# DONALD BREN SOLID
# DESIGN DOCUMENT

## —— Overview ——

Game Name:
Donald Bren Solid

Team Name:
The La-Li-Lu-Le-Lo

Names of Team Members:
Khoa Hoang, Patrick Zhang, Kha Tran, Vu Nguyen, Jeremy Chao

Game Overview:
Donald Bren Solid (DBS) is a 2D sneaking game with a top-down view. Although opportunities for the player to engage in projectile-based (non-lethal) combat will arise, the emphasis of the game is on sneaking past enemies rather than engaging them head-on. The game is set in an alternate-reality UC Irvine where its computer science professors live in Donald Bren Hall. The school has commissioned mercenary groups (armed with stun rifles) to guard their buildings because it makes the professors that live there feel safe against the legions of angry students who disapprove of the difficult projects assigned by them throughout the school year. The object of the game is to sneak through the floors of Donald Bren Hall. If the player is spotted by a guard, then all guards within the current vicinity will go into alert phase and zero in on the player to subdue them. In order to escape this alert phase, the player must hide (e.g., in the bathrooms) for a designated period of time. The goal is the 5th floor where Professor Frost's office is located, and the player must place a CD (i.e., an in-game CD) containing all of the data for their CS 113 project, which they failed to submit earlier in the day, into the professor's desk without him noticing. On the way to the 5th floor, the player will also encounter members of the KTP Unit, consisting of the ICS 31, 32, and 33 professors ((Kay, Thornton, Pattis) == KTP), whom they must subdue in order to advance to the next floor.

As one might have guessed, DBS is based off of the famous *Metal Gear* series of games. In particular, the graphics and gameplay mechanics will be influenced directly by the first game of the series, which was called *Metal Gear* and released in Japan on the MSX home computer in July of 1987. The character to be controlled in DBS will be none other than the beloved protagonist of several of the *Metal Gear* titles themselves — Solid Snake. That is, in this alternate reality, Solid Snake was once a student at UC Irvine. Character sprites for Snake and the guards will be taken directly from *Metal Gear* on the MSX, while sprites for the professors of DBH will be drawn by our team members using image editing software such as GIMP and Photo Shop. The code for the game will be written entirely in python using the pygame library.

## —— Game Specification ——

**Background Story:**
You play as Solid Snake who, at the time of the events of Donald Bren Solid, is a senior computer science major in his last quarter at UCI. Due to having been deployed on a top secret sneaking mission to the jungles of Vietnam for a vast majority of the quarter, he was incapable of turning in his final CS 113 project on time. As such, he has resorted to the desperate measure of calling forth his skills as a world-class infiltrator of military bases to sneak through the mercenary-ridden floors of DBH, lest he spend another quarter at UCI. He must make his way through to the 5th floor where his CS 113 professor, Dan Frost, resides. Although Frost has been regarded as one of the more personable professors at UCI, he is not to be crossed, for he has been known to shoot frost from his hands towards misbehaving students — a frost that would leave students with a cold for weeks on end, preventing them from doing any of their work for days. An encounter with Frost on the 5th floor could spell disaster for Snake's future unless he does battle with the professor and, in the process, can find a way to wipe clean the professor's recollection of Snake having ever been there.

Initially armed with nothing more than a knack for stealth and an iron will to graduate, Snake will not only brave through floors of hardened mercenaries, but the powerful force that is the KTP Unit as well. Comprised of the introductory programming professors David Kay, Alex Thornton, and Richard Pattis, the KTP unit (perhaps with the exception of Kay) are known (in the game world) for their ruthless curriculum and have made many aspiring computer science majors drop out of the program in favor of more gentle academic pursuits at UCI — e.g., the humanities. Each member of the KTP Unit occupies a floor (starting from the 2nd) in DBH and is in command of the mercenaries on their assigned floor. In addition, they each hold keys to the floor above their own. If Snake is to make it to Professor Frost's office, he must certainly confront these three men that had made his first year at UC Irvine the most painful of his life.

**Characters:**
*Solid Snake*

. Determined to not spend another quarter at UCI, Snake has committed his existence to a dishonest act that will serve as his ticket to liberty. Snake is the only character the player will control.

*Dan Frost*

. Although he exudes a generally happy-go-lucky demeanor, underneath this spirited veil lies the ability to unleash a vicious frost attack against students exhibiting questionable conduct. He is also

known for having a powerful “shush” that knocks chatty students to the ground. Professor Frost is the final boss of the game.

*Richard Pattis*

. As a man of infinite wisdom, he spreads it by releasing volumes of reading material (mostly computer science-related) that those who come into contact with can’t help but read — even if it means compromising their own physical safety. Professor Pattis is the third boss of the game.

*Alex Thornton*

. A lover of dogs, U2, and othello. Carries around othello tiles in his pockets that he has become exceptionally skilled at throwing — an ability he had to develop to deal with un-attentive students at lecture. He is often seen with his dog Boo. Professor Thornton is the second boss of the game.

*Boo*

.The dog of Professor Thornton. Boo is a fast and agile creature with a fervent commitment to ensuring his owner’s safety. Boo is a companion to Professor Thornton during his boss battle. (Sprite taken from http://imageshack.com/f/585/catsdogs.png)

*David Kay*

. An all-around nice guy. He radiates a kindness that is simply painful to some students. Kay is the first boss that Snake faces.

*Mercenaries*

. Equipped with stun rifles that will make any trouble-doer think twice about their misdeeds, this mercenary group hired by UCI poses as a formidable group of foes. (Sprites taken from https://www.pinterest.com/pin/308215168219125496/)

**Rules and Mechanics**

*Main Menu Screen*

. The options available to the player here are “Play” and “High Scores”

*View*

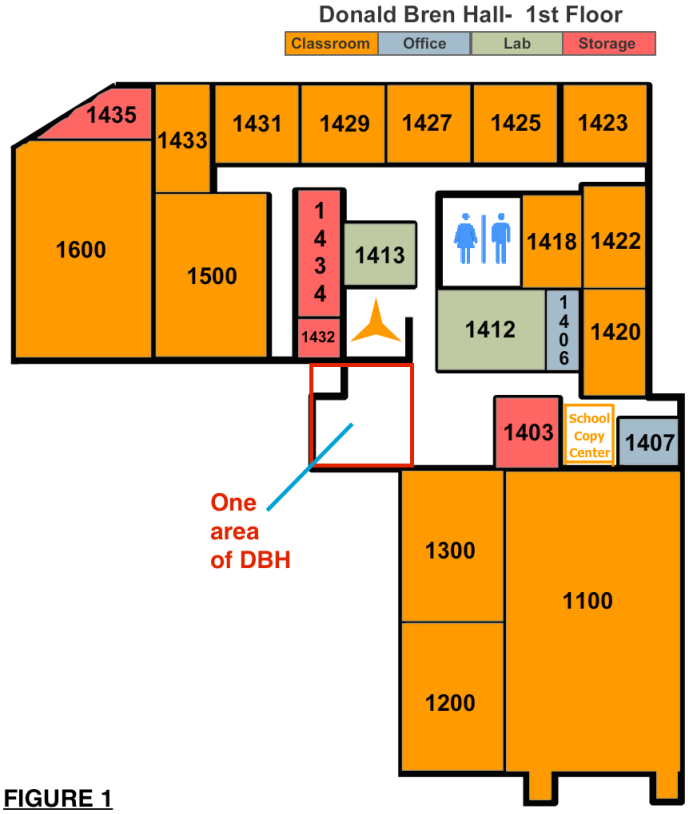



**FIGURE 1**

Snake will move through a 2D world, which the player observes from a top-down perspective. Conceptually, the world is simply floors 1 through 5 of the DBH building. However, only particular sections of the floors, which we’ve called ***areas***, will be drawn onscreen at a time. One may visualize an area as a square portion of the Donald Bren floor plan. For instance, labeled in figure 1 to the left is a single area of the first floor of DBH. The shift from area to area will not be achieved through the scrolling of a top-down camera; rather, movement through the world will be represented by transitions between areas. For example, to get from some area A to area B, Snake will have to walk to a designated transition section, which we’ve called ***doors*** (even though

they might not be depicted as actual doors in the game), within the current area. Once this occurs, the screen will instantly be filled with the contents of area B with no scrolling whatsoever. Areas that are adjacent to one another are deemed ***neighbors***. Snake cannot move between two areas that are not neighbors. Snake moves between floors by going up the stair room (the elevator will only work on the fifth floor).

*Movement*
Snake and all NPCs will only be able to move in the four cardinal directions — north, east, south, and west. Movement is controlled by the arrow keys. Jumping or crouching cannot be performed. Snake is only capable of moving at a constant speed, and mercenaries will move slightly slower than him. However, while in alert phase, mercenaries will be able to move slightly faster. Bosses may have a faster rate of movement as well, and Boo the dog will for sure have an added spring in his step. Bosses, as well as mercenaries in alert phase, will by and large be programmed to move in Snake's general direction. Snake is free to move anywhere in one area as long as there isn't an obstacle in his path. Mercenaries in each area will have a predefined patrol route that are adhered to while they are not in alert phase. There will not be mercenary patrols for every area — i.e., some areas will be free of mercenaries. Moreover, there will be surveillance cameras in certain areas that scroll vertically or horizontally along the wall in which they've been placed upon.

*How Alert Phases are Triggered:*
- Colliding with a mercenary while he is on patrol
- Coming into the line of sight of a mercenary
- Coming into the line of sight of a surveillance camera.
. Alert phases subside if the player hides in the bathroom, hides behind an object in one of the rooms on a floor, or initiates a boss battle

*Interaction With The World*
Snake can move through the world but will have no actions that can produce any visible effects on it. Physical objects will block his path and only doors (i.e., transition points) will be able to be walked through. Whatever items may be lying around can be procured upon coming into contact with them.

*Items*
Listed below are items that can be obtained:
- Weapons: banana peels, straws (for shooting spitballs), rubber bands, water balloons
- Ammo: spitballs (for the straw. To get ammo for the other weapons, just pick up more banana peels, rubber bands, or water balloons)
- Health Power-ups: bowls of instant ramen
- Extra Lives: a job offer from Google (only 1 exists throughout the whole game)
- CD with CS 113 game data: Snake starts the game with this item and must equip it in front of Frost's desk
- Miscellaneous: cardboard boxes (MAYBE. For hiding from NPCs), source code documents for Metal Gear Solid V: The Phantom Pain (these will raise the player's score. 5 exist throughout the whole game), iPod and iPod songs to collect (MAYBE)

*Inventory*
Players can view Snake's inventory by pressing the M key. Weapons that Snake has obtained, along with their ammo count, will be displayed in the inventory. In addition, the amount of source code documents collected (out of 5) will be displayed.

*HUD*
Snakes health, current weapon, an indication of whether or not he has a bowl of ramen equipped, and number of lives will be displayed in the HUD. Snakes health consists of 100 "hit points".

*How Enemies are "Taken Out":*
No one is ever killed. If anything, they are "subdued". That is, after being hit with one of Snake's weapons by a certain amount, they will become unconscious for a designated period of time. Mercenaries can wake up within seconds of being subdued, but bosses will be subdued for the remainder of the game once you've defeated them (the case is a bit different for Frost. See *Boss Fight Descriptions*). Boo the dog cannot take damage and will only be subdued once Professor Thornton is subdued. However, if you try to attack Boo, then your score will decrease (see *Scoring System*)

*How Boss Fights are Triggered*
Kay, Thornton, and Pattis will be assigned a particular area on a floor — namely, in an area that represents a room on that floor. The whole fight will take place in that room. For Frost, entering the 5th floor will trigger the battle. The battle with Frost will take place throughout the entire 5th floor.

*Boss Fight Descriptions*
. Kay: Will simply chase you around the room looking to hit you with his radius of kindness
. Thornton: Will stay fixed at the top of the screen moving left and right while shooting othello pieces at you. At the same time, Boo will be chasing you around the room trying to run into you.
. Pattis: Will chase you around the room tossing reading material at you, which has an immobilizing effect. If they don't hit you, they end up on the floor and you can become immobilized if you come into contact with them. Pattis will then proceed to run into you. Readings can disappear if you shoot them.
. Frost: Will chase you around the 5th floor shooting big beams of frost at you. No other weapons except banana peels will take away his health (story-wise, the idea is that he slips, hits his head on the floor, and forgets that you were ever there). You can only slip your CD into his desk once he's been subdued, but if you take too long to get to his office, Frost will wake up and fight you again.
. As an added layer of suspense, health bars will not be displayed for bosses

*Attacks and Damage For Snake*
At the moment, no precise mathematical descriptions for damage have been decided. Generally speaking, water balloons will do the most damage. Water balloons are thrown

and will trace an arc through the air. A step below water balloons are banana peels, then rubber bands, and finally spitballs. Banana peels are placable weapons while rubber bands and spitballs will travel in a straight line starting from Snake. If water balloons are used outside of boss battles, then an alert phase will trigger (they're considered noisy)

*Attacks and Damage for Enemies*

. Attacks on Snake will diminish his health by a certain number of units
. Mercenaries: running into you (5 units), firing stun rifle projectiles (10 units)
. David Kay: running into you with a radius of kindness that extends from his body (15 units)
. Alex Thornton: firing othello pieces at you (15 units) while Boo, his companion during the boss battle, attempts to run into you (5 units).
. Richard Pattis: throwing reading material at you (that traces an arc through the air and stays on the ground) that immobilizes you (because you're compelled to read it) for three seconds (3 units after done reading). While you're immobilized, the professor will run into you (10 units)
. Dan Frost: running into you (10 units), shooting beams of frost from his hands (20 units) that extends across the width or height of the screen, knocking you to the ground for 2 seconds with a "shush" attack (5 units), which then leaves you open to other attacks
. After an attack by an enemy is processed, you will have 2 second invincibility frame.

*Health Recovery*

Consuming bowls of ramen is the only way Snake can recover health through items. The health effect will take place if Snake presses space on the inventory screen or if he has it equipped when his health goes to 0. A single bowl will recover 45 units of his health. In addition, after defeating each boss, Snake will recover 50 units of his health, with the exception of Kay who will give you all your health back.

*Game Ending Moments*

If Snake's health bar reaches 0 while he still has lives, he will restart at the entrance of the floor that he was on when he was subdued. Else, it will be game over and Snake will have to restart on the very first floor. In addition, game overs can also be encountered if you are spotted by a mercenary on any floor after you've put Snake's CD in Frost's desk regardless of how many lives you might have had. Finally, you win the game by successfully sneaking to the door to the outside on the first floor.

Checkpoints

If Snake is subdued (by either mercenaries or bosses), he will be sent back to the entrance of the floor that he was subdued on

*Number of Lives*

Snake will start the game with 3 lives but can gain an extra life if you can find a job offer from Google lying around somewhere

*Scoring System*

A numeric score will be given once the player successfully completes the game (a negative score is possible). Although we have not decided on exact numbers, the following is a tentative list of gameplay actions that will increase/decrease your score:
. Increase: successfully sneaking through an area with no alerts, successfully sneaking through the entire game with no alerts, collecting Metal Gear Solid V source code documents, defeating a boss without taking any hits, a fast completion time.
. Decrease: subduing mercenaries, causing alerts, getting subdued, eating bowls of instant ramen, taking hits from bosses, if you battle Frost more than once

**Gameplay and Balance**
As mentioned in the game overview, the focus of the game is on sneaking past enemies rather than engaging them in battle (except for boss fights, which are unavoidable). The scoring system reflects this emphasis, as actively seeking a fight with the mercenaries will hinder your final score while striving for stealth will augment it. Weapons and ammo are scattered throughout each floor, and source code documents for the latest Metal Gear game (The Phantom Pain) will be randomly placed in a room on each floor (one document per floor). Such documents have no bearing on the story and are not mandatory to collect. They are simply there for fun and for increasing your score. Furthermore, the sole item for a one-up (represented as a job offer from Google) will also be randomly placed in some area on one of the floors.

The first floor will only have mercenaries for Snake to sneak past and the key to the next floor will be randomly placed in a room. Mercenaries will be placed in particular areas of the floor and will have a set patrol route. For added variety, patrol routes can change depending on which door Snake entered the section through. Such an arrangement for the mercenaries applies to all floors except the fifth where there will be no guards. Once on the second floor, Snake must confront and subdue Professor Kay (in a designated room) who serves as the guardian for the third floor key. Professors Thornton and Pattis assume similar roles as Kay's for keys to floors above theirs. Even after progressing to the next floor, the player is free to move about any area in any level of DBH. Upon entering the fifth floor, Snake will immediately encounter Professor Frost. This battle can span across the entire fifth floor, and Snake's CS 113 data CD can only be put into Frost's desk once Frost has been subdued. The player must be careful not to stray too far away from Frost's office, or else during the time it takes to walk to the office (or even as the player is walking out of there), Frost will recover and start another battle. There's an elevator key that you can pick up in the office so that you don't have to go back down floor by floor. As an added twist, if a guard on any floor sees you after you've placed the CD in Frost's desk, it will be instant game over. Snake will have successfully completed his mission once he makes it to the outside through the first floor.

**Music and Sound**
Select pieces from the soundtrack of the original Metal Gear on the MSX will be used in our game. "Theme of Tara", in particular, will serve as the primary sneaking music for this game. A playlist of the soundtrack can be found by visiting the following link:

https://www.youtube.com/playlist?list=PL1Y35eXe45afJClZfKZqs58Avq0H8VvWZ

Moreover, we will include an undecided assortment of 8-bit renditions of famous songs from the 60s, 70s, and 80s that can be played on Snake's iPod. As for the sound, we will try to find a source that offers sound effects from Metal Gear or from any other 8-bit action game.

**Artwork and User Interface**
For Solid Snake, we will be using the following frames of animation taken off of the sprite sheet from https://www.pinterest.com/pin/308215168219125496/

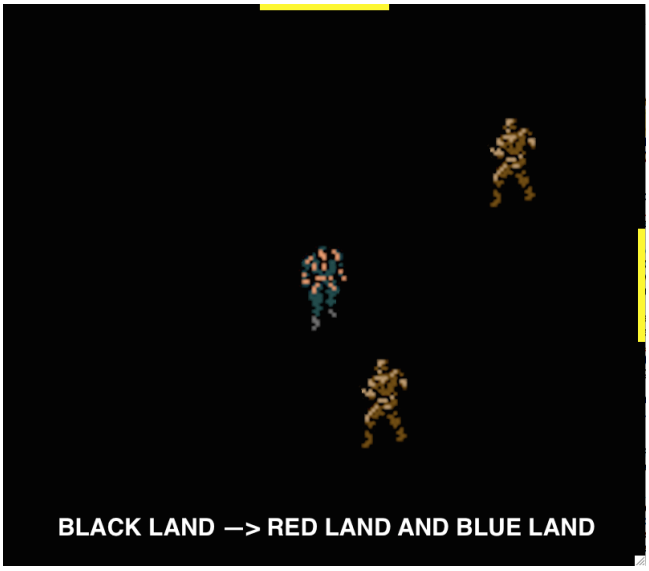
Also from the same sprite sheet, the following frames of animation will be used for the mercenaries and surveillance cameras:

And for Boo (http://imageshack.com/f/585/catsdogs.png):

All the other characters will be drawn using GIMP or Photo Shop. We will look continue to look for resources to pull sprites from for physical objects such as furniture.

Prototypes for areas and enemy patrols are provided below. Labeled on each area are their names and their neighbors (i.e., the areas that they transition to). The yellow lines represent the doors (or transition sections).





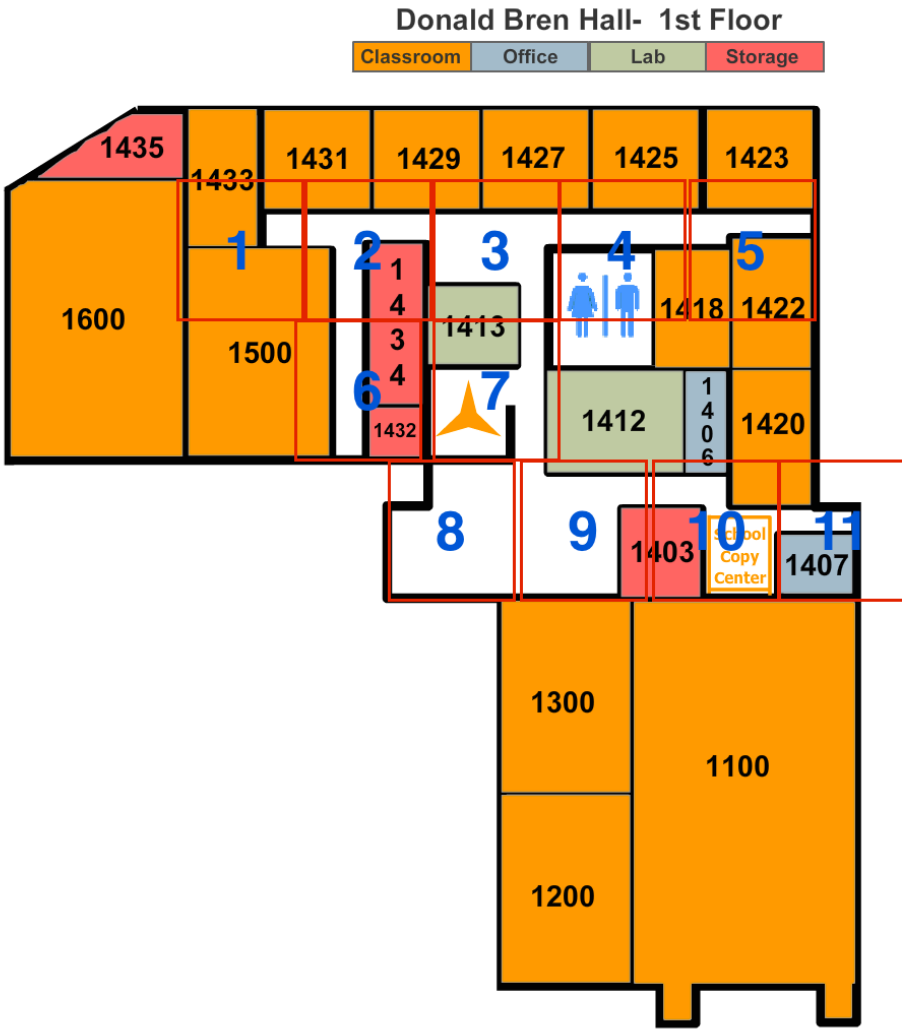


In the final product, each area will represent a square section of DBH with all of the peculiarities of that section such as couches, pillars, garbage cans, etc. However, we do

plan on taking liberties with the design in order to add to the gameplay, such as extra crates to hide behind in the more open areas of a DBH floor.

**Levels (or in our case, Areas)**
Below is our vision of how the main hallways and open areas of DBH's floors will be divided into areas. We've only included diagrams for the first and second floor because fortunately, the layout of DBH is virtually identical from the second to fifth floor, so we've only included a division of the second floor to represent the others above it. Only some rooms on each floor will be accessible (e.g., all the bathrooms for sure).

*1st Floor*

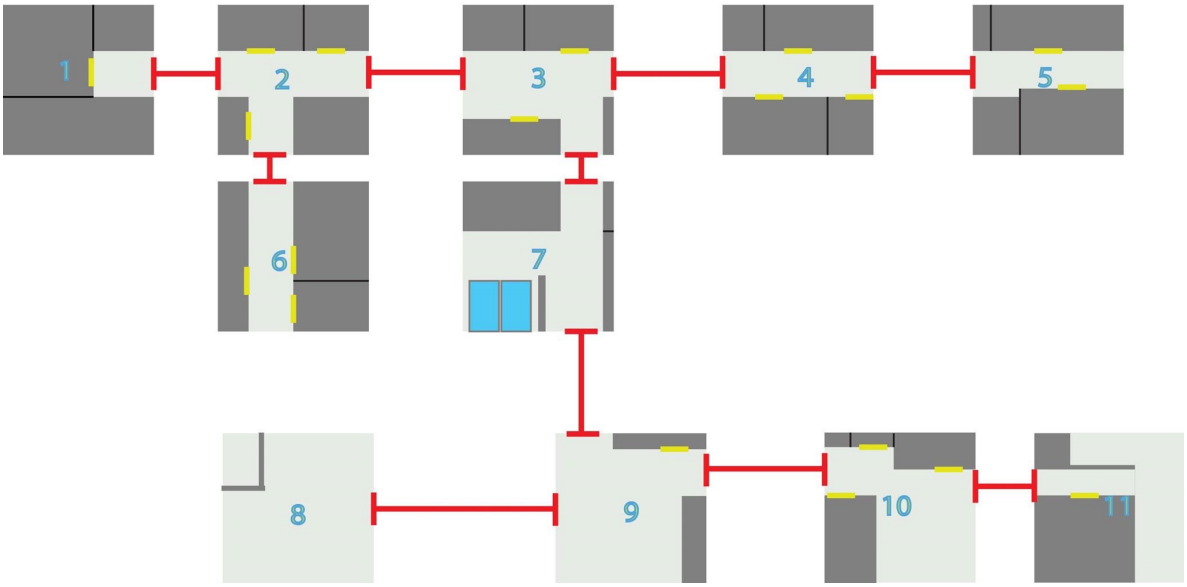


*2nd Floor*

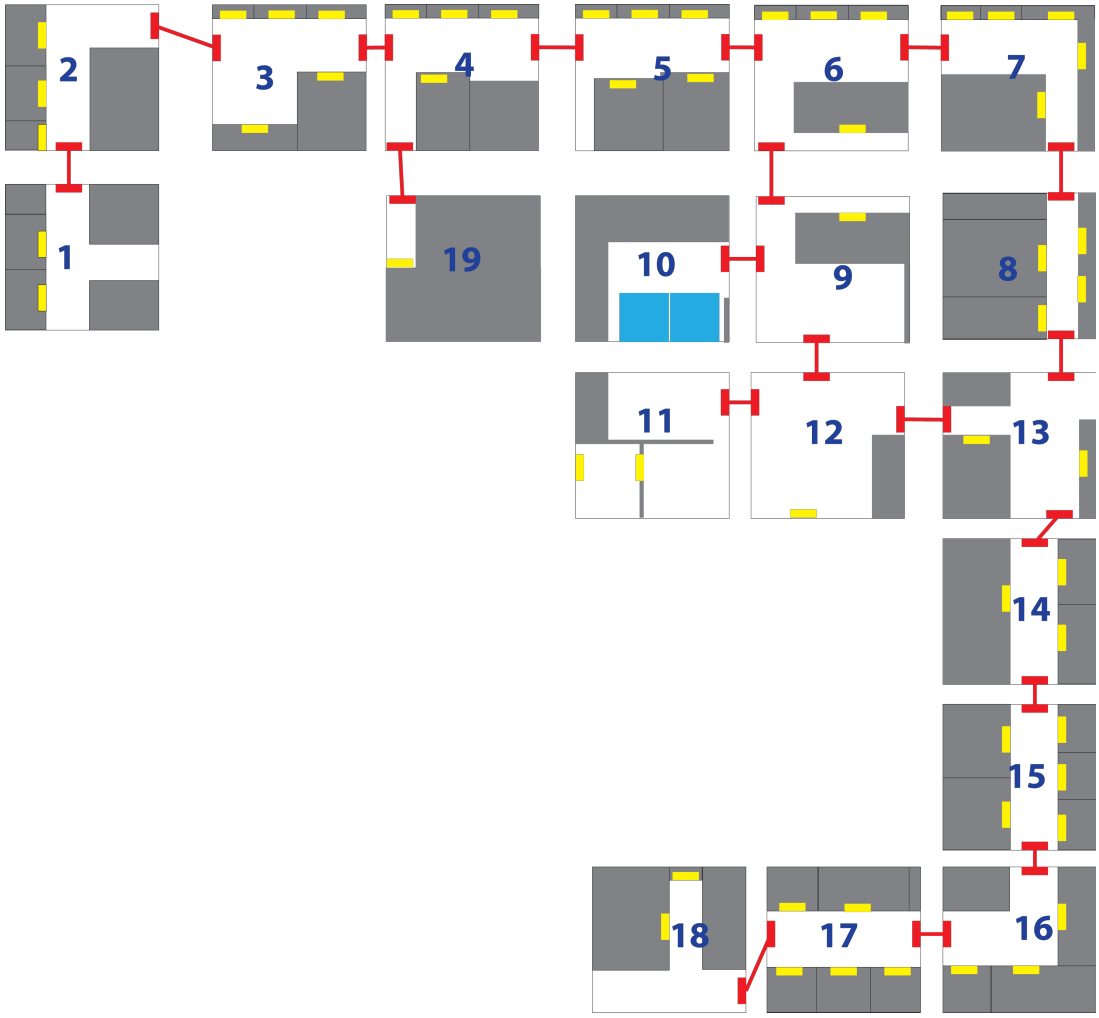

Note how some areas aren't the same size as all the others, but this was really only for the sake of specifying the span of an area. We will design the areas as though they were all the same size in the game. Moreover, the labeled numbers here do not represent the order in which the player will visit areas. Players are free to roam the entire floor and will enter floors 1 and 2 through areas labeled 8 and 11, respectively.

Below are sketches of how the areas will look and the connections between each area. The number labels correspond to the areas specified in the area division diagrams above. The yellow lines in each area represent doors that may or may not be accessible. Doors that are accessible will lead to another area, though such areas have not been depicted in the following diagrams.

*1st Floor*




*2nd Floor*



## —— Technical Specification ——

*PROGRAMMING LANGUAGE AND LIBRARIES*
Donald Bren Solid is written entirely in Python using the pygame libraries.

*TARGET HARDWARE AND OPERATING SYSTEM*
Any of today's processors are more than good enough to handle what Donald Bren Solid will demand from them, and the game will work well for any operating system.

*GAME LOOP*
Our game implements the standard pattern for a game loop. Written below pseudocode for this established pattern:

```
## Game Loop ##
while game_is_running:
     ## Event Loop ##
     for event in events:
          event_handler(event)

     update_sprites()

     logic_tests()

     draw_everything_to_screen()

     update_screen()

     delay_framerate()
```

. **Event Loop:** pygame facilitates the process of capturing events with pygame.event.get(), which essentially gets all of the events (e.g., mouse clicks or key presses) that have taken place in the current iteration of the game loop. The for loop runs through all of these events and processes them accordingly. For example, if the up key is pressed, then Snake will move north 5 pixels (more on this later in *Classes*). It is relevant to note that the if statements associated with the event handling for Snake are all contained within the Snake class itself rather than directly in the game loop. We decided on this arrangement because we wanted all data processing that is particular to Snake to be contained in a Snake object.
. **Update Sprites:** The states of all relevant Sprite objects are updated. For our game, we update the states of Snake and the enemies in the current area. Note that updating sprites does not entail a visual change of the sprite to coincide with the update. The visual change is made in draw_everything_to_screen().
. **Logic Tests:** In response to particular updates in state, particular things in the game world can occur. For example, checks for if Snake has entered a transition section, which prompts a transition to a new area, would be carried out here

. **Draw Everything to Screen:** When sprites are updated, certain updates must be visually expressed. Sprites are redrawn here in response to those updates. For example, if an update to Snake involved moving him 5 pixels south, then Snake will be redrawn in a position 5 pixels south of where he was in the previous iteration of the game loop. It it important to note that before all sprites are redrawn, the game display is "wiped clean" by, for example, filling the screen with white (if the game background is white) so that drawings from previous game loop iterations don't stick around. The illusion of movement in this game is thus achieved by constantly "wiping clean" and redrawing sprites.
. **Update Screen:** pygame requires the programmer to call pygame.display.update() in order for all the drawings/redrawings of sprites to take form.
. **Delay Framerate:** pygame allows the programmer to adjust the frames per second of the game with pygame.time.Clock().tick(FPS). This tick() will delay the time between one iteration of the game loop and the next. That is, the call to tick() tells the game to sleep for the next 1/FPS seconds, which effectively limits the game speed to a frames per second equal to whatever value FPS is. The smaller the FPS value, the longer the delay while the larger the FPS value, the shorter the delay. For our game, we've settled with an FPS of 20.

*OBJECT INSTANTIATIONS FOR THE GAME LOOP*
. **Window Instantiation:** A non-resizable window with an initial width of 1000 pixels and a height of 800 pixels will be used for DBS.
. **Player and NPC Sprite Instantiations:** Snake and all NPCs (e.g., mercenaries and professors) will be instantiated and placed in their own specialized container types for Sprite objects called Groups. Groups are highly useful for processing collisions between other Groups.
. **Area Instantiation and Link Establishment:** Each area in the game, which will all inherit from a class called Area, will be constructed and the connections between them will be established. The connections between all Areas will be represented by a python dictionary (see the section *Area Transitions* for further details). Each Area object will have a Group of their own sprites, which will be specified in their respective __init__ methods.

*CLASSES*
Our game makes heavy use of object-oriented programming. For basically every element one sees in the game display, there exists a class to represent that element. The game is primarily built around the **Snake**, **Area**, and **Enemy** classes.

**Snake (inherited from pygame's Sprite class)**
Instance Variables
- **image:** represents the visual look of the Sprite object
- **rect:** the rectangle in which the Sprite "lives". This is a necessary attribute for any Sprite so that we may manipulate the position of the Sprite by coordinates on its rect
- **fired:** a boolean that indicates if Snake is in his animation of firing a weapon. Initialized to False.

- **fire_frame:** an int used as a counter that allows Snake to hold the animation for firing a weapon. Initalized to 0.
- **is_moving:** a boolean that indicates if snake is moving
- moving_north/moving_east/moving_south/moving_west: each are booleans indicating if Snake is moving in either of the cardinal directions. Each is initialized to False.
- **cur_frame:** An index into the list of animation frames associated with the direction that that Snake is moving in (see next instance variable). Initialized to 0.
- **north_imgs/east_imgs/south_imgs/west_imgs:** each is a list of strings that is the name of an image file used as an animation frame. north_imgs is initialized to ["north0.png", "north1.png", “north2.png”], and the others are initialized similarly for their respective direction
- **h_move:** an int that indicates the amount of pixels by which snake has moved horizontally since that last time he stopped moving. Initialized to 0.
- **v_move:** an int that indicates the amount of pixels by which snake has moved vertically since the last time he stopped moving. Initialized to 0.
- **health:** An int that will take on values between 0 and 100 that represents the health of Snake. Initialized to 100.
- **life:** An int that represents the number of lives Snake has. Initialized to 3.

Methods (underscores prefixing method names denote helper methods):
- **__init__:** constructor for a Snake Sprite that initializes all of its attributes
- change_move: changes the position of Snake through the game world as he walks through it. Responsible for making him start to move or to make him stop moving. This method contains checks to see if he is moving horizontally or vertically and increments the h_move and v_move attributes, respectively. Since Snake will always move at the same speed (as mentioned in the game specification), the amount by which each of the h_move and v_move attributes are incremented is a constant value (called SPEED) of 7. That is, each increment provides a 7 pixel movement from one iteration of the game loop (i.e., frame) to the next (if Snake is moving). If Snake is not moving, the the h_move and v_move attributes are set to 0
- **init_position:** initializes the position of Snake within an Area
- **update:** sets all of the necessary updates to the state of a Snake Sprite given an event (e.g., a key press). If statements specific to event handling for Snake are processed for the event that is passed to this function. E.g., if the up key was pressed, the appropriate event handler will be called to update his movement. Moreover, checks exist to see if Snake is moving. If he is, then his rect.x and rect.y coordinates (i.e., the top-left coordinates of his rect) are incremented by the h_move and v_move values that were set in change_move and the list of animation frames corresponding to the direction he is currently facing is stepped through. A check also exists to see if Snake has gone into the firing animation. We’ve set the firing animation for 3 frame, so as long as fire_frame is less than three, Snake’s image attribute will be set to the firing animation. Else, his image will be set to one that is associated with the current direction in which he is facing. Finally, collision handling is carried out — a task that is largely accomplished by pygame’s spridecollide() function. Snake may potentially collide with physical objects in the Area or Enemies. Handling the former is described in the next

method, while the latter is handled by decrementing Snake's health by 1 for as long as he is touching an Enemy.
- **_handle_obj_collisions:** sets the necessary attributes for when Snake collides with physical objects in an Area. The illusion of Snake being blocked by a wall is achieved by setting, if he is moving east into an object, his rect.right attribute to the rect.left attribute of the Sprite object he is colliding with. Collisions from other directions are handled in a similar fashion.
- **_animate:** loads an image for the Snake sprite based on what the value of cur_frame is. Each call to _animate() will increment cur_frame until cur_frame == 3 (since Snake has a total of 3 frames for any direction of movement), in which case it is set back to 0. This helper method is called inside the update() method.
- **_handle_snake_movement_up:** handles a key up event by setting is_moving and moving_north to True. Also sets movement in all other directions to false and calls change_move to move Snake up.
- **_handle_snake_movement_down/right/left:** Similar to _handle_snake_movement_up but varies depending on the key that was pressed
. **_event_handler:** a series of if statements (one for key down and another for key up) that determine what event is to be processed for Snake. Snake starts moving on key down events and stops moving on key up events. This helper method is called inside the update() method.

**Area**

. Area is a base class that all areas in the game will inherit from. Each area in the game will be named with the convention Area[floor_number]_[area_number_on_that_floor]. floor_number is of course the floor number and area_number_on_that_floor is an integer between 0 and however many areas there are on that floor. Listed below this description of the Area class will be a description of how subclasses of Area will behave. Before we discuss the attributes and methods of an Area, however, let us describe how the illusion of Area switching is achieved.

*Area Transitions*

One may visualize the set of all Areas in the game and the connections between them as a connected graph (i.e., from each Area, one may traverse to all the other Areas). In python, graphs can be represented by the dictionary data type. If we view Areas as nodes and their connections as edges, then we can have the keys of the dict be Area objects while the values are containers that store Areas that the key Area shares an edge with. The type of container we chose was a namedtuple where its attributes are strings indicating a direction (e.g., "north", "south") where a door is located on the border of an Area. There's also an attribute called "inner" for doors that don't lie on the edge of an Area, but within it instead. The instantiation of this namedtuple would look as such:

```
Neighbors = namedtuple("Neighbors", "north east south west inner")
```

Each attribute will be assigned an Area object that coincides with the direction in which said Area object is a neighbor of the key Area object. For example, say we have two

objects from two classes that inherited from Area — call them area_a and area_b. Let area_b be an east neighbor of area_a, which means that area_a is a west neighbor of area_b. Each has no other neighbors otherwise. The dict representing the area graph would be instantiated as follows:

```
area_dict = {}
area_dict[area_a] = Neighbors(None, area_b, None, None, None)
area_dict[area_b] = Neighbors(None, None, None, area_a, None)
```

Thus, if we wish to access the east neighbor of area_a, we would say area_dict[area_a].east. Similarly, to access the west neighbor of area_b, we would say area_dict[area_b].west. Checks for Snake being at a door are made in check_transition(), and if he is, the method will return the appropriate Area using the syntax just described and set this Area to the current one inside the game loop. To be more specific, when Snake goes to a particular section of the current Area, the current Area will redraw Snake in a position where he *would* be in the Area (i.e., neighbor) that you're transitioning to. The appropriate neighbor will then be accessed in the dictionary of (Area, Neighbor) pairs. That neighbor will become the current Area.This area_dict would be passed as an argument to an Area's check_transition() method. Lastly, it's important to mention that since an Area can have multiple doors to its neighbors, we will be including a few extra attribute names to the named tuple in the final product — e.g.,

```
#(Assuming that any Area will have at most 3 entrances to its neighbors)
Neighbors = namedtuple("Neighbors", "north1 north2 north3 east1 east2
                        east3 south1 south2 south3 west1 west2 west3
                        inner1 inner2 inner3")
```

The following will be descriptions of the instance variables and methods of an Area object.

*Instance Variables:*

. **obj_group**: a Group (i.e., a container type specifically suited for holding Sprite objects) containing all of the sprites representing physical objects that are contained within this Area.
. **width**: a float representing the width of the game display. Used in specifying the transition sections (i.e., doors) of the Area.
. **height:** a float representing the height of the game display Used in specifying the transition sections of the Area.
. **player_obj:** a Sprite object representing the controllable character, i.e., Snake. An Area needs a handle on Snake so that it can check if Snake has entered a door within the Area.
. **e_group:** a Group of Sprite objects representing Enemies that are contained within the area.
. **snake_group:** a Group that will store Snake.

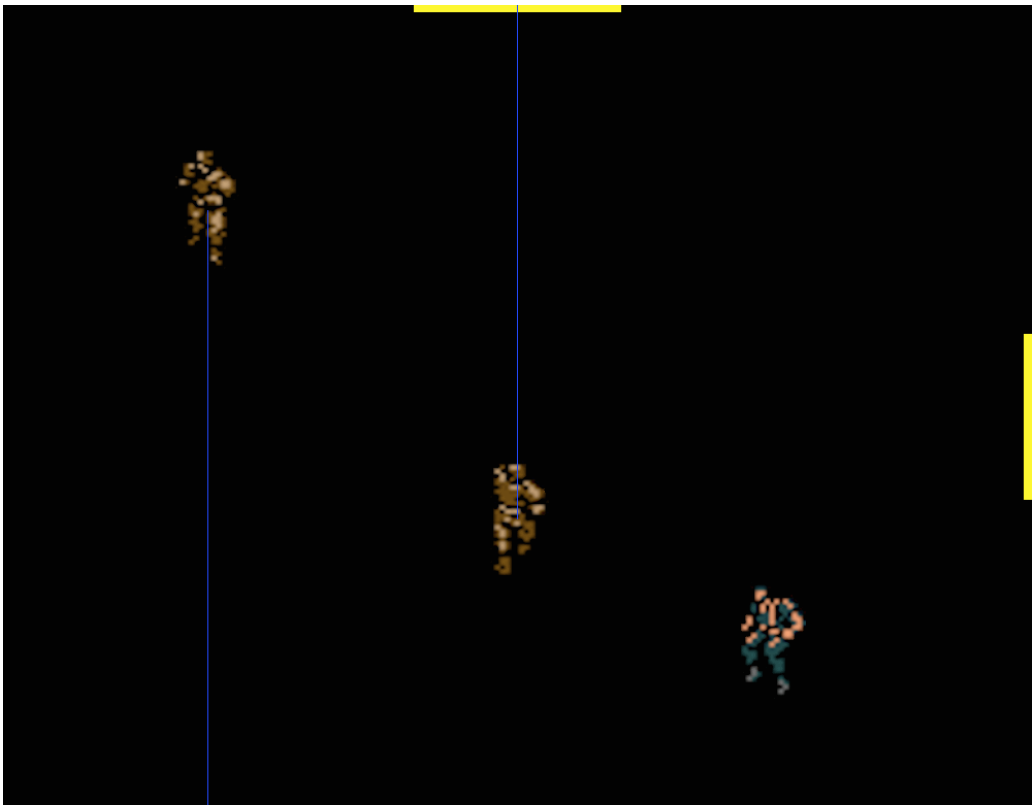
*Methods (underscores prefixing method names denote helper methods):*

. **update:** updates this Area's obj_group by calling update() on obj_group, which is a Group of Sprites representing the physical objects in this Area. Calling update() on a Group will have the effect of calling the update() methods belonging to all sprites in the Group. The update() method will be passed obj_group and snake_group so the appropriate collisions can be handled.
. **draw:** calls draw() on obj_group. This has the effect of calling draw() on all the sprites in obj_group.
. **draw_enemies:** similar to the draw() method but in this method, draw() is called on e_group.
. **check_transition:** a method that will be overridden by subclasses of Area. In these subclasses, check_transition will check if Snake passed through a door within the Area (note that not all Areas will have the same door locations). The checks are carried out by testing if Snake is within a certain interval in the Area. For example, if there is a door at the north edge of the Area, then the check would involve testing if Snake's top-left y-coordinate is <= 0 and if Snake's center x-coordinate is between 250 and 350. If Snake is indeed within a specified interval, then Snake's position will be re-initialized to where he would be in the new Area, which in this case could be, if we assume a game display of 500 by 500, at around (490, 250). In addition, the states of all Enemy Sprites in the new Area just transitioned to will be reset to what they were when they were first constructed and a formation for the Enemies will be set (in more detail in the **Enemy** class description)
. **_make_transition:** accesses the neighbor of this Area through the dictionary that represents the connection between all Areas
. **_handle_corners_and_boundaries:** checks if Snakes is at the corner or boundary of a screen by carrying out tests similar to the ones in check_transition but that correspond exclusively to the edges and corners of the screen.
. **update_enemies:** calls the update() method on all Enemy Sprites in this Area
. **_reset_enemy_state:** resets the state of Enemy Sprites to how it was like when they were first constructed. If Snake leaves an Area and enters that Area again, not resetting the states of all Enemies in that Area will cause them to engage in unwanted behavior when we go back there (e.g., walking off the screen)

*How Subclasses of the Area Class Will Behave*
In addition to overriding Area's check_transition function (described in the previous section), subclasses of Area will set patrol routes for individual Enemies within the Area with the helper method _gen_patrol_route[n] where n is an integer between 1 and the number of patrol routes we plan on providing to Enemies for an Area. Each _gen_patrol_route method defined in the class will be called in the method formation[n] where n is an integer between 1 and the number of formations we plan on establishing for the Area. In other words, a formation is set of patrol routes and patrol routes are assigned to each individual Enemy. As will be described in the next section about **Enemy** classes, Enemies will have a patrol_route attribute that is a list that stores instructions, essentially, for the patrol route.

**LOS (Line of Sight. Inherited from pygame's Sprite class)**

. One may imagine a LOS as a line that extends the width or height of the screen from some initial point (which we've called an eye_point). A visual depiction of this description has been provided:



The line was achieved by creating a pygame Surface object with either *a height of 1 for a horizontal line* or *a width of 1 for a vertical line*. If Snake passes through these lines (thereby having collided with them), then Snake has been spotted. The line will not be drawn in the final version as it is done so here.

*Instance Variables:*

- **image:** represents the "look" of the LOS, which has been set to be a very thin Surface object. The Surface constructor is passed a width and a height, which will then correspond to the width and height of the LOS.
- **rect:** the rectangle in which the Sprite "lives".
- **eye_end:** the starting point of an LOS, which is to be the x and y coordinates of an Enemy object corresponding roughly to where its eyes would be.

*Methods:*

- **set_position:** sets the rect.x and rect.y attribute of this LOS to the x and y values passed to this method. It is important to note how when an Enemy is looking east, then one can simply set the eye_end of this LOS to the location of where the Enemy's eyes would roughly be. However, if an Enemy is looking west, then the x-component of the eye_end must be shifted to the left (i.e., subtracted) by an amount equal to the height of the current LOS. This effectively flips the LOS over the Enemy's x-position so that the

LOS is properly established for a westward gaze. A similar procedure is carried out for when an Enemy is looking north, but it is the y-component that is subtracted.

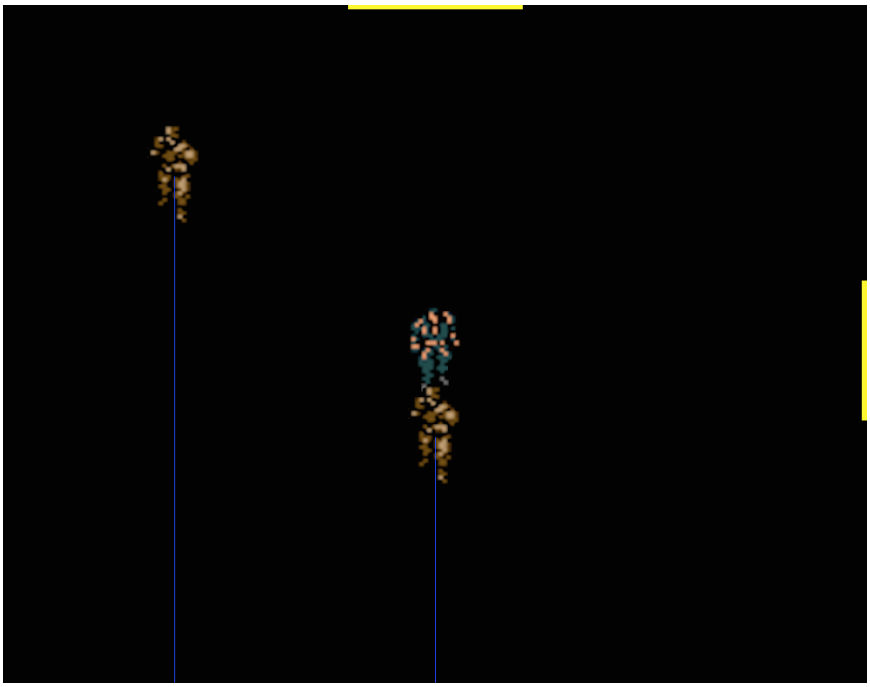
**Enemy (inherited from pygame's Sprite class)**
*Instance Variables:*
- **image:** represents the visual look of the Sprite object
- **rect:** the rectangle in which the Sprite "lives".
- **patrol_route:** the patrol route is represented by a list of 2-tuples. The entries in the 2-tuple are a string and an integer. The string is a movement, and the integer represents the units of movement (e.g., for walking, think of the unit of movement as a single step). For example, if you want to make an Enemy walk back and forth, you can set its patrol_route attribute to the following list:
`[("go_west", 50), ("go_east", 50), ("end", 0)]`
The ("end", 0) instruction must be present. The Enemy will repeat the patrol route after the "end" instruction. The patrol route is processed in an Enemy's update() function. As mentioned in the previous section, a subclass of Area will have methods that set patrol routes. When we write these methods, we will have the ability to program how a patrol route for an Enemy looks like. Initially, the patrol_route is an empty list
- **instr_index:** an index into the list of patrol route instructions
- **current_step_num/current_wait_num/curren_look_num:** ints associated with how many iterations of the game loop an Enemy will walk, wait, or look in a certain direction. Initialized to 0.
- **is_moving:** a boolean that indicates if the Enemy is currently moving. Initialized to False
- **looking_north/looking_east/looking_south/looking_west:** a boolean that specifies the direction in which an Enemy is looking. Initialized to False.
- **cur_frame:** An index into the list of animation frames associated with the direction that that the Enemy is moving in (see next instance variable). Initialized to 0.
- **north_imgs/east_imgs/south_imgs/west_imgs:** each is a list of strings that is the name of an image file used as an animation frame. north_imgs is initialized to ["enemy_north0.png", "enemy_north1.png"], and the others are initialized similarly for their respective direction
- **view_obstructed:** a boolean indicating if this Enemy's line of sight has been obstructed by Sprite object representing a physical object. Initialized to False.
- **in_alert_phase:** a boolean indicating if this Enemy is in alert phase, which is switch on when either the LOS of this Enemy or another Enemy collide with Snake. Initialized to False.
- **window:** a pygame Surface object that serves as the display for the game. This attribute is needed to configure the lengths of the LOS in cases where the LOS collides with Sprites representing physical objects.
- **los:** a LOS obect that represents the line of sight of this Enemy
- **log_group:** a pygame Group container that stores the los of this Enemy for collision processing

Methods (underscores prefixing method names denote helper functions):
- **init_position:** initializes the position of an Enemy within an Area

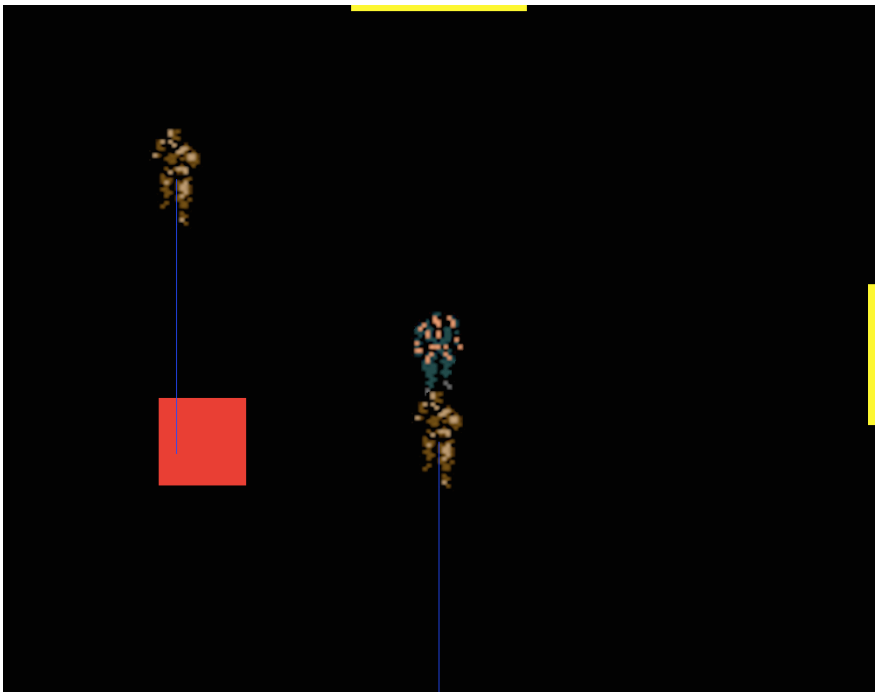
- **update:** processes an instruction in the patrol route of an Enemy. Initially, the first instruction in an enemy's list of instructions (i.e., patrol_route) is parsed. The first component in the two tuple (the movement) is set to a variable called movement and the second component (the units of movement) is set to a variable called units. The following is a list of all the movements an Enemy can "understand":
1. "go_east"
2. "go_west"
3. "go_north"
4. "go_south"
5. "wait"
6. "look_east"
7. "look_west"
8. "look_north"
9. "look_south"
10. "end"

A series of if statements checks for one of these movements. Whenever a movement that isn't "end" is processed, instr_index is incremented so that the next instruction in patrol_route can be handled. movements 1-9 are handled quite similarly, and the helper methods that define the handling will be described shortly. When the "end" movement, is encountered, the instr_index attribute is set back to 0 so that the first instruction in the patrol_route may be repeated.

Furthermore, there exists a check in update() to determine if this Enemy is moving. If it is, then the animation corresponding to the direction that they are moving in will be stepped through. Finally, the last section of this method involves handling collisions between an Enemy and a physical object, an Enemy and Snake, and an Enemy's LOS with physical objects and Snake. Collisions with physical objects are handled similarly to what was described in Snake's update method, and whenever collisions are made between an Enemy and Snake, the Enemy's alert_phase will be set to True. LOS collisions are a bit more complex. a new LOS for this Enemy will be constructed based on the direction it is looking in. This will happen every time this Enemy's update() method is called. Collisions with the LOS are checked with pygame's spritecollide function. The first type of collision to be checked is



between the newly constructed LOS and any Sprite representing a physical object in the current Area. This is done to determine obstructions to this Enemy's LOS, which could shorten the length of it. For example, in the picture shown on the previous page, the Enemy on the far left of the screen does not have its LOS obstructed. But then suppose a physical object (represented by the red square in the following picture) obstructs the Enemy's LOS.



The shortening is achieved by constructing yet another LOS whose length is the difference between coordinates on the Enemy's rect and coordinates on the obstructing Sprite's rect. Two different LOS needed to be created so that Snake wouldn't be able to be spotted through objects. The first time an LOS is constructed in this method, it is a LOS that extends from its eye_end to an edge of the screen. The LOS would ignore everything in its path if it weren't for this adjusment of shortening it (by constructing a completely new LOS) in the event that it is obstructed. That is, without the adjustment, the LOS would pierce through any Sprite, which could then allow Enemies to "see" Snake through any other Sprites. An Enemy's LOS will only ever collide with one object.
- **_handle_LOS_collision_for_obj:** Iterates through a list of Sprites to determine the Sprite that had collided with the first LOS that was constructed. Shortens the LOS by creating an LOS with a new size corresponding to the distance between this Enemy and the Sprite object that was "spotted". The orientation of this LOS (e.g., horizontal or vertical) depends from which direction this Enemy's LOS spotted the object.
- **_handle_LOS_collision_for_snake:** Handles a collision between Snake and this Enemy's LOS that may or may not have been obstructed. If Snake has been spotted, then this Enemy's alert_phase attribute will be set to True.
- **_create_LOS:** Constructs an LOS with a given width and height and also sets the position of the LOS with a give x and y position.

- **_animate:** loads an image for the Enemy sprite based on what the value of cur_frame is. Each call to _animate() will increment cur_frame until cur_frame == the total number of animation frames that an Enemy has in each direction, in which case cur_frame is set back to 0. This helper method is called inside the update() method.
. **_set_[some_movement]:** Each helper method for processing a movement has the same basic structure. Bear in mind the variable called unit, which was set after an instruction was parsed. For the “go” movements, if current_step_num <= units, then the Enemy’s rectangular coordinates will be incremented/decremented based on which direction was given. For the “look” movements, if current_look_num <= units, then the image of the Enemy will be set to the animation frame associated with the given direction, effectively “holding” this image for as long as current_wait_num <= units is true. For the “wait” movement, if current_wait_num <= units, then current_wait_num is incremented, making the Enemy wait in whatever animation frame it was last in before the wait instruction. Once any of the relational expressions just described evaluate to False, then current_step_num/current_wait_num/curren_look_num are set to 0 and _instr_index is incremented so the next instruction in the patrol route can be processed.

*BACK-UP AND VERSION CONTROL PLANS*

At the moment, each group member has been developing sets of their own code that they send to all other group members (through Facebook and Slack) at the beginning of each week. We plan to use GitHub in the near future.