Algorithm and Programming Final Project Report

Bunn's Christmas Tree Tale

BY

Name : Ella Raputri

Student ID : 2702298154

Class : L1AC

Computer Science Program

School Of Computing and Creative Arts

Bina Nusantara International University

Jakarta

2023

Table of Contents

BUNN'S CHRISTMAS TREE TALE

A. Introduction

1. Background

As a part of our Python Algorithm and Programming class, we are expected to create a comprehensive application or game to further deepen our knowledge of Python. The project has to be extended more than what has been taught in class. In other words, this final project expected us to make an application that is not taught yet and it is outside our comfort zone to learn new concepts in Python and solve or debug the problems that we encountered during the creation of this specific application.

Hence, after some research, I decided to make a story game that focuses on a farm using Python, specifically the Pygame module that isn't taught in the course. I learnt it by reading the documentation in pygame.org and watching some Youtube videos. I also learned how to make a map in Tiled and incorporate it into my game using the PyTMX (Python Tiled Map XML) module which is also not taught in this semester's course.

I was heavily inspired by the Youtube video that Clear Code made about Pydew Valley. I adapted some of the farming physics that he taught in his video into my game. However, my main point in making this game isn't to let players only experience a farm game, but I was more inspired to make a wholesome story that happens on a farm for players to experience during the Christmas holiday.

2. Problem Identification

The first thing that I decided to do when I first thought about my Python Algorithm and Programming Final Project was to make a simple game that can be solved in only like few minutes but gives rather useful information. My inspiration is to make a bite-sized simple game, like Google games (for example: Google Doodle Champion Island, Pangolin Love, Black Cat Academy, etc.). So, after some thought, I decided to make a farm game, and the useful information that I picked for this game is a story to cherish your family and not let yourself drown in depression or other kinds of sadness because life continues. We have strived and do our best to not let the people who care about us be

disappointed. The game is aimed only to be played for a few minutes but will give players useful information that also fits into the holiday theme.

At the beginning of the game, the player will be directed to an intro page and if the player clicks the play button, then the game will be started. The game will start with the first cutscene about the introduction of the story's main character (Bunn's) background story. Then, after each cutscene, there will be some simple tasks that the player has to accomplish so that they can proceed with the story. One of the tasks is to raise money and this can be achieved by farming and raising animals.

After the story, the player will get some money and a decoration will be built in the player's house. Players can choose to continue playing the game (but there is no story anymore) or exit the game. The story is not too long too, so this will also let the player feel that their time didn't get wasted by playing this game.

So, my main objectives for creating this game are:

a. Let players experience a warm-hearted holiday story.
b. Let the player draw a message from the story.
c. The player doesn't have to waste that much time to experience the story.

B. Project Specification

1. Game Name

"Bunn's Christmas Tree Tale"

The game is related to a Christmas tree and the main character's name is Bunn. So, the game name is simply based on them.

2. Game Flow Summary

"Bunn's Christmas Tree Tale" is a game where the player has to walk through the story of Bunn and how he got his Christmas tree. The player has to control Bunn to move around the map and do things, like going to some places or raising money.

The player or Bunn can raise money by planting seeds, which are differentiated into two types (tomato and corn), cutting wood, or raising cows for their milk. Players can sell their crops (corn and tomatoes), wood, or milk by selling them to the merchant. The player has to use specific keys on the keyboard to move around the map, toggle menus (inventory, help, and shop), and use specific tools.

The player also has to press only the key ‘C’ on their keyboard to start a cutscene. If the objectives or missions from the last cutscene are not completed yet, then the player can’t move to the next part of the story.

3. Game Display

The objects of this game are created using the sprite sheets from the Sprout Land assets, which are 2D-pixel art assets. Meanwhile, the background is the map image that I created using Tiled with the tilesets that are also from the Sprout Land assets.

When players first open this application, they will not be directed to the map instantly, but they will rather be directed to an intro menu with the background of a pixeled grass image. There will be also a play button and our main character Bunn. If the player clicks the play button, then a cutscene will start.

Every cutscene display is divided into two parts, the main picture and the dialogue part. The main picture is about 550 – 700 pixels in height, showing the relevant pictures of what is told in the story. Meanwhile, the dialogue part is also differentiated into two different types. The first type is the narrator’s line, in which the background is a brown rect, while the second type is the dialogue of the character. The background of the second type of dialogue is a dialogue box and the head profile of the character that is talking.

After that, there is the game's default background, which is the map or the world as the background. There are many objects in this game background, from plants, trees, players, cows, the merchant, houses, etc. However, there are also overlay displays on the bottom left and top right of the screen. On the bottom left, there is the display of the tool and the seed type that the player currently uses. While, on the top

right, there are two clickable menus. The first one is the inventory with the cart symbol and the second one is the help menu about the keyboard shortcuts in this game with the question mark symbol. Both of them can also be toggled using the keyboard shortcut ‘P’ for inventory and ‘H’ for help.

4. Game Physics

Below are the explanations of the game physics that exists in the game:

a. Collision detection

Some objects like trees, plants, fences, and others have hitboxes so that they can detect collisions. In this game, if a collision happens, then the player can’t move in that direction. For example, if the player crashes or hits a tree when moving to the right, then the player can’t move to the right further. In this game, except for objects that are derived from the class Generic, I also made some collisions in the map. The collisions can be seen through Tiled which is marked by red colour. With this, we can control players, so that they can’t walk on the water or walk out of the map.

b. Tree

If the player uses an axe and the target position is a tree object, then the tree will be cut. If the tree is cut, then the stump will be blitted on the screen rather than the tree because it is cut. There are two types of trees, the large one and the small one. Different trees have different sizes of stumps. After that, if the player goes to sleep, then the tree will be reset. The tree will revive again (the stump surface will be replaced by the tree surface again) and the player can chop it again.

c. Tilling Soil

The player can only till the soil if the player uses a hoe and the soil is farmable. If the soil is not farmable, for example: soil tile that is near to water body or below a tree, then the soil will not be tilled. Tilled soil will lose its green colour and change colour to brown as it is tilled.

d. Growing Plants

Every plant has four types of stages when they are planted. The first state is when they are still a little sprout. This stage occurs when the player first uses a seed on plantable soil. The second and third stages are the plants when they are still growing, while the last (fourth) stage is the stage when the plant is harvestable. Every seed has to be planted into a plantable or tiled soil, if not, then the seed will not be planted and the amount of the seed will not be deducted. If the seed is planted, then it won't grow even if the player resets it for several days if the seed is not watered. So, the player has to water the seed. If the seed is watered, then it will grow to become a plant on the next day. If the player watered it every day, then after several days, when the plant reaches the fourth (harvestable stage), they can harvest it.

e. Harvesting Milk

The mechanism for harvesting milk is similar to the growing plants. The different is that when growing plants, we have to water them every day, but for getting milk, we have to feed the cow with grass every day. If the cow reaches the maximum age, then we can harvest the milk by again giving the cow grass. The status of the milk (whether it is ready or not) can be seen by checking the milk bottle inside the cow farm. If it's full, then the player can harvest the milk.

f. Weather (Rain and Sunny)

The weather system in this game is achieved by using the random library. The probability for rain to happen in the game is 0.3 or 30%. If it's raining, then the tilled soil will be wet, so the player doesn't have to water the soil anymore. There will be also a water drop that is blitted on the screen.

g. Day-night Transition

There is also a day-night transition in this game, so the sky will be darker along with the pass of time. If the player goes to sleep, everything will be reset, including the sky colour. The sky will become light again and if the player plays for quite a long time, it will become dark again.

5. Game Input

a. Mouse’s left button, to click the play button in the intro menu, quit, or click the overlay help and inventory menu on the top right of the game when the game has started.

b. Keyboard keys:

- 'Esc' to quit the game.
- 'A' to move the player to the left.
- 'D' to move the player to the right.
- 'W' to move the player upward.
- 'S' to move the player downward.
- 'Space' to use current tools and if the merchant menu is on the screen, ‘Space’ is used to purchase or to sell something.
- 'Q' to switch the current tool.
- 'J' to plant the current type of seed.
- 'I' to switch seed with another type of seed.
- 'P' to open the player’s inventory menu.
- ‘H’ to open the help menu.
- 'O' to close the help menu, inventory menu, or merchant menu if they are active.
- 'Enter' to let the player sleep (reset the day) if it is pressed near the player’s bed, open the merchant menu if it is pressed in front of the merchant, and feed the cow if it is pressed inside the cow farm.
- 'Arrow' to navigate between items that are available when the merchant menu is active.
- 'C' to start a cutscene after doing some missions.
- ‘LCtrl’ to proceed to the next line of dialogue in a cutscene.

6. Game Output

a. Player image. The player has many kinds of images based on what they are doing at the moment, from idle, using specific tools, and walking (based on the player’s input). The direction in which they do something also determines which player

image will be blitted to the screen. The images of the player are all contained in the ‘graphics/character’ folder.

b. Background image. The background image can be accessed through the ‘graphics/world’ folder which the name is ‘ground.png’.
c. Object images. The rest of the objects’ images like cuttable trees, flowers, fences, houses, cows, milk, stumps, and plants. are also in the ‘graphics’ folder, but in a different type of folder based on their name. The reason why they don’t exist in ‘ground.png’ is because they are planned to have a hitbox so that they can detect collisions.
d. Water and rain image. The water images can be accessed through the ‘graphics/water’ folder, while the rain image can be accessed through the ‘graphics/rain’ folder where the water and rain have several states (it is animated).
e. Button and overlay images can be accessed in the ‘graphics’ folder with their respective name. The button and overlay image are blitted on top of the screen so that it will move around with the player.
f. Soil images. The default soil has the colour of green, but if it’s farmable and is tilled by the player, then it will change its image. The brownish image is available in the ‘graphics/soil’ image. If the soil is further watered, then there will be also water on top of the soil. The water that is on top of the soil is one of the images in ‘graphics/soil_water’.
g. Cutscene images. The images for the dialogue box, character’s emoticon, and cutscene picture are available in the ‘graphics/dialogue’ folder where for every cutscene, there will be a specific folder for them based on their order.
h. Player’s money.
i. Player’s inventory. The list of items that the player currently has can be accessed in the game by toggling the inventory menu. There will be also an image for each item that the player has. The images for those are available in the ‘graphics/inventory’ folder.
j. Background music is in the audio folder and its name is ‘bg.mp3’.
k. Sound effects are also available in the audio folder. The sound effects in this game are

- When the player uses an axe to chop a tree (axe.mp3)
- When a player buys something (buy.mp3)
- When the player clicks on the play button in the intro menu or toggle the help and inventory menu (click.mp3)
- When the player presses LCtrl to proceed to the next line of dialogue in a cutscene (cutscene.mp3)
- When the player feeds the cows with grass (grass.mp3)
- When the player uses a hoe to till soils (hoe.mp3)
- When the player harvests milk (milk.mp3)
- When the player plants a seed (plant.mp3)
- When the player sells something (sell.mp3)
- When the player acquired crops or woods (success.mp3)
- When the player waters a plant (water.mp3)

7. Game Libraries (Modules)

a. Pygame

Pygame is a set of Python modules to create a game. Used for creating the game window, and the game objects and blitting them to the screen by using surface and rect, drawing text, playing sounds, and detecting collisions.

b. PyTMX

PyTMX or Python Tiled Map XML is a set of Python modules specified to load and handle maps for games. So, we can easily edit our map in Tiled and use it in Pygame by utilizing PyTMX. In this game, PyTMX helped me to get the tile layer's and object layer's names and properties based on Tiled, so that I can use it in the program.

c. random

The random module in Python is a built-in module that is used to generate random choices or numbers. In this game, I used the randint and choice from the random module. The randint function is used to help generate the probability of rain happening, while the choice function is used to choose one of the elements

inside a list. The choice function is used when generating the water on top of the soil tiles and the water drop and water floor when it is raining.

d. os

Python has a built-in module, called the os module that contains methods to handle and interact with the operating system. In this game, I use the walk function from the os module that generates the file names in a directory tree by walking the tree. I used it to help me to return the surfaces from the images in a specific folder.

e. sys

The sys Python module provides various functions and variables to manipulate the Python runtime environment. In this game, I use the exit function to exit the program when the player quits the game or presses the 'Esc' key.

8. Game Important Files and Folders

a. 'audio' folder, contains the background music and the sound effects for this game.

b. 'code' folder, contains the Python code of this game.

- cutscene.py, contains the class CutSceneOne, CutSceneTwo, CutSceneThree, and CutSceneFour. Each of them is used to handle their own respective cutscene. cutscene.py also contains the class CutSceneManager to manage cutscene and update the status of a completed cutscene.

- game_display.py, contains the class Display that is used to manage all the game functions and displays if the player has started the game and the class CameraGroup to move the camera based on the player's movement so that the player is still in the centre of the screen.

- game_settings.py, contains the constant values of some variables that are used in the game, like the screen width, screen height, and many more.

- intro.py, contains the class Intro that is used to draw and manage the intro menu screen (the display screen before the player starts the game).

- main.py, contains the class Game that is used to handle methods to run the game (contains the game loop).

- merchant_menu.py, contains the class Menu that is used to handle the merchant shop, so that the player can buy and sell something through the shop menu.

- overlay_menu.py, contains the class Overlay_Menu which handles the help menu display and the class Inventory which handles the inventory menu. Both of them are displayed overlay on the screen, that's why they are in the same file.

- overlay.py, contains the class Overlay that is used to handle the images on the bottom left of the screen that represents the currently used tool and current type of seed.

- player.py, contains the class Player that handles the player's movement, animation, tool usage, collisions, and many more that are related to the player.

- sky.py, contains the class Sky which draws a darker sky over time, the class Drop handles the movement of the raindrops and their lifetime, and the class Rain creates the raindrops on the screen and updates it.

- soil.py, contains the class Plant to handle methods for growing plants, the class SoilTile as the tilled soil tiles after the player uses a hoe, the class WaterTile as the water on top of the soil tiles after the player waters the soil tiles, and the class SoilLayer to handle methods of tilling and planting in the game.

- sprites.py, contains the class Generic for every kind of object in the game. Besides Generic, there are also the classes Interaction, Water, Particle, Tree, and Animal which are all inherited from Generic.

- The class Interaction is used to handle non-visible objects but the player can interact with them, for example, in front of the merchant, beside the bed, etc.
- The class Water is used to draw water and animate them.
- The class Particle is used to draw a Particle when a crop is harvested.
- The class Tree is used to handle a method that is related to a cuttable tree.
- The class Animal is used to draw and animate the cows.

- support.py, contains the class Timer to know and handle the duration of a player when doing something and the class Button to handle and draw a button on the screen.
- transition.py, contains the class Transition to draw the transition if the player goes to sleep and reset the day.

c. ‘data’ folder, contains the .tmx file map for this game’s ground background and several .tsx files inside the tilesets folder. The .tsx files are used in the process to create the complete map in the ‘map.tmx’ file.

d. ‘document’ folder, contains the documentation (report and diagrams) for this project.

e. ‘font’ folder, contains the font types that are used in the game.

f. ‘graphics’ folder, contains all of the game picture assets. The ‘graphics’ folder contains 16 different folders that are all the pictures or sprites for the game. The folders are ‘animal’, ‘button’, ‘character’, ‘dialogue’, ‘environment’, ‘fruit’, ‘inventory’, ‘milk’, ‘objects’, ‘overlay’, ‘rain’, ‘soil’, ‘soil_water’, ‘stumps’, ‘water’, and ‘world’. The ‘environment’ and ‘objects’ folder contains the images to create the tilesets in the previous ‘data’ folder, meanwhile the rest besides of them are images that are going to be blitted on the screen when the game runs.

g. ‘story’ folder, contains four .txt files that are the story or the narration for each cutscene of this game.

C. Solution Design

1. Design Choices

For this game, I wanted to have a light-hearted and cute character as the main character. The game atmosphere should also be light-hearted and casual so that the player can experience a warm-hearted story in this casual game.

For the image design, I decided to use the online free sprite sheet assets from Sprout Land for my project because the character and the game elements are as cute as I intended. They also have more free assets than the other farm assets that I have searched. Below are the images of the Sprout Land assets.

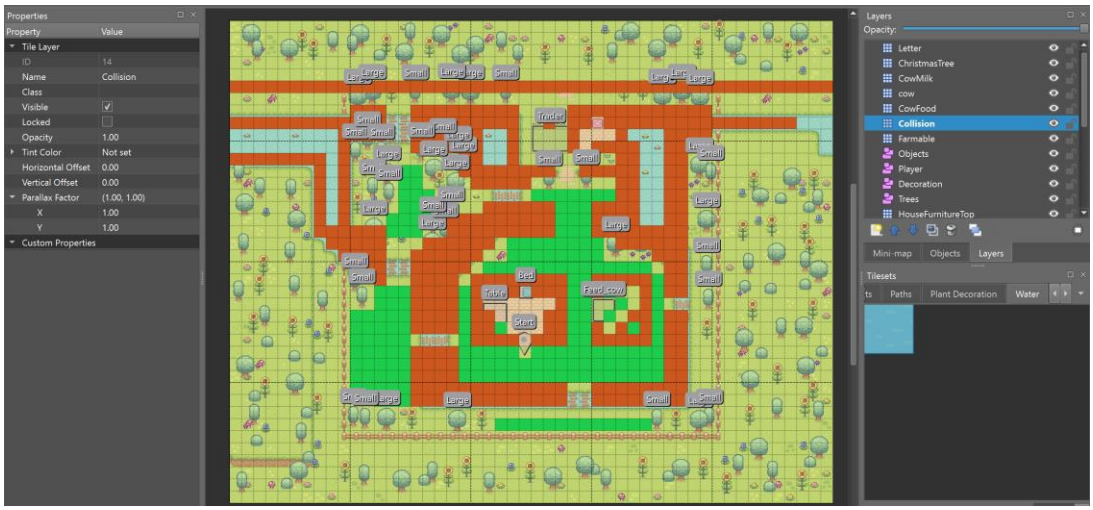
Source: https://cupnooble.itch.io/sprout-lands-asset-pack

The only objects that I didn't use from this assets pack are the Christmas tree and the Christmas hat which are shown below.

Source: https://www.istockphoto.com/id/vektor/pohon-natal-seni-piksel-dengan-ikon-vektor-bintang-dan-pernak-pernik-untuk-game-8bit-gm1431396087-474006614

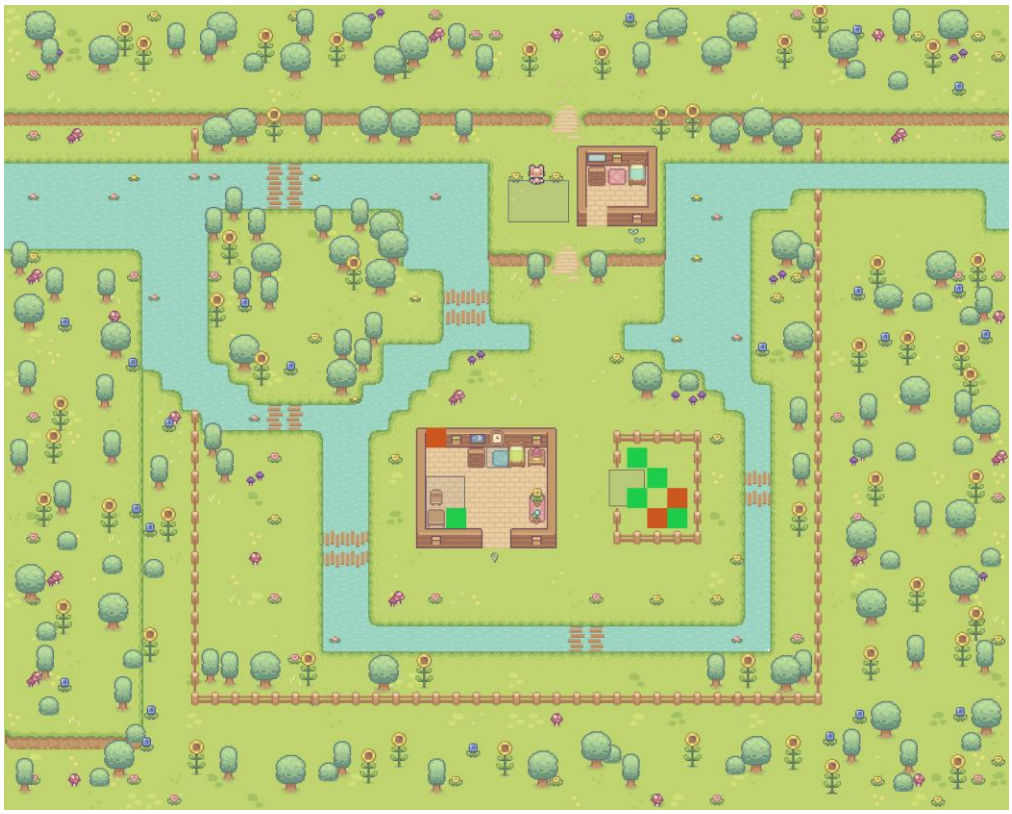
Source: https://www.freepik.com/premium-vector/santa-hat-pixel-art-style_35166269.htm

From the assets pack that I downloaded, I then created a map in Tiled for the world environment for this game. The map can be seen below. The green tiles are the farmable tiles (the tiles that can be tilled), while the red tiles are the collision tiles (the player can't walk across these tiles). The fence is not marked as collision tiles because the fences will be generated as a Generic object later and the Generic object has a hitbox.



Below is the map's appearance if the collision and farmable tiles are not visible. The red and green tiles inside the cow farm and Bunn's house are not farmable and collision tiles.

The red tile inside the house is for the Christmas tree, while the green one is for the letter. Meanwhile, the green tiles inside the cow farm are for the cows and the milk bottle, while the red one is for the cows' food.



Result

2. Font and Sound

The fonts that I used for this project are LycheeSoda and Rubik (Bold and Semibold). LycheeSoda is used mostly in the game because it's in pixel style, so it is very suitable for the overall design, while Rubik is only used for the help menu because I think that the help menu should have a straighter and easier-to-see font so that the player can read it more easily.

Source for LycheeSoda: https://www.dafont.com/lycheesoda.font

Source for Rubik: https://fonts.google.com/specimen/Rubik?query=rubik

Meanwhile, for the sound, as I said before, I wanted to have a light-hearted song, so I chose a light-hearted background music created by MusicTown from Pixabay (https://pixabay.com/music/search/light-hearted/) and renamed it to bg.mp3.

Below are the sources for the other sound effects that I have used throughout the game:

- axe.wav
  Source: https://www.soundsnap.com/axe_impact_hit_wood_blastwavefx_26204

- buy.mp3
  Source: Pixabay from Pixabay (https://pixabay.com/sound-effects/coin-c-02-102844/)

- click.mp3
  Source: UNIVERSFIELD from Pixabay (https://pixabay.com/sound-effects/interface-124464/)

- cutscene.mp3
  Source: UNIVERSFIELD from Pixabay (https://pixabay.com/sound-effects/button-124476/)

- grass.mp3
  Source: https://www.youtube.com/watch?v=4eVmKIfZFpY

- hoe.mp3

Source:https://www.videvo.net/sound-effect/hoe-chop-dirt-grub-soil-pehd069902/247265/#rs=audio-download

- milk.mp3
  Source: https://www.youtube.com/watch?v=3aOnFCVz9iY

- plant.mp3
  Source: Pixabay from Pixabay (https://pixabay.com/sound-effects/21-seed-46670/)

- sell.mp3
  Source: Pixabay from Pixabay (https://pixabay.com/sound-effects/cash-register-fake-88639/)

- success.mp3
  Source: https://www.zapsplat.com/music/retro-80s-arcade-game-jump-8/

- water.mp3
  Source: Pixabay from Pixabay (https://pixabay.com/sound-effects/pouring-drink-sound-effect-by-rea-102018/)

3. Diagrams

All the diagrams (the use case diagram, the activity diagram, and the class diagram) are inside the 'document' folder.

D. Essential Algorithms

1. game_settings.py

This file includes the constant variables that will be used throughout the game. Below is the explanation for the code in this file.

```
from pygame.math import Vector2
```

First of all, I imported Vector2 from pygame.math. Vector2 is used to define a two-dimensional vector.

```
#game name
GAME_NAME = "Bunn's Christmas Tree Tale"
```

The first constant variable that I define is the game's name which is Bunn's Christmas Tree Tale. The game circles around Bunn and his story with the Christmas tree, so I think this is a suitable name for this game.

```
#screen configuration
SCREEN_HEIGHT = 720
SCREEN_WIDTH = 1280
TILE_SIZE = 64
```

These three variables are the screen and the map's tile size for this game. The window's dimension is 1280px x 720px, while the tile size is 64px. So, for every tile in map.tmx, when it is blitted on the screen when this game runs, the size will be 64px x 64px.

```
#overlay positions
OVERLAY_POSITIONS = {
    'tool' : (40, SCREEN_HEIGHT - 15),
    'seed' : (40, SCREEN_HEIGHT - 80)
}
```

This dictionary defines the position of the overlay pictures of the tool and the seed. As we can see, it is positioned on the bottom left of the game window.

```
#player tool offset to determine the target position of the used tool
PLAYER_TOOL_OFFSET = {
    'left': Vector2(-50,40),
    'right': Vector2(50,40),
    'up': Vector2(0,-10),
    'down': Vector2(0,50)
}
```

This dictionary defines the tool offset as when the player faces up, down, left, or right. This tool offset will be used to determine the target position when the player uses a tool.

```
#layers of objects
LAYERS = {
    'water': 0,
    'ground': 1,
    'soil': 2,
    'soil water': 3,
    'rain floor': 4,
    'house bottom': 5,
    'main': 6,
    'rain drops': 7
}
```

This dictionary contains the layer value of each kind of object in this game. The highest the value, then it will be blitted more top. Most of the object in this game is blitted in the ‘main’ layer.

```
#plant's grow speed
GROW_SPEED = {
    'corn': 1,
    'tomato': 0.7
}
```

This dictionary contains the growth speed for each plant in this game, which is corn and tomato. The lower the number, then the slower it will grow. In this case, corn grows faster than tomatoes.

```
#item's sale prices
SALE_PRICES = {
    'corn': 10,
    'milk': 35,
    'tomato': 20,
    'wood': 5
}
```

This dictionary contains all the sale prices for the items in this game. If a player sells an item to the merchant, then the sale price of the specific item will be added to the player’s money.

```
#item's purchase prices
PURCHASE_PRICES = {
    'corn': 4,
    'tomato': 5,
    'grass' : 8
}
```

On the contrary, this dictionary contains all the purchase prices of each item in the game. So, if the player buys something, then the purchase price of each item will be deducted from the player's money. The corn and tomato here are also not the plants, but rather the seeds.

```
#keyboard keys input in this game
KEYBOARD_KEYS = {
    'Esc' : 'Quit the game',
    'A' : 'Move left',
    'D' : 'Move right',
    'W' : 'Move up',
    'S' : 'Move down',
    'Space' : 'Use tool and buy/sell things',
    'Q' : 'Switch tool',
    'J' : 'Plant seed',
    'I' : 'Switch seed',
    'P' : 'Check inventory',
    'H' : 'Open help menu',
    'O' : 'Close menu',
    'Enter': 'Go to bed, open merchant menu, or feed cow',
    'Arrow' : 'Navigate items in merchant menu',
    'C' : 'Start cutscene',
}
```

Next, it's the KEYBOARD_KEYS dictionary. This dictionary contains the keyboard shortcuts in this game when the cutscene is not running. If the cutscene is running, then the only key that the player can press to update the game (cutscene) is the LCtrl key.

```
#overlay menu's button position
BUTTON_POS = {
    'help' : (1220, 40),
    'inventory' : (1120, 40)
}
```

This dictionary contains the help menu and inventory button position. The position of both of them is on the top right side of the screen or window.

```
#colors that are common in the game
GAME_COLOR = {
    'dialogue box' : (232, 207,166),
    'text': (184,139,98),
    'inventory' : (244,231,196)
}
```

This dictionary contains three different shades of brown colour. These brown colours are commonly seen in the game, so for convenience, I also include them in this file.

2. **cutscene.py**

This file contains the classes CutSceneOne, CutSceneTwo, CutSceneThree, CutSceneFour, and CutSceneManager. So, this file is specially used to manage cutscenes in the game.

```
import pygame
from game_settings import *
```

Import pygame and all constant variables from game_settings.

```
def draw_text(screen, text, size, color, x, y):
    #a function to draw the text on the screen, the font is all LycheeSoda
    font = pygame.font.Font('../font/LycheeSoda.ttf', size)

    #render font to become text surface and sets its rect position as (x,y) from topleft
    text_surface = font.render(text, True, color)
    text_rect = text_surface.get_rect(topleft=(x,y))

    #blit the text on the screen
    screen.blit(text_surface, text_rect)
```

The draw_text function is used to draw the text on the screen. First of all, the function will create a new font object with the type of LycheeSoda and the size that is passed to the function. Next, it will make a text surface by rendering the font with the text that is passed to the function. Then, it gets the rect and sets its position. After that, it will blit the text surface based on the text rect position on the screen surface.

```
def blit_bg_image(screen, path):
    #get the background image surface and set its rect as (0,0)
    #because background image has to cover all of the screen
    bg_pict = pygame.image.load(f'../graphics/dialogue/{path}')
    bg_rect = bg_pict.get_rect(topleft=(0,0))
    #blit the image
    screen.blit(bg_pict, bg_rect)
```

The blit_bg_image function is used to blit the background image on the screen. It will first load the background pictures as a surface based on the given path. Then, it will get the rect. As it is a background image, so I set the position to start from (0,0). After that, it will blit the background image on the screen based on the rect's position.

```
def draw_chara(screen, path):
    #get the character image surface and set its rect as (65,510)
    #to fit in the dialogue box
    chara_surf = pygame.image.load((f'../graphics/dialogue/{path}'))
    chara_rect = chara_surf.get_rect(topleft=(65,510))
    #blit the chara avatar image
    screen.blit(chara_surf, chara_rect)
```

The draw_chara function is used to draw the avatar icon of the character that is speaking on its box inside the dialogue box. First off, it will get the character image surface based on the given path. After that, it gets its rect and sets its position based on the location of the box. Lastly, this function will blit the surface on the screen.

```
def draw_dialog_box(screen):
    #get the dialogue box image surface and set its rect to fit in the screen
    dialog_pict = pygame.image.load(f'../graphics/dialogue/dialog_box.png')
    dialog_rect = dialog_pict.get_rect(topleft=(0,450))
    #blit the dialogue box image
    screen.blit(dialog_pict, dialog_rect)
```

The draw_dialog_box_function takes on one parameter which is the screen. It also has a similar workflow as the other functions before it. It will first get the dialogue box image

surface and then set its rect so that it will fit on the screen. After that, it will blit the dialogue box surface on the screen.

- Class CutSceneOne

```
class CutSceneOne:
    def __init__(self):
        # Basic setup
        self.name = 'cutscene1' #cutscene's name
        self.step = 0 #on which scene now is the cutscene
        self.cut_scene_running = True #the status of the cutscene

        # Dialogue
        self.setup()
        # text counter is used to create the slide-right effect when playing the cutscene
        self.text_counter = 0

        # Sound when press LCtrl key to proceed the cutscene
        self.next_sound = pygame.mixer.Sound('../audio/cutscene.mp3')
        self.next_sound.set_volume(0.3)
```

The __init__ function is called when a new object is created from this class. The class CutSceneOne __init__ function has the attribute name, which is ‘cutscene1’, then by default (because the cutscene hasn’t started yet), the step will be 0. The step is on which scene or which line the cutscene is now. Also, the status of the cut_scene_running will be True when CutSceneOne is initialized, so it will automatically play the cutscene.

Next is the text counter. The text counter will be used to create a slide-right effect of the text when we play the cutscene. The default value will be 0 because we want to start the slide right from the first character in the line. And then, there is also the next_sound which is the sound effects that will be played when we press the LCtrl key to proceed with the cutscene. Here, I also lower the volume by setting the volume to 0.3.

```
    def setup(self):
        # Append all lines in the story1 in the self.text dictionary
        self.text={}
        file_story1 = open("../story/story1.txt", "r")
        for index, item in enumerate(file_story1.readlines()):
            self.text[index] = item
```

In the previous __init__ function, we also call the setup function. So, basically what the setup function does is read all the lines in the txt file of the story and then make a new key:value pair in the self.text dictionary. The key will be the index and the value will be the line of dialogue (string).

```
def update(self):
    #check if the LCtrl key get pressed or not
    pressed = pygame.key.get_pressed()
    lctrl = pressed[pygame.K_LCTRL]

    #for every lines of dialogue
    for i in range(0, len(self.text)):
        #if cutscene is not on the last line of dialogue
        if self.step < len(self.text) - 1:

            #if the text counter still shorter than amount of character
            #in the dialogue line, then add the text counter to provide the
            #slide-right effect of the text
            if int(self.text_counter) < len(self.text[i]):
                self.text_counter += 0.05

            #if LCtrl is pressed, then play next sound, go to the next step
            #text counter becomes 0 again
            else:
                if lctrl:
                    self.next_sound.play()
                    self.text_counter = 0
                    self.step += 1
```

Next is the update method. First off, we will check if the LCtrl key is pressed or not. Then, for every line of dialogue, if the cutscene is not on the last line of dialogue, then it will perform the text slide right animation of the dialogue line. This animation can be achieved by using the text counter. If the text counter is still shorter than the number of characters in the dialogue line, then add the text counter until it reaches the end of the dialogue line. If LCtrl is pressed, then play the next sound, the text counter will become 0 again (so that the next dialogue line can have the slide right animation too), and then go to the next step or the next line of dialogue.

```
            #if cutscene is on the last line of dialogue
            else:
                #the text animation slide-right effect
                if int(self.text_counter) < len(self.text[i]):
                    self.text_counter += 0.05
                else:
                    if lctrl:
                        #finish cutscene if LCtrl is pressed
                        self.cut_scene_running = False

        return self.cut_scene_running
```

Else, if the cutscene is already on the last line of dialogue. It will also perform the slide-right animation effect. But upon pressing LCtrl, the cutscene will be finished and the status of cut_scene_running will be False.

```
def draw(self, screen):
    #for every lines of dialogue
    for i in range (0, len(self.text)):
        if self.step == i:
            #if it is the first line, blit a black rect
            if self.step == 0 :
                pygame.draw.rect(screen, (0,0,0), (0, 0, screen.get_width(), screen.get_height()))

            #blit background image based on the line's number
            elif self.step == 1 or self.step == 2 or self.step == 3:
                blit_bg_image(screen, 'cutscene1/bg1.png')
            elif self.step == 4 or self.step == 5:
                blit_bg_image(screen, 'cutscene1/bg2.png')
            elif self.step == 6 or self.step == 7 or self.step == 8:
                blit_bg_image(screen, 'cutscene1/bg3.png')
            else:
                blit_bg_image(screen, 'cutscene1/bg4.png')

            #draw the narrator brown rect on the screen
            pygame.draw.rect(screen, GAME_COLOR['dialogue box'], (0, 550, screen.get_width(), 400))
            #draw the dialogue line text
            draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 50, 600)
```

The last method of this class is the draw method. The draw method is specialized to draw the background picture of each scene in the cutscene and also the dialogue or the narrator box, and the dialogue text. So, for every line of dialogue, if it's the first line, then draw a black rect as the background image. If that's not the case, then blit the background image based on the line's number or the step because each line of dialogue has a different scene. Lastly, it will draw a narrator box (rect) and the text for each dialogue's line. The first cutscene doesn't have any dialogue or monologue,

only narrator lines, that's the reason why we only need to draw the narrator box for every line of dialogue in this cutscene.

- Class CutSceneTwo

```
    self.name = 'cutscene2' #cutscene's name
```

CutSceneTwo is very similar to CutSceneOne. It has the same attributes as CutSceneOne, the only difference is its name is cutscene2.

```
    file_story2 = open("../story/story2.txt", "r")
```

It also has four methods which are the __init__, setup, update, and draw. The setup method is slightly different because it gets the lines from another txt file. The main difference between CutSceneOne with CutSceneTwo and also with the other cutscene classes is the draw method.

```
def draw(self, screen):
    #for every lines of dialogue
    for i in range (0, len(self.text)):
        if self.step == i:
            #if it is the first line, draw the narrator box rect, black background rect, and the text
            if self.step == 0 :
                pygame.draw.rect(screen, (0,0,0), (0, 0, screen.get_width(), screen.get_height()))
                pygame.draw.rect(screen, GAME_COLOR['dialogue box'], (0, 550, screen.get_width(), 400))
                draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 50, 600)

            #blit background image based on the line's number
            elif self.step == 1 or self.step == 4 or self.step == 5 or self.step == 6:
                #if it is the 2nd, 5th, 6th, or 7th line
                blit_bg_image(screen, 'cutscene2/bg1.png')
                #draw the dialogue box, chara avatar, and text
                draw_dialog_box(screen)
                draw_chara(screen, 'cutscene2/emote1.png')
                draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 275, 575)

            else:
                blit_bg_image(screen, 'cutscene2/bg1.png')
                #draw the narrator box
                pygame.draw.rect(screen, GAME_COLOR['dialogue box'], (0, 550, screen.get_width(), 400))
                draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 50, 600)
```

For every line of dialogue, if it's the first line, then draw the narrator box, black background image, and the text. But, if it isn't, then draw the other background images. And if it is the second, fifth, sixth, or seventh line, then draw the character avatar and the dialogue box. Otherwise, draw the narrator box again. So, as we can see, in this cutscene, there are dialogue boxes beside the narrator boxes.

- Class CutSceneThree

The differences are also the same, which is the name and the story file in the setup method and also the draw method.

```
self.name = 'cutscene3' #cutscene's name
```

```
file_story3 = open("../story/story3.txt", "r")
```

```
def draw(self, screen):
    #for every lines od dialogue
    for i in range (0, len(self.text)):
        if self.step == i:
            #draw black background, draw narrator box, and text
            if self.step == 0 :
                pygame.draw.rect(screen, (0,0,0), (0, 0, screen.get_width(), screen.get_height()))
                pygame.draw.rect(screen, GAME_COLOR['dialogue box'], (0, 550, screen.get_width(), 400))
                draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 50, 600)

            #draw corresponding background image, draw narrator box, and text
            elif self.step == 5 or self.step == 12:
                blit_bg_image(screen, 'cutscene3/bg1.png')
                pygame.draw.rect(screen, GAME_COLOR['dialogue box'], (0, 550, screen.get_width(), 400))
                draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 50, 600)

            else:
                #draw the corresponding background image, dialogue box, and text
                blit_bg_image(screen, 'cutscene3/bg1.png')
                draw_dialog_box(screen)
                draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 275, 575)

                #update the chara avatar and expression based on the character speaking at the moment
                if self.step == 1 or self.step == 2: draw_chara(screen, 'cutscene3/emote1.png')
                elif self.step == 4: draw_chara(screen, 'cutscene3/emote2.png')
                elif self.step == 9 or self.step == 10: draw_chara(screen, 'cutscene3/emote3.png')
                else: draw_chara(screen, 'cutscene3/emote4.png')
```

The draw method in CutSceneThree is more complex than the one in the previous cutscenes because here, there are several character icons, so it has to be more specified. Here, it blits a black background image in the first line as usual. After that, it will draw the narrator's box only on the sixth and thirteenth lines. Besides those, all of the rest are dialogue boxes. There are also different kinds of character icons for each specific line in the cutscene.

- Class CutSceneFour

  The differences are also the same, which is the name and the story file in the setup method and also the draw method.

```
self.name = 'cutscene4' #cutscene's name
file_story4 = open("../story/story4.txt", "r")
```

```
def draw(self, screen):
    #for every lines of dialogue
    for i in range (0, len(self.text)):
        if self.step == i:
            #if it's tthe first line, draw black background
            if self.step == 0 :
                pygame.draw.rect(screen, (0,0,0), (0, 0, screen.get_width(), screen.get_height()))

            #else, blit background image
            else:
                blit_bg_image(screen, 'cutscene4/bg1.png')
                #if it's the 7th or 8th line, blit grandpa's letter image
                if self.step == 7 or self.step == 8:
                    letter_grandpa_surf = pygame.image.load('../graphics/dialogue/cutscene4/letter.png')
                    letter_grandpa_rect = letter_grandpa_surf.get_rect(center=(SCREEN_WIDTH//2, SCREEN_HEIGHT//2 - 90))
                    screen.blit(letter_grandpa_surf, letter_grandpa_rect)

            #draw narrator box for all of them because there is no dialogue in this cutscene and draw text
            pygame.draw.rect(screen, GAME_COLOR['dialogue box'], (0, 550, screen.get_width(), 400))
            draw_text(screen, self.text[i][0:int(self.text_counter)], 35, GAME_COLOR['text'], 50, 600)
```

Here, in CutSceneFour, the draw method is more similar to the one in CutSceneOne because it only contains the narrator box. As usual, the first background image is still the black rect. The rest of the line has the picture in cutscene4/bg1.png as the background image. However, there is also a special image in the eighth and ninth lines of this cutscene, which is a letter image that will be blitted on top of the background image.

- Class CutSceneManager

```
class CutSceneManager:
    def __init__(self, screen):
        #manage cutscene status
        self.cut_scenes_complete = [] #contains name of all completed cutscenes

        #initially there will be no cutscene running
        self.cut_scene = None
        self.cut_scene_running = False

        #screen to be drawn on
        self.screen = screen
```

CutSceneManager is a class that helps to manage cutscenes. When we initialize an instance of this class, it will have the attribute: cut_scene_complete which is a list that contains all completed cutscenes' names, cut_scene which will be None at the beginning because there is no cutscene object, the status of cut_scene_running is still False, and the screen.

```
def start_cut_scene(self, cut_scene):
    #if a cutscene is started, then if it's not completed yet,
    #append the cutscene to the list of completed cutscenes
    if cut_scene.name not in self.cut_scenes_complete:
        self.cut_scenes_complete.append(cut_scene.name)

        #set the cutscene to be running
        self.cut_scene = cut_scene
        self.cut_scene_running = True
```

The start_cut_scene method is used to append the cut_scene_name in the list of completed cutscenes if it's not completed yet and the player starts the cutscene. When starting the cutscene, the cutscene object will run, and the status of the cut_scene_running will become True.

```
def end_cut_scene(self):
    #the cutscene will stop running
    self.cut_scene = None
    self.cut_scene_running = False
```

The end_cut_scene method is to set the cutscene back to None and the status of cut_scene_running back to False because the cutscene has ended.

```
def update(self):
    #update the cutscene status based on each cutscene
    #if the cutscene reaches the end, then the status will be False
    if self.cut_scene_running:
        self.cut_scene_running = self.cut_scene.update()
    else:
        #if the cut_scene_running becomes False, end the cutscene
        self.end_cut_scene()
```

The update method is used to update the cutscene based on the update method in each cutscene. If the cutscene reaches the end of the dialogue line, then it will return False for cut_scene_running in the previous cutscene classes. If not, then it will be True. So, here it basically updates the cut_scene_running status periodically and if it becomes False, the cutscene manager will end the cutscene.

```
def draw(self):
    #draw the running (active) cutscene
    if self.cut_scene_running:
        self.cut_scene.draw(self.screen)
```

The draw method is used to call the draw method of each cutscene. So, if the cutscene is active or still running, then call the draw method of the running cutscenes, so that it can blit text, boxes, background image, etc.

3. support.py

This file contains functions and classes that are widely used in other files, especially the import_folder function. That's why its name is "support.py" because it supports other files.

```
from os import walk
import pygame
```

Import the walk function from the os module and import pygame.

```
def import_folder(path):
    #import folder and its content
    surface_list = []

    #get the image files names, generate a path for them, and then make them a surface
    for _, __, img_files in walk(path):
        for image in img_files:
            full_path = path + '/' + image
            image_surf = pygame.image.load(full_path).convert_alpha()
            #after that, append them in a surface list
            surface_list.append(image_surf)
    return surface_list
```

The import_folder function is used to return a list of surfaces based on the path given in the parameter. First off, it will generate an empty surface list. After that, using the walk function, it will generate the name of the image files in the folder path. Then, it will load them as surface and append them to the surface list.

```
def import_folder_dict(path):
    surface_dict = {}

    #get the image files names, generate a path for them, and then make them a surface
    for _, __, img_files in walk(path):
        for image in img_files:
            full_path = path + '/' + image
            image_surf = pygame.image.load(full_path).convert_alpha()
            #after that, append the surface in a dictionary with the name of the image
            #as the key and the surface as the value
            surface_dict[image.split('.')[0]] = image_surf
    return surface_dict
```

Next is the import_folder_dict function. This function is similar to the import_folder function but it returns a surface dictionary instead with the image file name as the key and the surface as the value. First, it will initialize a new surface dictionary. Then, it will walk the folder path and get the image file name. Then, it will generate a full path, so that it can load the image surface using that. After that, it will split the image name by '.' so that the key will be only the image name, not containing the .png and the value will be the surface.

- Class Timer

  The class Timer is used to let players have a more normal switching or doing some function experience. What this means is that for example, when we want to switch tools in this game, we have to press the 'Q' key. When we press it one time, pygame will not detect it as just one time, but it will detect it as a series of presses instead. So, the switching animation will be too fast and not comfortable for the player. So, it is important to have a timer, so that the player can only switch tools after some milliseconds.

```
class Timer:
#timer to know the duration of a player when doing something
    def __init__(self,duration,func = None):
        self.duration = duration #timer duration
        self.func = func #function to do
        self.start_time = 0
        self.active = False
```

  When we create an instance of the class Timer, there will be the duration of the timer, the function (that is by default None), the start time, and the status (active) of the timer which is by default False.

```
    def activate(self):
        #when the timer is activated
        self.active = True
        self.start_time = pygame.time.get_ticks()
```

The activate method is used to activate the timer by setting the status of the active to True and the start_time of the timer to the current time when the timer is started.

```
    def deactivate(self):
        #when the timer is deactivated
        self.active = False
        self.start_time = 0
```

On the contrary, the deactivate method is used to deactivate the timer by setting the status of the active back to False and the start_time back to 0.

```
    def update(self):
        #update timer
        current_time = pygame.time.get_ticks()
        #if current time minus the start time is greater than the duration
        if current_time - self.start_time >= self.duration:
            #if the function is available and start time is not 0
            if self.func and self.start_time != 0:
                self.func() #do this function
            self.deactivate() #deactivate the timer
```

The update method is used to get the current time. If the current time minus the start time is greater than the duration of the timer and if the function is None and the start time is not 0, the player can then do the function that they want. After that, the timer will deactivate. In other words, this Timer is something like ICD (Internal Cooldown).

- Class Button

```
class Button:
    def __init__(self, image, pos):
        #image
        self.image = image
        #position
        self.x_pos = pos[0]
        self.y_pos = pos[1]
        #rect
        self.rect = self.image.get_rect(center=(self.x_pos, self.y_pos))
```

The Button class is used to create buttons. When creating an object of this class, it will have the attribute image, position, which is the x and y position, and also the rect. The rect itself is initialized by getting the image rect and setting it to center by the x and y positions.

```
    def update(self, screen):
        #if image is not None, blit image to the screen
        if self.image is not None:
            screen.blit(self.image, self.rect)
```

The update method is used to blit the button image on the screen. If the image is not none, then blit the button image on the screen based on the image surface and its rect.

```
    def checkForInput(self, position):
        #if the x and y position is inside the button image, return True, else False
        if position[0] in range(self.rect.left, self.rect.right) and \
            position[1] in range(self.rect.top, self.rect.bottom):
            return True
        return False
```

The checkForInput method is used to track the mouse position of the player. If the mouse's x position is between the left and right of the rect and the y position between the top and bottom of the rect (the player mouse is inside the button), then return True. Otherwise, return False.

4. sprites.py

This file contains all the sprite classes that are used in this game.

```
import pygame
from game_settings import *
from support import *
```

Import pygame, and from game_settings and support module import all.

- Class Generic

```
class Generic(pygame.sprite.Sprite):
    #class for every generic elements in the game
    def __init__(self, pos, surf, groups, z = LAYERS['main']):
        super().__init__(groups)
        #contains image, rect, layer(z), and hitbox
        self.image = surf
        self.rect = self.image.get_rect(topleft = pos)
        self.z = z
        self.hitbox = self.rect.copy().inflate(-self.rect.width * 0.2, -self.rect.height * 0.75)
```

Generic is the standard class object for all the objects in this game as most of them will inherit from this class. Generic is also an inheritance of the pygame Sprite. Upon creating an object of the Generic class, the object will have the attribute of image (surface), the rect that is got by the image and position is set to top left to pos that is given, layers (which is by default the main layer because most of the objects are in this layer), and the hitbox. The hitbox is the copy of the rect and it is inflated to ensure a nicer collision detection effect.

- Class Interaction

```
class Interaction(Generic):
    def __init__(self, pos, size, groups, name):
        #has a name, so that we can define what is the interaction
        #when player collides with this object
        surf = pygame.Surface(size)
        super().__init__(pos, surf, groups)
        self.name = name
```

Interaction is inherited from Generic and the surface is a non-visible pygame Surface because this sprite's main usage is to detect if a player collides with the object of this class or not. If yes, then do something based on the name of the object. That’s why the Interaction class has the name attribute.

- Class Water

```
class Water(Generic):
    def __init__(self, pos, frames, groups):
        #animation setup
        self.frames = frames
        self.frame_index = 0

        #sprite setup
        super().__init__(
                pos = pos,
                surf = self.frames[self.frame_index],
                groups = groups,
                z = LAYERS['water'])
```

The Water class is also inherited from the Generic class. Upon initializing an object from the Water class, there will be additional attributes, which are the frames and the frames index. The reason is that, unlike ground, the water object will have animation.

```
    def animate(self,dt):
        frame_change = 5
        #animation, change by frame_change and delta time
        self.frame_index += frame_change * dt
        if self.frame_index >= len(self.frames):
            self.frame_index = 0
        #update the image with the new surface
        self.image = self.frames[int(self.frame_index)]

    def update(self,dt):
        #update the water
        self.animate(dt)
```

The animate method is used to animate the water object based on the frame change and delta time. Delta time is used to ensure smooth animation across framerate. So, to animate this, the frame index will change based on the frame changing speed and the delta time, then if the frame index is greater or equal to the length of the list of the frames, then the frame index will go back to 0. The image surface will be updated periodically based on the frame index.

The update method basically updates the water by animating it.

- Class Particle

```
class Particle(Generic):
    #particle when plant is destroyed
    def __init__(self, pos, surf, groups, z, duration = 200):
        super().__init__(pos, surf, groups, z)
        #the start time and duration
        self.start_time = pygame.time.get_ticks()
        self.duration = duration

        #generate white surface that is a masked surface from the image
        mask_surf = pygame.mask.from_surface(self.image)
        new_surf = mask_surf.to_surface()
        new_surf.set_colorkey((0,0,0)) #white color
        self.image = new_surf
```

Particle is also inherited from Generic. An object of the Particle class is a white surface that has the same shape as the object and is normally seen when the player destroys the plant. The object from this class has the additional attributes of start time and duration. The image or the surface is the masked surface from the plant image and has a white colour.

```
    def update(self,dt):
        current_time = pygame.time.get_ticks()
        #can only be seen for some miliseconds during the duration
        if current_time - self.start_time > self.duration:
            self.kill()
```

The update method is used to update the particle. First off, it will get the current time. If the duration of the particle is lesser than the current time minus the start time, then the particle object will be killed. So, we can only see the object from the Particle class only for some milliseconds.

- Class Tree

```
class Tree(Generic):
    def __init__(self, pos, surf, groups, name, player_add):
        #tree attributes
        self.health = 5
        self.alive = True #by default, alive

        #stump image
        stump_path = f'../graphics/stumps/{"small" if name == "Small" else "large"}.png'
        self.stump_surf = pygame.image.load(stump_path).convert_alpha()

        #tree image
        tree_path = f'../graphics/objects/{"tree_small" if name == "Small" else "tree_medium"}.png'
        self.tree_surf = pygame.image.load(tree_path).convert_alpha()
        super().__init__(pos, surf, groups)

        #add item
        self.player_add = player_add

        #sound
        #when tree is damaged, play axe sound
        self.axe_sound = pygame.mixer.Sound('../audio/axe.wav')
        self.axe_sound.set_volume(0.6)
```

The Tree class is also inherited from the Generic class. An instance of this class has additional attributes, like health, the status of the tree (alive or not), the stump surface, the tree surface, the player add (to add the cut tree or wood to the inventory) and the axe_sound (which is played when the tree is damaged). The stump surface is different between small and large trees, and the tree surface is also the same. I also set the volume of the axe sound there to 0.3.

```
    def damage(self):
        # damaging the tree
        self.health -= 1
        #play sound
        self.axe_sound.play()
```

The damage method is called when the player damages a tree. If that's the case, then it will decrease the tree's health one by one and play the axe sound.

```
def check_death(self):
    #if health is lesser than or equal to 0
    if self.health <= 0:
        #the surface will become stump
        self.image = self.stump_surf
        self.rect = self.image.get_rect(midbottom = self.rect.midbottom)
        self.hitbox = self.rect.copy().inflate(-10,-self.rect.height * 0.6)
        self.alive = False #tree is not alive
        self.player_add('wood') #add wood to player inventory
```

The check_death method is used to check if the tree is dead or not. If the tree is dead (the health is lesser or equal to 0), then the tree will become a stump. The surface will be changed to a stump surface and then get the rect and hitbox for the stump. The tree is dead, so the alive status of the tree also changes to False. It also adds 1 item of wood to the player's inventory.

```
def revive(self):
    #tree alive again and has full health
    self.alive = True
    self.health = 5

    #tree image change backs from stump to tree again
    self.image = self.tree_surf
    self.rect = self.image.get_rect(midbottom = self.rect.midbottom)
    self.hitbox = self.rect.copy().inflate(-10,-self.rect.height * 0.6)
```

If we reset the day, the tree will be alive again and has full health (the alive status will back to True and the health becomes 5 again). The surface also changes from stump back to tree surface again, so we also have to get (update) the rect and hitbox of the new surface (the tree surface).

```
def update(self, dt):
    #update the tree if it's dead
    if self.alive:
        self.check_death()
```

The update method is used to update the tree when the player is not sleeping. If the tree is still alive, check if it dies or not to constantly update them to stump if the player cuts them.

- Class Animal

```
class Animal(Generic):
    def __init__(self, pos, frames, groups):
        #animation setup
        self.frames = frames
        self.frame_index = 0

        #sprite setup
        super().__init__(
                pos = pos,
                surf = self.frames[self.frame_index],
                groups = groups,
                z = LAYERS['main'])
```

The Animal class is also inherited from the Generic class, but it has additional attributes, which are frames and frame_index (for animation). It is similar to the Water class, but it's initialized on the 'main' layer.

```
    def update(self,dt):
        frame_change = 5
        #animation based on frame change and dt
        self.frame_index += frame_change * dt
        if self.frame_index >= len(self.frames):
            self.frame_index = 0
        #update the image with the new surface
        self.image = self.frames[int(self.frame_index)]
```

The update method in the Animal class is used to update the frame for the animal (cow). So, the frame index will change (goes up) by adding itself with the frame change multiplied by the delta time. If the frame index is greater than the length of the frames, then the frame index goes back to 0. After that, update the image surface with the new surface from the frame index. The frame index has to be explicitly typed integer because it has the possibility to be a double and it will lead to an error.

5. **overlay.py**

This file contains the Overlay class that is used to handle the overlay picture of the tool and seed status on the bottom left of the screen.

```
import pygame
from game_settings import *
```

Import pygame and from game_settings import all constant variables.

- Class Overlay

```
class Overlay:
    def __init__(self,player):
        #general setup
        self.display_surface = pygame.display.get_surface()
        self.player = player

        #the surface of the tool and the seed, so that it can be drawn in the screen
        overlay_path = '../graphics/overlay/'
        self.tools_surf = {tool: pygame.image.load(f'{overlay_path}{tool}.png').convert_alpha() for tool in player.tools}
        self.seeds_surf = {seed: pygame.image.load(f'{overlay_path}{seed}.png').convert_alpha() for seed in player.seeds}
```

When an object of the class Overlay is initialized, then it will have several attributes. The first one is the display_surface, which is obtained by getting the display surface using pygame. After that, the player. Then, there is an overlay_path which is the path to the image of the tools and seeds. Then, there are the attributes tools_surf and seeds_surf. The attribute tools_surf is a dictionary of the name of the tool as the key and the image surface of the corresponding tool as the value. Similar to tools_surf, seeds_surf is a dictionary of the name of the seed as the key and the image surface as the value.

```
    def display(self):
        #display tools in the screen
        tool_surf = self.tools_surf[self.player.tool_status]
        tool_rect = tool_surf.get_rect(midbottom = OVERLAY_POSITIONS['tool'])
        self.display_surface.blit(tool_surf,tool_rect)

        #display seeds in the screen
        seed_surf = self.seeds_surf[self.player.seed_status]
        seed_rect = seed_surf.get_rect(midbottom = OVERLAY_POSITIONS['seed'])
        self.display_surface.blit(seed_surf,seed_rect)
```

The display method is used to display the tools and the seeds status overlay image on the screen. First off, it will get the tool surface based on the status of the tool that the player uses and the seed surface based on the seed that is currently active. After that,

it will get the rect for the tool surface and the seed surface, and then blit both of them on the screen (display surface) with slightly different positions based on their rect positions.

6. transition.py

This file contains the class Transition which mainly handles the transition that is played when the player goes to sleep (reset the day).

```
import pygame
from game_settings import *
```

Import pygame and from game_settings import all of the constant variables.

- Class Transition

```
class Transition:
    def __init__(self, reset, player):
        #basic setup
        self.display_surface = pygame.display.get_surface()
        self.reset = reset #reset function
        self.player = player

        #overlay surface
        self.image = pygame.Surface((SCREEN_WIDTH,SCREEN_HEIGHT))
        #color
        self.color = 255
        #changing color speed
        self.speed = -2
```

The object from the Transition class has several attributes, which are the display surface that is obtained by using pygame to get the display surface, the reset function, the player, the overlay surface (image), the colour (initial), and the colour change speed.

```
def play(self):
    #playing color transition from white to black
    self.color += self.speed
    #if the color is black, then change the color slowly to white again
    #by multiplying the speed with -1 while also doing the reset function
    if self.color <= 0:
        self.speed *= -1
        self.color = 0
        self.reset()
    #if the color number is greater than 255, then change the color number back to 255
    #or white and player wakes up, the speed also becomes negative again
    if self.color > 255:
        self.color = 255
        self.player.sleep = False
        self.speed = -2

    #fill the surface with the corresponding color
    self.image.fill((self.color,self.color,self.color))
    self.display_surface.blit(self.image, (0,0), special_flags = pygame.BLEND_RGBA_MULT)
```

The play method is to play the colour transition. First of all, the overlay surface is still invisible (doesn't have any colour when we initialize it). Then, the play method will increase the colour number by adding it with the colour-changing speed. After that, if the colour number is 0 or lesser now, set the colour number to 0, multiply the speed by -1 to add 0 into 255 again and also do the reset function.

If the color number is now 255 again, then the player will wake up and the speed and colour number will be changed to the initial value. The surface will be filled periodically based on the colour number. The surface will also be blitted with the special flag multiply (kind of giving a shade layer on top of the surface). So, basically what this method does is to change the shade layer from white into black and then white again.

7. sky.py

This file contains the class Sky for day-night transition, class Rain and class Drop for the weather system (if it's raining).

```
import pygame
from game_settings import *
from support import import_folder
from random import randint, choice
from sprites import Generic
```

Import pygame and from game_settings import all, from support import import_folder, from random import randint and choice, and from sprites import Generic.

- Class Sky

```
class Sky:
    def __init__(self):
        self.display_surface = pygame.display.get_surface()

        #full surface of the sky
        self.full_surf = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT))
        #starts with white color and ends with rather blueish color
        self.start_color = [255, 255, 255]
        self.end_color = (38, 101, 189)
```

When initializing an instance of this class, there will be several attributes. The display surface attribute is obtained by getting the display surface with pygame. Then, the full surface is the shade surface on top of the screen, so its size has to be the same as the screen width and screen height. After that, there are the attributes start colour which is white and the end colour which is dark blue. The start colour is a list because it will change over time.

```
    def display (self, dt):
        #change the color if the start color is greater than the end color
        color_change = 2
        for index, value in enumerate(self.end_color):
            if self.start_color[index] > value:
                self.start_color[index] -= color_change * dt

        #fill the full surface with the start color that change over time
        self.full_surf.fill(self.start_color)
        #blit it with special blend multiply
        self.display_surface.blit(self.full_surf, (0,0), special_flags=pygame.BLEND_RGBA_MULT)
```

The display method is used to change the sky colour. The colour-changing speed is 2. For every item in the end colour tuple, if the start colour's item that has the same index as the end colour's item is still greater than the end colour's item, then change the start colour's item based on the colour change speed and delta time. After that, it will fill the still no colour full surface with the colour of the start colour that changes over time. It is also blitted in multiply.

- Class Drop

```
class Drop(Generic):
    def __init__(self, surf, pos, moving, groups, z):
        #general setup
        super().__init__(pos, surf, groups, z)
        self.lifetime = randint(400,500)
        self.start_time = pygame.time.get_ticks()

        #moving raindrop
        self.moving = moving
        if self.moving:
            #if the raindrop is moving, then set the position, direction, and speed
            self.pos = pygame.math.Vector2(self.rect.topleft)
            self.direction = pygame.math.Vector2(-3,4)
            self.speed = randint(200,250)
```

The Drop class is inherited from the Generic class, so it has the attributes of the Generic class. It also has additional attributes, like lifetime, start time, and moving. The Drop class will also be used to create the rain floor beside the raindrop, so if it's moving (it's a raindrop), then it has position, direction, and speed. The position and direction are two-dimensional vectors, while the speed is randomized between 200 and 250.

```
def update(self, dt):
    #movement
    if self.moving:
        #update the position
        self.pos += self.direction * self.speed * dt
        self.rect.topleft = (round(self.pos.x), round(self.pos.y))
    #timer
    if pygame.time.get_ticks() - self.start_time >= self.lifetime:
        #if it's duration already greater than it's lifetime then kill it
        self.kill()
```

The update method is used to update the raindrops and rain floors. If it is moving (raindrop), then it will update the position of the raindrop by adding its position with the direction multiplied by the speed and delta time. Then, it sets the rect based on the position that is rounded (so, it will be not in double). Next, there is also a timer regardless it is moving or not. If the current time minus the start time is greater or equal to the lifetime of the Drop object, then kill it.

- Class Rain

```
class Rain:
    def __init__(self, all_sprites):
        self.all_sprites = all_sprites
        #list of rain drops images
        self.rain_drops = import_folder('../graphics/rain/drops')
        #list of rain floor images
        self.rain_floor = import_folder('../graphics/rain/floor')

        #size of the map
        self.floor_width, self.floor_height = pygame.image.load('../graphics/world/ground.png').get_size()
```

When we create an object of the class Rain, it will have the attributes all_sprites, rain_drops (list of raindrop image surfaces), rain_floor (list of rainfloor image surfaces), and floor width and floor height (the width and height of the map).

```
    def create_floor(self):
        #create rain floors
        Drop(
            surf = choice(self.rain_floor),
            #the surface is randomized from the rain_floor surfaces
            pos = (randint(0, self.floor_width), randint(0, self.floor_height)),
            #the position is randomized based on the size of the map
            moving = False, #it is not moving because it is rain floor
            groups = self.all_sprites,
            z = LAYERS['rain floor'])
```

The create_floor method is used to create a Drop object with the surface being randomized between all the choices inside the rain floor list, the position is randomized between 0 until the width and height of our map, moving is False because it is rain floor, the group is all sprites, and the layer is rain floor.

```
    def create_drops(self):
        #create rain drops
        Drop(
            surf = choice(self.rain_drops),
            #the surface is randomized from the rain_drops surfaces
            pos = (randint(0, self.floor_width), randint(0, self.floor_height)),
            #the position is randomized based on the size of the map
            moving = True, #rain drops is moving from topleft
            groups = self.all_sprites,
            z = LAYERS['rain drops'])
```

The create_drops method is used to create a Drop object with the surface being randomized between all the choices inside the raindrops list, the position is

randomized between 0 until the width and height of our map, moving is True because raindrops move from top to bottom, group is all sprites, and the layer is rain drops.

```
    def update(self):
        #create rain floors and drops if it's raining
        self.create_floor()
        self.create_drops()
```

If it's raining, the updated method will create rain floors and raindrops by calling the create_floor and create_drops methods.

8. overlay_menu.py

This file contains the class for the overlay menus in this game, which is the Overlay_Menu class for the help menu and the Inventory class for the inventory.

```
import pygame
from game_settings import *
from support import import_folder
```

Import pygame, from game_settings import all constants, and from support import import_folder function.

- Class Overlay_Menu

```
class Overlay_Menu:
    def __init__(self, player, toggle_menu):
        #basic setup
        self.player = player
        self.display_surface = pygame.display.get_surface()
        #font
        self.font = pygame.font.Font('../font/Rubik-Bold.ttf', 35)
        self.font2 = pygame.font.Font('../font/Rubik-SemiBold.ttf', 25)

        #graphics
        self.width = 800
        self.height = 650

        #toggle
        self.toggle_menu = toggle_menu
        self.setup()
```

When initializing an object from this class, the object will have some attributes. The attributes are the player, the display surface that is obtained by using pygame, the font

(font is for the header and font2 is for the text), the width and height of the menu, and the toggle function. It will also call the setup method.

```
    def input(self):
        #check if key get pressed
        keys = pygame.key.get_pressed()
        #if player press O, toggle help menu
        if keys[pygame.K_o]:
            self.toggle_menu()
```

The input method will check if there is any key that gets pressed. If the help menu is active and the player presses the ‘O’ key, then toggle or close the help menu.

```
    def setup(self):
        self.text_surfs = []
        #for every key and its description, append it to text_surfs
        for key, desc in KEYBOARD_KEYS.items():
            text_surf = self.font2.render(f"{key} : {desc}", False, GAME_COLOR['text'])
            self.text_surfs.append(text_surf)
```

The setup method is used to initialize the text_surfs list that contains all the text surfaces that will be blitted on the screen later on. So, it will first initialize an empty list and, then for every key and its description in the keyboard shortcut dictionary, it will render font2 with the key and its description as the text surface. After that, the text_surfs list will append the text surface.

```
def show(self):
    #the main background for help menu
    self.menu_top = SCREEN_HEIGHT / 2 - self.height / 2
    self.main_rect = pygame.Rect(SCREEN_WIDTH/2 - self.width/2, self.menu_top, self.width, self.height)
    pygame.draw.rect(self.display_surface, GAME_COLOR['dialogue box'], self.main_rect)

    #the topic text for the help menu
    text_surf = self.font.render("KEYBOARD SHORTCUT", False, 'BLACK')
    text_rect = text_surf.get_rect(midtop = (self.main_rect.centerx, self.main_rect.top+10))
    self.display_surface.blit(text_surf, text_rect)

    space = 10
    for item_index, item_surf in enumerate(self.text_surfs):
        #blit the key and description text
        item_rect = item_surf.get_rect(topleft = (self.main_rect.left+40, self.main_rect.top + 60 + space))
        self.display_surface.blit(item_surf, item_rect)
        space += 35 #every key's description has about 35 px space from each other

        #under all the key and description, blit this sentence
        if item_index == len(self.text_surfs) - 1:
            instruction_surf = self.font2.render('Play the story to get more money and a decoration', False, GAME_COLOR['text'])
            instruction_rect = instruction_surf.get_rect(topleft = (self.main_rect.left+40, self.main_rect.top + 60 + space))
            self.display_surface.blit(instruction_surf, instruction_rect)
```

The show method is used to show the menu and the text. First off, it will get the top of the menu position and draw the main rect based on the positions. The main rect will be drawn on the display surface with the colour of light brown as the menu's background. Next, it will render the text "KEYBOARD SHORTCUT" by using self.font into a surface and then get the rect. After that, it will also blit it on the screen.

After that, for every item in the text_surfs list, blit the key and description text into the display surface. The y position of each text is affected by the space which is the space for each line of text. The space will be added by 35 every iteration so that every line has about 35px for the vertical space.

After all the keys and descriptions are blitted or the item is already the last item, blit the instruction surface on the lowermost by first rendering the text using font2 and getting the rect.

```
def update(self):
    #update the menu based on player's input
    self.input()
    self.show() #show the menu
```

The update method will update the help menu based on the player input (toggle) and also show the content of the menu.

- Class Inventory

```
class Inventory:
    def __init__(self, player, toggle_menu):
        #basic setup
        self.player = player
        self.display_surface = pygame.display.get_surface()
        #font
        self.font = pygame.font.Font('../font/LycheeSoda.ttf', 40)
        self.font_amount = pygame.font.Font('../font/LycheeSoda.ttf', 28)

        #graphics
        self.width = 480
        self.height = 480
        self.vertical_space = 80
        self.horizontal_space = 30
        self.padding = 10
```

```
        #toggle
        self.toggle_menu = toggle_menu

        #entries
        #list of the items
        self.list_items = list(self.player.item_inventory.keys()) \
            + list(self.player.seed_inventory.keys()) + list(self.player.other_inventory.keys())
        path = '../graphics/inventory'

        #list of the item images
        self.list_images = import_folder(f'{path}/item_inventory') + \
            import_folder(f'{path}/seed_inventory') + import_folder(f'{path}/other_inventory')
```

Class Inventory is used to handle the inventory in this game. The object of this class has several attributes: the display surface that can be obtained by using pygame, the player, the font (for header text), the font_amount (for amount text), the width, height, vertical space, and horizontal space, padding, toggle function, list of the items names, and list of the items image surfaces. The image surfaces list is generated by using the import_folder function based on the path given there.

```
    def input(self):
        #check if there are any keys got pressed
        keys = pygame.key.get_pressed()

        #if player press O, toggle inventory
        if keys[pygame.K_o]:
            self.toggle_menu()
```

The input method is used to check if the player presses any key or not. If inventory is active and the player presses the O key, then close the inventory.

```
def blit_image(self, row, column):
    #the formula for counting the vertical space and horizontal space for each item
    #each item vertical's and horizontal space depends on what row and column that they are in
    vert_space = self.vertical_space + (1.8 * row * self.vertical_space)
    horizontal_space = self.padding + self.horizontal_space + (5 * column * self.horizontal_space)

    #accesing the image surf and get the image rect
    image_surf = self.list_images[(row*3)+column]
    image_rect = image_surf.get_rect(topleft = (self.main_rect.left+ horizontal_space, self.main_rect.top + vert_space))
    #draw a darker brown rect as the background of the item's image
    pygame.draw.rect(self.display_surface, GAME_COLOR['dialogue box'], image_rect)
    self.display_surface.blit(image_surf, image_rect)
```

The blit_image will first calculate the vertical space and horizontal space of the item based on which row and column it is in. After that, it will get the image surface from the list_images. The index will be row times 3 added by column. The reason is that every row has a maximum of three items. After that, it will get the rect, blit the image, and draw a dark brown rect as the item's image background.

```
def show(self):
    #background for the inventory
    self.menu_top = SCREEN_HEIGHT / 2 - self.height / 2
    self.main_rect = pygame.Rect(SCREEN_WIDTH/2 - self.width/2, self.menu_top, self.width, self.height)
    pygame.draw.rect(self.display_surface, GAME_COLOR['inventory'], self.main_rect)

    #text header
    text_surf = self.font.render("INVENTORY", False, 'BLACK')
    text_rect = text_surf.get_rect(midtop = (self.main_rect.centerx, self.main_rect.top+ self.padding))
    self.display_surface.blit(text_surf, text_rect)

    #money
    text_surf = self.font.render(f'Money: ${self.player.money}', False, 'Black')
    text_rect = text_surf.get_rect(midtop = (self.main_rect.centerx, self.main_rect.top - (5*self.padding)))
    #the white background for the money text
    pygame.draw.rect(self.display_surface, 'White', text_rect.inflate(10,10), 0, 5)
    self.display_surface.blit(text_surf, text_rect)

    #blit all image on their respective position
    for row in range (0, (len(self.list_images) // 3)):
        for column in range (0, 3):
            self.blit_image(row, column)
    #blit the last image because it is not included in the for loop
    self.blit_image(2, 0)
```

The show method is used to show the content of the inventory. It will first calculate the top of the menu and draw the main rect (the background of the menu) on the screen. After that, it will render the text header, get its rect, and blit it on the screen. Then, it will also render the player's money text above the menu and blit it. The money text also has a white surface as the background. The white surface is generated by using the rect of the text and it is inflated.

It will also blit all the images on their respective row and column. For every row in range 2 (because 7//3=2), for columns in range (0, 3), blit image on their row and column. The reason why the column is in the range 0 until 3 is because 1 row has a maximum of 3 items. So, this code has a little flaw, where it will only blit the first 6 items. So, we have to blit the seventh item after that.

```
def show_amount(self, row, column, text, amount):
    #the amount text vertical and horizontal space also depens on the row and column which they are in
    vert_space = 60 + self.vertical_space + (1.8 * row * self.vertical_space)
    horizontal_space = self.padding + self.horizontal_space + (5 * column * self.horizontal_space)

    #generate and blit the amount text
    amount_surf = self.font_amount.render(f'{text} : {amount}', False, 'Black')
    amount_rect = amount_surf.get_rect(topleft = (self.main_rect.left+ horizontal_space, self.main_rect.top + vert_space))
    self.display_surface.blit(amount_surf, amount_rect)
```

The show_amount method is used to show the amount of the item. It will first calculate the vertical and horizontal space too based on the row and column in which

the item is in. After that, it will render the item's name and its amount on the surface, get its rect, and blit it on the display surface.

```
def update(self):
    self.input() #update based on player's input
    self.show() #show inventory

    #showing amount based on the most updated version of player's inventory
    amount_list = list(self.player.item_inventory.values()) + \
        list(self.player.seed_inventory.values()) + list(self.player.other_inventory.values())
    for row in range (0, (len(self.list_items) // 3)):
        for column in range (0, 3):
            amount = amount_list[(row*3) + column]
            text = self.list_items[(row*3) + column]
            self.show_amount(row, column, text, amount)

    #show amount for grass because it is not included in the for loop
    self.show_amount(2, 0, self.list_items[6], amount_list[6])
```

The update method is used to update the menu based on the player input and show the content of the menu. The amount of the item in the inventory is subject to change, so it has to be included in the update method too. So, first, it will periodically update the amount_list based on the values of the player's inventory. After that, in a similar way to the blit image, it will blit the first 6 items' amounts on the inventory menu. So, we have to type manually for the seventh item which is the grass because it is not included yet in the loop.

9. merchant_menu.py

This file contains the Menu class to handle the shop menu in this game.

```
import pygame
from game_settings import *
from support import Timer
```

Import pygame, from game_settings, import all constants, and from support import Timer.

- Class Menu

```
class Menu:
    def __init__(self, player, toggle_menu):
        #basic setup
        self.player = player
        self.toggle_menu = toggle_menu
        self.display_surface = pygame.display.get_surface()
        self.font = pygame.font.Font('../font/LycheeSoda.ttf', 30)

        #graphics
        self.width = 400
        self.space = 10
        self.padding = 8

        #entries
        self.options = list(self.player.item_inventory.keys()) + \
            list(self.player.seed_inventory.keys()) + list(self.player.other_inventory.keys())
        self.sell_border = len(self.player.item_inventory) - 1 #to determine which item can be bought and can be sold
        self.setup()

        #movement
        self.index = 0 #starts from the top one
        self.timer = Timer(200) #used to normalize movement

        #sound
        #when player sells something
        self.sell_sound = pygame.mixer.Sound('../audio/sell.mp3')
        self.sell_sound.set_volume(0.6)
        #when player buys something
        self.buy_sound = pygame.mixer.Sound('../audio/buy.mp3')
        self.buy_sound.set_volume(0.6)
```

When initializing an object from this class, it will have several attributes from the player, toggle function, the display surface (obtained using pygame), font, width, space, padding, options (the keys or the name of the items in the player's inventory), sell_border (to determine which item can be bought and which item can be sold), the index, timer, sell_sound, and buy_sound. The index and the timer are used for movement when navigating to buy or sell an item. By default, the player will start on index 0 or the top of all of the entries. The timer is used to normalize the navigation so that when we press the arrow button, it will not loop that fast. The sell sound is played when the player sells something and the buy sound is when the player buys something. Here, we also call the setup function.

```
def display_money(self):
    #render font and get its rect to display the money text
    text_surf = self.font.render(f'Money: ${self.player.money}', False, 'Black')
    text_rect = text_surf.get_rect(midbottom= (SCREEN_WIDTH/2, SCREEN_HEIGHT -600))
    #the background of the text is white
    pygame.draw.rect(self.display_surface, 'White', text_rect.inflate(10,10), 0, 5)
    self.display_surface.blit(text_surf, text_rect)
```

The display_money method is used to display the player's money. Same as the inventory, it will first render the money text into a surface, get its rect and blit it on the screen. It will also draw a white rect to become the background of the money text. The rect is obtained by inflating the text rect.

```
def setup(self):
    #all prices of the item regardless it's sale prices or purchase prices
    ALL_PRICES = []
    for price in SALE_PRICES.values():
        ALL_PRICES.append(price)
    for price in PURCHASE_PRICES.values():
        ALL_PRICES.append(price)

    #create the text surfaces
    self.text_surfs = []
    self.total_height = 0
    #for every item
    for item_index, item in enumerate(self.options):
        #render the name and the price of each item, append it to the text_surfs list
        text_surf = self.font.render(F'{item} (${ALL_PRICES[item_index]})', False, 'Black')
        self.text_surfs.append(text_surf)
        #calculate the total height of the menu
        self.total_height += text_surf.get_height() + (self.padding * 2)
```

The setup method will first append all the sale and purchase prices in a list. After that, it will initialize an empty list for the text surfaces. It will also initialize the total height to 0. The item index is used to help assess the price of each item based on its index. So, it will render the name and the price of each item and append it to the text surfs list, then it will calculate the total height based on the text surfaces' height.

```
    #calculate the total height of the menu
    self.total_height += (len(self.text_surfs) - 1) * self.space
    #set the position of the top of the menu
    self.menu_top = SCREEN_HEIGHT / 2 - self.total_height / 2
    #the menu rect
    self.main_rect = pygame.Rect(SCREEN_WIDTH/2 - self.width/2, self.menu_top, self.width, self.total_height)

    #buy and sell text surface
    self.buy_text = self.font.render('buy', False, 'darkred')
    self.sell_text = self.font.render('sell', False, 'chartreuse3')
```

Then, it will calculate the total height by adding it to the length of the text surfaces multiplied by the space. The reason is that the shop menu, unlike the inventory and help menu, doesn't have a big rect as the background image, but instead, every entry has its own rect. It will also calculate the top of the menu and generate a main rect (but doesn't blit it). There will be also the text 'buy' and 'sell' in the shop menu for

every entry. The ‘buy’ text will be in dark red and the ‘sell’ text colour will be in green.

```
def input(self):
    #check if key is pressed
    keys = pygame.key.get_pressed()
    self.timer.update()

    #if player press o, then toggle shop
    if keys[pygame.K_o]:
        self.toggle_menu()

    #if the timer for the movement is not active
    if not self.timer.active:
        #if player press arrow up, then go to the item above
        if keys[pygame.K_UP]:
            self.index -= 1
            self.timer.activate() #activate timer
        #if player press arrow down, then go to the item below
        if keys[pygame.K_DOWN]:
            self.index += 1
            self.timer.activate() #activate timer
```

The input method will first check if there are any pressed keys and update the timer. If the player presses O, then close the shop menu. If the timer movement is not active, the player can navigate up or down. If they start to navigate up or down, the timer will activate. If they navigate up, then the index will be minus by 1, else if they navigate down, then the index will be added by 1.

```
    if keys[pygame.K_SPACE]:
        self.timer.activate()
        #get item
        current_item = self.options[self.index]

        #sell item if the item is greater than 0
        if self.index <= self.sell_border:
            if self.player.item_inventory[current_item] > 0:
                self.sell_sound.play()
                #deduct player's item amount and add player money
                self.player.item_inventory[current_item] -= 1
                self.player.money += SALE_PRICES[current_item]
```

If the player presses space, then activate the timer too and get the current item based on the index (the position player is right now). If the index is lesser than the sell bolder, then it means that it's still in the item_inventory. The item in the item_inventory can be sold. If the player's current item is greater than 0, then the player can sell the item. Sell sound will be played, the item will be deducted from the player's inventory, and the player's money will be added based on the sale price of the current item.

```
        #buy item if the money is enough
        else:
            if self.player.money >= PURCHASE_PRICES[current_item]:
                self.buy_sound.play()
                #if it's a grass, then add grass item by 1
                if current_item == 'grass':
                    self.player.other_inventory[current_item] += 1
                #else, add seed amount based on the plant type
                else:
                    self.player.seed_inventory[current_item] += 1
                #deducts player money
                self.player.money -= PURCHASE_PRICES[current_item]
```

Otherwise, if the current item is not in the item inventory, it means that it can be bought. If the player's money is greater or equal to the purchase price of the current item, then play the buy sound and deduct the player's money based on the purchase price of the current item. If the current item is grass, then add it to the other inventory because grass is in the other inventory. Otherwise, it will be added to the seed inventory.

```
#if player press the up arrow key while in the first entry, then the
#active entry goes to the last entry
if self.index < 0:
    self.index = len(self.options) - 1
#if player press the down arrow key while in the last entry, then the
#active entry goes to the first entry
if self.index > len(self.options)-1:
    self.index = 0
```

If the player presses the arrow key up when they are in the first entry (index becomes lesser than 0), then go to the last entry (index will be the length of the list minus 1). If the player presses the down arrow key although they are in the last entry (index becomes greater than the length of the list minus 1), then set the index back to 0 (go to the first entry).

```
def show_entries(self, text_surf, amount, top, selected):
    #background surface (white) for each entry
    bg_height = text_surf.get_height() + (self.padding * 2)
    bg_rect = pygame.Rect(self.main_rect.left, top, self.width, bg_height)
    pygame.draw.rect(self.display_surface, 'White', bg_rect, 0, 5)

    #text surface
    text_rect = text_surf.get_rect(midleft = (self.main_rect.left + 20, bg_rect.centery))
    self.display_surface.blit(text_surf, text_rect)

    #amount surface
    amount_surf = self.font.render(str(amount), False, 'Black')
    amount_rect = amount_surf.get_rect(midright = (self.main_rect.right - 20, bg_rect.centery))
    self.display_surface.blit(amount_surf, amount_rect)

    #selected
    if selected:
        pygame.draw.rect(self.display_surface, 'black', bg_rect, 4, 4)
        if self.index <= self.sell_border: #sell item
            pos_rect = self.sell_text.get_rect(midleft = (self.main_rect.left + 200, bg_rect.centery))
            self.display_surface.blit(self.sell_text, pos_rect) #display 'sell' text
        else: #buy item
            pos_rect = self.buy_text.get_rect(midleft = (self.main_rect.left + 200, bg_rect.centery))
            self.display_surface.blit(self.buy_text, pos_rect) #display 'buy' text
```

The show_entries method will first get the background height for each entry, get their rects, and draw them with white colour on the display surface. After that, it will also blit the text surface and text rect to the display screen (the name and price of the item). Then, it will also blit the amount surface and amount rect. If an item is selected, then draw a black rect as the border for the selected entry. If the index is lesser than the sell border, then get the position rects and display ‘sell’ text. Otherwise, display ‘buy’ text.

```
def update(self):
    self.input() #update based on player's input
    self.display_money() #display money

    #for every text index and text surface in text surfs
    for text_index, text_surf in enumerate(self.text_surfs):
        #top of the background rect
        top = self.main_rect.top + text_index * (text_surf.get_height() + (self.padding *2) + self.space)
        #the amount of each item
        amount_list = list(self.player.item_inventory.values()) + \
            list(self.player.seed_inventory.values()) + list(self.player.other_inventory.values())
        amount = amount_list[text_index]
        self.show_entries(text_surf, amount, top, self.index == text_index)
```

The update method is used to update the shop menu based on player input. It will also update the money by calling the display money method. The amount of every item also needs to be updated, so for every text index and text surface in the text surfs list, it will first get the top of the background rect for each entry and then from the amount list, it will get the amount of the corresponding item based on the text index. After that, it will call the show_entries method based on the up-to-date numbers.

10. soil.py

This file contains the class Plant, SoilTile, WaterTile, and SoilLayer. This file mainly handles the soil and plant systems in this game.

```
import pygame
from game_settings import *
from pytmx.util_pygame import load_pygame
from support import import_folder, import_folder_dict
from random import choice
```

Import pygame, from game_settings, import all constants, from pytmx import load_pygame, from support import import_folder and import_folder_dict functions, and from random import choice.

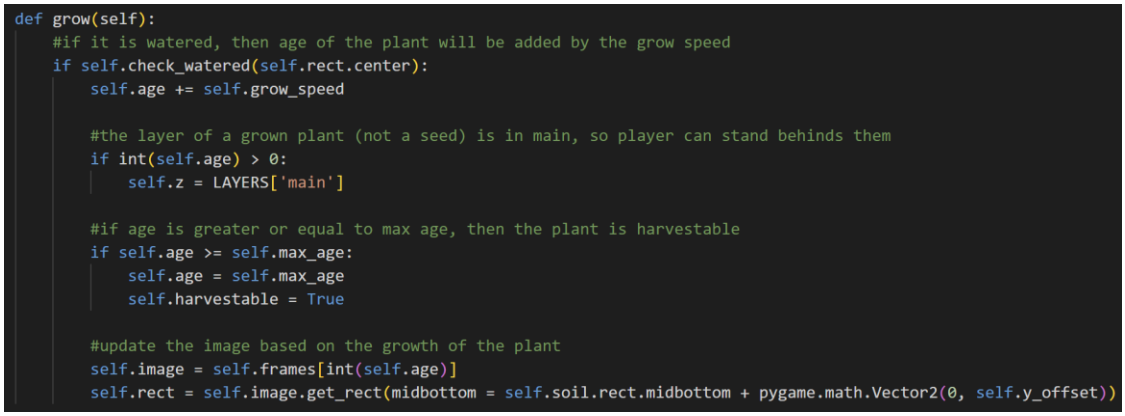
- Class Plant

```
class Plant(pygame.sprite.Sprite):
    def __init__(self, plant_type, groups, soil, check_watered):
        #the basic setup
        super().__init__(groups)
        self.plant_type = plant_type
        self.frames = import_folder(f'../graphics/fruit/{plant_type}') #list of surfaces based on the plant type
        self.soil = soil
        self.check_watered = check_watered

        #the plant growth
        self.age = 0
        self.max_age = len(self.frames) - 1
        self.grow_speed = GROW_SPEED[plant_type]
        self.harvestable = False #plant's is not harvestable by default

        #animation
        self.image = self.frames[self.age] #the image depends on the age of the plant
        self.y_offset = -16 if plant_type == 'corn' else -8 #y offset of the plant
        self.rect = self.image.get_rect(midbottom = soil.rect.midbottom + pygame.math.Vector2(0, self.y_offset))
        self.z = LAYERS['house bottom']
```

The Plant class is inherited from the Sprite class in pygame. There are several attributes upon creating a plant object, which is the plant_type (in this game, there are two, which are corn and tomato), frames (list of surfaces based on the plant type), the soil, the check_watered function, the age of the plant (initially 0), the max_age (the

length of the frame minus 1), the grow_speed (depends on the plant type), the harvestable status (initially False), the image (that depends on the frames and the age), the y offset of the plant, the rect of the plant (depends on the image), and the layer.

Initially, the layer of the plant is the house bottom (equivalent to carpet) because it is still a seed, so the player can step on top of it. The y offset of each plant type is also different. The y offset helps to determine the rect of a plant because each plant has different shapes.



```
def grow(self):
    #if it is watered, then age of the plant will be added by the grow speed
    if self.check_watered(self.rect.center):
        self.age += self.grow_speed

        #the layer of a grown plant (not a seed) is in main, so player can stand behinds them
        if int(self.age) > 0:
            self.z = LAYERS['main']

        #if age is greater or equal to max age, then the plant is harvestable
        if self.age >= self.max_age:
            self.age = self.max_age
            self.harvestable = True

        #update the image based on the growth of the plant
        self.image = self.frames[int(self.age)]
        self.rect = self.image.get_rect(midbottom = self.soil.rect.midbottom + pygame.math.Vector2(0, self.y_offset))
```

The grow method is used to grow plants. If it is watered, then the age of the plant will be added to the growth speed. If the plant is not a seed anymore, the layer will also become ‘main’ because they are not seeds or sprouts anymore, so they have quite a distinguishable height. If the age of the plant is greater or equal to the max-age, then set the age as the max age and the plant is harvestable. Based on the growth of the plant, this method will also update the plant’s image surface based on the age of the plant and gets it rect again.

- Class SoilTile

```
class SoilTile(pygame.sprite.Sprite):
    def __init__(self, pos, surf, groups):
        super().__init__(groups)
        #image, rect, and layer for soiltile
        self.image = surf
        self.rect = self.image.get_rect(topleft = pos)
        self.z = LAYERS['soil']
```

The SoilTile class is also inherited from the Sprite class in pygame. Objects of this class have the attributes images, rect, and the layer (z) is in the soil.

- Class WaterTile

```
class WaterTile(pygame.sprite.Sprite):
    def __init__(self, pos, surf, groups):
        super().__init__(groups)
        #image, rect, and layer for water on top of the soil tile
        self.image = surf
        self.rect = self.image.get_rect(topleft = pos)
        self.z = LAYERS['soil water']
```

The WaterTile class is also inherited from the Sprite class in pygame. Objects of this class have the attributes images, rect, and the layer (z) is in the soil water. It is higher in the layer than SoilTile because objects from WaterTile is actually the water tiles on top of the soil tile when the player waters the soil.

- Class SoilLayer

```
class SoilLayer:
    def __init__(self, all_sprites, collision_sprites):
        #sprite groups
        self.all_sprites = all_sprites
        self.soil_sprites = pygame.sprite.Group()
        self.water_soil_sprites = pygame.sprite.Group()
        self.plant_sprites = pygame.sprite.Group()
        self.collision_sprites = collision_sprites

        #graphics
        self.soil_surfs = import_folder_dict('../graphics/soil')
        self.water_soil_surfs = import_folder('../graphics/soil_water')

        #create tiles that can be tiled as soil
        self.create_soil_grid()
        self.create_hit_rects()

        #sound
        #hoe sound for tilling soils
        self.hoe_sound = pygame.mixer.Sound('../audio/hoe.mp3')
        self.hoe_sound.set_volume(0.5)
        #plant sound for planting seeds
        self.plant_sound = pygame.mixer.Sound('../audio/plant.mp3')
```

The class SoilLayer is used to maintain all the soil tiles in this game. Upon initializing an object of this class, it will have several attributes. The sprite groups are the all sprites, the soil sprites (which is a sprite Group), the water soil sprites (which is a sprite Group), the plant sprites (which is a sprite Group), and collision sprites. Next, it will also import the dictionary of the surface's name and the surface for the soil surfaces and import a list of the water soil surfaces. It will also call the create soil grid and create hit rects method which will be explained below. Lastly, it will also have the sound attributes for hoe and plant. The hoe sound is played when tilling soils and it also sets the volume to 0.5, while the plant sound is played when the player plants seed.

```
    def create_soil_grid(self):
        #load the ground image and get the size of it
        ground = pygame.image.load('../graphics/world/ground.png')
        h_tiles = ground.get_width() // TILE_SIZE
        v_tiles = ground.get_height() // TILE_SIZE

        #list for each individual cell (tile) inside the col in horizontal tiles and row in vertical tiles
        self.grid = [[[]for col in range(h_tiles)] for row in range(v_tiles)]

        #if the cell is farmable, then the list for that cell will contain a F
        for x, y, _ in load_pygame('../data/map.tmx').get_layer_by_name('Farmable').tiles():
            self.grid[y][x].append('F')
```

The create_soil_grid method will first load the ground image and get the size of it. It will calculate how many horizontal tiles and vertical tiles the ground has. It will also

initialize the grid attribute which is an empty list for each individual tile or cell. Then, it will load the map.tmx and check the tiles in the Farmable layer. If it's a tile in the farmable layer, then the corresponding list for that tile or cell will contain an 'F' which means it is farmable.

```
def create_hit_rects(self):
    self.hit_rects = []
    #if the cell contains F, then create a rect based on it and append it as hit rect
    for index_row, row in enumerate(self.grid):
        for index_col, cell in enumerate(row):
            if 'F' in cell:
                x = index_col * TILE_SIZE
                y = index_row * TILE_SIZE
                rect = pygame.Rect(x, y, TILE_SIZE, TILE_SIZE)
                self.hit_rects.append(rect)
```

The create_hit_rects method is used to create a rect that can be hit or tilled. It will first initialize an empty list of hit rects. Now, for every cell in the grid, if it is farmable, then it will get the x and y positions from this cell and create a rect. The rect will be appended to the hit rects list.

```
def get_hit(self, point):
    #for every rect in hit_rects
    for rect in self.hit_rects:
        #if rect collides with player's hoe target position
        if rect.collidepoint(point):
            self.hoe_sound.play()
            x = rect.x // TILE_SIZE
            y = rect.y // TILE_SIZE

            #if it's farmable and it's tilled, then the tile wiill contain X
            #and create a soil tile
            if 'F' in self.grid[y][x]:
                self.grid[y][x].append('X')
                self.create_soil_tiles()

                #if it's raining, then tiles that contain X will be watered
                #and there is water tile above them
                if self.raining:
                    self.water_all()
```

This method is called when a rect is hit or tilled by the player. So, it will check for every rect in the hit rects list, if there is a rect that collides with the player hoe's target position, then play the hoe sound. It will also get the index column and index row for the cell so that in the grid attribute, the corresponding cell with the index column and

the index row will have a ‘X’ beside the ‘F’. It will also called the create_soil_tiles method to create the soil tiles. Then, if it’s raining, all tiles containing X will be watered (a water tile will be blitted above them).

```
def create_soil_tiles(self):
    #draw all the soil tiles together, so that when a soil is
    #tilled next to a tilled tile, then the tiles will be connected
    self.soil_sprites.empty() #empty the soil sprites, so we can create it all together
    for index_row, row in enumerate(self.grid):
        for index_col, cell in enumerate(row):
            #if the soil is plantable or already tilled
            if 'X' in cell:
                #tile options (are there any tilled soil on every positions of the tile)
                top = 'X' in self.grid[index_row -1][index_col]
                bottom = 'X' in self.grid[index_row + 1][index_col]
                right = 'X' in row[index_col + 1]
                left = 'X' in row[index_col - 1]

                tile_type = 'o' #original type of tile if it's not connected to other tiles

                #if the tile is connected to other tiles, then change the tile image
                #connected to all sides
                if all((top, bottom, right, left)): tile_type = 'x'
```

The create_soil_tiles method is used to create and redraw all the soil tiles if a farmable tile is tilled. So, first, it will empty the soil sprites, so that it can create the soil tiles together. Then, for every cell, if the cell is already tilled (contains ‘X’), then check if there are any tilled cells too on the top, bottom, right, and left of this cell. If no, then the original type (o) will be drawn. However, if the fourth of them is already tilled, then change the tile image to the one that connects with all sides. There are also several more cases for this.

```
#connected to horizontal sides
if left and not any ((top, bottom, right)): tile_type = 'r' #left only
if right and not any ((top, bottom, left)): tile_type = 'l' #right only
if left and right and not any ((top, bottom)): tile_type = 'lr' #left and right

#connected to vertical sides
if top and not any ((right, bottom, left)): tile_type = 'b' #top only
if bottom and not any ((top, right, left)): tile_type = 't' #bottom only
if top and bottom and not any ((left, right)): tile_type = 'tb' #top and bottom

#connected to corners sides (L shape)
if left and bottom and not any ((top, right)): tile_type = 'tr' #bottom and left
if right and bottom and not any ((top, left)): tile_type = 'tl' #right and bottom
if left and top and not any ((bottom, right)): tile_type = 'br' #left and top
if right and top and not any ((bottom, left)): tile_type = 'bl' #right and top

#connected to three sides/T shapes
if all((top, bottom, right)) and not left: tile_type = 'tbr' #top, bottom, right
if all((top, bottom, left)) and not right: tile_type = 'tbl' #top, bottom, left
if all((top, left, right)) and not bottom: tile_type = 'lrb' #top, left, right
if all((left, bottom, right)) and not top: tile_type = 'lrt' #left, bottom, right

#create the soiltile
SoilTile(
    pos = (index_col * TILE_SIZE, index_row * TILE_SIZE),
    surf = self.soil_surfs[tile_type],
    groups = [self.all_sprites, self.soil_sprites])
```

The first case is to check whether the horizontal sides of this cell is tilled or not. It will check if it's connected to left only, right only, or only left and right. Similarly, it will also check for the vertical sides, whether it's connected to the top only, bottom only, or only top and bottom.

Besides that, there are also the L shape cases (bottom and left, bottom and right, top and left, top and right) and the T shape cases (top and bottom and right, top and bottom and left, top and left and right, bottom and left and right). Every possible case for the soil has a different surface, so after knowing what is their type, it will be able to create a suitable soil tile based on their type (or connection with other tiles).

```
def water(self, point):
    #for every soil tile
    for soil_sprite in self.soil_sprites.sprites():
        #if soil tile collides with water can's target pos
        if soil_sprite.rect.collidepoint(point):
            #the soil is watered, then append 'W' in the list
            x = soil_sprite.rect.x // TILE_SIZE
            y = soil_sprite.rect.y // TILE_SIZE
            self.grid[y][x].append('W')
            soil_pos = soil_sprite.rect.topleft

            #create a water tile above the soil tile
            WaterTile(
                pos = soil_pos,
                surf = choice(self.water_soil_surfs),
                groups = [self.all_sprites, self.water_soil_sprites]
            )
```

The water method is used to create a water tile above the soil tile. First, it will check for all the soil tiles inside the soil sprites. If the soil tile rect collides with the water can's target position, then append 'W' on the list that the cell has. After that, it will create the water tile based on the position of the soil tile and the surface is randomized between the surfaces that are available in the water soil surfaces.

```
def water_all(self):
    #for every tile
    for index_row, row in enumerate(self.grid):
        for index_col, cell in enumerate(row):
            #if the tile is plantable, but is not watered, then water it
            if 'X' in cell and 'W' not in cell:
                cell.append('W')

                #create watertile above the soiltile
                WaterTile(
                    pos = (index_col * TILE_SIZE, index_row * TILE_SIZE),
                    surf = choice(self.water_soil_surfs),
                    groups = [self.all_sprites, self.water_soil_sprites]
                )
```

The water_all method is called when it is raining. It basically checks for every cell in the grid attribute. If it contains 'X' (it is tilled) and is not watered yet (does not contain 'W'), then append 'W' in the list that the cell has and create a water tile above the soil tile.

```
def remove_water(self):
    #destroy all the water sprites
    for sprite in self.water_soil_sprites.sprites():
        sprite.kill()

    #empty the cell from W if there is W
    for row in self.grid:
        for cell in row:
            if 'W' in cell:
                cell.remove('W')
```

The remove_water method is called when the player resets the day. It will destroy all the water tiles inside the water soil sprites so that there will be no soil tiles available. After that, it will also remove the ‘W’ from the list that the cell has.

```
def check_watered(self, pos):
    #if W is in the cell, then return True
    x = pos[0] // TILE_SIZE
    y = pos[1] // TILE_SIZE
    cell = self.grid[y][x]
    is_watered = 'W' in cell
    return is_watered
```

The check_watered function is used to check if the cell is watered or not. The algorithm is easy, the function only needs to check whether there is a ‘W’ in the cell or not. If yes, then return True, else, return False.

```
def plant_seed(self, target_pos, seed):
    #if the player plant seeds in the soil tile, then the soil tile will contains the code P
    #means that the soil is planted
    for soil_sprite in self.soil_sprites.sprites():
        #if soil tile collides with the target position
        if soil_sprite.rect.collidepoint(target_pos):
            self.plant_sound.play()
            x = soil_sprite.rect.x // TILE_SIZE
            y = soil_sprite.rect.y // TILE_SIZE

            #create plant object only if the soil is not planted before
            if 'P' not in self.grid[y][x]:
                self.grid[y][x].append('P')
                Plant(
                    plant_type = seed,
                    groups = [self.all_sprites, self.plant_sprites, self.collision_sprites],
                    soil = soil_sprite,
                    check_watered = self.check_watered
                )
```

The plant_seed method is used to create a plant object when the player plants a seed. So, for all soil sprites (that are plantable or contain ‘X’), if the soil sprite collides with

the player's use seed's target position and there is still no plant in the soil tile, then create a plant object. The cell will now also contain 'P' which means that it now has a plant on top of it.

```
def check_plantable(self, target_pos):
    for soil_sprite in self.soil_sprites.sprites():
        if soil_sprite.rect.collidepoint(target_pos):
            x = soil_sprite.rect.x // TILE_SIZE
            y = soil_sprite.rect.y // TILE_SIZE

            #if plantable and still no plant then return true and the seed can be planted
            #if not then, the seed cannot be planted and it doesnt deduct the seed amount
            if 'X' in self.grid[y][x] and 'P' not in self.grid[y][x]: return True

    return False
```

The check_plantable method is used to check if the soil can be planted or not. For every soil sprite in soil sprites, if the target position collides with the soil sprite, then if the soil is tilled, but there is no plant, returns True, else returns False. This method will help the system to control the seed amounts that the player has, so that if the player uses a seed in a plantable, but there is already a plant on top of the soil, the seed amount will not be deducted.

```
def update_plants(self):
    #grow every plant
    for plant in self.plant_sprites.sprites():
        plant.grow()
```

The update_plants method is used to update the plant every day. It will grow the plant if the player resets the day.

11. player.py

This file contains the setup for the player in this game.

```
import pygame, sys
from game_settings import *
from support import *
from sprites import Generic
from cutscene import *
```

Import pygame and sys module, from game_settings, support, and cutscene import everything, and from sprites import Generic.

- Class Player

```
class Player(pygame.sprite.Sprite):
    def __init__(self, pos, group, collision_sprites, all_sprites, tree_sprites,
                 interaction, soil_layer, toggle_shop, toggle_help, toggle_inventory, tmx_data_map, cutscene):
        super().__init__(group)
        self.import_assets() #import player's image assets

        #default image for the player
        self.animation_status = 'down_idle'
        self.frame_index = 0

        #general configuration, getting surface based on the animation status and index
        #after that generate a rect based on the surface with the layer on the main
        self.image = self.animations[self.animation_status][self.frame_index]
        self.rect = self.image.get_rect(center = pos)
        self.z = LAYERS['main']

        #movement setup
        self.direction = pygame.math.Vector2() #direction of the player
        self.pos = pygame.math.Vector2(self.rect.center) #position of the player
        self.speed = 200 #player's walking speed

        # collision
        self.hitbox = self.rect.copy().inflate((-126,-70))
        self.collision_sprites = collision_sprites

        #timers
        self.timers = {
            'tool use': Timer(350,self.use_tool), #for using tool
            'tool switch': Timer(200), #for switching tool
            'seed plant': Timer(350,self.use_seed), #for using seed
            'seed switch': Timer(200), #for switching seed
        }
```

The object of the Player class has many attributes. First, it will call the import_assets method to import all the image surfaces of the player, then by default the animation of the player is on down idle and starts from the first frame (frame index 0). Next is the general setup for the player's default animation. It will update the player's image surface based on the animation status and the frame index of the player. After that, it will generate the rect and blit this surface on the 'main' layer.

Next is the movement setup. The direction and the position of the player are 2-dimensional vectors and then there is also the player's walking speed. For collision setup, the player will also have a hitbox that is copied from the player's rect and it is inflated and the collision sprites. There are also timers for the player when using tools, switching tools, planting seeds, and switching seeds.

```
#tools used
self.tools = ['hoe','axe','water'] #tools that can be used
self.tool_index = 0 #default tool will be hoe
self.tool_status = self.tools[self.tool_index] #tool status

#seeds available
self.seeds = ['corn', 'tomato'] #seeds that can be used
self.seed_index = 0 #default seed will be corn
self.seed_status = self.seeds[self.seed_index] #seed status

#player's inventory by default will be 0
self.item_inventory = {
    'corn':   0,
    'milk': 0,
    'tomato': 0,
    'wood':   0
}

#player's seeds and grass by default will be 5
self.seed_inventory = {
    'corn': 5,
    'tomato' : 5
}
self.other_inventory = {
    'grass': 5
}
```

```
#player's money by default
self.money = 0

#sprites
self.all_sprites = all_sprites #all object that needs to be updated
self.tree_sprites = tree_sprites #all trees
self.grass_sprites = pygame.sprite.Group() #all grass for the cows
self.interaction = interaction #interaction sprites
```

Next, there is also the list of the tools that the player can use, the tool status right now, and the tool index (by default 0 or the first tool (hoe) will initially be active). Same as the tool, there is also the seed list, seed status, and the seed index. Next, there are the player's inventories, starting from item inventory (items that can be sold), seed inventory (plants' seeds), and other inventory (grass). Seed inventory and other

inventory items can be bought from the merchant and initially, the player has 5 of them. The player's money will be 0 by default.

Below them are the sprite groups in this game, from all sprites, tree sprites (for tree objects), grass sprites (for grass for the cow), and interaction sprites to determine what should the system do when the player collides with one of the interaction sprites.

```
self.sleep = False #player by default is awake
self.soil_layer = soil_layer #soil layer
self.toggle_shop = toggle_shop #to toggle shop
self.toggle_help = toggle_help #to toggle help
self.toggle_inventory = toggle_inventory #to toggle inventory

#sound
#water sound is for water can
self.water_sound = pygame.mixer.Sound('../audio/water.mp3')
self.water_sound.set_volume(0.3)

#giving grass to cows
self.grass_sound = pygame.mixer.Sound('../audio/grass.mp3')
self.grass_sound.set_volume(0.4)

#collecting milk from cows
self.milk_sound = pygame.mixer.Sound('../audio/milk.mp3')
self.milk_sound.set_volume(0.5)
```

Next up is the player's sleep status which by default is False because the player is awake, and then the soil layer, and the toggle functions of each menu (shop, help, and inventory).

After that, there are also sound attributes which are the water sound (played when using a water can), the grass sound (played when giving grass to cows), and the milk sound (played when collecting milk). It also has set the volume.

```
    #button
    self.display_surface = pygame.display.get_surface() #display surface

    #help button image
    self.help_image = pygame.image.load('../graphics/button/help.png')
    self.help_image = pygame.transform.rotozoom(self.help_image, 0, 3) #scale

    #inventory button image
    self.inventory_image = pygame.image.load('../graphics/button/inventory.png')
    self.inventory_image = pygame.transform.rotozoom(self.inventory_image, 0, 4) #scale
```

After that, there is also the button configuration. There is the display surface for the button to be displayed, the button image (help and inventory) and it is scaled to achieve the right size for the button.

```
        #animal
        self.food = False #cow by default is hungry
        self.cow_max_age = 5 #cow's max age
        self.cow_age = 0 #cow's age is 0 by default
        self.harvest_milk = False #milk can't be harvestable yet

        self.tmx_data = tmx_data_map #game's map

        #cutscene
        self.cut_scene_manager = cutscene
```

Next, there is the animal setup. The cow by default will be hungry (the food is not given yet). There is also the max-age of the cow, the cow's age right now, and the status of the milk whether it's harvestable or not. There is also the game's map data and the cutscene manager.

```
def import_assets(self):
    #importing assets for player animation
    #initialize a dictionary
    self.animations = {'up': [],'down': [],'left': [],'right': [],
                       'right_idle':[],'left_idle':[],'up_idle':[],'down_idle':[],
                       'right_hoe':[],'left_hoe':[],'up_hoe':[],'down_hoe':[],
                       'right_axe':[],'left_axe':[],'up_axe':[],'down_axe':[],
                       'right_water':[],'left_water':[],'up_water':[],'down_water':[]}

    #getting the full path for the folder and then append the image surface to corresponding keys
    for animation in self.animations.keys():
        full_path = '../graphics/character/' + animation
        self.animations[animation] = import_folder(full_path)
```

The import_assets method is used to import player image surfaces for each condition of the player. First, it will generate a full path for each type of animation that the player has and will return a list of surfaces based on the animation status.

```
def animate(self,dt):
    frame_change = 4
    #if frame_index greater than number of surfaces in the animation frame, then
    #make it back to the beginning state
    self.frame_index += frame_change * dt
    if self.frame_index >= len(self.animations[self.animation_status]):
        self.frame_index = 0
    #update the image with the new surface
    self.image = self.animations[self.animation_status][int(self.frame_index)]
```

The animate method is used to animate the player. It will first change the frame index based on the frame change and delta time. If the frame index is greater than the length of the frame, then it will become 0 again. Lastly, it will update the image surfaces of the player periodically based on the animation status and the frame index of the player.

```
def input(self):
    #get input to determine which direction player moves
    keys = pygame.key.get_pressed()

    #we can only move and start using a tool when we are not using tools
    if not self.timers['tool use'].active and not self.sleep:
        #get and set movement's animation and direction of the player
        if keys[pygame.K_w]: #if w is pressed, player goes up
            self.direction.y = -1
            self.animation_status = 'up'
        elif keys[pygame.K_s]: #if s is pressed, player goes down
            self.direction.y = 1
            self.animation_status = 'down'
        else:
            self.direction.y = 0 #player doesn't move vertically

        if keys[pygame.K_d]: #if d is pressed, player goes right
            self.direction.x = 1
            self.animation_status = 'right'
        elif keys[pygame.K_a]: #if a is pressed, player goes left
            self.direction.x = -1
            self.animation_status = 'left'
        else:
            self.direction.x = 0 #player doesn't move horizontally
```

The input method is used to check the player's inputs. First off, it will checks if the player presses a key or not. If we are not using a tool and not sleeping (resetting the day), we can move up, down, left, or right. When moving vertically, the y factor of the direction vector will change, while when moving horizontally, the x factor of the direction vector will change. Otherwise, if the player doesn't move, then the y and x

factor will become 0. To move, the player has to press A (for left), W (for up), S (for down), and D (for right).

```
#activate the timer when player start using a tool (press space)
#and set the direction to (0,0)
#to prevent player moving when using a tool
if keys[pygame.K_SPACE]:
    self.timers['tool use'].activate()
    self.direction = pygame.math.Vector2()
    #frame index is set to 0 to make the animation of using tool starts from beginning
    self.frame_index = 0

#player can only switch tool when player press q and is not in the middle of switching tool
if keys[pygame.K_q] and not self.timers['tool switch'].active:
    self.timers['tool switch'].activate()
    #update tool index and tool status based on tool index to change the tool that player use
    self.tool_index += 1
    self.tool_index = self.tool_index if self.tool_index < len(self.tools) else 0
    self.tool_status = self.tools[self.tool_index]
```

If Space is pressed, then activate the timer for using the tool. The direction of the player will become (0,0) to prevent player moves when using a tool and the frame index returns to 0 because the animation for using the tool will start from the beginning. Then if the tool switch is not active and the player presses Q, the player can also switch the tool. The tool index will go up to change the currently active tool status to another tool. If the tool index reaches the last tool in the tool list, then set it back to 0.

```
#activate the timer when player start planting a seed and set the direction to (0,0)
#to prevent player moving when planting
if keys[pygame.K_j]:
    self.timers['seed plant'].activate()
    self.direction = pygame.math.Vector2()
    self.frame_index = 0

#player can only switch seed when player press i and is not in the middle of switching seed
if keys[pygame.K_i] and not self.timers['seed switch'].active:
    self.timers['seed switch'].activate()
    #update seed index and seed status based on seed index
    self.seed_index += 1
    self.seed_index = self.seed_index if self.seed_index < len(self.seeds) else 0
    self.seed_status = self.seeds[self.seed_index]
```

The plant seed and switch seed are also the same as the system checks for tool usage and tool switch. The difference is when the player uses a seed, they press J and when the player switches seeds, they press I.

```
#show help menu
if keys[pygame.K_h]:
    self.toggle_help()

#show inventory menu
if keys[pygame.K_p]:
    self.toggle_inventory()
```

If the player presses H, it will toggle the help menu. Meanwhile, if the player presses P, it will toggle the inventory.

```
#if player press Enter
if keys[pygame.K_RETURN]:
    collided_interaction_sprite = pygame.sprite.spritecollide(self,self.interaction,False)

    if collided_interaction_sprite:
        #if player collides with the trader, then toggle shop
        if collided_interaction_sprite[0].name == 'Trader':
            self.toggle_shop()

        #if player collides with the feed cow area, then
        elif collided_interaction_sprite[0].name == 'Feed_cow':
            #if milk is not harvestable, give food to the cow
            if self.harvest_milk == False:
                self.give_food()
                self.grass_sound.play()
            #if milk is harvestable, harvest milk
            else:
                self.milk_sound.play()
                self.get_milk()

        #if player collides with the bed, then player goes to sleep and play transition
        elif collided_interaction_sprite[0].name == 'Bed':
            self.animation_status = 'down_idle'
            self.sleep = True
```

If the player presses Enter, the input method will check for the collided interaction sprite. If the interaction sprite's name is Trader, then it will toggle the shop menu; if its name is Feed_cow, then when the milk is not harvestable, the player can give food, and grass sound will also be player, if the milk is harvestable, then harvest milk and play milk sound; else if it's in the bed, then the player will go to sleep (the sleep status will be True and the player's image will be reset to down idle).

```
    #checking mouse event
    mouse_pos = pygame.mouse.get_pos()
    self.button_setup()

    for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
            #if player clicks the help button, open help menu
            if self.help_button.checkForInput(mouse_pos):
                self.toggle_help()
            #if player clicks the inventory button, open inventory
            elif self.inventory_button.checkForInput(mouse_pos):
                self.toggle_inventory()

        #exit the game
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                pygame.quit()
                sys.exit()
```

Besides keyboard keys, the input method will also check for the mouse position of the player. It will call the button setup function to set up the button and then when the player clicks the help button, it will open the help menu. However, if the player clicks the inventory button, then it will open the inventory. Player can also exit the game by clicking quit or pressing Esc key.

```
def button_setup(self):
    #help button
    self.help_button = Button(image=self.help_image,
                              pos=BUTTON_POS['help'])
    #blit help button
    self.help_button.update(self.display_surface)
    #inventory button
    self.inventory_button = Button(image=self.inventory_image,
                              pos=BUTTON_POS['inventory'])
    #blit inventory button
    self.inventory_button.update(self.display_surface)
```

The button_setup method is used to configure the help button and inventory button. It will first create the help button and inventory button as objects from the class Button. After that, it will update or blit it on the display surface.

```
def get_status(self):
    #to change the animation when player goes idle
    if self.direction.magnitude() == 0:
        self.animation_status = self.animation_status.split('_')[0] + '_idle'

    #to change animation when player use particular tool
    if self.timers['tool use'].active:
        self.animation_status = self.animation_status.split('_')[0] + '_' + self.tool_status
```

The get_status method is used to get the animation status of the player. If the direction magnitude is 0 (or the player doesn't move), then the animation status will become idle and the position is obtained by splitting animation_status and getting the first word before the _. When the tool is in use, then the animation status will be updated based on the position of the player and the tool status (current tool used).

```
def update_timers(self):
    #update timers
    for timer in self.timers.values():
        timer.update()
```

The update_timers method is used to update the timer for every timer in the Player class (tool switch, tool use, seed switch, seed use).

```
def collision(self, direction):
    #if the sprite has the attribute of hitbox and it coliides with the player's hitbox
    for sprite in self.collision_sprites.sprites():
        if hasattr(sprite, 'hitbox'):
            if sprite.hitbox.colliderect(self.hitbox):
                if direction == 'horizontal':
                    if self.direction.x > 0: #the player moving right
                        self.hitbox.right = sprite.hitbox.left
                    if self.direction.x < 0: #the player moving left
                        self.hitbox.left = sprite.hitbox.right
                    #update the rect based on the hitbox
                    self.rect.centerx = self.hitbox.centerx
                    self.pos.x = self.hitbox.centerx

                if direction == 'vertical':
                    if self.direction.y > 0: #the player moving down
                        self.hitbox.bottom = sprite.hitbox.top
                    if self.direction.y < 0: #the player moving up
                        self.hitbox.top = sprite.hitbox.bottom
                    #update the rect based on the hitbox
                    self.rect.centery = self.hitbox.centery
                    self.pos.y = self.hitbox.centery
```

The collision method will check for all sprites in the collision sprites. If the sprite has the attribute hitbox and it collides with the player's hitbox, then if the direction is

horizontal and the player moving right, the player can't proceed to the right further because the player's right side of the hitbox is always set to the sprite hitbox left side. The logic for the left side is also similar. After that, this method will update the player x rect and x position based on this logic, so that a collision is happening. Meanwhile, if the collisions happens when the direction is vertical, then the one changes is not the x rect and x position, but the y rect and y position.

```
def move(self,dt):
    #normalize the speed of objects when moving diagonal
    if self.direction.magnitude() > 0:
        self.direction = self.direction.normalize()

    #horizontal movement
    self.pos.x += self.direction.x * self.speed * dt
    #update the hitbox and the rect for x value
    self.hitbox.centerx = round(self.pos.x)
    self.rect.centerx = self.hitbox.centerx
    self.collision('horizontal') #update horizontal collision

    #vertical movement
    self.pos.y += self.direction.y * self.speed * dt
    #update the hitbox and the rect for y value
    self.hitbox.centery = round(self.pos.y)
    self.rect.centery = self.hitbox.centery
    self.collision('vertical') #update vertical collision
```

The move method is used to move the player. When the magnitude of the player is greater than 0, then, normalize the movements, so that when the player moves diagonally, the player's speed still feels normal (not faster). After that, if the movement is horizontal, then update the x position based on the direction, the speed, and the delta time and update the x rect and x hitbox based on the rounded position (so that the position will be integer) and also check for horizontal collisions by called the collision method. That is also the same for the vertical movement, the difference is it will update the y factors and check vertical collisions.

```
def use_tool(self):
    #if player uses hoe, then till soil layer
    if self.tool_status == 'hoe':
        self.soil_layer.get_hit(self.target_pos)

    #if player uses axe and there is a tree that collides with the axe target position
    #then damage the tree
    if self.tool_status == 'axe':
        for tree in self.tree_sprites.sprites():
            if tree.rect.collidepoint(self.target_pos):
                tree.damage()

    #if player uses water can, then water the soil
    if self.tool_status == 'water':
        self.water_sound.play()
        self.soil_layer.water(self.target_pos)
```

The use_tool method defines what will the player do if they use a specific tool. If the player uses a hoe, then it will check if the soil layer gets hit by the target position. If the player uses an axe, then for every tree in tree sprites, if the tree collides with the axe, damage the tree. If the player uses the water can, then play the water can sound and water the soil tile based on the target position.

```
def get_target_pos(self):
    #get tool's target position
    self.target_pos = self.rect.center + PLAYER_TOOL_OFFSET[self.animation_status.split('_')[0]]
```

The get_target_pos method is used to get the tool's target position based on the player's position right now and the tool's offset for each direction that the player faces.

```
def use_seed(self):
    #if the seed is greater than 0 and the soil is plantable,
    #then plant the seeds and deduct the seed amount by 1
    if self.seed_inventory[self.seed_status] > 0 and self.soil_layer.check_plantable(self.target_pos):
        self.soil_layer.plant_seed(self.target_pos, self.seed_status)
        self.seed_inventory[self.seed_status] -= 1
```

The use_seed method will first check if the seed amount is greater than 0 and if the soil layer is plantable. If that's the case, then plant the seed based on the target position and the type of the seed. The amount of the seed will also be deducted by 1.

```
def give_food(self):
    #if grass is greater than 0 and cow is hungry, then cow becomes full,
    #deduct grass amount by 1, add cow age, and grass image will be blitted
    if self.other_inventory['grass'] > 0 and self.food == False:
        self.other_inventory['grass'] -= 1
        self.food = True
        self.cow_age += 1
        #create Generic object for the grass, so that it's visible
        grass_surf = pygame.image.load('../graphics/milk/grass.png').convert_alpha()
        for x, y, __ in self.tmx_data.get_layer_by_name('CowFood').tiles():
            Generic((x * TILE_SIZE,y * TILE_SIZE), grass_surf, [self.all_sprites, self.collision_sprites, self.grass_sprites])

    if self.cow_age >= self.cow_max_age:
        self.harvest_milk = True
```

The give_food method is used to give food to the cow. If the grass is still available and the cow is hungry, then the player will give grass to the cow. The grass in the inventory will be deducted by 1, the cows will become full, and the cow age will be added by 1. This method also creates a Generic object for the grass under the cow in the CowFood layer. The grass surface is loaded and converted to help boost the game speed. If the cow's age is greater or equal to the cow's max-age, then the milk status is now harvestable.

```
def get_milk(self):
    #if milk is harvestable
    if self.harvest_milk:
        #add the milk item by 1
        self.item_inventory['milk'] += 1
        #milk is not harvestable again and cow age is 0
        self.harvest_milk = False
        self.cow_age = 0

        #milk bottle become empty again
        milk_surf = pygame.image.load('../graphics/milk/milk_item/0.png').convert_alpha()
        for x, y, __ in self.tmx_data.get_layer_by_name('CowMilk').tiles():
            Generic((x * TILE_SIZE,y * TILE_SIZE), milk_surf, [self.all_sprites, self.collision_sprites])
```

The get_milk function is used to harvest milk. If the milk is harvestable, then add the milk item by 1 in the inventory and set the milk to become unharvestable again and the cow age to 0. It will also load the empty milk bottle surface and then create a Generic object on the display surface.

```
def update(self, dt):
    #update the player's position, animation, timer, etc based on the input
    self.input()
    #get player status (using tool or idle)
    self.get_status()
    #update timers
    self.update_timers()
    #get player's tool target position
    self.get_target_pos()

    #animate player and movements setup
    self.move(dt)
    self.animate(dt)
```

The update method is used to update the player based on the input, and then get its status and target position of the tool, and update the timers. It will also update the player's position based on the movement and animate the player.

```
def update_cutscene(self):
    keys = pygame.key.get_pressed()
    #if C is pressed
    if keys[pygame.K_c]:
        #if cutscene3 is already completed
        if 'cutscene3' in self.cut_scene_manager.cut_scenes_complete:
            collided_interaction_sprite = pygame.sprite.spritecollide(self,self.interaction,False)
            if collided_interaction_sprite:
                #and the player collides with the table
                if collided_interaction_sprite[0].name == 'Table':
                    #start cutscene4
                    self.cut_scene_manager.start_cut_scene(CutSceneFour())

                    #create a Christmas tree in Bunn's house after cutscene4 ended
                    christmas_tree_surf = pygame.image.load('../graphics/dialogue/cutscene4/christmas_tree.png').convert_alpha()
                    for x, y, __ in self.tmx_data.get_layer_by_name('ChristmasTree').tiles():
                        Generic((x * TILE_SIZE,y * TILE_SIZE), christmas_tree_surf, [self.all_sprites, self.collision_sprites])

        #if cutscene2 is already completed
        elif 'cutscene2' in self.cut_scene_manager.cut_scenes_complete:
            #if player's money >= 200, then start cutscene3
            if self.money >= 200:
                self.cut_scene_manager.start_cut_scene(CutSceneThree())
                #add player's money by 2000 after cutscene3 ends
                self.money += 1800

        #if cutscene1 is already completed
        elif 'cutscene1' in self.cut_scene_manager.cut_scenes_complete:
            collided_interaction_sprite = pygame.sprite.spritecollide(self,self.interaction,False)
            #if player collides with table, then start cutscene2
            if collided_interaction_sprite:
                if collided_interaction_sprite[0].name == 'Table':
                    self.cut_scene_manager.start_cut_scene(CutSceneTwo())
```

The update_cutscene method is used to update the cutscene based on the player. First, it will check if there is any C key that is pressed. If that's the case, then it will check the completed cutscene in the completed cutscenes list. If cutscene3 is completed and the player collides with the table (the mission from cutscene3), then start the last cutscene, After the fourth cutscene, load the Christmas tree surface and create a Generic object out of it.

If cutscene2 is completed and the player's money is greater than $200 (the mission from cutscene2), then start cutscene3, and add the money by 1800 (related to the story). Meanwhile, if cutscene1 is completed and the player collides with the table (mission from cutscene1), then play the cutscene2.

12. game_display.py

This file contains the game function and the map setup for this game.

```
import pygame
from game_settings import *
from player import Player
from overlay import Overlay
from sprites import *
from pytmx.util_pygame import load_pygame
from support import import_folder
from transition import Transition
from soil import SoilLayer
from sky import Rain, Sky
from random import randint
from merchant_menu import Menu
from overlay_menu import Overlay_Menu, Inventory
from cutscene import CutSceneManager
```

Import pygame and from game_settings import all. Then, import Player class from player, Overlay class from overlay, all class from sprites, load_pygame from pytmx, import_folder from support, Transition class from transition, SoilLayer class from soil, Rain and Sky classes from sky, randint from random, Menu class from merchant_menu, Overlay_Menu and Inventory classes from overlay_menu, and CutSceneManager from cutscene.

- Class CameraGroup

```
class CameraGroup(pygame.sprite.Group):
    def __init__(self):
        super().__init__()
        #the display surface and the offset
        self.display_surface = pygame.display.get_surface()
        self.offset = pygame.math.Vector2()
```

When initialising an object of this class, it will have the attributes as the sprite Group from pygame (because it is inherited from that), the display surface that is obtained using pygame, and the offset which is a 2-dimensional vector.

```
def custom_draw(self, player):
    #how much we want to shift the sprite based on the player's movement
    self.offset.x = player.rect.centerx - SCREEN_WIDTH / 2
    self.offset.y = player.rect.centery - SCREEN_HEIGHT / 2

    #it is sorted to draw the layers based on the y index
    #if the player y is lesser than the object, then the player will be blitted behind that object
    for layer in LAYERS.values():
        for sprite in sorted(self.sprites(), key = lambda sprite: sprite.rect.centery):
            if sprite.z == layer:
                offset_rect = sprite.rect.copy()
                offset_rect.center -= self.offset
                self.display_surface.blit(sprite.image, offset_rect)
```

The custom_draw method will first calculate the offset or how much we want to shift the sprite based on the player's movement. Then, it will blit the layer from the lowest one and sort all the sprites based on the y index. So, if the player's y factor is lesser than the object, then the player will be drawn behind it. It will draw the sprite based on the offset rect.

- Class Display

```
class Display:
    def __init__(self):
        # get the display surface
        self.display_surface = pygame.display.get_surface()
        #cutscene manager
        self.cut_scene_manager = CutSceneManager(self.display_surface)

        # sprites groups
        #to know which object has to be updated based on player's point of view (camera)
        self.all_sprites = CameraGroup()
        self.collision_sprites = pygame.sprite.Group()
        self.tree_sprites = pygame.sprite.Group()
        self.interaction_sprites = pygame.sprite.Group()
```

When initializing an object from the Display class, it will have the attributes display surface that is obtained using pygame, the cut_scene_manager by initializing the CutSceneManager object, and the sprites groups. There is the all_sprites which is a CameraGroup (so that it will be updated frequently), and then the collision, tree, and interaction sprites which are a standard pygame sprite Group.

```
        #the soil layer (has to be updated and also contains plant that is collidable)
        self.soil_layer = SoilLayer(self.all_sprites, self.collision_sprites)
        self.setup() #set up game elements

        #the overlay tool and seed status on the bottom left of the screen
        self.overlay = Overlay(self.player)
        #the transition to reset a day
        self.transition = Transition(self.reset, self.player)
```

Next, there is the soil layer which is an object from the SoilLayer class, overlay which is an object from the Overlay class, and transition which is an object from the Transition class. It will also call the setup function to set up the game elements based on the map.

```
#rain
self.rain = Rain(self.all_sprites)
#percentage of the rain
self.raining = randint(0,10) < 3
#to update the all soil to water soil tiles if raining
self.soil_layer.raining = self.raining
#day-night transition
self.sky = Sky()
```

There is also the rain created from the Rain class and then the percentage of the rain which is 3/10 or 30% because integers that are lesser than 3 in range (0,10) is 0,1,2 and the total number of integers in range (0,10) is 10. If it's less than 3, then raining will be True, else False. The code below that line is to pass the raining variable to the soil layer. After that, it also creates a sky object from the Sky class.

```
#shop
self.menu = Menu(self.player, self.toggle_shop)
self.shop_active = False #shop is not default open

#help
self.help_active = False #help is not defaultly opened
self.help = Overlay_Menu(self.player, self.toggle_help)

#inventory
self.inventory_active = False #inventory is not default opened
self.inventory = Inventory(self.player, self.toggle_inventory)
```

Here, we also create the merchant menu from the class Menu, the help menu from the class Overlay_Menu, and the inventory from the class Inventory. All of them are inactive (by default) or are not opened.

```
#Sound
#success acquiring plants and woods
self.success_sound = pygame.mixer.Sound('../audio/success.mp3')
self.success_sound.set_volume(0.3)

#background music, (loops=-1) to loop it until player exits
self.bg_sound = pygame.mixer.Sound('../audio/bg.mp3')
self.bg_sound.set_volume(0.25)
self.bg_sound.play(loops=-1)

#when clicking a button or toggling menus
self.click_sound = pygame.mixer.Sound('../audio/click.mp3')
self.click_sound.set_volume(0.3)
```

There are also sound attributes. The first one is the success sound which is played when the player successfully harvests a plant or cuts a wood. Then, the background music is played from the beginning until the end and loops until the game is quit (that's why loops = -1). There is also the click sound that is played when clicking a button or toggling the menus (shop, help, inventory).

```
def setup(self):
    #load the map for the game elements
    self.tmx_data = load_pygame('../data/map.tmx')

    #house bottom surface
    for layer in ['HouseFloor', 'HouseFurnitureBottom']:
        #for every tile in the tilelayers, create a Generic object with the layer in house bottom
        for x, y, surf in self.tmx_data.get_layer_by_name(layer).tiles():
            Generic((x * TILE_SIZE,y * TILE_SIZE), surf, self.all_sprites, LAYERS['house bottom'])

    #house top surface
    for layer in ['HouseWalls', 'HouseFurnitureTop']:
        #for every tile in the tilelayers, create a Generic object with the layer main (default)
        for x, y, surf in self.tmx_data.get_layer_by_name(layer).tiles():
            Generic((x * TILE_SIZE,y * TILE_SIZE), surf, self.all_sprites)
```

The setup method is used to load the map and create the object in the layer of the map. First off, it will create a Generic object for each tile on the house floor and house furniture bottom in the 'house bottom' layer. Similar to the house bottom surface, the

setup method will get all the surfaces in the house walls and house furniture top layer and create a Generic object out of it in the layer ‘main’ (default).

```
#fence
for x, y, surf in self.tmx_data.get_layer_by_name('Fence').tiles():
    #for every surface in the tilelayer, create a Generic object with the layer in main
    #it needs collision sprite to restrain the player from moving out of the fences
    Generic((x * TILE_SIZE,y * TILE_SIZE), surf, [self.all_sprites, self.collision_sprites])

#water
water_frames = import_folder('../graphics/water') #to animate water
for x, y, surf in self.tmx_data.get_layer_by_name('Water').tiles():
    #for every surface in the tilelayer, create a Water object with the layer in water (default)
    Water((x * TILE_SIZE,y * TILE_SIZE), water_frames, self.all_sprites)
```

Similar to the housetop surface, the surfaces in the fence layer will also be a Generic object, but it needs collision sprites because its collision is not set up yet in the collision layer in the map. The fence helps restrain the player from moving out of the explorable area. The water is animated, so we have to import the surfaces first. After that, for each surface in the Water layer in the map, create a Water object.

```
#trees
for obj in self.tmx_data.get_layer_by_name('Trees'):
    #for every object in the object layer, create a Tree object with the layer in main (default)
    Tree(
        pos = (obj.x, obj.y),
        surf = obj.image,
        groups = [self.all_sprites, self.collision_sprites, self.tree_sprites],
        name = obj.name,
        player_add = self.player_add)

#wildflowers
for obj in self.tmx_data.get_layer_by_name('Decoration'):
    #for every object in the object layer, create a Generic object with the layer in main (default)
    Generic((obj.x, obj.y), obj.image, [self.all_sprites, self.collision_sprites])

#collision tiles
#no need to be updated and visible, so don't have to be included in all sprites
for x, y, surf in self.tmx_data.get_layer_by_name('Collision').tiles():
    #simulate collisions on objects from the other tilelayers
    Generic((x * TILE_SIZE, y * TILE_SIZE), pygame.Surface((TILE_SIZE, TILE_SIZE)), self.collision_sprites)
```

For every object in the object layer Trees, create a Tree object and the layer is also in ‘main’ (default), it is also in all sprites, collision sprites (because it can collide with the player), and tree sprites. Then, for wildflowers which are in the ‘Decoration’ object layer, create a Generic object out of it (it can also collide, so it is in collision sprites too besides all sprites). Meanwhile, collision tiles, don’t need to be visible, so this method creates a Generic object with the image only a pygame Surface (no

colour) and also it only needs collision sprites (because its image doesn't need to be updated).

```
#the player
for obj in self.tmx_data.get_layer_by_name('Player'):
    #player will be first be blitted on the start object in the player object layer
    if obj.name == 'Start':
        self.player = Player(
            pos = (obj.x,obj.y),
            group = self.all_sprites,
            collision_sprites = self.collision_sprites,
            all_sprites = self.all_sprites,
            tree_sprites = self.tree_sprites,
            interaction = self.interaction_sprites,
            soil_layer = self.soil_layer,
            toggle_shop = self.toggle_shop,
            toggle_help = self.toggle_help,
            toggle_inventory= self.toggle_inventory,
            tmx_data_map= self.tmx_data,
            cutscene= self.cut_scene_manager)

    #if the object is bed, then create an Interaction object, so that we can interact with it (reset the day)
    if obj.name == 'Bed':
        Interaction((obj.x,obj.y), (obj.width,obj.height), self.interaction_sprites, obj.name)

    #if the object is trader, then create an Interaction object, so that we can interact with it (open shop)
    if obj.name == 'Trader':
        Interaction((obj.x,obj.y), (obj.width,obj.height), self.interaction_sprites, obj.name)

    ##if the object is feed_cow, then create an Interaction object, so that we can interact with it (feed the cow)
    if obj.name == "Feed_cow":
        Interaction((obj.x,obj.y), (obj.width,obj.height), self.interaction_sprites, obj.name)

    #if the object is table, then create an Interaction object, so that we can interact with it (play specific cutscene)
    if obj.name == "Table":
        Interaction((obj.x,obj.y), (obj.width,obj.height), self.interaction_sprites, obj.name)
```

For objects in the object layer Player, if its name is Start, then the player will start the game there. If the object name is bed, trader, feed_cow, or table, then create an interaction sprite based on its name. The interaction sprite is used to detect if the player collides with it or not. If collides, then do some function. For bed, it will reset the day; for trader, it will open the shop menu; for feed_cow, it will feed the cow, and for table, it will start some cutscene (based on the mission given).

```
cow_frames = import_folder('../graphics/animal') #to animate cow
#Cow
for x, y, surf in self.tmx_data.get_layer_by_name('cow').tiles():
    #for every tile in the tilelayers, create an Animal object with the layer main (default)
    Animal((x * TILE_SIZE,y * TILE_SIZE), cow_frames, [self.all_sprites, self.collision_sprites])

#Cow Farm (Fence)
for x, y, surf in self.tmx_data.get_layer_by_name('CowFarm').tiles():
    #for every tile in the tilelayers, create a Generic object with the layer main (default)
    Generic((x * TILE_SIZE,y * TILE_SIZE), surf, [self.all_sprites, self.collision_sprites])

#to set the milk bottle to the first image (the empty one)
milk_surf_list = import_folder('../graphics/milk/milk_item')
self.index_milk = 0
milk_surf = milk_surf_list[self.index_milk]

for x, y, __ in self.tmx_data.get_layer_by_name('CowMilk').tiles():
    #for every tile in the tilelayers, create a Generic object with the layer main (default)
    Generic((x * TILE_SIZE,y * TILE_SIZE), milk_surf, [self.all_sprites, self.collision_sprites])
```

This is the configuration for the cow farms, from the cows, the fences for the farm, and then the milk bottle (that represents the status of the milk). The cow is animated and the milk can change per day, so it will import the surfaces first. After that, it will create Generic objects for them. Meanwhile, the fences are the same as the regular fences, so the code to create them is the same.

```
#the floor background, so the layer is ground or the lowest layer
Generic(
    pos = (0,0),
    surf = pygame.image.load('../graphics/world/ground.png').convert_alpha(), #load the image
    groups = self.all_sprites,
    z = LAYERS['ground'])
```

Lastly, the setup method will also create a Generic object for the ground and the layer is in ‘ground’ or the lowest layer in this game.

```
def toggle_shop(self):
    #toggling on and off for the shop
    self.click_sound.play()
    self.shop_active = not self.shop_active

def toggle_help(self):
    #toggling on and off for the help menu
    self.click_sound.play()
    self.help_active = not self.help_active

def toggle_inventory(self):
    #toggling on and off for the inventory
    self.click_sound.play()
    self.inventory_active = not self.inventory_active
```

The toggle_shop, toggle_help, and toggle_inventory methods are almost the same. It will first play the click sound when the player toggles one of them. The code active = not active means that if the menu is active then set it to not active, meanwhile if the menu is not active, set it to not 'not active' (active). The toggle_shop is used to toggle the shop menu, toggle_help for toggling help menu, and toggle_inventory for toggling inventory menu.

```
def player_add(self,item):
    #+1 item to the inventory
    self.player.item_inventory[item] += 1
    self.success_sound.play()
```

The player_add is used to add the plant and wood to the inventory. When this happens, the amount of the item will be added by 1 and success sound will be played.

```
def reset(self):
    #plants growth
    self.soil_layer.update_plants()
    #remove water on the soils
    self.soil_layer.remove_water()

    #randomize the rain again
    self.raining = randint(0,10) < 3
    self.soil_layer.raining = self.raining
    #if it's raining, all plantable soils will become water soils
    if self.raining:
        self.soil_layer.water_all()

    #tree alive again
    for tree in self.tree_sprites.sprites():
        tree.revive()

    #sky color back to day
    self.sky.start_color = [255,255,255]
```

The reset method is used to reset the day when the player goes to sleep. This method will update all the plants or grow the plants and remove all water in the soil layer. If it's raining, then water all soil tiles. This method will also revive all the trees in the tree sprites and change the sky colour back by setting it to be multiplied by white again.

```
#update milk bottle
if self.player.food == True:
    milk_surf_list = import_folder('../graphics/milk/milk_item')

    #if cows are feeded and index_milk is lesser than 4 (because the index milk is from 0 to 4)
    #then index_milk will be added 1
    if self.index_milk < (len(milk_surf_list) - 1):
        self.index_milk += 1

    #if the index_milk = 4, then if we feed the cow again, we will harvest the mik
    #the milk bottle will become empty again
    else:
        self.index_milk = 0

    #create a Generic object based on the updated milk bottle
    milk_surf = milk_surf_list[self.index_milk]
    for x, y, __ in self.tmx_data.get_layer_by_name('CowMilk').tiles():
        Generic((x * TILE_SIZE,y * TILE_SIZE), milk_surf, [self.all_sprites, self.collision_sprites])

#cow hungry again and grass disappear
self.player.food = False
for grass in self.player.grass_sprites:
    grass.kill()
```

It will also update the milk bottle. If the cow is fed and index milk is lesser than the length of the frame for the milk bottle, index milk will be added by 1, else it will become empty again. After that, we update that milk bottle surface and create a Generic object out of it. After a day, the cow will also be hungry again and the grass disappears (because the cow has eaten them).

```
def plant_collision(self):
    #if there is plant in the soil layer
    if self.soil_layer.plant_sprites:
        #for every plant
        for plant in self.soil_layer.plant_sprites.sprites():
            #if plant is harvestable and the player collides with the plant
            if plant.harvestable and plant.rect.collidepoint(self.player.target_pos):
                #add the crops to the inventory and kill the plant object
                self.player_add(plant.plant_type)
                plant.kill()

                #generate a white mask particle when the plant is killed
                Particle(
                    pos = plant.rect.topleft,
                    surf = plant.image,
                    groups = self.all_sprites,
                    z = LAYERS['main']
                )

                #the soil layer can be used to plant seeds again
                self.soil_layer.grid[plant.rect.centery // TILE_SIZE][plant.rect.centerx // TILE_SIZE].remove('P')
```

The plant_collision method is used to harvest the plant. If there is a plant in the soil sprite and for every plant in the soil sprites, if the plant is harvestable and the plant's rect collides with the player's target position, then add the plant item to the player's inventory and kill the plant object. When harvesting, there will also be a white particle for some milliseconds that signifies that the plant is killed. The tile will also not contain 'P' anymore because the plant is harvested.

```
def run(self,dt):
    #run and update all the game elements
    #if no cutscene runs at the moment
    if self.cut_scene_manager.cut_scene_running == False:
        #draw player and its environment based on the camera
        self.all_sprites.custom_draw(self.player)
        #day-night transition
        self.sky.display(dt)
        #if shop is active and help and inventory is not active, update shop
        if self.shop_active and not self.help_active and not self.inventory_active:
            self.menu.update()
        #if help is active and shop and inventory is not active, update help menu
        elif self.help_active and not self.shop_active and not self.inventory_active:
            self.help.update()
        #if inventory is active and help and shop is not active, update inventory
        elif self.inventory_active and not self.shop_active and not self.help_active:
            self.inventory.update()
```

```
        else:
            #update or play cutscenes if player done the missions and press C
            self.player.update_cutscene()
            #update all objects in all sprites
            self.all_sprites.update(dt)
            # player can harvest plant
            self.plant_collision()
            #display the tool and seed overlay
            self.overlay.display()
    else:
        #if cutscene is running, then update and draw scene
        self.cut_scene_manager.update()
        self.cut_scene_manager.draw()
```

The run method is used to run the game elements that have been set up. If there is no cutscene (the running cutscene is False), then draw the player and its environment based on the camera and display the day-night transition, and then update the shop menu (if the shop is active and help and inventory are not active), update the help menu (if the help is active and shop and inventory are not active), and update the inventory (if the inventory is active and help and shop are not active).

Otherwise, if the menus are not active, update the cutscene if the player does the mission and presses C, update all sprites, so the player can harvest the plant, and the overlay tool status and seed status can be seen on the bottom left. Otherwise, if the cutscene is running, then update and draw the scene in the cutscene.

```
#if it's raining and shop, help, inventory menu is not active, and no cutscene runs
if self.raining and not self.shop_active and not self.help_active and not self.inventory_active \
    and self.cut_scene_manager.cut_scene_running == False:
    #then update rain drops and rain floors
    self.rain.update()

#transition if player goes to bed
if self.player.sleep:
    self.transition.play()
```

If it's raining and all the menus and cutscene are not active, then update the rain drops and rain floors. If the player goes to sleep, then play the transition.

13. intro.py

This file contains the intro menu configuration. The intro menu is the display screen before player starts the game.

```
import pygame
from game_settings import *
from support import *
from sprites import Generic
from cutscene import CutSceneOne
```

Improt pygame, from game_settings and support import everything, from sprites import Generic and from cutscene import CutSceneOne.

- Class Intro

```
class Intro:
    def __init__(self, display_game):
        self.display_surface = pygame.display.get_surface() #display screen
        self.display_game = display_game #game function
        self.topic_font = pygame.font.Font('../font/LycheeSoda.ttf', 100) #topic font
        self.text_font = pygame.font.Font('../font/LycheeSoda.ttf', 30) #Bunn's dialogue font

        #button animation
        self.frames = import_folder('../graphics/button/play')
        self.frame_index = 0

        #character animation
        self.chara_frames = import_folder('../graphics/world/intro_emote')
        self.chara_frame_index = 0
```

Upon initializing an object from this class, there will be several attributes for it. The attributes are the display surface that is obtained by using pygame, the display_game (game state), the topic_font and text_font, the button's and character's frames and frame indexes. The topic font is used for the game title, while the text font is used for the text dialogue for Bunn in the intro screen.

```
def blit_all(self):
    #load the Christmas hat image, rotate and scale it, then blit it
    hat_surf = pygame.image.load('../graphics/world/hat.png').convert_alpha()
    hat_surf = pygame.transform.rotozoom(hat_surf, 50, 1.5)
    hat_rect = hat_surf.get_rect(topleft=(110, 0))
    self.display_surface.blit(hat_surf, hat_rect)

    #load the Christmas tree image and blit it
    tree_surf = pygame.image.load('../graphics/world/christmas_tree.png').convert_alpha()
    tree_rect = tree_surf.get_rect(topleft=(1050, 0))
    self.display_surface.blit(tree_surf, tree_rect)

    #load the first part of the topic text as a surface and blit it
    topic_surf1 = self.topic_font.render((GAME_NAME.split()[0] + " " + GAME_NAME.split()[1]).upper(), False, 'chartreuse4')
    topic_rect1 = topic_surf1.get_rect(center= (SCREEN_WIDTH/2, 100))
    self.display_surface.blit(topic_surf1, topic_rect1)

    #load the second part of the topic text as a surface and blit it
    topic_surf2 = self.topic_font.render((GAME_NAME.split()[2] + " " + GAME_NAME.split()[3]).upper(), False, 'darkgreen')
    topic_rect2 = topic_surf2.get_rect(center= (SCREEN_WIDTH/2, 200))
    self.display_surface.blit(topic_surf2, topic_rect2)

    #load the dialogue box and blit it
    dialog_surf = pygame.image.load('../graphics/world/dialog.png').convert_alpha()
    dialog_rect = dialog_surf.get_rect(topleft=(350, 500))
    self.display_surface.blit(dialog_surf, dialog_rect)

    #load the dialogue text and blit it
    text_surf = self.text_font.render("Hello, I'm Bunn and this is my Christmas story.", False, 'Black')
    text_rect = text_surf.get_rect(topleft=(400, 560))
    self.display_surface.blit(text_surf, text_rect)
```

The blit_all method is used to blit all the images and text to the intro menu screen. The method for all the images is the same, it will first load them as surfaces and convert them to boost the game speed. After that, it will get the rect position of the surface and blit the surface on the display surface based on the rect. The hat surface is also get rotated and scaled, meanwhile the rest (the tree and the dialogue box) is not.

As for the text, it will first split the game name and get the first two words to become the topic text1 and the last two words to become topic text2. Both of them have slightly different colours and the topic text2 is slightly lower than the topic text1's position. After that, they and the dialogue text will be rendered as surfaces and then blittes on the screen.

```
def animate_all(self,dt):
    #button animation
    self.frame_index += 2 * dt
    if self.frame_index >= len(self.frames):
        self.frame_index = 0
    self.image = self.frames[int(self.frame_index)]

    #the play button
    self.play_button = Button(self.image, (SCREEN_WIDTH/2, SCREEN_HEIGHT/2))
    self.play_button.update(self.display_surface)

    #chara animation
    self.chara_frame_index += 2 * dt
    if self.chara_frame_index >= len(self.chara_frames):
        self.chara_frame_index = 0

    #load the chara image based on the index and blit it
    self.chara_image = self.chara_frames[int(self.chara_frame_index)]
    self.chara_rect = self.chara_image.get_rect(topleft=(70, 425))
    self.display_surface.blit(self.chara_image, self.chara_rect)
```

The animate_all method is used to animate the button and the character. It will first change the frame index and then set the new surface based on the frame index. After that, it will get the rects of the new surfaces and blit the surface on the display surface.

```
def run(self, dt):
    #display the background image on the screen
    background = pygame.image.load('../graphics/world/intro.png').convert()
    background_rect = background.get_rect(topleft= (0,0))
    self.display_surface.blit(background, background_rect)

    self.blit_all() #blit objects
    self.animate_all(dt) #animate objects
```

The run method is used to load the background image surface, get its rect, and blit the surface on the screen. The run method also calls the blit _all method to blit all the other objects and the animate_all method to animate specific objects.

```
def play_cutscene(self):
    #start cutscene one
    self.display_game.cut_scene_manager.start_cut_scene(CutSceneOne())

    #create a letter Generic object besides the table after cutscene1
    letter_surf = pygame.Surface((20,10))
    letter_surf.fill('white')

    for x, y, __ in self.display_game.tmx_data.get_layer_by_name('Letter').tiles():
        Generic((x * TILE_SIZE,y * TILE_SIZE), letter_surf,
                [self.display_game.all_sprites, self.display_game.collision_sprites])
```

The play_cutscene method is used to play the first cutscene when the player first clicks the play button. After that, it will also create a Generic object for the letter surface (its colour is white) as it is related to the story.

14. main.py

This file contains the main game loop for this game and there are two game states for this game, which is the intro menu and the game.

```
import pygame, sys
from game_settings import *
from game_display import Display
from intro import Intro
```

Import pygame and sys, from game_settings import everything, and import Display from Game_Display and Intro from intro.

- Class Game

```
class Game:
    #initialize game
    def __init__(self):
        #init pygame
        pygame.init()
        #set the screen and the game title
        self.screen = pygame.display.set_mode((SCREEN_WIDTH,SCREEN_HEIGHT))
        pygame.display.set_caption(GAME_NAME)
        #clock will be used to count delta time below
        self.clock = pygame.time.Clock()

        #the game function
        self.display = Display()
        #the intro menu
        self.intro = Intro(self.display)
```

Upon initializing an object of this class, there will be several attributes. First of all, it will initialize pygame. After that, it will have the screen attribute with the width and height of the screen. Then, we display the name of the window using pygame. We also set the attribute clock that will be used to count delta time (dt). After that, we create the object Display for the game and the object Intro for the intro menu.

```
def intro_menu(self):
#intro menu state
    while True:
        #counting delta time
        #delta time is used to ensure smooth consistent animation across framerate
        dt = self.clock.tick() / 1000

        #display the intro menu
        self.intro.run(dt)

        #checking pygame event
        for event in pygame.event.get():
            #if player clicks quit or press escape, then quit the game
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    pygame.quit()
                    sys.exit()

            #if player clicks the play button, then
            #play click sound, play the first cutscene, and run the game
            if event.type == pygame.MOUSEBUTTONDOWN:
                mouse_pos = pygame.mouse.get_pos()
                if self.intro.play_button.checkForInput(mouse_pos):
                    self.display.click_sound.play()
                    self.intro.play_cutscene()
                    self.run_game()

        #update display
        pygame.display.update()
```

The intro_menu method is for the intro menu state. So, first, it will count the delta time. Delta time is used to ensure smooth consistent animation across framerate. After that, this method will display the intro menu and check for player input. If the player presses Esc key or quit the game, then exit this program. However, if the player clicks

the play button, the click sound and the first cutscene will be played, and then the game will start. It will also update the entire display.

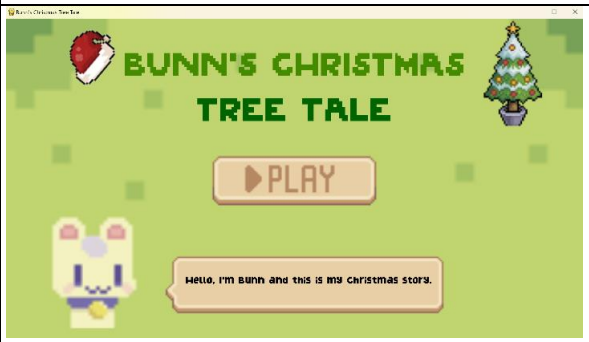
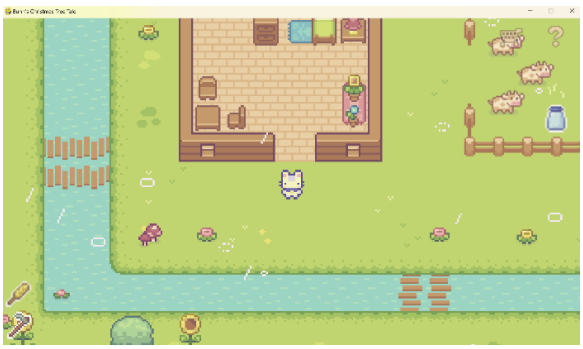
```
def run_game(self):
    #while running the game
    while True:
        #formula for delta time
        dt = self.clock.tick() / 1000

        #checking pygame event
        for event in pygame.event.get():
            #if player clicks quit or press escape, then quit the game
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    pygame.quit()
                    sys.exit()

        #run all the game elements and update the display
        self.display.run(dt)
        pygame.display.update()
```

The run_game method is used to run the game or for the actual game state. First, it will also calculate the delta time and then check for player input. If the player quits the game or presses the Esc key, then it will quit the game. It will also run all of the game elements and update the entire display.

```
#This code will only be executed when we run this program as a script, not an imported module
if __name__ == '__main__':
    #make a game object and then run the intro menu
    game = Game()
    game.intro_menu()
```

This code is not inside the game class. It is in the main loop. So, this code will only be executed when we run it as Python script or the main file not if we run it as an imported module. If that's the case, then we create a game object and then run the intro menu for the game first.

E. Evidences of Working Program

Name	Picture
The intro menu	
The cutscene	
The day time	
The night time	

Other features can be viewed through the demo video that is attached in the link of the README file.

F. Lesson Learnt (Reflection)

From this project, I learnt a lot of things from technical skills and also non-technical skills. First of all, I would like to thank Sir Jude for giving us this project, so that we can learn Python in our own way. When I first got this project as my assignment, I didn't know what I was going to make and I was also confused because my Python skills are still basic. I have learnt C++ before, but it's only until OOP and I haven't tried to implement it in a fully functional program. However, after this project, my understanding of OOP, especially in Python has deepened. I can create my own classes and modify them.

I also learnt a lot about the pygame and pytmx module by doing this project because, in the end, I decided to make a game. I am quite satisfied with myself as the map that I created myself is functional and the layout is beautiful (in my opinion). I created this game by modifying the program that Clear Code has (the Pydew Valley). I watched his videos and learned pygame a lot from him. This game's mechanic is modified from his Pydew Valley so that it has a new map, a full story for the character, the cow system, some new menus, etc.

Besides pygame, the pytmx module is also new for me. I also learned to use Tiled, an application to edit tmx files for maps and tiles. The designing process is quite satisfying because the assets from Sprout Land are cute.

Besides technical skills in Python, I also learned that I should manage my time wiser next time I have several big projects. I admit that I spent several months for the HCI (Human and Computer Interaction) project, so I didn't go to think about what should I do for this project. I got the idea of this project only when the HCI project almost ended (almost the Christmas holiday). I also finished this almost 100-page report in only 4 consecutive hard-working days (from 10.00 A.M. until 03.00 A.M.), so I quite regretted not starting this sooner because it's actually quite fun although tiring. I also planned to develop more features for this game, for example chicken and egg system, longer storyline, fishing, quiz quest from the merchant, more seed types, etc. However, I regretted that I didn't have enough time for that.

I rejoice too that there is no major bug for this game, so I can debug it. After the Christmas holiday, when I went to the campus, I realised that several people (maybe) also tried to make the Pydew Valley game, but they just only started. So, I hope with the story and other features and map of this game, players won't get bored with this kind of game mechanic and will enjoy this game's story.

G. Resources

Learning resources:

- https://youtu.be/T4IX36sP_0c?si=SWBr1FJOJsnNRTLp (Pydew Valley)
- https://youtu.be/N6xqCwblyiw?si=cSlMqJROhVaVNBzm (Tiled and pytmx)
- https://thorbjorn.itch.io/tiled (download Tiled)
- https://youtu.be/AY9MnQ4x3zk?si=Rw2qzcAVISgUi93h (pygame)
- https://youtu.be/rWtfClpWSb8?si=7XojNVWWtFJcYA3w (framerate and delta time)
- https://www.pygame.org/docs/ (pygame)
- https://youtu.be/Mp57mHfOXTw?si=MDjZUzdBcTMz5NyZ (cutscene in pygame)
- https://youtu.be/GMBqjxcKogA?si=JYnufJLPwULyNH00 (several states in pygame)
- https://stackoverflow.com/ and https://www.quora.com/ (debugging)

Resources for images and audio assets are in part C of this document.