CS3021 Big Project
Max Geiszler
Chris Norine
Greg Norton

Setup:
Clone the repo using git:

```
git clone https://github.com/maximusg/GameShazy.git
```

Our game has two dependencies:

1. Pygame
   Install PyGame on your machine:
   A. Use the pip examples here: https://www.pygame.org/wiki/GettingStarted
      -python -m pip install -U pygame --user
   B. Also needed to run the following because I guess it is a pygame dependency:
      -pip install msgpack
   C. Run to verify correctly installed
      -python3 -m pygame.examples.aliens

2. Numpy
   Install Numpy on your machine
   A. Use the pip example here: https://scipy.org/install.html
      -python -m pip install --user numpy

Once these have been installed, run GUI.py to launch the game. Originally, we were planning to use Pyganim to handle our sprite animations, but we ended up handling all animations within pygame, so we didn't use Pyganim at all.

Code documentation can be found inside the documentation subfolder, under _build/html/index.html.

Data Structures:
Our project employed two different data structures:

1. A linked list used for implementing high scores, located in the "highscore.py" file (with persistence).
2. A JSON level maker was used for creating persistent game levels that can be written to disk, and subsequently loaded into the game using our level loader. These can be found in the "levelMaker.py" and "levelLoader.py" files.

3. An asset “manager” utilizing a dictionary and loader methods. The first time a particular path is “loaded” by pygame, it serializes and stores the surface in the dictionary with the path name as the key. Subsequent requests for a specific file (determined by path) are given an unserialized version of the stored surface, rather than executing a new call to the underlying file system. The best example is that without this, each individual bullet sprite created during a level will generate a call to the underlying file system (so thousands of calls just for the “spitfire” bullet) while the asset manager, acting as a middle man, can ensure that the program only ever reaches into the file system once for the “spitfire” image, and keeps the rest in the working dictionary.

Required Techniques:

1. Basic object functionality: We use an Entity class to represent the various in-game entities that the player can encounter, such as enemies and items. This can be found in “Entity.py”, line 21
2. Exception handling: When a level is loaded, the JSON structure is examined, and if it doesn’t fit our intended format, an exception is raised. Found in “levelLoader.py” line 72.
3. Basic inheritance: The Enemy class inherits from the Entity class, as mentioned in point 1 above. Found in “Entity.py” line 269.
4. GUI: Our GUI provides the following functionality: starting a game, loading a game, viewing the credits, viewing the high scores, and quitting gracefully. These buttons are defined in “GUI.py”, line 758.
5. Functional Programming: When the playerShip fires a bullet, the appropriate function is called within the playerShip’s Weapon instance. Found in “Entity.py” line 191 which maps to “weapon.py” line 85.
6. Unit Tests: We have defined a test suite in “unitTestMain.py”, which runs Unit Tests defined in files such as “TestEntity.py”
7. Framework/Library: We built the game using Pygame.

Optional Techniques / Bonuses:

1. Grab Bag Topics: We use a library of commonly used constants and functions throughout the project, found in “library.py”
2. We used JSON as our second framework/library for our level structure.
3. Data export / persistence: We used JSON to export levels, as well as the python pickle module to export information for saved games.

Personal Experiences:

Christopher Norine:

- I was both a fan of and terrified by the amount of rope we were given working on this. I focused mostly on the window drawing and management (including game states), which was definitely an enlightening experience. I learned a lot and it has

convinced me to start deep diving on other more advanced game engines and research that might be accomplished using them.

- High Score List
    - We ended up utilizing a linked list as the underlying data structure. Adds are executed by iterating through the list until it finds where it belongs, then inserts itself there.
    - Persistence is achieved via a simple write out of the list in text to a file, which is then read back in whenever the high score list is initialized the next time the program runs.
- Game State
    - Each “state” of the game, whether it be the introduction crawl, the main menu, or the main game itself tends to follow the same overall pattern:
        - Initialize specifics for that state
        - Loop
            - Check if the loop should end
            - Get input for the frame
            - Update sprites (including collision detection)
            - Draw the screen
            - Advance time
    - The “level_loop” method helps facilitate the main game by ensuring that every level is treated as its own game state. This helped with garbage collection in between levels, since we’ve effectively “returned” from that level’s method, then loading into the next level’s method, while also allowing us to more easily insert hooks for the ability to load a game from a saved file.
- Drawing a UI plus “main” game window
    - My approach to drawing the screen was mostly to utilize a binning method where all sprites on the screen were grouped by what they were thus determining the order they were drawn (i.e. layer).
    - Out-of-bounds detection was also a large part of keeping frame rates high. Bullets that left the screen would continue on forever (getting drawn off screen), since pygame does not have any sort of built in “on screen” detection. I worked closely with Max and Greg to ensure that any sprites built had to abide by the rules. They either were forced to remain on screen (i.e. the player ship) or if they left the screen (bullets/enemy ships) then the sprites would immediately be removed from any active sprite groups and left for garbage collection (end of the level).
    - Pygame offers a number of different versatile options for collision detection, ranging from a simple “my rectangle is touching your rectangle” detection down to sprite mask collision or even point collision. For speed and simplicity (since there was a great deal of checking to do) we opted to utilize the simple rectangle collision method. Although it is not pixel perfect detection, at the speed sprites were moving it is almost

unnoticeable. Binning here came in supremely handy, as we were able to check bins against other bins (playerShip and enemy bullets, or player bullets and enemy ships), instead of doing some sort of check of every single sprite against every other sprite along with the specific handling for any of our cases.

- AssetLoader
    - In the quest for more framerate one thing I noticed with regards to pygame was that pygame Surfaces (the image itself) do not lend themselves to constant creation/duplication due to the nature of their storage in memory. Surfaces are not contiguous in memory, and as such, are not easily serializable by Python. My initial thought was to use some sort of dictionary to store an entire Entity, and just execute some form of deepcopy on the "master copy" every time I needed a new one, thus we could create a single copy of each sprite we would need, and then generate copies per requirements within the levels. What I ended up with was using a few more advanced pygame modules that allowed me to serialize any Surface into a 3-dimensional array (using numpy under the hood) which then could be stored in a dictionary. Personally, I want to think that this is a big boost by reducing pygame.image.load() calls, but I'll get into a little about why we don't really see any sort of improvement, even though I feel it is supremely clever.
- Save/Load Game
    - In the interest of not dumping the entire game state as a whole into a file, we discussed the things we *absolutely* needed to start a level somewhere that wasn't time = 0, level = 1. So we agreed that the state of the player ship, current time, and current level were all that was required. So the function to save will take those attributes, pickle into a file, and then gracefully return back to the main menu.
    - Loading was accomplished by inserting hooks into our "level loop" method. By giving level loop the ability to pass the player ship state from the file back into the main() method (which was already done since we needed persistence between levels in a single session), and being able to initialize our levelLoader with any level (assuming it exists) we were easily able to just "restart" a game by setting the "current time" in the level to the time saved, which allowed us to use levelLoader's getEvents() method to just start returning sprites at our new updated time, without forcing any sort of work to be done on times before our save game.
- The Good, the Bad, and the Ugly
    - The Good
        - Game states. By keeping everything in its own state, it simplified everything and kept us from having one massive "while(true):" loop that would make debugging a mess.

        - I strictly (almost ruthlessly, if you ask others) enforced the requirement that entities that found themselves on the screen at any given time were absolutely not allowed to modify the screen directly or modify other elements on the screen. This was mostly an extension of the rule “don’t mess with things that don’t belong to you” (Professor Peitso, CS3021, sometime during AY19 - Winter Quarter). However, it made debugging immensely easier, since a draw call being wrong is easier to pinpoint only having to search a single file at any time. Every sprite was updated on a frame-by-frame basis, so the flow was more like:

            GUI tells sprites to update -> sprites all update -> GUI now uses updates to redraw screen
    - The Bad
        - Given the opportunity, I do not know if I would utilize pygame as a framework in the future. Dealing with assets, I found myself constantly frustrated working through strange quirks. The biggest example was probably that alpha channel detection in pygame was lackluster at best. This resulted in a number of minor graphical glitches, the easiest to see is when a large number of explosion sprites are drawn on the screen, they have a tendency to not have the transparent pixels stay transparent, and instead draw them with the color black.
        - The AssetLoader is a constant source of mixed feelings for me. Although I felt like I really found a simple way to extend a dictionary’s functionality to increase performance, I felt that the actual effect is almost negligible. But as an idea, I felt it definitely had merit, and if we were utilizing it in some manner to execute calls to pygame’s gfxdraw module (which is still experimental and unstable), we may see some real increase in performance.
    - The Ugly
        - Timekeeping is a bit of an “other” with the GUI. We agreed initially to limit the game to 60FPS. At the time it seemed like a fairly pragmatic choice, since time updates became easier knowing that 1 frame was 1/60th of a second, and could do all our math based on that. However, frame rates aren’t always steady, which can cause some unintended side effects when you consider the fact that entities are spawned in the game based on time (we did not hard code in any spawns). As frame rate decreased, we would see entities (bullets/enemy ships/etc) spawn much closer to one another than intended, since their movement was coded on a per-frame basis. Pygame does have the ability to give us a “time differential” which is the amount of time between tick() calls. This would have allowed entities to update proportionally to the amount

of time between ticks, and give a more consistent presentation regardless of whether the program is rendering 300 frames per second or only 3. This also hobbled any sort of real performance gain seen from the AssetLoader since we were already limiting ourselves to 60 FPS and it would be difficult at best to quantify any gains from using it versus utilizing the standard under-the-hood processes that Python utilizes.

Max Geiszler:

This project was a lot of fun. A couple of things I would have done differently.

-Unit tests first. I know you mentioned to do these first at the beginning of the course, but we didn't know how until half way through our game on how they were supposed to be implemented. By then we were so far into our code that it all naturally got done towards the end.

-Use python packages over re-inventing the wheel. I ended up re-creating a physics engine for our game, and could have used a popular physics engine called pymonk, which did everything I was trying to accomplish + more.

-Do less. We went hard, and tried to do more than we had time to do. It ended up taking a bite out of time larger than necessary for this project, and made class-work balance a little off kilter putting much more time in to this project than you told us we needed from the beginning, and we ended up scrapping a lot of code which ended up not going to use.

-Level loading. The approach we took for loading level ended up working great, but after implementing, I found ways to streamline auto-generation and mass enemy production methods which where not thought of prior. This left us with a lot of legacy code after implementation, because that code got re-used a few other sections, which we didn't have time to go back through and change, so we carried that code fwd in different parts of the program. It was only towards the end that we did a massive cleanup of that old code, and it was a lot of work, plus we had to do bug-checking… AND we didn't have unit tests built then so we really were not helping ourselves in that respect….

Greg Norton:

I focused on the weapons and powerup system.

-Standard API: since the weapons involve lots of interaction with the various components of the game, I had to learn where all the different code lived in the project, so that I could add to it and modify it if necessary. Eventually we settled on a system: GUI polls for keystrokes from the user, if the weapon fire key is pressed, it creates a Bullet Entity using the playerShip's Weapon scheme, this is then blitted onto the screen. In a larger project with more moving pieces, it would have been very difficult to trace through a similar system. If we were to keep adding

to this project, it may be helpful to add standard API's to help manage the interaction between these pieces.
-Functional programming: At first, I had defined a weapons dictionary that included weapon attributes such as damage, speed, a path to the weapon image, etc. This dictionary also included a reference to a function that would define a Bullet sprite using these attributes. This was a major headache, because Python required the weapon functions to be defined before the dictionary that contains these references. In order to allow weapon functions to reference the weapons dictionary, I moved the references to functions into their own dictionary.
-Animations: Managing an animation associated with a weapon was difficult, as lots of communication had to happen between the Weapon object and the GUI. This is more of a Pygame drawback, and if we'd had more time or had more animations to incorporate, it would have been a good idea to use a library meant to facilitate Pygame animations, or defined our own protocol for dealing with them. But I am happy with how they came out overall for this project.
-Movement: Bullet and Bomb movement was implemented using the same technique as moving enemies, which involved a Movement class. This ended up being difficult, because as the Movement class was revised, bullets stopped working. Overall, this was a good investment, because using the Movement class allowed for greater flexibility and code readability, since there is a single system moving in game objects.