DONALD BREN SOLID
Design Document

Table of Contents

1. Overview……3

2. Game Specification……4
2.1 Background Story……4
2.2 Characters……4
2.3 Rules and Mechanics……6
2.4 Gameplay and Balance……13
2.5 Artificial Intelligence……13
2.6 Music and Sound……14
2.7 Artwork and User Interface……15
2.8 Areas……18

3. Technical Specification……23
3.1 Programming Languages and Libraries……23
3.2 Target Hardware and Operating Systems……23
3.3 Game Loop……23
3.3.1 Object Instantiations for the Game Loop……25
3.4 Classes……25
3.4.1 Snake……25
3.4.1.1 Instance Variables……25
3.4.1.2 Method……26
3.4.2 Area……28
3.4.2.1 Area Transitions……28
3.4.2.2 Instance Variables……29
3.4.2.3 Methods……29
3.4.2.4 Behavior of Area subclasses……30
3.4.3 LOS (Line of Sight)……31
3.4.3.1 Instance Variables……32
3.4.3.2 Methods……32
3.4.4 Enemy……32
3.4.4.1 Instance Variables……32
3.4.4.2 Methods……33
3.4.5 Block……37
3.4.5.1 Instance Variables……37
3.4.5.2 Methods……37
3.4.6 Classes for Physical Objects……37
3.4.7 Classes for Collectible Objects……37
3.4.8 Camera……38
3.4.9 HUD……38

3.4.9.1 Instance Variables……...38
3.4.9.2 Methods……..……..38
3.4.10 Classes for Weapons.....38
3.4.10.1 Snake's Weapons.....38
3.4.10.2 Mercenary Weapons.....39
3.4.11 Unique Characteristics of Bosses.....40
3.4.11.1 Professor Kay……...40
3.4.11.2 Professor Thornton……...40
3.4.11.3 Boo.....40
3.4.11.4 Professor Pattis.....40
3.4.11.5 Professor Frost.....40
3.5 Back-up and Version Control Plans……...41

4. Schedule and Personnel……41

1 —— 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 Piskel. The code for the game will be written entirely in python using the pygame library.

2 —— Game Specification ——

2.1 Background Story:
You play as a young 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 in 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. Snake 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.

2.2 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. 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 (probably 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. Professor Kay is the first boss that Snake faces.

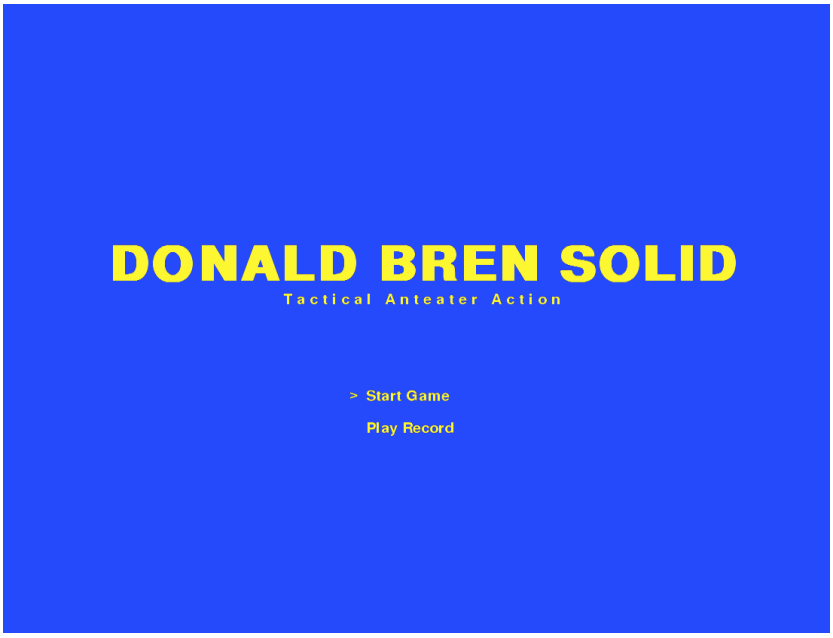
Mercenaries

. 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/)

2.3 Rules and Mechanics

Title Screen

* The options available to the player here are “Play” and “Play Record”, which displays the stats and score of the player’s best play-through.
* On starting up the game, the title will scroll to the center. The player can skip this scrolling animation by pressing space
* An 8-bit rendition of Jimi Hendrix’s “All Along The Watchtower” will play here



View

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 the diagram on 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***. 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 stairwell rooms.

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 much noticeably slower than him while on patrol. However, while in alert phase, mercenaries will be able to move slightly faster, but still slower than how fast Snake can move. Bosses will have a rate of movement comparable to alerted mercenaries, 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/Subsided:

* 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.
* Subduing an enemy with one of Snake's weapons while mercenaries are on patrol (the idea is that the grunt they let out alerts their comrades)
* Alert phases subside if the player hides in the girl's bathroom (since pretty much all characters are male) for several seconds, or hides within a designated room on a given floor of DBH (called a "safe haven"). Such rooms are marked with pink doors and will play a "ding!" sound when an alert phase has subsided. No ding sound will play if

Snake is in the safe haven without having triggered an alert.
* The mercenary who spotted Snake will have an exclamation point appear over his head

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. Inaccessible entrances will emit a buzzing sound to deny Snake entrance. For example, entrances leading to the stairwells of a floor on DBH will be inaccessible until Snake obtains a card key for that particular floor. The south door in the stairwell for the first floor is the door Snake must exit in order to complete his mission, but it will only be accessible once Snake drops off his CS 113 data disc.

Items
Listed below are items that can be obtained. Items are placed in specific areas and can be collected by walking over them. Items do not respawn.
* 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 noodles
* 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 in his inventory
* Card Keys: card keys exist for elevators (which can only go down to the first floor) and for accessing floors 2-5
* Metal Gear Solid Games (1 through 5): collectible items representing game packages for the Metal Gear Solid games. Collecting these items will raise your score.

Inventory

Players can view Snake's weapons list by pressing the W key and items list by pressing the Q key. Setting a weapon or item as the current weapon or current item involves the player pressing W or Q while having the cursor next to the desired weapon/item on the appropriate screen. The player cannot go from the weapons list to the items list or vice versa. Items and weapons that Snake has obtained, along with their ammo count, will be displayed in the inventory. In addition, the amount of Metal Gear Solid games collected (out of 5) will be displayed.

HUD
Snakes health, current weapon, current item, and number of lives will be displayed in the HUD. Weapons and items will have an associated stock value beside them. Snake's health consists of 100 "hit points" (HP). Here is a screenshot of the HUD:

How Enemies are "Taken Out":
No one is ever killed. If anything, they are "subdued". For mercenaries, after being hit with one of Snake's weapons by a certain amount, they will let out a grunt and become stunned for a designated period of time. A stunned state is represented by a mercenary facing the screen with what would be stars rotating above his head:

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.

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. Each room for a member of the KTP unit has a colored door that leads to their boss battle "arena". The arena itself is an obstacle-free area of the same color as the door that led to it. Kay has a blue door, Thornton has a yellow door, and Pattis has a green

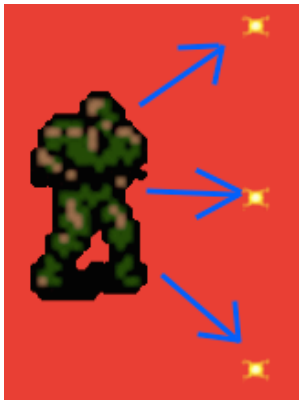
door. For Frost, entering the 5th floor will trigger the battle. The battle with Frost will take place throughout the entire 5th floor.

Boss Fight Descriptions
Boss fights with members of the KTP unit will take place in an open area with no obstacles to allow for unobstructed movement. The battle with Frost will take place throughout an entire floor of DBH. Players will have to rely on their reflexes as opposed to physical objects to deal with the bosses' projectiles. As an added layer of suspense, health bars will not be displayed for bosses. However, it should be mentioned that each member of the KTP unit will have 300 HP. Frost, on the other hand, will have 70 HP — a low amount compared to the other bosses, but unlike the other bosses, Frost can only be affected by one type of weapon and he never really "goes away", as will be explained shortly. Lastly, bosses do not regenerate health if the player decides to leave the boss battle arena nor do they regenerate health when the player needs to use a continue (this is to balance out the fact that weapons do not respawn).
* Kay: Will simply chase you around the room and spread hearts, which represents his kindness, in all directions when he comes within a certain distance of you. When his health is depleted, he will disappear and be replaced by the 3rd floor card key.
* Thornton: Will stay fixed at the top of the screen moving left and right while throwing othello pieces at you. At the same time, Boo will be chasing you around the room trying to run into you. When his health is depleted, he will disappear and be replaced by the 4th floor card key.
* Pattis: Will chase you around the room tossing reading material at you in a spread-like pattern (see *Attacks and Damage Done By Enemies* a few sections down for further description of this pattern). Each piece of reading that is thrown by Pattis will not traverse the length/height of the screen; rather, they will freeze after a randomized distance away from Pattis to simulate having been thrown on the ground. Snake takes damage if he comes into contact with them while they're either in motion or on the ground. Readings can disappear if you shoot them. When his health is depleted, he will disappear and be replaced by the 5th floor card key.
* Frost: Will chase you around the 5th floor shooting frost sparks at you in a spread pattern. 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). After repeatedly "slipping" on the banana peels, Frost will become stunned. You can only enter his office to slip your CD into his desk once he's been stunned, but if you take too long to get to his office, Frost will wake up and fight you again.

Attacks and Damage Done By Snake
The space bar is used to perform attacks. Water balloons will do the most damage to mercenaries in the sense that they will stay stunned the longest. Water balloons are thrown and will trace an arc through the air. They only do damage once they've "hit the ground" — i.e., they will not cause damage as they are soaring 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 where Snake is. Spitballs travel slower than rubber bands and are smaller,

while rubber bands travel quicker and are larger. Spitballs and rubber bands cannot travel through physical objects within an area.

Attacks and Damage Done By Enemies



* Attacks on Snake will diminish his health by a certain number of health points
* Mercenaries: being in contact with you (0.25 HP for every frame you're in contact with them. No harm will be done if they are stunned.), firing sparks from stun rifle (3 HP). A mercenary fires three sparks traveling at the same speed from his rifle — e.g., if he is facing east, then there'll be a spark that flies northeast, east, and southeast. Sparks cannot pass through physical objects in an area. A depiction of the attack is provided on the right (arrows placed for illustrative purposes).
* David Kay: releasing hearts in all directions (5 HP), making contact with Snake (1 HP/frame).
* Alex Thornton: making contact with Snake (1 HP/frame), firing othello pieces at you (4 HP) while Boo, his companion during the boss battle, attempts to run into you (5 HP)/ frame.
* Richard Pattis: running into you (1 HP/frame), throwing reading material at you in a spread pattern. If the reading is is still "in the air" when it hits you, damage is 8 HP. When you run into a reading that is "on the ground", damage is 4 HP.
* Dan Frost: running into you (1 HP/frame), shooting frost sparks from his hands in a spread pattern (9 HP).

Health Recovery

Consuming bowls of instant noodles is the only way Snake can recover health. The health recovery will take place if Snake presses the E key on the inventory screen after having already selected the noodles as his current item or if he has it equipped when his health goes to 0 (i.e., it would automatically take effect in this case). A single bowl will recover 50 health points.

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.

Continues and Checkpoints

If Snake's health bar reaches 0 while he still has lives, the player will be met with a continue screen. The player has a choice to either continue or quit the game. The third floor stairwell is the checkpoint of the game. Being subdued before having reached this checkpoint will cause Snake to respawn in the stair room on the first floor. Being subdued after having reached the third floor stairwell checkpoint will always respawn Snake at that checkpoint (even if he goes down to floors below the checkpoint and is subdued in those earlier floors).

Game Overs and Game Completion

Game overs occur when Snake's health reaches 0 after only having 1 life left. Upon being subdued for the final time, players will be shown a game over screen, which will play a heartrending 8-bit version of the guitar solo from Pink Floyd's "Comfortably Numb". The game is completed by successfully making it through the door in the first floor stairwell after dropping off Snake's CS 113 data disc into Frost's desk. Once completed, the player will be shown a congratulatory message followed by a screen displaying various stats and their final score for the play-through they just completed.

Scoring System

A list of various game stats and a related numeric score will be calculated once the player successfully completes the game (negative scores are possible). Players can view the stats and score of their best play-through (i.e., the play-through that garnered them the highest score they've ever made) from the "Play Record" option on the title screen. The following is a list of stats and their contributions to the scoring:

* Completion Time: +15000 points if the time is less than 10 minutes, +8000 if it's between 10 and 20 minutes, and +4000 otherwise
* Mercenary Alerts: -200 points for each time Snake is spotted by a mercenary. If the player goes through the game undetected, then +15000 points.
* Mercenary Stuns: -100 points for every mercenary that Snake stuns. The idea is that Snake shouldn't be using his weapons unless he absolutely needs to in order to progress (e.g., during boss battles). +5000 if no mercenaries were harmed.
* MGS Games Collected: +800 points for every Metal Gear Solid title Snake collects
* Continues: -350 points for each continue. +10000 points if the player makes it through the game without any continues.
* Instant Noodles Consumed: -100 points for every cup of noodles consumed to regenerate health. +10000 if the player never once needs to eat noodles to regain health.
* Score: The sum of all points

2.4 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 copies of the Metal Gear Solid titles are situated within a particular area on each floor (five games total, 1 for each floor). Such items 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 available for pick-up in some area on one of the floors.

The recommended play-style for an initial play-through is to be patient and willing to observe enemy movements while staying clear of sections in the area that would bring Snake into a mercenary's line of sight. The first floor will only have mercenaries for Snake to sneak past and the key to the next floor will be placed in a designated 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 transition section Snake entered an area from. Such an arrangement for the mercenaries applies to all floors except the fifth where there will be no guards. Sneaking past guards becomes increasingly difficult as Snake progresses through the floors, for their will be a higher number of mercenaries to sneak past with less predictable patrol routes. On 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 room beyond the fifth floor stairwell, Snake will immediately encounter Professor Frost. This battle can span across the entire fifth floor (excluding the stairwell room and Frost's office), and Snake can only enter Frost's office to drop off his CS 113 data disc once Frost has been stunned. 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 can be picked up in the office so that you don't have to go back down floor by floor. That is, the player can take the elevator on the fifth floor all the way down to (and only to) the first floor. The card key will also work for any of the other elevators on floors 2-4. Snake will have successfully completed his mission once he makes it through the door in the first floor stairwell.

2.5 Artificial Intelligence

Mercenaries operate in two modes of AI — patrol phase and alert phase. When in patrol phase, mercenaries will simply follow a patrol route specified by our programmers (see description for an Enemy object's `update()` method in **Section 3.4.4.2** for how patrol routes can be programmed). The speed of a mercenary's movement while on patrol can be adjusted (only up to as fast as the slowest chase speed in alert phase). When in alert phase, mercenaries will move towards Snake's position and fire intermittently. The chase speed of mercenaries in attack phase as well as the frequency in which they fire their stun rifles can be adjusted (chase speed cannot be up to as fast as Snake). Mercenaries will move towards Snake until they have him in an "attack box" whose size a programmer can specify. Mercenaries will keep Snake on the edge of their attack box and continue to fire intermittently. See **Section 3.4.4.1** for a description of how the firing logic is implemented.

These adjustable settings for AI were implemented so that mercenaries within a given area wouldn't all behave exactly the same way, particularly in alert phase. In such a case, there's the possibility of all the enemies eventually flocking together and essentially becoming one entity (at least visually) if the player is stationary or guides them towards such a formation. However, since the player is encouraged to leave an area immediately during alert phase (lest they are subdued), such flocking behavior may barely be noticeable. Moreover, the number of enemies to uniquely consider and establish distinct AI profiles for is much too high relative to the amount of personnel we have on hand. Thus, the AI settings for all enemies in the game is largely quite similar.

The bosses of the game basically use the same AI as the mercenaries, but with some slight adjustments. For one, bosses are never patrolling, so they're always in alert (i.e., attack) phase. The Kay boss will only blast hearts in all directions once he has Snake in his attack box. The Thornton boss will only move from side to side at the top of the screen while facing south and throwing othello tiles intermittently. Pattis and Frost behave virtually identically to the mercenaries, differing only in how their attack projectiles look and behave.

To navigate around obstacles, mercenaries will "cling" to the side of the obstacle it collided with and will, if the collision occurred while the mercenary was moving east or west, move up or down depending on if Snake is above or below the mercenary's y-coordinate position while hugging the collided side of the obstacle. Similar logic applies for when the mercenary is moving north or south. Once the mercenary is no longer colliding with the obstacle, it will resume the typical alert phase behavior described previously.

2.6 Music and Sound

Select pieces from the soundtracks of the Metal Gear games (mainly, the first Metal Gear game on MSX) will be used as well as some favorite tunes of the lead game designer (and of some of the actual real-life bosses of DBS themselves). All songs are "8-bit" renditions. The majority of the sounds in DBS are taken from the Metal Gear games. Below is a list of sources for the music and sound of the game:

All Along The Watchtower (Title Screen Music):
https://www.youtube.com/watch?v=O_1JxaVIXeI

Theme of Tara (Sneaking Theme):
https://www.youtube.com/watch?v=t4SHQSSeib4

Red Alert (alert phase music):
https://www.youtube.com/watch?v=6YQm2M-v4l0

-!- Alert (for the BREEEEEEP! sound when mercenaries spot you):
https://www.youtube.com/watch?v=iekGOi3IPcE&index=5&list=PLF63AFC7EB42E68C9

Separate Ways (Kay Boss Theme):
https://www.youtube.com/watch?v=NKSItY9DFlk

Knight Rider Theme (Thornton Boss Music):
https://www.youtube.com/watch?v=5T7mQRhs2oc

Immigrant Song (Pattis Boss Music):
https://www.youtube.com/watch?v=OkMU3hiFeeU

Ballroom Blitz (Frost Boss Music):
https://www.youtube.com/watch?v=3ynZGZE2_-A

Just Another Dead Soldier (Continue Screen Music):
https://www.youtube.com/watch?v=aVKRJhIyvhk

Comfortably Numb (Game Over Music):
https://www.youtube.com/watch?v=OODfSpbd08I

Snake Eater (Game Complete Music):
https://www.youtube.com/watch?v=nA6UDEsXuyk

Various Sound Effects From the Metal Gear Solid Games:
https://www.reddit.com/r/metalgearsolid/comments/1jt1vg/official_metal_gear_solid_sound_effects/

Various Sound Effects From Metal Gear Solid 1 in Particular:
http://www.sounds-resource.com/pc_computer/mgs/sound/328/

Metal Gear Ration (health power up) Sound:
http://www.zedge.net/ringtone/720690/

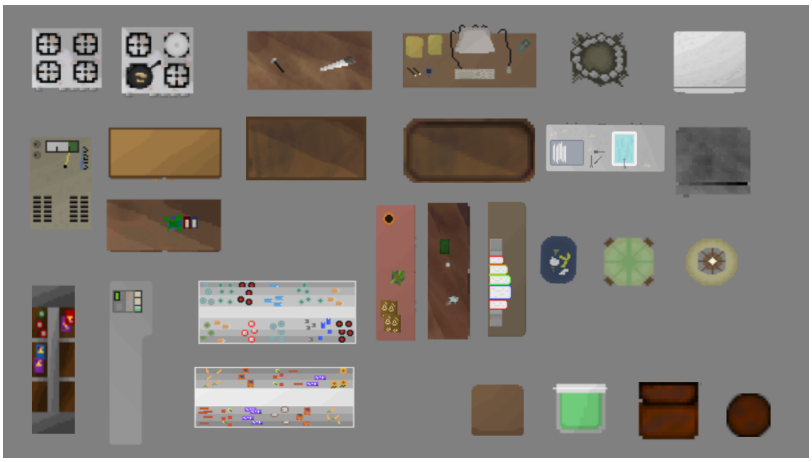
Ding! (for safe havens):
https://www.youtube.com/watch?v=FgcB7Lkvp44

2.7 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/

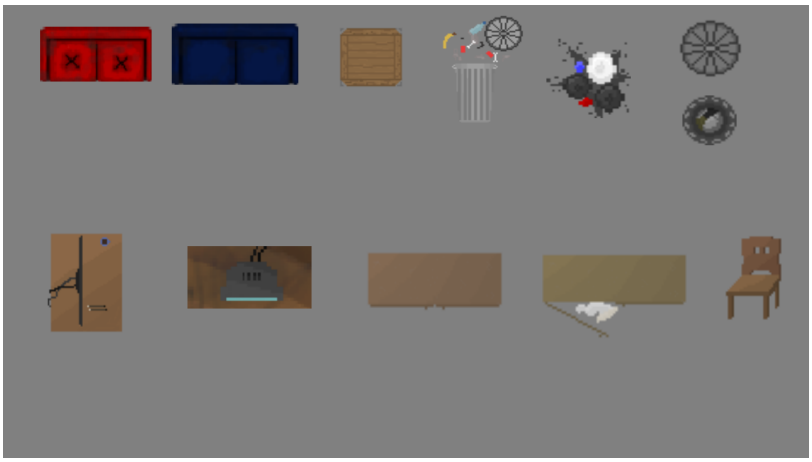
Also from the same sprite sheet, the following frames of animation will be used for the mercenaries and surveillance cameras:

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

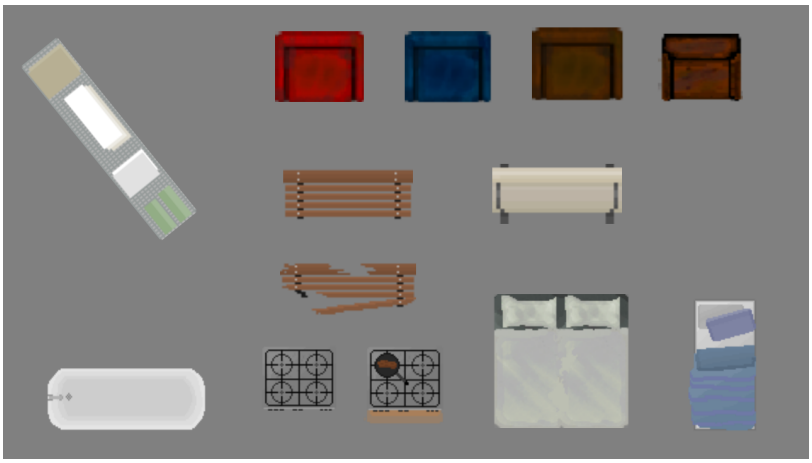
Various images for furniture and other physical objects were taken from the following sprite sheets:



https://marketplacecdn.yoyogames.com/images/assets/224/screenshots/451_original.png?1404568304



https://marketplacecdn.yoyogames.com/images/assets/224/screenshots/450_original.png?1404568302



https://marketplacecdn.yoyogames.com/images/assets/224/screenshots/452_original.png?1404568305

And here are some weapons and items:

* Spark (shot from a mercenary's stun rifle. Modified into a blue color for Frost's attack (http://pre12.deviantart.net/238d/th/pre/i/2013/052/6/b/mlss_thunder_luigi_sprites_by_pxlcobit-d33wjt5.png)

* Rubber Band (http://41.media.tumblr.com/31480454d53294ccbf515e3763b44d37/tumblr_mi2r2sQ2Ua1r413h3o1_400.jpg)

* Balloon (http://jermungandr.net/sprites/sheets/Kirby_Mass_Attack_%28DS%29/Level_Objects/Balloon_Ship.png) and accompanying splash image when it "hits the ground" (https://wreckageuk.files.wordpress.com/2015/03/splat-703894.png)

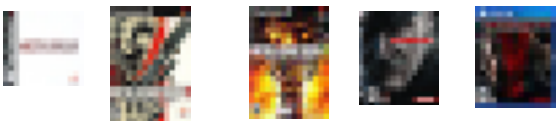
* Banana (left: deployed as a weapon [http://piq.codeus.net/static/media/userpics/piq_170502_400x400.png], right: as a collectible object [http://www.zeldaelements.net/images/games/links_awakening/trading_sequence/bananas.png])

* Straw (drawn using Piskel)

* Instant Noodles: https://s-media-cache-ak0.pinimg.com/236x/49/2a/5a/492a5afa6b102ff2a8c939c45e6d4410.jpg)

* Floor Card Key (also used for elevator card key): http://twitchplayspokemon.org/img/items/card_key.png

* Pixelated Google Logo (used for one-ups):
http://piq.codeus.net/static/media/userpics/piq_9991_400x400.png



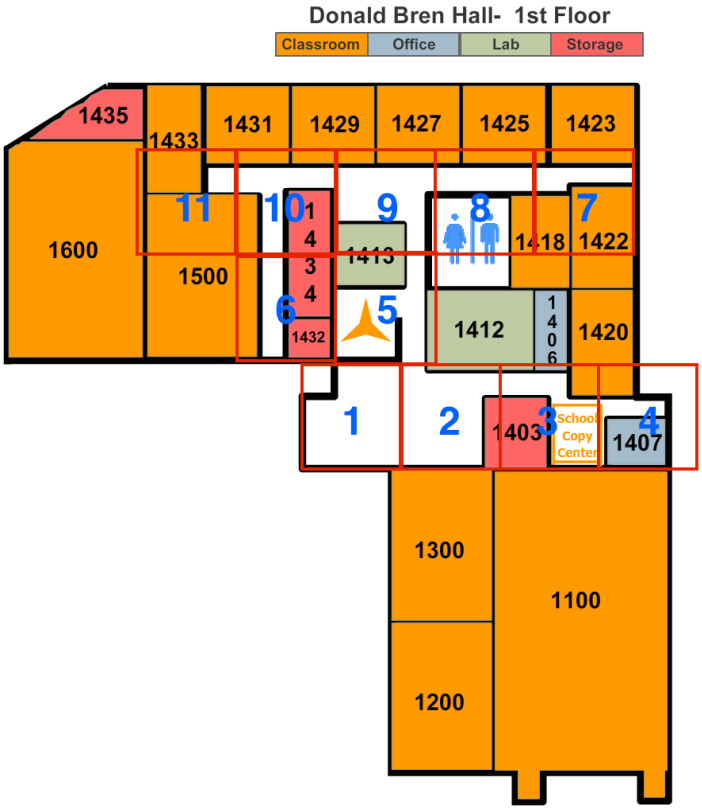
* Metal Gear Solid 1, 2, 3, 4, and 5 games (images were pixelated with GIMP for effect)
- MGS1: http://ocremix.org/files/images/games/ps1/8/metal-gear-solid-ps1-cover-front-48802.jpg
- MGS2: http://www.theicecave.org/damage_control/multimedia/mgs2covers_082410.jpg
- MGS3: http://ecx.images-amazon.com/images/I/6139QD2Z3ZL.jpg
- MGS4: http://vignette3.wikia.nocookie.net/metalgear/images/8/8d/MGS4_North_American_Cover.jpg/revision/20100401030142
- MGS5: http://cdn2.pu.nl/media/misc2/kon2.jpg

All the other characters and physical objects will be drawn using GIMP and Piskel. Most notably, the professors in the game were drawn using Piskel. Shown below are our drawings for each professor (in the interest of space, only their neutral south-facing images are depicted):

Kay Thornton Pattis Frost

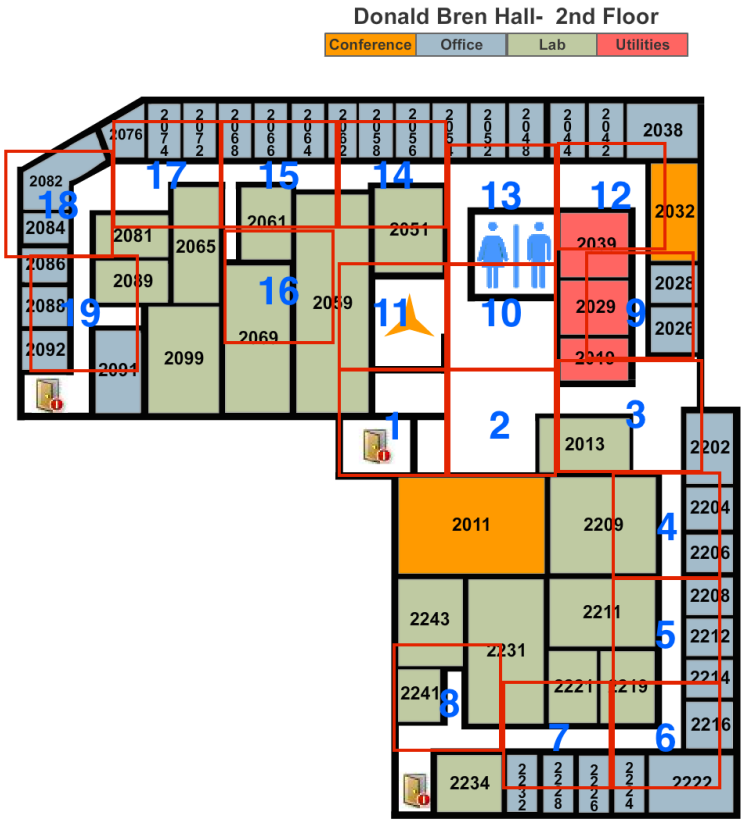
2.8 Areas
This section shows how we've envisioned the main hallways and open areas of DBH's floors will be divided into areas. Each area will represent a square section of DBH with all of the peculiarities of that section such as couches, pillars, garbage cans, etc. However, we do plan on taking liberties with the design in order to facilitate development. We've only included diagrams for the first and second floor in the following pages 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 doors on each floor will be accessible.

1st Floor



A screenshot of how Area 2 looks like in-game is provided below:

2nd Floor

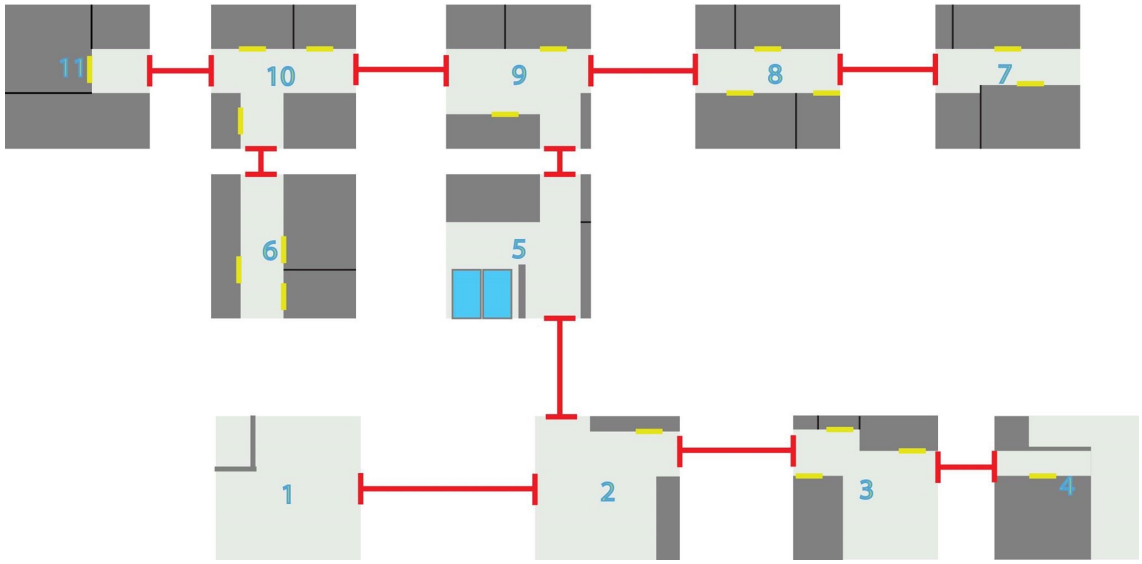


Here is a screenshot of how Area 4 looks like in-game (the alcove is a real-life detail not shown in the diagram that we took into consideration):

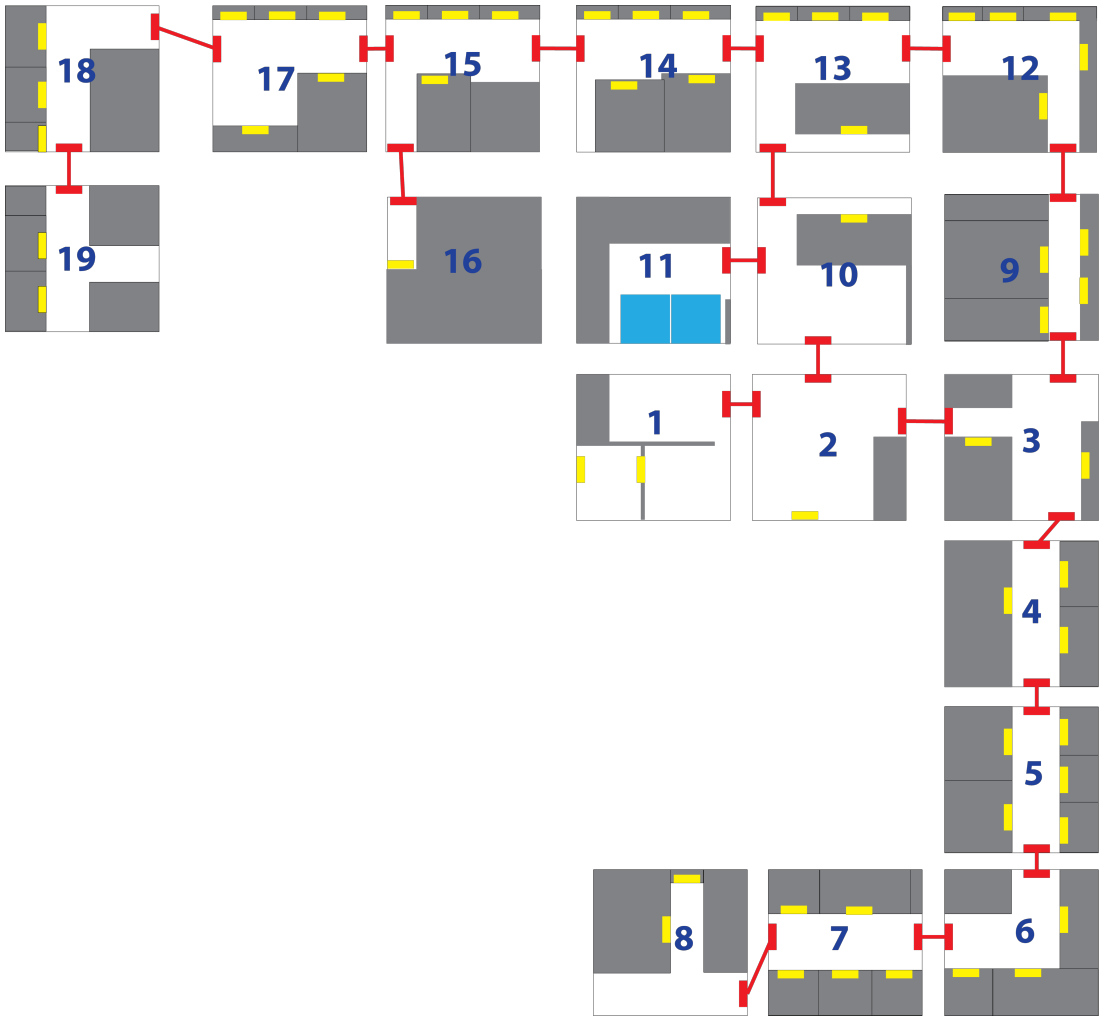
Note how some areas in the diagrams aren't the same size as all the others, but this was only for the sake of specifying the span of an area. We will design the areas as though they were all of the same size in the game. Moreover, the labeled numbers here do not necessarily represent the order in which the player will visit areas. Players are free to roam the entire floor but will gain entrance to floor 1 and floor 2 of DBH through their respective areas above labeled with 1.

The following diagrams sketch out how the areas will look and the connections between each area. The number labels here 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 (which we've dubbed "inner" Areas because the transition sections aren't accessed on the boundaries of the game display), though such areas have not been depicted in the following diagrams. Accessible doors will not be the same for all floors.

1st Floor



2nd Floor



3. —— Technical Specification ——

3.1 PROGRAMMING LANGUAGES AND LIBRARIES

Donald Bren Solid is written entirely in Python using the pygame library.

3.2 TARGET HARDWARE AND OPERATING SYSTEMS

Any desktop or laptop computers running the Windows or OS X operating systems with a screen display greater than or equal to 1100 x 900 pixels (smaller dimensions may incorrectly display the positioning of certain graphical elements. E.g., the HUD and title screen lettering).

3.3 GAME LOOP

Our game implements the standard pattern for a game loop. Written below is 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()

     process_music()
```

* **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.
*** Process Music:** Depending on the current state of the game, a certain song will need to be played (e.g., during alert phase). This section of the game loop sets a variable called `song_to_be_played` and checks it against a global variable called `current_song`. The pseudocode for how songs are loaded and played is provided below:

```
if current_song != song_to_play:
    current_song = song_to_play
    pygame.mixer.music.load(song_to_play)
    pygame.mixer.music.play()
```

All of these procedures are written under an if statement that checks if the weapon select or the item select screens, which are summoned with the W and Q keys, respectively. Additionally, the if statement checks if Snake has been subdued and if the end of the game has been reached. If weapon or item select screens have been summoned, then the appropriate text and event handling is established to represent them. The state of Snake being subdued will send the player to either the continue or game over screen. The state of Snake having reached the end of the game will bring up the congratulatory message screen followed by a display of the completed play-through's stats and score.

The game loop itself is contained inside of a `game_on` loop where all of the game data is loaded. Upon quitting the game (or receiving a game over), all of the game data is re-loaded to reset the entire game state. When the player successfully finishes the game, an `if` statement at the beginning of the game loop will process such a state and write the completed game's score and stats to a .txt file, which will be read from every time. The .txt file will be overwritten if the score of any new play through is greater than the one already stored.

3.3.1 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 **3.4.2.1 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.

3.4 CLASSES

Object-oriented programming drives the implementation of our game. For basically every element one sees in the game display, there exists a class to represent that element. Below is a description of each of the classes and important attributes/methods associated with them. NOTE: (underscores prefixing method names denote helper methods)

3.4.1 Snake (inherited from pygame's Sprite class)

3.4.1.1 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 for 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. Initialized 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 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.
- **current_weapon:** an object representing a particular weapon (see **3.4.8 Classes For Weapons** section) that Snake currently has equipped. Initialized to an Unequipped() object, which is used as a placeholder for a “None” slot, essentially.
- **current_item:** stores a Collectible object (**3.4.6 Classes for Collectible Objects**) Snake currently has equipped. Initialized to an Unequipped() object
- **weapons_lst:** stores a list of all the weapons Snake has his disposal. Used by the HUD to list out all available weapons.
- **items_lst:** stores a lis of all the items Snake has at his disposal. Used by the HUD to list out all available items.
- **reached_checkpoint:** a boolean to indicate if Snake reached the checkpoint of the game (the 3rd floor stairwell). Being subdued after reaching the checkpoint will restart Snake here.
- **lives:** an int that represents the number of lives Snake has. Initialized to 3.
- **is_subdued:** a boolean used to indicate if Snake has been subdued by the Enemies (i.e., when his health bar reached zero).
- **mgs_games:** int that represents the number of Metal Gear Solid games Snake has collected. Will be factored into total score.
- **time, merc_alerts, merc_stuns, continues, noodles_consumed, score:** ints that represent the values for these respective stats. Will be factored into total score.
- **has_floor[X]_card_key:** a boolean to indicate if Snake has the proper card key to access floor[X]
- **has_dropped_off_data_disc:** a boolean that will allow the player to exit the south door in the first floor stairwell and finish the game if True
- **has_elevator_card_key:** a boolean to indicate if Snake has the elevator card key that will allow him to traverse from any elevator on floors 2-5 to the elevator on the 1st floor
- **reached_end_of_game:** a boolean that indicates if the player has reached the end of the game. Used to exit the game loop.

3.4.1.2 Methods:
- **__init__:** constructor for a Snake Sprite that initializes all of its attributes
- **change_move(h_move: int, v_move: int):** 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 11. That is, each increment provides a 11 pixel movement from one iteration of the game loop (i.e., frame) to the next (if Snake is moving). If Snake is not moving, then the h_move and v_move attributes are set to 0
- **init_position(x: float, y: float):** initializes the position of Snake within an Area
- **update(event: Event, obj_group: Group, e_group: Group):** 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 frames, 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 with 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(collision_lst: [Group]):** 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(image_lst: [str]):** 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(event: Event):** 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. The space button is reserved for attacks that Snake can perform depending on his current weapon.

3.4.2 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. Each floor of DBH will have a module consisting of Area classes for all of the areas within that floor.

Section 3.4.3 will provide a description of how subclasses of Area will behave. Before we discuss the attributes and methods of an Area, however, let us describe how Area transitions are achieved.

3.4.2.1 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.The 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 namedtuple 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”)
```

Damage values for attack objects for Snake and Enemies are established in the module for Areas. They are represented as global constants. The following will be descriptions of the instance variables and methods of an Area object.

3.4.2.2 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.
. **boss**: a Kay, Thornton, Boo, Pattis, or Frost object that represents the boss the resides within the Area (or Areas if the object is Frost. See **Section 3.4.9.5 Professor Frost** for more details)
. **snake_group:** a Group that will store Snake.
. **in_alert_phase:** a boolean indicating if this Area is in alert phase. Used to propagate alerts to all Areas

3.4.2.3 Methods:
- **__init__(player_obj: “Snake", enemies: "Group"):** constructor for an Area that initializes all of its attributes
. **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.
. **draw(window: Surface):** calls draw() on obj_group. This has the effect of calling draw() on all the sprites in obj_group.
. **draw_enemies(window: Surface):** similar to the draw() method but in this method, draw() is called on e_group.
. **check_transition(area_dict: dict):** 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 (more detail in the **Enemy** class description)
. **_make_transition(x: float, y: float, area_dict: dict, direction: str):** 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
. **update_e_attack_objects():** updates the necessary attributes of all the attack objects for all Enemies in this Area. This method is responsible for moving the sparks fired from an Enemy's stun rifle across the screen. Collisions between an Enemy's attack objects (i.e., sparks) and Snake will cause Snake's health to be decremented by 1.
. **update_s_attack_objects():** updates the necessary attributes of all the attack objects for Snake in this Area (e.g., spitballs and water balloons). Collisions between Snake's attack objects and Enemies will update their hit counters related to a particular weapon (see **3.4.8 Classes for Weapons** for more information)
. **_reset_enemy_state():** resets the state of Enemy Sprites to how it was 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)
. **if_alert_then_propagate(area_lst: ["Areas"]):** Takes a list of Area objects representing all Areas in the game, which is a list that is initialized before the game loop, and checks if this Area is in alert phase. If it is, then sets the alert phase of all Areas to True.
. **if_save_haven_subside_alerts(area_lst: ["Areas"]):** Takes a list of all Area objects in the game and checks if this Area is a "safe haven". If it is, then sets the alert phase of all Areas to false. It takes a certain number of iterations of the game loop for the alert to subside.
. **_set_background(window: "Surface", filename: str):** Sets the background for this Area to the image referenced by filename

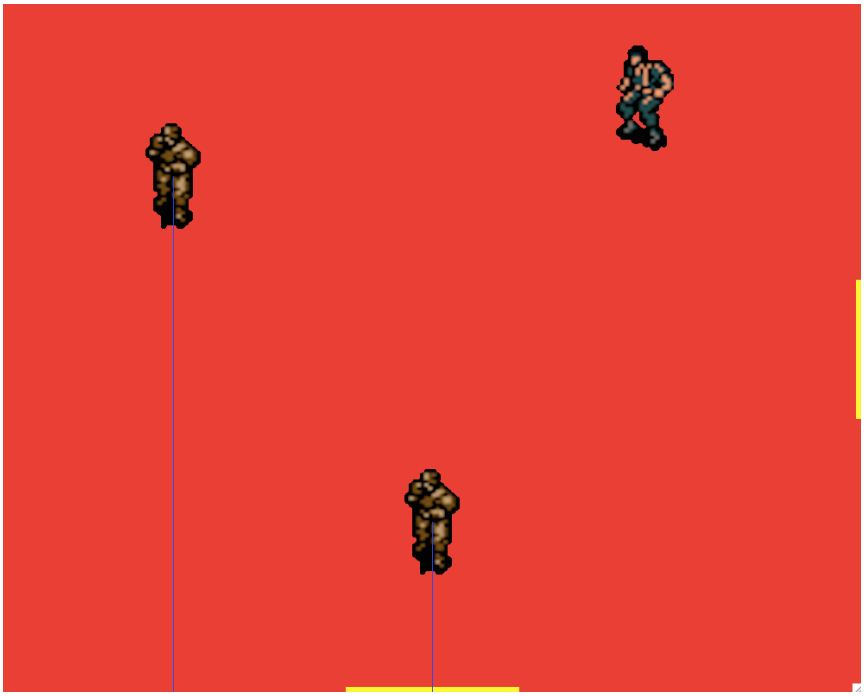
3.4.2.4 Behavior of Area Subclasses

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 a coming section about the **Enemy** class, Enemies will have a patrol_route attribute that is a list that stores instructions, essentially, for the patrol route. Enemy formations will become increasingly difficulty to get around as Snake progresses through floors 1 to 4. Depending on which direction Snake entered an Area from, the Area may set a different formation. This adds variety and sense of fairness to certain situations. For example, say that Area A has a formation consisting of patrol route that a mercenary begins by standing in front of a door, which leads to Area B. Snake goes through that door to Area B (after the mercenary walks away from it), leaves Area B to go back to Area A, and runs directly into the mercenary that began his patrol route by standing in front of the door. This is rather unfair. However, such leeway is not given for some transition sections that lie on the edges of the screen. For such areas, players will need to carefully inch into them to prevent colliding with a mercenary on patrol (this is, as it turns out, a general rule of thumb for how to approach each Area).

3.4.3 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 on the next page. 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 colliding with them), then Snake has been spotted. The line will not be drawn in the final version as it is done so in the following screenshot (they were drawn for testing).



3.4.3.1 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.

3.4.3.2 Methods:

- **__init__(eye_end: (int, int), width: int, height: int):** constructor for an LOS Sprite that initializes all of its attributes
- **set_position(x: float, y: float):** 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.

3.4.4 Enemy (inherited from pygame’s Sprite class)

3.4.4.1 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 (think of a single unit as a single iteration of the game loop). 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/current_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 a Sprite object representing a physical object. Initialized to False.
- **in_alert_phase:** a boolean indicating if this Enemy is in alert phase, which is switched on when Snake is spotted (collides with a LOS) or collides with any Enemy onscreen. 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 (see **update** method).
- **los:** a LOS obect that represents the line of sight of this Enemy.
- **los_group:** a pygame Group container that stores the los of this Enemy for collision processing.
- trigger_happiness: a list that contains 20 elements, each of which is either a 0 or 1. This attribute is what underlies the firing logic for an Enemy. A section of this Enemy’s update() method will use this list to randomly pick an element from. If the element is 1, then this Enemy will fire its weapon. Else, it will not. Have more 1s than 0s in this list will cause this Enemy to fire more often, while having fewer ones will cause the opposite.
- **keeps_dist:** a boolean that determines if this Enemy is of the type that tends to keep its distance from Snake as described in **2.5 Artificial Intelligence**
- **hover_dist:** a float that specifies how far Snake would have to be from this Enemy in order for this Enemy to exhibiting its behavior for keeping distance. E.g., if hover_dist is 500, then as long as Snake is greater than 500 pixels away (in Euclidean distance), then this Enemy will keep its distance and hover along the outside.
- **ab_left_x/ab_right_x/ab_top_y/ab_bottom_y:** ints that represents how many pills this Enemy’s attack box extends from its center to the left, right, top, and bottom of it
- **spitball_stunned/self.rubberband_stunned/self.waterballoon_stunned/ self.banana_peel_stunned:** booleans that indicate if this Enemy has been stunned by a particular weapon. Only one of these should be True at any time.
- **rubberband_hits/spitball_hits/wb_drench_time:** ints that represent counts of how many times this Enemy has been hit by rubberbands, spitballs, and/or colliding with the splash animation for the water balloon. This Enemy will have any one of the instance variables previously described representing a stunned state set to true once a corresponding counter reaches a certain amount (see **3.4.8 Classes for Weapons** for more information)

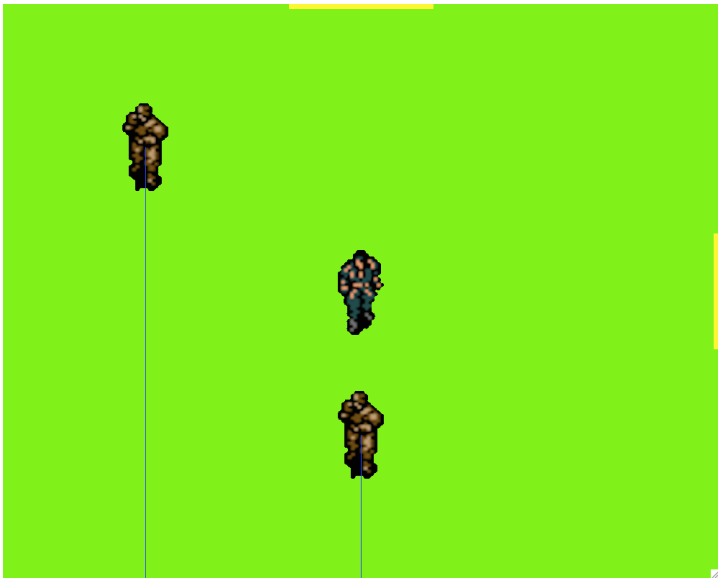
3.4.4.2 Methods:
- **__init__(window: Canvas, trigger_happiness: [int], keep_dist: bool, chase_speed: int, patrol_speed: int, attack_box: (int), hover_dis: int):** constructor for an Enemy Sprite that initializes all of its attributes. An Enemy can use information about the window it’s inside of to adjust the length of its LOS (described shortly). Settings that affect this Enemy’s AI are passed to this method as arguments.

- **init_position(x: float, y: float):** initializes the position of an Enemy within an Area.
- **update(collidables: Group, snake_group: Group, s_attack_obj_group: “Group”):** Sets the necessary attributes to allow a mercenary to engage in behavior specific to either patrol phase or alert phase. Such behaviors will only be exhibited if this Enemy is not stunned. When in patrol phase, update() processes instructions, which comes in the form of a list of 2-tuples, for the patrol route of an Enemy. The first instruction in an enemy’s list of instructions (i.e., patrol_route) will be parsed. The 2-tuple’s first component (the movement) is put into a variable called movement and the second component (the units of movement [or the number of iterations of the game loop, essentially]) is put into a variable called units. The following is a list of all the movements an Enemy can “understand”:

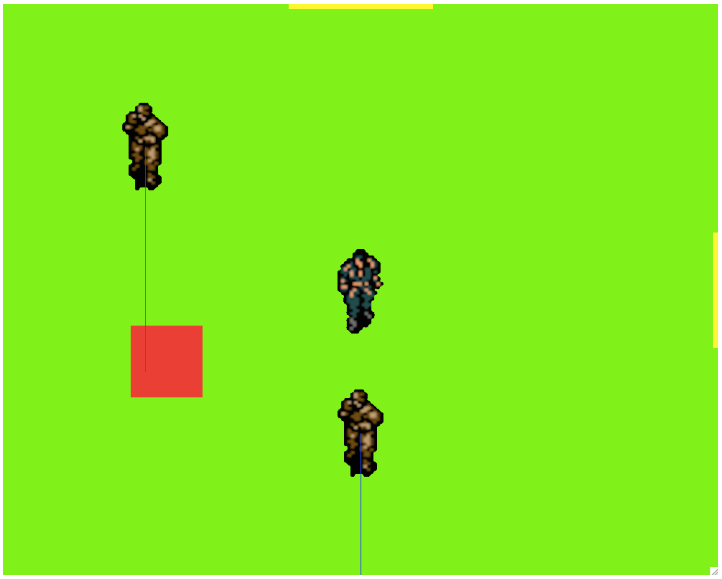
1. “go_east”
2. “go_west”
3. “go_north”
4. “go_south”
5. “wait”
6. “look_east”
7. “look_west”
8. “look_north”
9. “look_south”
10. “end”

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 end of this section for the Enemy’s patrol phase 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 and Snake will lose health for coming into contact with this Enemy. 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 at the top of the next 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 pierce every object in its path if it weren't for this adjustment of shortening it (by constructing a completely new LOS) in the event that it is obstructed. That is, without the adjustment, the LOS would go through 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.

When an Enemy is in alert phase, the behavior it follows corresponds with what was described in **2.5 Artificial Intelligence**. The basic principle behind the AI is to determine where Snake is with respect to this Enemy's x- and y-position. If Snake is less than/ greater than this Enemy's x-coordinate, then this Enemy will move left/right to get nearer to Snake until he is within this Enemy's attack box. Similarly, if Snake is less than/greater than this Enemy's y-coordinate, then this Enemy will move down/up to get nearer to Snake until he is within this Enemy's attack box. Having the self.keeps_dis attribute set to True will carry out checks to see if Snake is greater than the hover_dist attribute. If he is, then this Enemy will engage in movement that maintains his distance from Snake until it is essentially on the same vertical or horizontal line as Snake, in which case this Enemy will move directly towards Snake.

As mentioned, the update method() also handles cases where this Enemy has been stunned by Snake. Information on the different stun times associated with each weapon is described in **3.4.6 Classes for Weapons**. The principle behind being stunned is that an Enemy will not perform any of the behaviors associated with being in patrol or alert phase while stunned. Instead, it will assume a stunned state where it will perform no movement-associated updates for a designated number of game iterations (except for the animation of being stunned)

- **_handle_LOS_collision_with_obj(los_collision_lst: [Sprite]):** 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 based on which direction this Enemy's LOS spotted the object.
- **_handle_LOS_collision_with_snake(snake_group: Group):** 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(width: float, height: float, x: float, y: float):** Constructs an LOS with a given width and height and also sets the position of the LOS with a given x and y position.
- **_animate(imge_lst: [str]):** 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](units: int) helper methods:** 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.

3.4.5 Block (inherited from pygame's Sprite class)

A very simple class that is used to establish the walls of an Area.

3.4.5.1 Instance Variables:

- **image:** represents the visual look of the Block. An image file stored locally can be set to this attribute and will be scaled accordingly to fit the Block (a process carried out in this Block's __init__ method). If no image is specified, then this attribute is simply set to a Surface object with a blue fill.
- **rect:** the rectangle in which the Sprite "lives".

3.4.5.2 Methods:

- **__init__(width, height):** initializes the instance variables of this Block
- **init_position(x, y):** places this Block onto the game display. This is accomplished by setting this Block's top-left x and top-left y coordinates to the given x and y values

3.4.6 Classes for Physical Objects

. Classes for physical objects (e.g., sofas, tables, garbage cans) all contain very similar code. The chief difference among them is that for some physical objects, a directional/ positional orientation can be set from them to make them face a certain way. For example, long tables can be positioned horizontally and vertically while sofas can face north, east, west, or south. Square objects (e.g., square tables) require no such attention to orientation. The following is the basic set up for the class of some physical object:

```
class SomePhysicalObject(pygame.sprite.Sprite):

    def __init__(self):
        '''Initializes the attributes of some physical object.'''
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("name_of_image_file")
        self.rect = self.image.get_rect()

    def init_position(self, x, y):
        '''Initializes the position of some physical object.'''
        self.rect.x = x
        self.rect.y = y
```

3.4.7 Classes for Collectible Objects

. All classes for the collectible objects in the game (i.e., weapons and items) derive from a Collectibles base class, which inherits from pygame's Sprite class. All differ by what

has been set for their image attribute and all have an init_positon method. See **Game Specifications** for a list of all collectible items

3.4.8 Camera (inherited from pygame's Sprite class)

. Cameras are essentially Enemies. The chief difference between Cameras and Enemies is that Cameras don't take damage, they only ever "look" in a single direction, and their patrol routes can consist of only strictly north and south movements (if the Camera is facing east or west) or only strictly east and west movements (if the Camera is facing north or south). Cameras will be positioned with their back-end flush against a wall to simulate a real surveillance cameras typical positioning against walls.

3.4.9 HUD

. A class that represents the HUD for Snake.

3.4.9.1 Instance Variables:

- **font:** a pygame SysFont object that will be used to render strings onto the game display
- **health_label:** a string with the value "HEALTH"
- **lives_label:** a string with the value "LIVES"
- **weapons_label:** a string with the value "WEAPON"
- **item_label:** a string with the value "ITEM"
- **snake_health:** stores the current health of Snake
- **snake_weapon:** a Collectible object that stores the current weapon of Snake
- **snake_item:** a Collectible object stores the current weapon of Snake
- **snake_lives:** stores the number of lives Snake currently has

3.4.9.2 Methods:

- **__init__(width, height):** initializes the instance variables of this HUD
- **draw():** renders all strings stored as instance variables and blits them to this HUD. Snake's health bar is represented by a rect Surface object with a width equal to the number of hit points that he has
- **update():** updates the values represented in the HUD to coincide with Snake's current attributes

3.4.10 Classes for Weapons

. Classes that represent the weapons of the game when they are in attack form. E.g., when Snake presses the space bar to toss a water balloon through the air. That water balloon is represented by one of these classes. Likewise, the projectiles fired from a mercenary's stun rifle is represented by one of these classes.

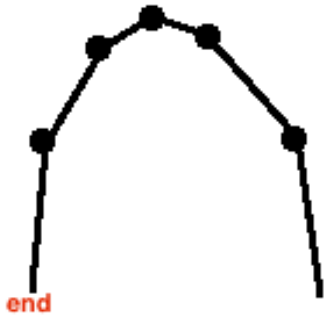
3.4.10.1 Snake's Weapons

All classes for Snake's weapons have a name and stock (think: ammo) attribute. All weapons that are fired have similar behavior in that they travel in a straight line

beginning at where Snake is. Spitballs travel at 25 pixels per frame while rubber bands are a bit faster at 45 pixels per frame. Banana peels are simply placed at Snake's current position. The amount of hits an Enemy would need to take before becoming stunned varies between weapons. In addition, the stun times also vary. The number of times an Enemy has been hit by a particular weapon will be accumulated in the appropriate instance variable and will not be reset until it is stunned.

* Spitball: 3 hits to stun. Stun takes effect for 35 frames
* Rubberband: 2 hits to stun. Stun takes effect for 40 frames
* Banana Peel: an Enemy must simply collide with a BananaPeel object in order for the stun to take effect. Stun takes effect for 50 frames.
* Water Balloon: an Enemy must have collided with the splash image for a WaterBalloon for a total of 5 frames in order for the stun to take effect. Stun takes effect for 55 frames.

The animation for the water balloon is the most complex out of all weapons. In order to represent the parabolic trajectory of the balloon, the WaterBalloon object would travel diagonally for a set number of ticks. Next, the object will switch directions towards a diagonal less steep than the last. The animation progresses in a similar fashion until it the parabolic arc is completed. The arc on the right represents a water balloon's trajectory (assuming Snake threw it while facing west) with dots to indicate changes in direction after a specified number of ticks.



3.4.10.2 Mercenary Weapons

The distinguishing behavior of a mercenary's (i.e., an Enemy object's) weapon (a stun rifle) is that it can travel in a spread pattern (see *Attacks and Damage Done By Enemies* in **Section 2.3 Rules and Mechanics**). Depending on which way an Enemy is facing, three stun rifle bullets (represented by the class StunBullet) will be created inside of an Enemy's update method. The individual StunBullets are passed a string argument that specifies the orientation of a StunBullet. For example, if an Enemy is facing south, then three StunBullets, each with an orientation of south, southwest, and southeast. Below is the update() method of a StunBullet, which shows how a StunBullet's movement progresses depending on what its orientation is (P_SPEED is the number of pixels the StunBullet will move per iteration of the game loop):

```
def update(self):
    '''Moves the StunBullet.'''
    if self.orientation == "north":
        self.rect.y -= P_SPEED
    elif self.orientation == "northeast":
        self.rect.y -= P_SPEED
        self.rect.x += P_SPEED
    elif self.orientation == "east":
        self.rect.x += P_SPEED
    elif self.orientation == "southeast":
        self.rect.x += P_SPEED
        self.rect.y += P_SPEED
```

```
elif self.orientation == "south":
    self.rect.y += P_SPEED
elif self.orientation == "southwest":
    self.rect.y += P_SPEED
    self.rect.x -= P_SPEED
elif self.orientation == "west":
    self.rect.x -= P_SPEED
elif self.orientation == "northwest":
    self.rect.y -= P_SPEED
    self.rect.x -= P_SPEED
```

3.4.11 Unique Characteristics of Bosses

A class exist for each of the individual bosses (and Boo). As alluded to in **Section 2.5 Artificial Intelligence**, the code is virtually identical to the Enemy class, but with a few adjustments. A common characteristic between all bosses is that, unlike mercenaries, they possess an animation frame associated with firing their respective projectiles.

3.4.11.1 Professor Kay

Represented by the Kay class. Movement is identical to mercenaries, but once he gets Snake in/on his attack box, which is 250 pixels all around, he will fire hearts in all directions (i.e., to the north, northeast, east, southeast, south, southwest, west, and northwest).

3.4.11.2 Professor Thornton

Represented by the Thornton class. Stays fixed at the top of the screen (in particular, 10% of the screen's height from the top) and can only move east and west. The othello titles he throws do not come out in a spread pattern like a mercenary's attack; rather, othello tiles travel in a straight line from his position until they collide with Snake or the bottom edge of the screen.

3.4.11.3 Boo

Represented by the Boo class. Identical to a mercenary except Boo cannot shoot any sorts of projectiles.

3.4.11.4 Professor Pattis

Represented by the Pattis class. Similar to Professor Kay and differs in the sorts of projectiles he fires. His projectiles are fired in a spread pattern and are represented by a class called Readings. Readings behave similarly to StunBullets, but they have an attribute called tick_lim that serves as a sort of time limit for how long the reading will soar through the air before it freezes, simulating a state of falling to the ground. An associated attribute, ticks, is incremented in every call to a Readings' update() method. Thus, a Reading will keep on moving as long as its tick attribute value is less than tick_lim. I.e., if the orientation of the Reading is north, then

```
if self.orientation == "north" and self.ticks < self.tick_lim:
    #move the Reading north
```

3.4.11.5 Professor Frost

Represented by the Frost class. The projectiles Professor Frost shoots moves like that of a mercenary's but are larger and do more damage. The main gimmick for Frost's battle is that he follows Snake from room to room. This is accomplished by letting each Area that Frost can travel through have a handle on a Frost object. That is, multiple Areas on the fifth floor will be passed a Frost object as an argument so that Frost can be drawn and manipulated within that Area. The effect of having Frost following Snake from Area to Area is achieved by a method called `_frost_transition`, which can be found in the Area base class. `_frost_transition` is called whenever a new Area transition is made, and it initializes Frost's position in the new Area several pixels *behind Snake's position.* This makes it so that Frost initially exists offscreen until Snake starts moving around the Area and causes Frost to pursue him.

3.5 BACK-UP AND VERSION CONTROL PLANS

Each group member will develop sets of their own code that they send to all other group members (through Facebook and Slack) at the beginning of each week. The project leader is in charge of integrating all of this code together.

4 —— Schedule and Personnel ——

. *Khoa Hoang:* Project Leader and Lead Game Designer. Responsible for directing all of the tasks each team member is to focus on each week. Will also be programming all of the game's core functionalities — essentially, all aspects of the game that aren't involved with the level design. Core functionalities include the likes of Enemy AI, enemy and player movement, and menu screens. Will also direct level designers on how levels (i.e., the areas in each floor of DBH) should look.
. *Jeremy Chao:* Lead Level Designer. Responsible for programming the in-game design of all the Areas in the game and collaborating closely with the project leader on how each Area needs to look.
. *Kha Trang:* Assistant Level Designer. Will assist the lead level designer on parts of the level design including placement of enemies and design of "inner" Areas (see **2.8 Levels (or Floors/Areas))**
. *Patrick Zhang:* Location Scouter and Lead Artist. Monitors the details of the real-life look of floors 1, 2, and 3 of DBH and provides the level designers with the appropriate details. Will also draw the images for the sprites associated with the game's bosses and miscellaneous objects (e.g., couches). May also provide assistance to the lead and assistant level designers in coding the floors.
. *Vu Nguyen:* Location Scouter and Assistant Artist. Monitors the details of the real-life look of floors 3, 4, and 5 of DBH and provides the level designers with the appropriate details. Will also assist the lead artist in drawing the images for the sprites associated with miscellaneous objects (e.g., couches). May also provide assistance to the lead and assistant level designers in coding the floors.

Week 6

. *Khoa:* Begin integrating all of the code written for the core functionalities of the game to the game world itself and program the AI for all of the bosses

. *Jeremy:* Begin adding finishing touches to floors 1 and 2 (e.g., colors of floors, walls, and 3D wall effects) and code inner transitions for accessible doors on floors 1 and 2 (Many doors will be accessible, but Khoa will decide on which ones to "close" once the whole building is complete)

. *Kha*: Get accustomed to mercenary patrol route programming and devise mercenary patrol routes for Area 1_2, Area 1_3, and Area 1_4

. *Patrick:* Draw Professor Thornton, Professor Kay, and all of their associated animations frames

. *Vu:* Get accustomed to mercenary patrol route programming and devise mercenary patrol routes for Area 1_5, Area 1_8, Area 1_9, and Area 1_10

Week 7

. *Khoa:* Implement surveillance cameras, the scoring system, and oversee/direct Jeremy on how to code the final touches to the floor designs. Decide which doors will be accessible on each of the floors for the sake of variety between floors

. *Jeremy:* Finalize the finishing touches to floors 1 and 2 (e.g., colors of floors, walls, and 3D wall effects), inner transitions for accessible doors on floors 1 and 2, and write code for floors 3, 4, and 5 (which should heavily reuse code from floors 1 and 2) to complete the DBH building

. *Kha*: Devise mercenary patrol routes for Areas 2_2, 2_13, 2_14, 2_15, 2_17, and 2_18. These patrol routes will be reused for floors 3 and 4

. *Patrick:* Draw Professor Pattis, Professor Frost, and all of their associated animation frames

. *Vu:* Devise mercenary patrol routes for Areas 2_19, 2_9, 2_4, 2_5, 2_6, 2_7, and 2_8

Week 8

. *Khoa:* Finalize integration of the game's core functionalities with all the level design code and the code for mercenary patrol routes, determine item placements, and test out the various core functionalities within this completed game world (e.g., enemy alerts, boss battles)

. *Jeremy:* Finalize code for floors 3, 4, and 5 and test out all transitions in and between floors 1 and 2 to see that they're functioning properly. Also test out mercenary patrol routes and alert phases for difficulty/bugginess

. *Kha:* Test out all transitions in and between floors 3, 4, and 5 to see that they're functioning properly. Also test out mercenary patrol routes and alert phases for difficulty/bugginess

. *Patrick:* Draw furniture and other physical objects that make up the Areas of each floor (e.g., pillars, trash cans, toilets)

. *Vu:* Assist Patrick in drawing the furniture and other physical objects that make up the Areas of each floor

Week 9

. *Khoa:* Incorporate surveillance cameras into some of the Areas of the floors. Play through the game to determine how to balance the scoring system

. *Jeremy:* Play through the game and fix any bugs. Ideally, allow colleagues to test out the game also and be open to any suggestions they might have. Tweak mercenary patrol routes on floors 3 and 4 so that they're a bit different from floor 2

. *Kha:* Play through the game and fix any bugs. Ideally, allow colleagues to test out the game also and be open to any suggestions they might have. Tweak mercenary patrol routes on floors 3 and 4 so that they're a bit different from floor 2

. *Patrick:* Continue to draw furniture and other physical objects that make up the Areas of each floor

. *Vu:* Continue to assist Patrick in drawing the furniture and other physical objects that make up the Areas of each floor

Week 10

. *Khoa:* Incorporate easter eggs. Play through the game for possible improvements/bug fixes. Make gameplay video

. *Jeremy:* Play through the game for possible improvements/bug fixes

. *Kha:* Play through the game for possible improvements/bug fixes

. *Patrick:* Continue to draw furniture and other physical objects that make up the Areas of each floor. Play through the game for possible improvements/bug fixes

. *Vu:* Continue to assist Patrick in drawing the furniture and other physical objects that make up the Areas of each floor. Play through the game for possible improvements/bug fixes