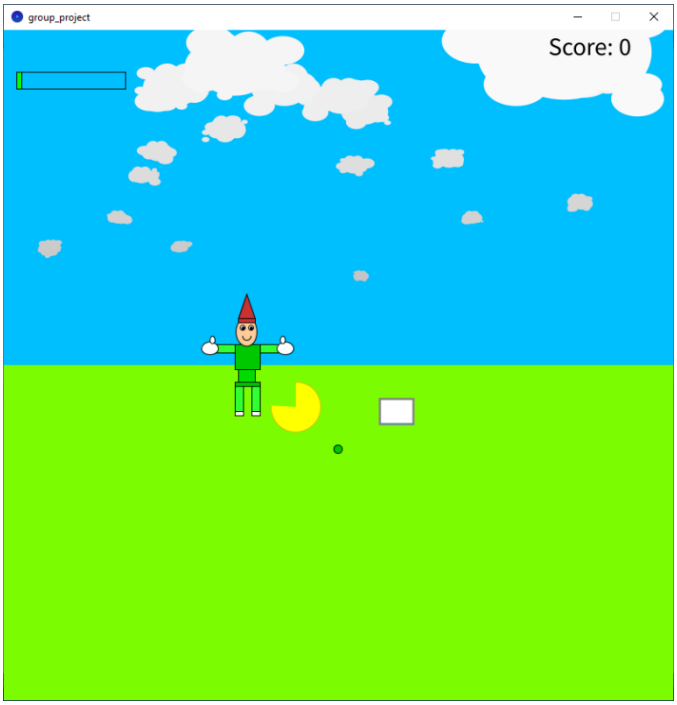


The main game screen, showing the player in an unplanted garden, with clouds moving across the sky, the current score in the top right, and the timer bar slowly filling. On this screen, the player can move side to side, and moving all the way to the right sends the player to the tool selection screen.

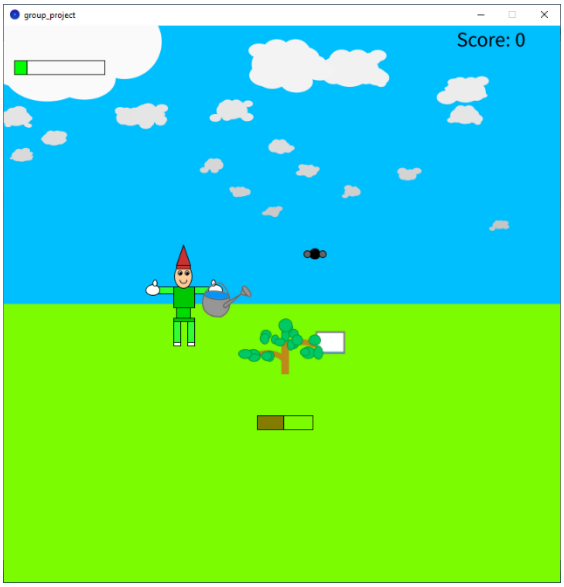


This showcases the second screen, from which the player can select the tool they would like to carry. When the player selects the planting pot, it is moved to the character’s right hand, and it now follows the player’s movements, including between screens.

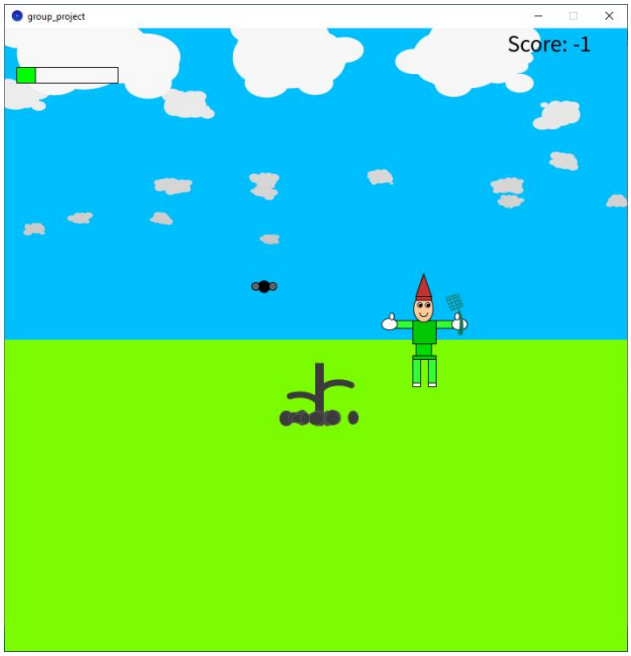
Here, the player has carried the pot to the garden, and has planted a seed. Immediately, the seed begins filling its sunlight meter, and once it is done so, the player can then begin watering the seed to grow it.



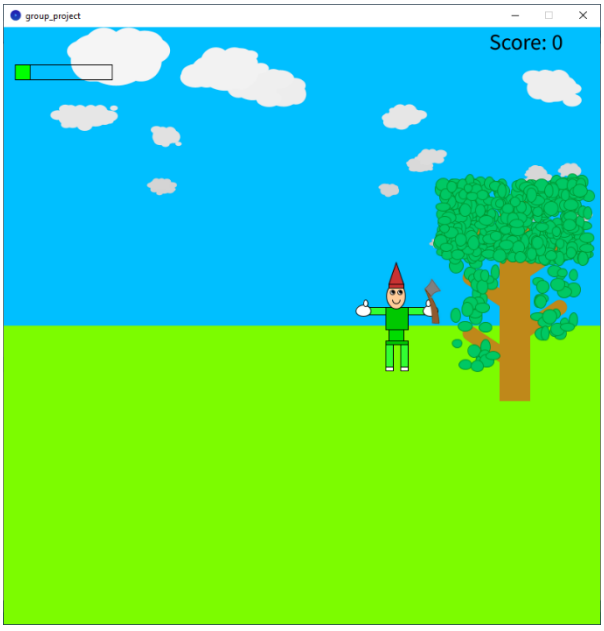
When the player is carrying the watering can, they can water the seed, which increases its water meter. A short time after the first seed is planted, flies will begin to spawn and move towards planted seeds to attack them.



This showcases a seed that has received enough sunlight and water to grow into a sapling. Its sunlight and water meters are reset and must be refilled to grow into a tree. When a fly reaches a plant, none of its meters will fill, its health bar will appear, and it will gradually shrink and redden the more health the plant loses. The player can retrieve the fly swatter tool from the shed and click when near the flies to remove it from the plant.



If a plant dies, it will turn black and its leaves (if any) will slowly descend to the ground. When a plant dies, the player's score is decreased, and the player must click with any tool to remove the dead plant for it to be used again.



When a sapling is given enough sunlight and water to grow, it becomes a tree, gets even more leaves, and now no longer requires sunlight or water. Trees instead must be clicked on repeatedly with the axe tool before it rotates ninety degrees, falls, and increases the player's score.

CODE SEGMENTS

```
//If plant is ready to grow, remove and replace with higher stage plant
public void grow(){
  if(readyToGrow){
    plots[plot] = null;
    switch(plantStage){
      case 1: {
        Sapling sapling = new Sapling(plot);
        plots[plot] = sapling;
        break;
      }
      case 2: {
        Tree tree = new Tree(plot);
        plots[plot] = tree;
        break;
      }
      case 3: {
        plots[plot] = null;
        score++;
        break;
      }
    }
  }
}
```

The grow function in the Plant class progresses plants to their next stage once they are ready by removing the current stored object and creating the next stage plant in its place. If the plant object was the final stage, it is not replaced, and instead increases the player's score.

```
//Tests if the current plant has enough water and sunlight, or chops to fall
void checkForGrowth(){
  if(plantStage < 3){
    if(waterCurrent >= waterNeeded && sunlightCurrent >= sunlightNeeded && alive){
        readyToGrow = true;
        grow();
    }
  }else{
    if(chopsCurrent >= chopsNeeded && alive){
      readyToGrow = true;
      grow();
    }
  }
}
```

The checkForGrowth method is checked in the draw function to constantly check whether a plant has enough of the resources it needs to grow. For everything smaller than a tree, it must have the necessary water and sunlight, and must not have been eaten by flies. For a tree, which does not require any water or sunlight, it must have received the necessary amount of chops to fall.

```
Tree(int plot){
  plantStage = 3;
  this.plot = plot;
  X = (1+plot)*(width/3) - width/6;
  Y = 500;
  waterNeeded = 1;
  sunlightNeeded = 1;
  chopsNeeded = int(random(20,30));
  healthMax = 1000;
  healthCurrent = healthMax;
  numberOfLeafs = 500;
  for(int i = 0; i < numberOfLeafs; i++){
    addLeafs();
  }
}
```

The Tree class constructor is the most complex version of the typical plant constructors. Like the other plant child class constructors, it sets the values the variables, including the X value (set as the center of the three plots of valid land), the amount of water, sunlight, and chops needed (because trees need no water and sunlight, their values can be initialized as any values). It also populates the leaf array for the object to be displayed afterwards.

```
  public void addLeafs(){
    if(leafs.size() < numberOfLeafs/8){ //Leafs on left branch
      Leaf newLeaf = new Leaf(int(random(-60,-5)), int(random(-290,-40)));
      leafs.add(newLeaf);
    }else if(leafs.size() < numberOfLeafs/4){ //Leafs on right branch
      Leaf newLeaf = new Leaf(int(random(45,100)), int(random(-290,-80)));
      leafs.add(newLeaf);
    }else if(leafs.size() < numberOfLeafs){ //Leafs on top
      Leaf newLeaf = new Leaf(int(random(-80,120)), int(random(-295,-190)));
      leafs.add(newLeaf);
    }
```

The addLeafs method creates leaf objects around the tree, with one eighth being on the left branches, one eighth being on the right, and the remaining three quarters being at the top. An earlier error in an individual project had the plants continually replace themselves, but this was resolved by populating the array in the constructor, and displaying it afterwards.

```
void move() {

  if (targetX>x) {
    this.x+=speed+moveDisX*random(-1, 1);
  } else if (targetX<x) {
    this.x-=speed+moveDisX*random(-1, 1);
  }
  if (targetY<y) {
    this.y+=moveDisY*random(-1, 1)-0.5;
  } else if (targetY>y) {
    this.y+=moveDisY*random(-1, 1)+0.5;
  }

  if (x>targetS*width/3 && x<targetS*width/3+width/3)
    attacks=true;
}
```

To give the flies a realistic flight pattern, they move slightly left and right, up and down, but ultimately always towards their target. This is achieved by checking whether the fly is to the right or left of its target and then taking random steps between -1 and 1 backward and two steps towards the target. Once the fly is in the target section, it changes its status to "attack".

The start screen consists of several aspects. The first one is the display method, which uses a switch to cycle through the modes, and depending on which mode is active, a different screen is shown.

```
void display() {
  background(30, 144, 255);
  switch (mode) {
  case "DONE":
    displayStart();
    break;
  case "SETTINGS":
    displaySettings();
    break;
  case "INSTRUCTIONS":
    displayInstructions();
    break;
  case "CREDITS":
    displayCredits();
    break;
  }
}
```

The different modes are determined by the buttons. Each button is an object of the Button class and has its own label.

```
String[] buttonNames = {"START GAME", "SETTINGS", "INSTRUCTIONS", "CREDITS"};
String[] buttonNamesSettings = {"TIME", "DIFFICULTY", "FILTER", "SOUND"};
```

In the mouse pressed method, all main buttons are looped through, and the mode is set to the label of each respective button. Consequently, the screen changes depending on which button was pressed.

```
boolean mousePressed() {
  if (mode.equals("DONE")) {
    for (Button button : buttons) {
      if (button.isMouseOver()) {
        mode = button.onClick();
      }
    }
  }
```

```
  boolean isMouseOver() {
    return mouseX > x && mouseX < x + w && mouseY > y && mouseY < y + h;
  }
  String onClick() {
    return label;
  }
}
```

By combining the methods `isMouseOver` and `onClick` from the Button class, it is determined which button was pressed and thus which label is returned to display the correct screen.

The populateClouds method fills the cloud array in the main screen. Each cloud is spawned at a random position that determines its size and shade. This array is, however, initially not sorted, which would cause smaller and shadier clouds (which should be in the background) to occasionally spawn in front of bigger, foreground clouds. The solution for this problem was to implement a simple sorting algorithm (bubble sort) that sorts out the cloud array in descending order, guaranteeing that clouds closer to the origin of y are drawn after (i.e. in front of) clouds that are further away (i.e. that have a higher y coordinate).

```
void populateClouds() {
  for (int i=0; i<20; i++) {
    float randomX = random(0, width);
    float randomY = random(20, height/2-100);
    Cloud cloud = new Cloud(randomX, randomY);
    cloudArray.add(cloud);

    for (int j=0; j<10; j++) {
      Cloud subBlob = new CloudSubBlob(randomX, randomY);
      cloudArray.add(subBlob);
    }
  }

  int n = cloudArray.size();
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      Cloud cloud1 = cloudArray.get(j);
      Cloud cloud2 = cloudArray.get(j + 1);
      if (cloud1.getY() < cloud2.getY()) {
        cloudArray.set(j, cloud2);
        cloudArray.set(j + 1, cloud1);
      }
    }
  }
```

```
switch (currItem) {
case NONE:
  break;
case POT:
  if (currItem == ItemsStates.POT) {
    figure.reachToGround();
    if (plots[currPlayerLocation] == null) {
      Seed seed = new Seed(currPlayerLocation);
      plots[currPlayerLocation] = seed;
    }
    currItem = ItemsStates.NONE;
  }
  break;
case AXE:
  if (!(plots[currPlayerLocation] == null)) {
    plots[currPlayerLocation].updateChops();
  }
  break;
case FLYSWATTER:
  for (Fly fly : flies) {
    if (fly.hit(int(figure.getX()))) {
      flies.remove(fly);
      if (plots[fly.getTargetS()]!=null && plots[fly.getTargetS()].flyIsOn)
        plots[fly.getTargetS()].setFly();
      break;
    }
  }
  break;
case WATERING_CAN:
  watering = true;
  break;
}
```

The currItem variable stores the information about which item the player is currently holding. This switch case block, determines which action the player can make, depending on which item the player is currently holding. Each case calls the appropriate method for executing the desired action. This code segment ensures that the player is only allowed to perform certain actions when their holding the correct tool.

```
class Figure{
  private Human[] bodyParts = new Human[10];
  private float x, y, scale;

  Figure(float xCoor, float yCoor, float scale){
    this.x = xCoor;
    this.y = yCoor;
    this.scale = scale;
    createFigure();
  }

  private void createFigure(){
    Human belly = new Belly(x, y);
    bodyParts[0] = belly;
    Human hip = new Hip(x, y);
    bodyParts[1] = hip;
    Human leftLeg = new LeftLeg(x, y);
    bodyParts[2] = leftLeg;
    Human rightLeg = new RightLeg(x, y);
    bodyParts[3] = rightLeg;
    Human chest = new Chest(x, y);
    bodyParts[4] = chest;
    Human leftArm = new LeftArm(x, y);
    bodyParts[5] = leftArm;
    Human leftHand = new LeftHand(x, y);
    bodyParts[6] = leftHand;
    Human rightArm = new RightArm(x, y);
    bodyParts[7] = rightArm;
    Human rightHand = new RightHand(x, y);
    bodyParts[8] = rightHand;
    Human head = new Head(x, y);
    bodyParts[9] = head;
  }

  void drawFigure(){
    for(Human part : bodyParts){
      pushMatrix();
      translate(-width/2*scale, -height/2*scale);
      scale(scale);
      translate(width/(2*scale), height/(2*scale));
      part.drawObj();
      popMatrix();
    }
  }
```

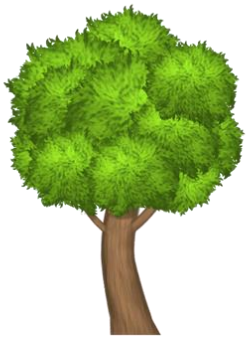
The class Figure represents the player's avatar and is responsible for initializing and saving all its body parts. In this code snippet it is possible to visualize how all body parts (i.e. all objects from classes that inherit from HumanImpl) are saved and drawn together. This makes it possible for 2d transformations (e.g. scaling the figure up or down) to be easily applied to the avatar as well as to any individual body parts.

REFERENCES

Visual Inspiration

Latakaki (n.d.). cleanpng.com *Sapling Botanical Plant Green Leaves Stem*. Retrieved February 8, 2024. https://www.cleanpng.com/png-sapling-botanical-plant-green-leaves-stem-7788535/

Green Tree*, Cartoon Sprite Caricature*. (n.d.). klipartz.com. Retrieved February 8, 2024. https://www.klipartz.com/en/sticker-png-tcmgx



The Legend of Zelda: The Wind Waker HD Art Gallery. (n.d.). Creative Uncut.

https://www.creativeuncut.com/art_the-legend-of-zelda-the-wind-waker-hd_a.html

Gameplay Inspiration

IMG 1198. (n.d.). plantsvszombies.fandom.com. Retrieved March 18, 2024. https://plantsvszombies.fandom.com/wiki/Zen_Garden_(PvZ)?file=IMG_1198.png

Happy Harvest. (2023, August 18). IGDB. Retrieved February 9, 2024, from https://www.igdb.com/games/happy-harvest

GAMEPLAY FEEDBACK

Video Link

https://www.youtube.com/watch?v=gGZDjxWrx_w

Feedback

The feedback we got from our gameplay tester, who requested to remain anonymous, told us that they appreciated the effort we put into the project. They thought the gameplay loop was somewhat simple but satisfying enough given the extent of the project. They liked that the gameplay required going back and forth between the screens quickly, especially when a fly was visibly moving to attack the plants, creating a sense of urgency. The UI elements were clear and easily understandable, but it was not obvious that a plant being attacked could not continue to grow, so we learned that in a future version of the program, we should be clearer about the consequences, potentially implementing a tutorial to show the player the nuances of the game.