Computer Seminar I Final Project
Spring 2024
Maximilian Fernaldy
C2TB1702

Sendai 11 PM

Sendai 11PM is a side-scrolling beat-em-up game inspired by Streets of Rage. The source folder consists of 3 module files (game_manager.py, entities.py and ui_elements.py), a main file (main.py), a JSON file containing level data, and the "resources" folder containing game assets. The rest are not required to run the game, but useful for virtual environment creation or type checking with pyright (currently, docstrings are incomplete, but will be improved later).

Pre-requisites

- Python 3.12
- pygame

All other dependency modules are built into Python 3.12. Other Python versions might work, but this project was built and tested with Python 3.12.3.

Classes in game_manager.py

GameInstance

This class is used to manage the topmost level of the game. On initiation, it assigns a new display surface object to `self.screen`, which will then be used by all draw methods later on. It then loads level data stored in `leveldata.json` and stores it in `self.level_data` for later use. This is done using `json`, a built in module used to write and read files in the JSON specification. This is particularly useful to simplify level creation. Finally, it sets the current stage to 1 (the first stage), initializes the level according to the current stage with `self.current_level = Level(self)` and declares `self.game_state` as `"start screen"` to show the home screen of the game.

Methods:

- `goto_next_stage(self)`: increments the current stage by 1, creates a new Level object and assigns it to `self.current_level`, then starts the level by calling `self.current_level.start_from_beginning()`.
- `restart_stage(self)`: keeps the current stage, creates a new Level object (essentially resetting all the object instances of the Level object, like the player, enemies, etc.) then starts the level from the beginning.
- `resume(self)`: This method simply sets `self.game_state` to `"play"`, and is defined so that it can be called by a UI element or keypress, like the resume button in the pause menu.

Level

This class handles all objects and events relating to a single level. Upon initiation, it assigns the `game_instance` parameter to `self.game_instance` so that its methods can have access to objects like the game state and current stage. The level data for the specific stage is loaded by indexing it with `str(stage)`. The numeric `stage` is converted to a string because the level data is stores in JSON, which does not allow number-type keys. Consequently, `json.load()` outputs a dictionary of **string-type keys and values**. Therefore to load the current level we use:

```
self.game_instance.level_data[str(stage)]
```

Next, the initial parameters of the entities in the level are assigned. These initial parameters are the player's maximum health, the initial position, and data relating to enemies. This consists of the initial positions of each enemy and how much damage they can inflict according to the level data loaded earlier. This approach allows the enemies to have more and more damage that can be inflicted as the levels become harder and harder.

Finally, the background for the level is loaded by creating a Map object.

Methods:

- `start_from_beginning(self)` : This method is executed when a level is started after completing the previous level, or when the restart button in the pause menu is pressed. Therefore, it needs to create/reassign the level instances, which are `self.player` and `self.enemies` . Creating `self.player` is done by calling the Player() constructor and passing in initial parameters defined in the `Level` constructor. However, creating enemies is a little harder. As the data is stored as Enemy objects, simply assigning the `self.enemy_data` array into `self.enemies` will pass them by reference, which will change the values in `self.enemies_data` if `self.enemies` is changed. We do not want this, because we want to be able to manipulate `self.enemies` (inflict damage, remove enemies) without changing `self.enemies_data` , so that it can be used later, maybe to reinitialize the level when it needs to be restarted.
  To do this, we need to add a method to the `Enemy` class called `Enemy.copy(self)` , which simply returns a one-to-one copy of the enemy with the same parameters. Adding this enemy object to `self.enemies` will literally copy the properties of the `Enemy` object in `self.enemies_data` to an object and place it in the `self.enemies` array, giving us a carbon copy without accessing the same object.
  Finally, the UI elements can be constructed. Currently it only consists of the player's health bar, but in the future, this would be where other UI elements need to be initialized, like a close-up portrait of the player character, text indicating how many enemies are left in the level, control schemes/tooltips, or maybe a dedicated pause button. Lastly, `self.game_instance.game_state` is set to `"play"` to give the player control and start updating the level.
- `update(self, screen):` The update method that gets called every frame when `game_instance.game_state == "play"` . This method updates the enemy and player objects, UI elements like the player health bar and text, and checks the win condition by checking if the last frame of an enemy's knockdown animation have been shown for all enemies in the level. If this condition is true, that means all enemies have been defeated *and* have finished their knockdown animation, allowing the player to continue to the next level, or the credit roll if it's the last level in the game. Conversely, player health is also checked, and if it is zero, then `game_state` is changed to `"game over"` , which will stop updating the level and show the Game Over screen. After everything is updated, they are all shown to the screen by calling `camera_group.camera_draw(self.player, self.map, [self.health_bar])` . An array is passed for UI elements as there might be a possibility that more UI elements will be added other than the health bar. Finally, `pygame.display.update()` is called to update the screen and reflect changes made.

Map

This class manages the Surface and Rect object of the background image so that it displays correctly. This involves scaling the image correctly so that the image fills up the height of the screen, and creating a separate Rect object called self.traversable_rect, which serves as the rectangle that clamps the player rectangle. This will not allow the player to go outside its bounds. For example, the player may only roam in the rectangular area in the bottom 200-300 pixels of the map. This is achieved by getting the size of the map after it is scaled to fill the height of the screen, and then adjusting the width of self.traversable_rect to be the same as the width of the scaled background image, and the height to be as specified by the level data. Then the top argument is calculated by subtracting the height of the traversable rectangle from the height of the screen.

There are no methods in this class.

CameraGroup(pygame.sprite.Group)

This class handles the drawing of all sprites on screen. It inherits `pygame.sprite.Group` , which is a utility class that simplifies handling sprites like `Player` and `Enemy` objects. Upon initialization, the object will declare a `self.screen` to give its methods access to the display object without passing it every time. Then, a "camera box" is created. Essentially, this is used to move the "camera" when the player goes too far right or too far left of the screen. The parameters are its boundaries `self.camera_boundaries` , which is a dictionary containing keys `"left"` and `"right"` , containing values that correspond to the number of pixels of space that should always be on the left or right of the player, except for when the player is at the leftmost or rightmost section of the map. This is set to 200 pixels to have enough room so that the player can have time to react to enemies from the right, but not too much that it's too easy to anticipate. A `Rect` object is made using these parameters, and a "camera offset" is defined as a `Vector2` for later use.

Methods:

- `move_camera(self, target, map)` : This method "moves" the "camera" to the right or left depending on which side of the camera box the player is hugging. If the player hugs the left side of the camera box, the "camera" moves left, and if the right side is hugged, the "camera" moves right. An exception is if the player is at the leftmost or rightmost 200 pixels of the map, in which the camera stops moving so that it doesn't show the black space unoccupied by the background image.
  I used quotation marks when talking about this camera, because in pygame, there isn't a literal camera that can be moved intuitively. The camera mechanic is done by using relative velocity, which means instead of following the player around, the camera is actually moving everything else in the opposite direction relative to the player, which will give the illusion of a camera moving with the player. This is why `self.offset` was defined in the constructor, because it will be used to display all other elements other than the "target entity" (which in this case is the player).
  I also put quotation marks around "moves" as the method really only sets the correct offset value, and this value is what is then used by the draw method to display surfaces in the correct position. This is done by setting the x-axis offset to the leftmost pixel of the camera box

subtracted by the left boundary space (in this case, 200 pixels). Then when the camera box is moved by the player, the relative positions of all other surfaces will also change according to how much the camera box has moved.

- `camera_draw(self, player, map, ui_elements)` : All this method does is draw. First it fills the screen with bright green so that it's easy to tell when a bug is in the draw method or game assets. This will get covered up by the map background, so the color will not show when the game is working correctly. Then, `self.move_camera(player, map)` is called to get the correct offset. This offset is then used to display the background image. For entities, another type of offset needs to be added, as the top-left pixel of the PNG file used for the entities don't correspond to the top-left pixel of their rectangles. `entity_offset` ensures that the hit detection and movement boundaries are somewhat accurate. The shadow image also needs a little adjustment after scaling, and is blit before the entities so that they appear behind their feet.

Classes in entities.py

This file contains all the classes that relates to the characters on the screen. Currently it contains `Player` , `Enemy` and their dependencies, like animation classes and the `Attack` class.

LinearAnimation

This class shows an animation that consists of a series of frames that do not loop. After the last frame is shown, it will hold that frame until the sprite is either killed or no longer shown. Upon initialization, it declares `self.actor` for context, sets `self.frame_index` to `0` , and loads the frames specified by the `frames` parameter, which is an array of `Surface` objects of the animation frames. The `convert_alpha()` method is then used to correctly show transparent PNG, and `self.flipped_frames` is defined by horizontally flipping the frames so that the characters can face left and right. The period of the animation and speed is also defined here.

Methods:

- `animate(self)` : This method sets the frame that should be displayed by changing `self.actor.surf` to the correct frame. If the last frame is shown, `self.last-frame_shown` is set to `True` , and `self.frame_index` is reset to 0.

LoopingAnimation(LinearAnimation)

This class inherits LinearAnimation, and its initialization is just a super function of its parent.

Methods:

- `animate(self)` : Instead of stopping when the last frame is shown, this animate method continues animating until another animation is selected.

AttackAnimation:

This class does not inherit any other animation methods, as it works in a more complicated way. Its frames are put into three stages: windup, linger and cooldown. As the names suggest, windup corresponds to the stage of the animation where the character prepares to throw a punch, kick, or any other attack. Linger are the frames where the punch or kick lands and stays for a moment, and cooldown is when the character recovers from throwing the attack, going back to the idle stance.

Methods:

- `animate(self)` : This method changes the actor surface to the correct frame. Unlike LinearAnimation, this method selects frames depending on which stage the player is in. Additionally, `self.frame_index` is not incremented here, but in the player object's `Attack` instance.

HitterBox

The odd name is to differentiate it from the commonly known term "hitbox", which refers to the area or volume of a character that is prone to hits/attacks. `HitterBox` , on the other hand, is the collider that is responsible to *inflict* damage instead. Upon initialization, it assigns `parent` as its parent object/actor, `dimensions` for how big the collision zone is, `attachpoint` for where the box should attach to the actor, and `linger` for how many frames it should stay active.

Methods:

- `update(self)` : Increments `self.time_on_screen` by 1, for use by parent object to kill it when it should be inactive.

Attack

This class is responsible to handle all entities' attack animation and detection. Upon initialization, it makes note of who is attacking in `self.attacker`, the reach of the attack (how far the attack can reach from the attachpoint) in `self.reach`, the sweep of the attack (vertical dimension of the attack) in `self.sweep`, the damage it should inflict in `self.damage`, the animation it should execute in `self.animation`, and declares a variable that keeps track of whether or not an enemy has been hit in `self.enemy_hit`.

Methods:

- `queue(self)`: This method changes the state of the attacker/actor to `"windup"` to start the attack. It also assigns itself to `self.attacker.current_attack` so that methods in the `Player` object can execute the correct attack.
- `windup(self)`: Shows the frames of the windup stage. When the last frame is shown, `self.should_start_new_phase` is set to `True`.
- `linger(self, enemies)`: This method creates a `HitterBox` object according to the attack's paramaters and adds it to `self.attacker.child_objects`, or in other words, the child object array of the attacker. The next frame then checks if there is an enemy Rect that collides with the `HitterBox` object, and if there is, damage is inflicted to the enemy, and the state of the enemy is changed to `"hurt"`. This will trigger a hurt animation for the enemy. If the last frame is shown, `self.attacker.state` is changed to `"cooldown"` and this next phase is started.
- `cooldown(self)`: Similar to `windup(self)`, this method simply shows the cooldown frames until the last frame is shown, in which it will set the attacker's state back to idle to give it control over the character again and start the idle animation.

Player(pygame.sprite.Sprite)

This class handles everything that has to do with the player character, from what side it is facing, the image surface, Rect, state, current attack, health points, movement variables, and child objects. The animations for each state are also initialized here, by loading the game assets and passing them into their respective animation class constructors. Finally, the attacks are intialized using the `Attack` class.

Methods:

- `update(self, enemies, map)`: This method updates the player's movement variables based on the keys being pressed, executes the correct animation based on which state is active, and updates its child objects.

Enemy(pygame.sprite.Sprite)

This class is largely similar to the `Player` class. Future refactoring might see the two classes inherit from a parent class called `Entity` or something related. The difference is the damage for `Enemy` objects are fixed as they only have one attack, and is specified when the object is created with the constructor. They also have a knockout animation, which the `Player` class doesn't have. This is simply due to the fact that there was no knock out animation for the player character in the asset pack that I used, but they can be replaced in the future.

Methods:

- `copy(self)`: This method copies the initial position and damage points of the object and returns an `Enemy` object with those parameters. This is useful for copying objects in lists so that the originals don't get changed.
- `update(self, player)`: This method checks if the player is inside the enemy's detection zone, and if yes, approaches the player. Once the enemy is close enough, it will try attacking the player. If it gets hit, its state will change to `"hurt"`, which will trigger the hurt animation, or the knockout animation if its health is zero or lower.

Classes in ui_elements.py

Menu

This class is the boilerplate for all the menus in the game, which are the start screen, pause menu, continue menu and game over screen. Upon initialization, it gets the display surface object for context, loads the menu background image, and centers the rectangle used to position the menu. It also initializes the rectangles for each menu item.

Inner classes:

- `MenuItem`: This class is defined to manage variables of the menu items, like the text, their positions, function to execute, and any function parameters, if any.

Methods:

- `update(self)` : This method updates and shows the menus. It first gets the mouse position and state of mouse buttons, then blits the background. Items are then displayed with their respective positions, and if the cursor is hovering over them, the alpha is decreased and the blit position is moved slightly down and to the left, for a hover effect. It sets `item.pressed` to `True` when MB1 is pressed, and when it is released with `item.pressed` set to `True` , a "release" is detected, and the function is executed. This approach will execute the function at button release instead of click, and will let users cancel their click by dragging off the button before releasing the mouse button if they change their mind. Lastly, the display surface is updated to reflect changes. Note that this update function is separate to `game_instance.camera_group.camera_draw()` , because they are used in different states. This is why the display update needs to be done here.

HealthBar

This class manages the health bar of the player on the top right. Initialization will take note of the maximum value of the player's health, then a maximum width and fixed height is set for the bar, and the top left position for the bar is set. A font object is also created for the text detailing the value of health left.

Methods:

- `update(self, new_value)` : This method takes the new value of the player's health to update the health bar's length. To avoid a negative length, which will result in an error, the `max()` function is used to set the value to 0 if it was smaller than 0. A new surface is created to reflect the change, with the length equalling the portion of health the player has left multiplied by the maximum width. The surface is filled with a bright green color. As for the text, the value is updated, and the color is determined by whether or not the amount of health left is larger than 20. If it is, then the color is white, and if not, it would be red. The text is then rendered with these parameters.

RollingScreen

This class is used to display the rolling credits after completing the final stage of the game.

Methods:

- `update(self, game_instance)` : Fills the screen with black, displays the items vertically, updates the position of every item, and lowers the opacity slightly each frame until it is zero after the last item gets close to the top of the screen.

Main file

The main file contains two global-scope functions, `quit_game()` and `main()` , the latter of which gets executed when the file is executed with `python main.py` in the shell.

Functions:

- `quit_game()` : Quits pygame, closes the window and exits python properly.
- `main()` : This function is the main script that gets executed when the game is launched. First of all, it initializes the global variables by running `setup()` . These variables include the game instance, menu specifications, credits, the clock and FPS the game will try to run at. Then, it enters a `while True` loop.

The `while True` loop

First, `clock.tick(frames_per_second)` is done to keep the FPS in check. Then, events are checked for keypresses and quit requests. The keys checked are Escape, W, A, S, D, J, and K. Escape will bring up the pause menu while in the `"play"` state, WASD keys function as the player's directional movement keys, and J and K executes light attack and heavy attack, respectively.

After checking for events, the rest of the loop simply updates whichever menu or process is appropriate, according to the `game_instance.game_state` variable.

Known Issues

The following are known issues with the game as of May 7th, 2024.

- Player punch/kick hit detection can sometimes be spotty. The player can also only hit one enemy at a time. The variable that checks if an enemy has been hit and stops the attack from inflicting further damage should be moved to the enemy object so that it is only true for each enemy, not for the player.
- Classes `Player` and `Enemy` have similar contents. Refactoring to inheritance from a parent class might be the better way of writing them.

- A cleaner solution than the current `setup(0` function might exist for initializing the global variables in the main file (menus, clock, FPS, game instance, etc.)
- A more accurate collision detection system using masks will get rid of moments where the player is hitting the air slightly in front of an enemy's face and the hit still registering. Currently it's using rectangles and this is unfortunately unavoidable.
- Enemies' player detection can be jittery sometimes, where they can flicker from facing left to facing right rapidly when the player is close to the enemy in the x-axis but not in the y-axis, and is in movement. The logic for player detection and approach should be rethought.