**AICore** – Defining movement and AI for monsters, 71 lines. *This is how the monster behaves. Depending on your level of creativity, you can do a lot with this, including using different cores for different moves and swapping cores in the middle of a fight to have the monster use a new and exciting move! At their base, each core will be used to influence the Movement of a monster. Because this movement is checked every frame, they have to be programmed such that the position is determined reactively.*

* **Class- AICore**
  + **\_\_init\_\_()***Initializes this AICore*
  + **setMovement(func):** *Sets the movement for this core to the passes function. Be careful, because this method needs to have both the monster it is acting on and the slime as formal parameters. If they are not present, the assignment will fail.* 
    - **update(slime):** *calls self.movement using this core’s monster and slime. This is called every frame by the main game loop when the update is called on the monster.*
  + **DefaultMovement():** *This is a placeholder designed before the Cluster was implemented. It’s the same as agressiveMovement. Probably won’t see a lot of use, but hey, I’m sentimental about the old versions of this program.*

**Anthology** – collection of all the Scenarios, 113 lines *These are all predefined scenarios. As long as these are added to the anthology list at the end of the file, they will show up as buttons on the main menu, and the scenario can be played. Defining a scenario is simple enough, please refer to the documentation for Scenario for that.*

* **Class DesertScenario(-> Scenario.Scenario)** *This scenario pits the player in a desert with two coyotes. The goal is simply to defeat the Coyotes without dying. Tumbleweeds are bouncing across the screen that will deal a small amount of damage to the player on contact, and they can be destroyed with a sword.* 
  + **\_\_init\_\_()** *Establishes the Scenario and it’s attributes*
    - reset() *Calls the \_\_init\_\_ function to reset the scenario*
  + **winCondition(horde):** *determines if the two wolves for this scenario have been killed, meaning the player has won. Sets self.win to true or false depending on what the state of the horde[] is*
  + **alldestroyed() [returns bool]***: determines whether or not all the setpieces in this Scenarios trove are destroyed*
* **Class PlainsScenario(-> Scenario.Scenario)** *This scenario has the player eating 5 cabbages without being defeated by the slow moving enemies on the screen.* 
  + **\_\_init\_\_()** *Initializes the Plains Scenario*
  + **reset():** *calls the \_\_init\_\_() function, resetting the Plains Scenario*
  + **winCondition(horde):** *sets self.win to true if all of the cabbages in the trove have been eaten.*
* **Class TestZoneScenario(->Scenario.Scenario):** *This is a “Boss fight” against omen, who has multiple moves and is generally more challenging to defeat.* 
  + **\_\_init\_\_()** *Initializes the TestZoneScenario Scenario*
  + **reset():** *resets this scenario to it’s defaults*
  + **winCondition(horde):** *sets win to true if Omen has been defeated.*
* **retrieveScenario(target\_name:str)** **[returns scenario]** *This is used by the main menu to return the scenario that is called. The Scenario that is called is based on the label for that scenario, so retrieveScenario looks at all of the Scenarios and returns the one that matches. Logically, it is not possible for a button to not be connected to a valid scenario, as the Label on the chosen scenario is what is displayed on the menu.*

**Armory**, used to define and update weapons and their positions when swinging, 172 lines. *This is maybe a bit ambiguous. This is not to be confused with Arsenal, which is a collection of predefined weapons. This is used to control the proper displaying of the weapons. When defining a new weapon, you can declare how wide the swing for that weapon will be and how much damage it will deal.*

* Class Weapon(->Sprite) *This is the base* 
  + **\_\_init\_\_()** *Initializes the weapon, giving it a rect and an image. It’s position is defaulted to (0,0)*
  + **setStats(self, power, arc)** *sets the arc (swing angle) and power of the weapon to the specified number*
  + **setImage(self, image):** *sets the image for this weapon to image, then sets the rectangle appropriately. This can be set to any image, so honestly, have fun, go nuts.*
  + **position(self, player):** *uses the direction of the player to determine where the sword swing should start. It uses the direction attribute of the player to decide how to swing, as well as whether or not the player is facing right. This is how the game can always have the sword swing in the proper direction without having to always define a different swing animation for each possible direction. A possible limitation of this is that the player is unable to strafe, that is, move in one direction while swinging the sword in a different direction. This is only called if the sword is being swung.*
  + **weaponPosition(self, player, SWINGTIME, elapsed, swingAngle, startAngle):** *Alright, big explain here. this is a method that will Uniformly generate the position and angle of the weapon based on its current angle and how far through it's swing animation it is. it directly modifies the passed weapons position information, changing the angle and the pos attributes. Pass it the player that is using the weapon, the global variable SWINGTIME, the amount of time elapsed (SWINGTIME - swingtime), the total angle that this sword swing will occupy(in degrees), and the starting angle(also in degrees). Each frame that the sword is still being swung, this calculates the correct position, further mitigating redundant code for each frame of the animation. This also works with any possible weapon size: because the animation is mathematically determined, each sword swing has the potential to look a little bit different.*
  + **Update(self, player)-** *Uses the position of the player in conjunction with weaponPosition to determine the current position of the sword and set it accordingly.* *This is determined globally, setting the position in relation to the screen, whereas weaponPosition only sets the sword position relative to the player. Updates for all displayed objects are called every frame, so it is necessary to specifically call this one Update.*

**Arsenal,** 21 lines, *This is not to be confused with Armory, which contains the class definition for the Weapon Class. This is simply a collection of predefined weapons and their properties. To save a lot of headache, there are three swords defined at the moment:*

*RedSword: Damage 15, arc 180, called REDSWORD*

*BlueSword: Damage 10, arc 180, called BLUESWORD*

*GreenSword: damage 25, arc 180, called GREENSWORD*

*At the end of this file there are the three declarations for these swords. To add them to a scenario, simply use Armory.NAME as the parameter for the weapon of your choice.*

**Atlas** – 26 lines. *This is a collection of some predefined backgrounds that can be used for scenarios. This is similar in function to Arsenal*

*All of the following methods build the background with their respective themes based on the current window size, then return the sprite group. To add them to a scenario, simply pass Atlas.NAME as the vista parameter. If you want to add a new background, make a tileably-shaped tile (the ones I use are 100 X 100 and I make them in Piskel) and save them to a folder with an appropriate name. From there, retrieve it from the list that is loaded in for this program and create a new variable that holds it, following the convention for the other ones.*

* GRASS()
* DESERT()
* LAVA()
* SPACE()

**Button,** 87 lines, *button is used as a homebrew button class. I ran into an issue where there are no default buttons, so menus are entirely difficult. I made this file to allow me to create new buttons. While it is pretty limited, it decidedly serves it’s purpose. Use this to add buttons to menus, or any other button you may want in the game.*

* **Class Button(-> Pygame.Surface)** *The class that holds the button. Each one has its own text, size, color, and position.* 
  + **\_\_init\_\_():** *initializes the button with no text, position (0,0) and the default color, which is an admittedly boring kind-of-gray.*
  + **modify( \_text="Click Me", \_height=20, \_width=100, \_pos=(0, 0), \_background\_color=(255, 255, 255, 255), \_label=""):** *This sets the buttons attributes accordingly, or defaults as listed. The listed defaults are the same way as the \_\_init\_\_() function sets them. This directly modifies an existing button, so call this immediately after button creation to customize buttons a bit.*
  + **hovered():** *This is run if the button is hovered, and slightly darkens the background color slightly by subtracting 5 from each RGB Component. This is just a nice and easy way to give visual feedback for a hovered button.*
  + **Quickrender():** *Counterpart to hovered, used to quickly color the background without changing it.*
  + **isHovered()[Bool]** *Checks if the position of the mouse is colliding with the rectangle of the button in question. I used this in conjunction with hovered() to make menus look a bit more reactive, and I also used it with clicked() to make the menus functional*
  + **Clicked():** *Called if the button is clicked, and sets is\_clicked to true and modifies the background color to be even darker than hovered.*
  + **Update():** *Checks and activates if the button is hovered and/or clicked. If the button is clicked, it returns the LABEL on the button as a STRING, and will otherwise will return an empty string, indicating that the button is not clicked.*

**Cluster,** 200 lines: *These host the movement instructions of various types. This one can get a bit messy, as almost all of the clusters are unique. Most of them are designed around an idea, or a goal movement that I try to emulate through the code. Most of them have pretty indicative names, but I’ll still do my best to explain them. They all work as AICores to directly modify the position of the object they are attached to.*

*Some of them are just specific moves, like lunge, which quickly moved the monster towards the player, and are meant to be switched in and out during an encounter.*

*These will almost always be called once per frame as the Update function for the monster is called. Keep that in mind as you make new ones, that this calculation may end up happening almost 60 times per second.*

*All of these set the movement for the attached core to the predefined core of the same name. They only serve to define constant movement for a monster.*

* **Class AgressiveMonement(->AICore.AICore):**
* **Class SimpleMonement(->AICore.AICore):**
* **Class ChaoticMonement(->AICore.AICore):**
* **Class TumbleweedMovement(->AICore.AICore):**
* **agressiveMovement(monster, slime):** *The monster always moves towards the player.*
* **chaoticMovement(monster, slime):***the monster will randomly move in one of 8 directions, at random intervals.*
* **tumbleweedMovement(monster, slime):** *the monster will constantly move to the left, and bounce according to the function. height width can be universally set. Basically, as the name implies, a tumbleweed.*
* **simpleMovement(monster, slime):***Similar to agressive monement, always moves towards the player. I defined this separately because if I ever want to change wither the simple or agreaaive movement, I am free to do so without losing the simplicity of this movement style.*
* **Stay(monster, slime):**
  + The monster doesn’t move. Literally just contains a *pass* statement.
* **printDirection(monster, slime):**
  + prints the monster current direction, used for debugging
* There are cores initialized as constants for all movement types.

Crypt, 72 lines. *Used to hold all the monsters created.*

* imgPath(image):
  + returns the current image path for monster images, with *image* appended to the end.
* Class WOLF(monster):
  + Used to create a WOLF monster, setting all of its attributes.
* Class TUMBLEWEED(monster):
  + Used to create a TUMBLEWEED monster, setting all of its attributes.
* Class AGGRABAGE(monster):
  + Used to create a AGGRABBAGE monster, setting all of its attributes.
* There is a list at the end of all currently initialized monsters.

Game, 176 lines. *This is the brain of the game, through which the menu and game are run*

* playScenario(scenario):
  + runs the scenario. Renders all the attributes of each scenario each frame, and checks for victory or defeat. Returns nothing.
* runScenario(scenario):
  + calls playScenario with *scenario*
* below this is the main game loop, calling the main menu while the user makes a selection.

GroundMaker , 87 lines *This is used to randomly generate the background for scenarios*

* class Land(->pygame.sprite):
  + init():
    - this just creates a land tile with an image and a rectangle
* class Plant(->pygame,sprite):
  + init():
    - similar to Land.init(), but this class is only used on the title screen for the grass along the bottom
  + update(): this is unused as an animation holder, where the grass was originally meant to blow in the wind.
* BuildTheLand(width, height, locale):
  + Locale is the folder to pull image files from. Beyond that, it will use the size of the screen to create a tiled background by randomly choosing background tiles and creating a larger surface out of them. It then returns the stitched tiles as a single surface.
* PlantTheGrass(width, height):
  + This uses the height and width of the screen to make random little patches of grass appear at the bottom of the screen, using a method similar to BuildTheLand. It then returns a single surface

HealthBar, 46 lines. *This is used to generate healthbars for enemies.*

* Class Healthbar(->sprite.Sprite):
  + *This class is the tedmplate for all the healthbars in the game, except the players.*
  + *\_\_init\_\_():*
    - this creates a healthbar using global defaults that is two sprites, one on top of another. The top bar is red, and shrinks to the left in proportion to the creatures current health.
  + noShow():
    - this toggles the visibility of the healthbar by setting a filter color.
  + Update():
    - Simply modifies the healthbar by checking the calling creatures current health.
  + Dump():
    - Shows the current total health and remaining health to the console, used for debugging.

Jukebox:

* JukeBox Currently has no implementation.

Main:

* Main is unused, as the game runs from Game. This is a remnant of the original program.

Meta:

* Meta is a text file containing basic global game info in a comma-delineated list.
* In order, the contents are:

*X-width of the screen : Y-Width of the screen : swing time for all weapons : framerate*

MonsterMash, 169 lines : *This one is important. It is used to define and control monsters.*

* Class Monster(->Sprite):
  + *This holds the current image of the monster.*

Overlay, 106 lines – This controls and updates the players health bar, will eventually control HUD

* Class PlayerHealthBar(->pygame.sprite.Sprite)
  + *The healthbar is shown at the top of the screen. The red portion is the amount of health that the player currently has. The fraction on the left side represents the current health over the total health.*
  + \_\_init\_\_():
    - Initializes the players healthbar to be 1/50 of the screen +15 pixels high and 2/3 of the screen wide.
    - Sets the HPbar to a red, and the text on it to”0/0” (representing 0 out of 0 total health
  + Update(target):
    - Uses the targets health to resize the bar appropriately and change the text on the bar.
* Class Overlay
  + *This is meant to hold text that will be shown on screen. It is currently not used, and does not work.*
  + \_\_init\_\_():
    - Generates the object for the players health bar.
  + Update():
    - Updates the players health bar, and will eventually show text to the screen.
  + ShowText(text, color=(255, 255, 255), size=12, duration=100):
    - Uses the parameters to display text to the screen for the time given.

PropStorage, 139 lines *This is the place where predefined props are defined and held. To add them to a scene, pull them from here. Adding a setpiece to a scenario is easy, you just need to create a new instance of that setpiece and add it to the local scenario.*

* Class DefaultSetPiece(->SetPiece.Setpiece):
  + *Creates a tester setpiece. Which spawns a wolf every few seconds until destroyed.*
  + Reset(): calls the init() function to reset the setpiece to its initial conditions
  + wolfResetEnemy(): called init() on the wolf to return the wolf enemy to its default state
  + destroyDefaultSetPiece(): stops the setpiece from spawning enemies and marks it as passable
* Class Round\_Cactus(->SetPiece.SetPiece): *This is a class that generates a small, round cactus.* 
  + \_\_init\_\_(): Initializes the cactus. Cacti are small, not passable, and deal damage to the player when the player comes into contact with them
  + Reset(): calls \_\_init\_\_() to reset the cactus.
* Class Tall\_Cactus(->SetPiece.SetPiece): *this class generates a cactus that is 2x taller than the round cactus*
  + *\_\_*init\_\_(): Initializes the cactus. It is taller, not passable, and deals damage to the player on contact.
  + Reset(): resets this cactus to it’s initial state
* Class TumbleweedLord(->SetPiece.SetPiece):
  + \_\_init\_\_(): *Initializes the TumbleWeedLord. It is a small bug-shaped thing. While the prop itself does not do anything, it generates a small armada of tumbleweeds that move across the screen*
  + tumbleweedResetEnemy()*: this resets the tumbleweed lord to its default*
* Class Cabbage(->SetPiece.SetPiece): *A small, tasty-looking cabbage.*
  + \_\_init\_\_(): *initializes the cabbage. They are small, green, and round. When the player touches one, it gets eaten, healing the player.*
  + eat(): *sets the cabbage to passable, and the healing property is disabled.*

Scenario, *This is used to create every scenario in the game! For suck a short file, it really packs a punch!*

* Class Scenario: *this is the container for all the levels in the game, and they all share a few common properties:*

*an array of monsters, horde = []*

*an array of setPieces, trove = []*

*the background, vista = []*

*slimes original position, slimyPOS = (0, 0)*

*slimes weapon, weapon = Armory.Weapon()*

*slime himself, TheWanderer = SlimesDelight.Slime()*

*In order to create a new scenario, every one of these has to have something in it. There are no “defaults” so to speak. If no enemies are specified, none are added. If there is no slime declared, he wont be added. This is arguably the most important file in this project, as it acts as a backbone for every scenario, and allows the main game file to engage with the created scenarios.*

* **\_\_init\_\_():** *Again, another important file. Here are the attributes, and their defaults.*

**self.name = ""** *This is the name of the scenario, and this is the text that appears on the button on the main menu*

**self.horde = []** *It’s an array, and it holds the monsters. Even while the scenario is running, new monsters can be added in. each time an enemy dies, it is removed from the horde for that scenario. That is to say that if the goal for your scenario is for the player to beat all of the enemies on screen, when the length of the horde is 0, there are none left on screen.*

**self.trove = []** *The trove is all of the setPieces. In similar fashion to the horde, pieces can be added to or removed from the trove while the scenario is running.*

**self.vista = []** *The vista for the scenario does not change, though realistically it could. Use the defaults in the ATLAS file as the vista. The reason that the Vista is a list is because the ATLAS file returns the background tiles as a list of 100x 100 tiles, which are then displayed in order.*

**self.slimyPOS = (0, 0)** *Pretty straightforward, the pixel coordinates of the slimes original position. Changing this in the scenario does not move slime, only changes the very first position.*

**self.weapon = Armory.Weapon()**  *The weapon is defined at the beginning of the scenario, and can change in the middle of the scenario with some finesse. Simply look at the predefined weapons in ARMORY and choose one.*

**self.TheWanderer = SlimesDelight.Slime()** *If you really want to change specific arrtibutes about slime, you can modify this to create that. If you do change the stats, make sure that if slime is every reset that those stats carry over.*

**self.Win = False** *Every scenario should have a win condition, ive been told that it’s good game design. This can be literally any trackable measure. Defeating all enemies, breaking all setpieces, spending enough time in the level, moving in a specific direction, reaching a certain location, etc. It’s really up to you. The main game loop will look every frame to see if this condition is fulfilled, so keep that in mind. Note, as a default, this will be set to true if the horde is empty. So if you have defined a new scenario and it immediately wins, make sure that you either change the win condition or add an enemy.*

*There are three more defined attributes: instructions, isScreenText, and and screenText([]). These are all experimental, and they will eventually be used to display text on screen, but I haven’t gotten them to work.*

* **setTheScene(self, \_horde, \_trove, \_vista, \_slimyPOS, \_weapon, \_TheWanderer):***As you can probably guess, passing in the corresponding parameters here sets this scenes attributes. Define ahead of time all of your monsters, weapons, etc. then pass them in here. For instance, if you have defined monA, monB, and monC as Crypt.WOLF enemies, then passing [monA, monB, monC] as your \_horde parameter is perfectly valid.*
* **\_\_del\_\_():** *this deletes all the attributes. Not exclusively useful, but I used it while creating the scenario system to make sure that the scenarios were not causing a memory leak.*
* **winCondition():** *This is what the main game loop checks every frame to see if the player has won. This is initially defined to set win to true if the horde is empty, but there are endless ways to redefine it.*
* **Generate():** *This was used to generate a new version of the scenario, and not carry over any memory from prior iterations.*

SetPeice

SlimesDelight

StatusBlock

TestScenario

TitleSlide