



ENG EC 327

Professor Densmore

Fall 2021

Team 13

Michelle Thevenin

Cole Wentzel

Jiawei Liao

Giacomo Coraluppi

Thomas Bowler

**Summary**

Protect the Planet is an entertainment game made for EC327 as a way to demonstrate our understanding of C++. In the game, asteroids spawn from the edge of the screen, as the player controls a spaceship and shoots down the asteroids to keep the planet safe. The game is controlled by using the WASD keys on the keyboard. Protect the Planet can be used by anyone for entertainment purposes in digital gaming. It is most catered to kids from the ages of 3-17. Protecting the Planet is on the agenda everyday!

**Front End**



**Figure 1**: Screenshot of game in action

The objective of the game is to prevent asteroids from plowing into Earth’s surface or striking your ship. If either Earth or the ship is hit too many times, the day is lost. The user moves the ship using the WASD keys. Holding A or D allows the player to spin the ship counterclockwise and clockwise respectively. Meanwhile W causes the ship to thrust forward in the direction that it is facing, and S will send the ship in reverse. Pressing the Spacebar will launch a bullet from the front of the ship. With just 5 user inputs, the game is quite user friendly since it does not take long to learn and memorize 5 intuitive commands, especially for anybody accustomed to flash games.

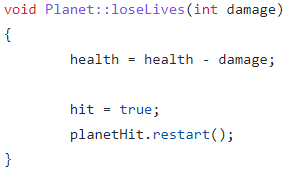
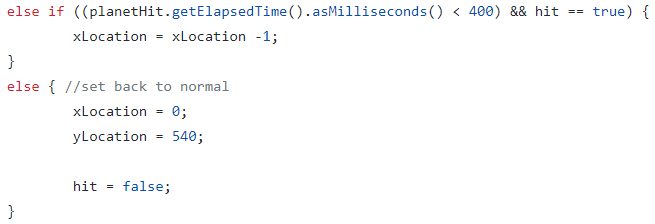
Half of the planet is visible on the left edge of the screen, where it appears to rotate by alternating through a few different sprites. Asteroids cascade unpredictably from any point along the edge on the right half of the screen. Once the asteroid spawns, it moves linearly to a randomized point along the planet. To really spice things up, the asteroids spawn in at a random speed so the user must prepare for all kinds of impending disaster. If the asteroid collides with another object, both objects lose 1 health point. The asteroid itself has 1 health point, so it is destroyed upon any impact. Asteroids also rotate a bit as they fly toward Earth, but this is achieved by actually rotating its orientation. 

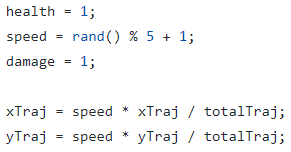
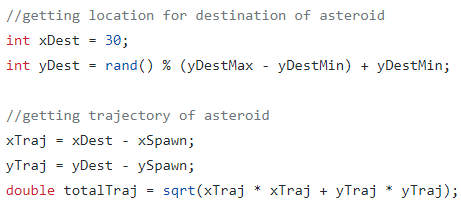
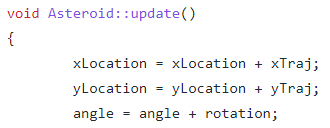
**Figure 2**: Possible asteroid spawn points, only along border of screen

In order to save the planet from such catastrophe, the user-controlled spaceship is of utmost importance. By firing bullets, the ship is responsible for destroying all asteroids that threaten the safety of the planet. Bullets emit from the ship and travel in a straight line from where they were launched. They only have 1 health point, as their sole purpose is to equalize any incoming asteroids. Upon impact, both objects are destroyed. The user should be very careful because the ship is not invincible: if it collides with asteroids 3 times, the user has lost. With no ship around, nothing can prevent the doom of the planet, so keeping the ship alive is just as important as keeping the planet alive.

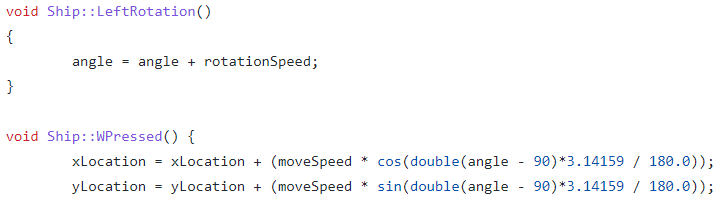
**Back End**

The game was created utilizing object oriented programming. Each object is rather easy to guess: Planet, Asteroid, Ship, and Bullet. The most fundamental part of ‘Protect the Planet’ is of course the planet itself, which is a simple stationary object. Earth has a health value of ‘10,’ which is the variable determining whether an object is currently active in the gameplay. The functions in Planet are loseLives() and planetShake(). loseLives() appears in each object, and decrements health based on the amount of damage dealt. planetShake is a visual effect that reacts to an impact, and quickly shifts positions based on the amount of time based since the impact before returning to its original position.

**Figure 3**: Planet::loseLives() and a snippet from planetShake()

The asteroid object is a bit more complicated because its position and angle is constantly changing; it can spawn in from a variety of locations, and there are multiple sprite possibilities. To first determine sprite and which screen edge it spawns along, rand() is used for 3 possible options. The update() function uses trajectory and speed calculations to determine the movement.

**Figure 4**: Trajectory logic (health and damage not related in top right image)

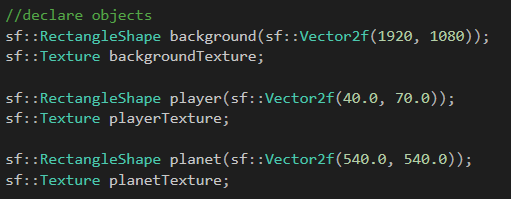
Ship is constructed with default location, health, and move/rotation speed values. The functions of ship are focused on its movement. To rotate left it adds a number of degrees from the current angle, while rotating right subtracts some degrees. Similarly, moving forward adds a value based on the speed of the ship, as well as what angle it is currently facing. If the ship crosses past the top pixel, it sets location to a bottom pixel and vice versa.

**Figure 5**: Example of ship’s rotation and linear motion

Conveniently, the Bullet object reuses a lot of logic from asteroid since they are both projectiles. The main difference is that its spawn and path are completely different. Its spawn location is based on ship’s location, and its path is determined by the angle of ship. Therefore, the bullet could travel anywhere and to account for this there has to be logic to test if it hits the edge of the screen, so that ‘graphicsmain’ can delete it.

One of the primary difficulties in making the game came from learning how to utilize SFML, the chosen graphics library. None of the code that we had done in class up to this point required a graphics library so creating a game with a user interface was something new.

We had to learn not only the functionality added by the library but also the best way to implement that functionality. For example, we initially tried to create our sprite objects in the main function, as demonstrated below:



**Figure 6**: Initial attempt with SFML

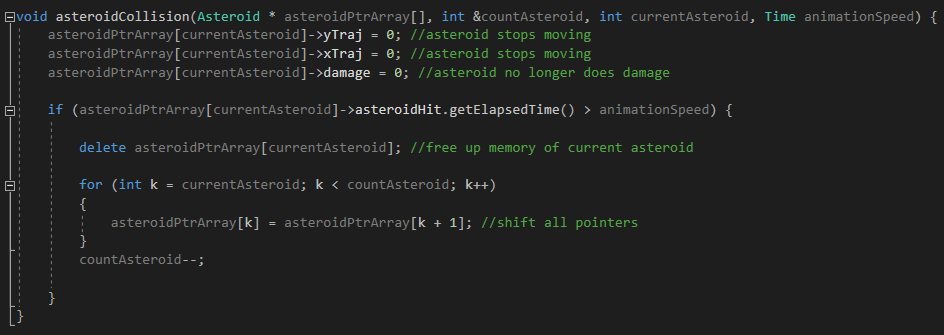
However, when it came to adding sprites for the asteroids, we realized that each asteroid needs its own unique sprite. Doing sprites in this way would require an array of sprites that is handled in the same way as the array for the asteroid pointers, which would be inefficient and redundant. Instead, we opted to add the RectangleShape and Texture as member variables of each class:



**Figure 7**: Putting SFML as variables in each class

We also had an issue with figuring out how to handle the procedurally generated asteroids. Assigning each new one an ID would require a huge array, and asteroids are destroyed fairly soon after being created, so the amount of game time would be limited by the size of the array storing the asteroids.

We solved this problem by creating asteroids on the heap and deleting them once they were destroyed due to colliding into a bullet, the ship, or the planet. Then, we added the asteroid collision function, which shifts all the pointers to the asteroids so there are no blank spaces in the array:



**Figure 8**: Asteroid array logic

We were also able to do this array pointer shift with the bullet class, which allows for infinite asteroids and infinite bullets without needing to use memory space for each new instance of each class.

Overall, the graphicsmain.cpp file handles all of the spawning and despawning logic, as well as all of the graphics. It also handles all of the timing operations, for example to keep the game interesting as the user learns how to play the game and adapts to the environment, difficulty ramps up every 15 seconds. Difficulty is based solely on the asteroid spawn rate, which decreases by 200 milliseconds every 15 seconds.

**Conclusion**

Protect the Planet is a fascinating and unique game that entertains the user through its fluid mechanics and simple but enriching gameplay. While the controls are fairly easy to learn, the game provides an increasing challenge that keeps the player absorbed over time. Meanwhile the graphics are visually appealing and captivate the user in the theme of outer space.

All of this was accomplished through the use of C++ object oriented programming learned in the EC 327 course. Creating several classes with different functions and variables was somewhat challenging on its own, but tying everything together to produce the actual game was much more challenging and complex. This required the skills of heap initialization and pointers, arrays, class dereferencing and function calling, nested loops, and plenty of boolean logic. The end product of Protect the Planet demonstrates a proficient understanding of an introduction to software engineering.