To The Stars: An Interactive Simulation To Explore Our Galaxy's Stars!

Janna Alyssa Lim, Jenny Nguyen

April 3, 2023

Introduction

To say that space is gigantic would be an understatement; there are perhaps an infinite amount of galaxies out there that we have yet to explore. It's intriguing to know what might be out there, and so we have dedicated organizations like NASA whose purpose is to learn more about the world beyond our planet.

With all of this new information constantly being discovered, it can be difficult to deliver this data in a digestible way that's easy for everyone to understand, especially the young students of today. This begs the question: **How do we create an informative, yet simple and interesting way to present complex information about our universe to everybody?**

Datasets

To display information about our universe, we'd first need to obtain that information. This is why we decided to settle on a dataset that we found on Kaggle (https://www.kaggle.com/datasets/thedevastator/properties-of-stars-in-our-galaxy). Specifically, we used one of the five .csv files that it contained: final.csv. It contains information about the stars in the Milky Way. Specifically, it contains information on a star's name, distance from Earth in light years, radius in solar radii, and mass in solar masses. We used the Star_name, Distance, Mass, and Radius columns.

Computational Overview

We created a Star, Galaxy, Path, and Ship data class to represent the data in the data set, and to display its contents interactively.

The Star data class represents each star and is a subclass of the `pygame.sprite.Sprite` class. Its name is a str value, and the star's distance from Earth, radius, and mass are stored in float values individually within the class. The Galaxy class represents the Milky Way galaxy, where all of the data set's stars are from. This data class keeps information on what stars are found inside of it using a mapping of star names to its Star object and contains several useful methods which we'll describe later on. The Path data class represents the "connections" between the stars and their neighboring stars and keeps a record of which Stars it's connecting by having one of its instance attributes be a tuple with each of its endpoint Stars in it. The Ship data class is a subclass of the `pygame.sprite.Sprite` data class and it represents the player inside the simulation. It contains information on the current Star that it's at (i.e., the Star that the Ship had last touched).

Here, graphs are central to our project and its methods to make our display fun, educational, and interactive. Each Star, Path, and Galaxy are analogous to the Node, Channel, and Graph classes that we learned in lecture, respectively. Using these paths between the Stars, the player can generate different visible paths based on the Stars radius or mass, and the Ship's current Star. These paths provide an educational way to learn about the attributes of the stars in a fun and interactive way, which helps create a welcoming learning environment.

As for how these simulated stars, galaxies, and paths came to be, we'll start by describing our first major computation: reading the CSV file and initializing a Galaxy with all of its Stars. First, we skip reading the first row in the final.csv file, as it refers to the header row with the column names, then initialize an empty Galaxy. Then, for every column in the row, we assign it to a variable that corresponds to what data it represents of the star. Finally,

we create all of the Stars and add them to the new, initialized Galaxy and return a tuple containing the Galaxy and a list of all of its Stars. Afterward, we'd then organize the list of all of the Stars by lowest distance to highest distance.

For our second major computation, we generated Paths between each Star using the method `create_paths` under our Galaxy data class. Once we initialize a Galaxy with all of its Stars using the `read_dataset` function (described above), we then call create_paths on that Galaxy to generate Paths that connect every Star with its four surrounding Stars (i.e., the two on its left and the two on its right, or the four neighboring Stars of the Star from the list that we generated from `read_dataset`). For the first and last Stars, they'd only be connected to their two closest Stars, and the second-first and second-last Stars would be connected to their three surrounding Stars.

Creating a visible path for the player requires an algorithm that filters the data of the Ship's current Star's neighbors, then its chosen star, and so on. To describe how this algorithm works we'll be focusing on the radius_path function (as the `mass_path` function fundamentally does the same thing, but instead accesses the Stars' mass attribute rather than the radius one). This function is supposed to analyze the neighbors of the `starting_star` (which would first be the current Star) and then out of its neighbors (i.e., the stars that it shares an immediate Path with) it chooses the one that has the lowest radius and also hasn't been visited yet. Then, that Star is chosen, and the cycle continues using recursion until it hits a point where all of the last Star's neighbors have already been visited, and so the generated visible path comes to an end. The returned value of this function is a list of tuples, referring to the Stars' coordinates on the screen to display later on in Pygame. These coordinates help us generate a line that visualizes these Paths.

To show all of these computations visibly, we used Pygame to create a game-like simulation of the Galaxy and its Stars ordered from left to right starting with the lowest distance to the highest distance. The Ship becomes a moveable Sprite that the user can control with their arrow keys to explore more of the Galaxy and the Stars that are off-screen. When the Ship touches a Star, in the bottom-right corner it will display the Star's name, radius, and mass, and in the bottom-left there are instructions for the viewer on how to create a visible Path, using the algorithm that we described above. When the player clicks 'r', the Paths will be shown using the `radius_path` method, and when the player clicks 'm', the Paths will be shown using the `mass_path` method.

Instructions

Download the data set that we used for our project from MarkUs. There should be one CSV file within it, labeled "final.csv".

After installing all of the libraries in `requirements.txt`, run `main.py` in the python console and turn on your volume.

Once this happens, you should expect to see a Pygame window open within around twenty seconds (if there's a Pygame window that appears with a black screen, don't click anything. Instead, keep waiting until you see the visuals appear).

You'll know when the visuals appear when you see multiple stars on the screen with a UFO, with a black picture and white spots on it in the background (to represent stars in the distance). This should look similar to the screenshots in the "screenshots" folder that we uploaded to MarkUs.

Use the arrow keys (i.e., up, down, left, right) to move the UFO. Once the UFO collides with a star, there will be information about the star displayed in the bottom-right corner of the screen, displaying the star's name, radius, and mass.

To create different visible paths using the algorithm that we describe in our computational overview, press 'r' to create the path via `radius_path`, and press 'm' to create the path via `mass_path`. You can hold down one of either key and use the arrow keys to explore more of what the path looks like off screen. If the UFO touches another star, the path will re-calibrate, with the ship's newest `current_star` attribute becoming the `starting_star` argument passed into the **..._path** functions.

Changes to our project plan

Much of our initial major computation ideas mentioned in our proposal have been cut out in our final implementation due to the complexity of each function. The majority of our time was spent figuring out how to visually represent the results of our `radius_path` and `mass_path` functions using Pygame; thus we did not have time to implement our Star Search function and flagging feature, as well as our fuel instance attribute for our ship. Initially, one of our core features was supposed to be generating all possible paths from one star to another star. However, since we represented our stars linearly there would only be one possible path, so we ended up scrapping that idea. Although we did not incorporate an actual view distance parameter for our ship as our TA suggested, we did end up implicitly integrating this into our game to make sure the screen didn't show too many stars. We made sure that we only showed at most 5 stars on the screen at a time. In the end, we maintained the initial visualization of our program which was a two-dimensional side-scrolling video game as described in our proposal.

Discussion

Our program successfully achieved our goal to display information about the universe, albeit only a small subset, in an interactive and digestible format. We were successfully able to map the stars in our dataset onto our game, and visually represent the relationships between the stars based on different parameters given by player input. The first challenge we faced while creating this program was generating the correct position values for the stars. Initially, we wanted to show the varying distance between the stars based on the value in the distance column on our dataset. However, this distance was in lightyears and its value was quite large - so we were unsure about how to represent such a great distance in a finite amount of pixels. In the end, we decided to sort the stars from smallest to largest distance from Earth, and then separate each star by the same amount of pixels.

The main challenge we faced was our side-scrolling feature. We created a `CameraGroup()` class that managed the positions of the stars relative to the current position of the ship. However, this class could only move sprite objects; so we had a lot of difficulty moving the lines generated by `radius_path` and `mass_path`. Initially, we were only able to see a small portion of the line. The lines were produced through `pygame.draw.lines()` which did not generate a sprite, but a rect object. We had hoped to tackle this by converting the line into a Line sprite class so we could use `CameraGroup()` to move the line accordingly. However, we had no value to pass into `self.image` and so we were unable to do this. We planned on just passing in an image of a straight line and creating a method to adjust the angle and length of the line depending on what stars it had to reach, but we realized that it was too complicated of a task. We also tried creating a `CameraGroup()` method to draw the lines instead of drawing it within the main game loop, but it didn't work either. In the end, we extracted the camera offset from our `CameraGroup()` class and used that to adjust the coordinates of the line in our game loop.

Some next steps for further exploration are implementing the functions we had omitted in the future, and choosing to visualize our stars in a circular motion akin to how the solar system is visualized. We could draw circles of increasing radii around Earth with stars dotting the outline of the circle. This would provide an accurate depiction of the distance of the stars from Earth; and allow us to work with a larger dataset of stars since we would be able to visualize it on a much larger scale. In the future, we could also incorporate planets into our program as well; and work on generating galaxies revolving around a certain planet besides the Milky Way and Earth. If we can implement this successfully, we could also work on turning this into a three-dimensional interactive map of the galaxy and show how the planets rotate and revolve around the Earth. The possibilities are endless. As we learn more information about space, we can implement even more functions into this program.

References

@TheDevastator. "Planets & Stars in Our Galaxy." *Kaggle*, 26 Oct. 2022, https://www.kaggle.com/datasets/thedevastator/properties-of-stars-in-our-galaxy.

"Pygame." Pygame Front Page - Pygame v2.1.4 Documentation, https://www.pygame.org/docs/.

"Cameras in Pygame." *YouTube*, uploaded by Clear Code, 12 Feb. 2022, https://www.youtube.com/watch?v=u7LPRqrzry8t=1939s.

"The ultimate introduction to Pygame." *YouTube*, uploaded by Clear Code, 11 Jul. 2021, https://www.youtube.com/watch?v=AY9MnQ4x3zk.

Mr. Scruff. "Kalimba (Ninja Tuna)." *YouTube*, uploaded by Calum White, 6 Aug. 2010, https://www.youtube.com/watch?v=tCO4i2t-Aso.

"video game beep-Sound effect." *YouTube*, uploaded by The Relaxer, 17 May 2022, https://www.youtube.com/watch?v=6KA7oa33pgk.