Grant Chen, Elizabeth John
COS 426 Spring 2022

2D Top Down Bullet Hell

Abstract

For our final project, we implemented a 2-dimensional bullet hell game where the player, represented by a yellow circle, is able to click to shoot a linear stream of circular bullets at randomly spawned enemies, represented by red circles. These enemies shoot a large and continuous pattern of bullets at the player, and the player's main goal is to dodge the bullets and shoot at the enemies until they are destroyed. The game begins with a start screen, in which the player is prompted to press the spacebar to begin. The player is then presented with the game scene, in which the player remains centered in the window but is able to move around with the W, A, S, and D keys. The score is also displayed at the top right corner of the screen, and is updated each time the player destroys an enemy. When one of the enemy's bullets hits the player, the game ends and displays "Game Over," and the player is able to restart by pressing the spacebar.

Introduction

Our main goal in this project was to implement our own version of a bullet hell game. Bullet hell games belong to the genre of video games called shoot 'em up games (also referred to as shmups or STGs) and primarily involve a single player whose main goal is to shoot at and destroy continuously spawned enemies, all the while dodging the large quantity of enemy bullets. Through this project, we hoped to utilize the techniques learned throughout the course to design and render such a game and implement interesting and unique patterns of enemy bullets to make the game more entertaining for those interested in playing a single player shooting game that is easy to play on their browsers.

The insurgence of bullet hell games can be dated back to the 1990s, specifically with the creation of Batsugun in Japan in 1993, which incited the term "danmaku," which directly translates into "bullet curtain." In the years following, numerous other bullet hell games were developed in Japan such as DonPachi and Ikaruga, some with more complex scoring systems or types of bullets (Davison 2013). This extended to the Western market, and these games continue to remain popular today, with versions available for gaming systems such as XBox as well as mobile versions.

For our project, we wanted to take inspiration from the graphics and gameplay in these games. Previous approaches succeeded when the quantities and patterns of bullets fired by enemies were not so large and dense that it became nearly impossible for the player to dodge them, but also were complex enough for the player to feel a sense of difficulty and remain engaged with the game, especially with its 2-dimensional nature.

Methodology

In order to implement our project, we would have to transform the scene into two dimensions, while also placing the player, the player's bullets, the enemies, and the enemies' bullets onto the scene, and then adding in UI elements such as the start menu and the scoreboard. Starting with the scene, Three.Js scenes are normally three dimensional, so in order to use Three.Js for our two dimensional game, we would fix the camera at a set Z value (in our case we used 10), and then the game would be in the z = 0 plane. Every other object would solve the 2d versus 3d issue by extending the Sprite class, but that's not possible for the scene. Many other methods would eventually be included for the scene object, but since these methods pertain to some of the other objects listed above, they will be discussed when their relevant objects are discussed.

Next, the player object was needed, and since the player is always in the scene until colliding with a bullet and game-overing, the player starts in the scene rather than being added to it with a method. In a bullet hell game, the bare minimum a player needs is directional movement and a key to shoot. Traditionally, the arrow keys are used to control the player while a key on the left side of the keyboard (such as Z) is used to fire, since player bullets are normally shot straight up, but since we wanted the player to be able to fire in all directions, we chose to use WASD for movement and the mouse for aiming and firing. For movement, there was an event handler for key down that set a movement direction to true as well as an event handler for key up that set a movement direction to false. In the update method, if that movement direction was true, the player's position was incremented. Since we wanted the game to be infinite, the camera needed to be centered on the player so the player couldn't go off screen, so it is in this place that the camera was moved as well. To add some complexity to the game, a focused movement setting was added, triggered by the shift key. When shift is held, the player's movement becomes slowed, allowing for more precise micro-dodging.

For firing, we decided to use the mouse's position for direction (updating mouse position using the mouseMove event) as well as left click for firing. This leads to the first issue with implementing a two-dimensional game with three-dimensional Three.Js: mouse coordinates needed to be converted from 2d to 3d (with z = 0). This was solved by taking advantage of the innate Vector3.unproject() method. Because of bugs resulting from camera centering, we had to take a roundabout way to convert mouse coordinates to scene coordinates. Rather than un-projecting using the camera's coordinates, we would un-project using the camera with x=0, y=0 and then add the x and y position of the player, since the relative position of the mouse with respect to the player was always the same. This implementation was chosen because it was simple and successful, although a more math-intensive approach that used the camera's actual position might save some math in the long run if methods such as screen rotated are implemented. For shooting the bullets, we needed bullets to fire as long as left-click was held, and this was done using timeOuts. As long as the player was holding down left-click, a timeout would be called every 100 milliseconds, firing a bullet. Another possible approach would be to use an internal lifetime counter and fire bullets using the update() method if a certain timer was reached. These implementations are pretty much the same, and in later objects the internal lifetime counter would be used.

Next, the player's bullets needed to be implemented. Since the bullets fire in a straight line, the bullets really only need one parameter to be passed in: the direction. Within the update() method, the bullet travels in its direction until it hits a given maximum range. Things we did not implement for the player bullet would be dynamic change of parameters such as bullet speed, bullet range, and bullet damage. These are commonly modified through power-ups, which were not implemented primarily due to time constraints. Because of this, there was no need to pass these parameters in to modify them.

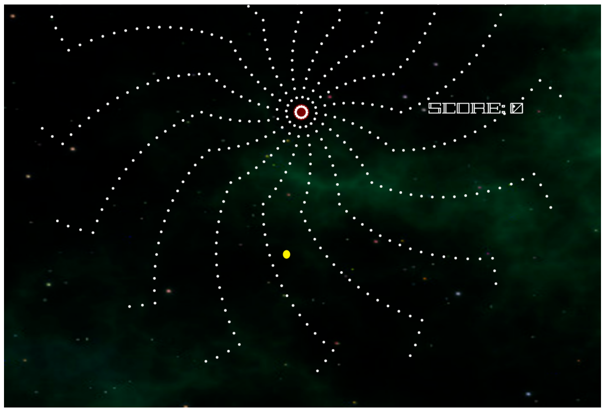
The addition of the player's bullets requires some methods to be added to the scene. Firstly, since bullets are not meant to be infinite, a remove() method is needed to remove an outdated bullet from the update list. This one is simple: remove the object from the scene as well as the array of objects to update. The more difficult addition is for player bullet – enemy collisions. We decided to have an array containing every enemy as well as an array containing every player bullet, and then iterate through every pair in order to check if there was an intersection for their bounding boxes (since both player bullets and enemies are circular, bounding boxes are the largest square fully enclosed by the object. This drastically simplifies computation because of built in Three.Js methods while also having a minimal effect on gameplay). Choosing to detect collisions within the scene rather than within the player bullets or the enemies saves memory because it prevents each player bullet or enemy from storing references to each enemy or player bullet, respectively. (This is in contrast to player - enemy bullet collisions, which will be discussed later).

Next, the enemies needed to be implemented. The most complex part of the project is the patterns each enemy shoots, and that was using another object: the enemyPattern object. This was done for clarity reasons. It is entirely possible for the enemy object to have methods for every pattern, but it's easier on the programmer for all the patterns to be in one place and all the other enemy related methods to be in another. Enemy patterns will be discussed further in a later paragraph. Other than incrementing the enemy's shot pattern, the only other method is one that removes the enemy upon taking lethal damage. Things that were not implemented include enemy movement, not included because it adds a level of randomness to the patterns while also not adding much in exchange.

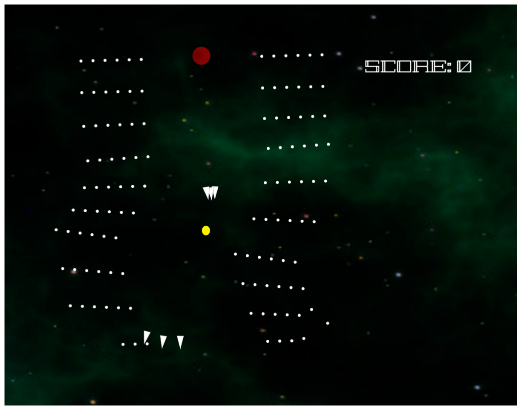
The addition of the enemies also requires additional methods to be added to the scene. Firstly, enemies are meant to spawn dynamically, so there needed to be a method controlling that in the scene. The way this was done was through a few internal timers. One internal timer tracks how long it has been since an enemy has spawned, and if it has been too long, the game spawns another enemy, even if there is one on the field. Because the game is meant to be infinite, the player will sometimes flee from patterns that are too difficult. As a result, the game needs to be able to create enemies even when there already are enemies present. This internal timer goes down as enemies are killed, so as the player progresses the game potentially gets harder. The other internal timer tracks how long it has been since an enemy has died. This is done to prevent too much down time from where there is no enemy, as well as preventing enemies from spawning immediately, catching the player off guard. Enemies are spawned by placing them into the scene within a circular ring of the player in order to utilize the whole 360 degrees of movement and firing. Since the player is able to flee from difficult enemies, the game also needs to be able to despawn enemies so a large amount of bullets offscreen doesn't lag a player's computer. This was done by removing an enemy if it's too far off the screen (utilizing the mouseToSceneCoordinates() conversion to identify the coordinates of the screen). Another approach to this problem would be to give an enemy an internal timer that starts counting down when the player gets too far away from it. However, for practical purposes this approach will rarely be different with a suitable value for how far away the enemy can be before despawning. A player that is too far away from an enemy will take long enough to reach it that it will despawn, so it's much easier to implement a distance check rather than a lifetime counter that also needs a distance check. Player bullet - enemy collisions were discussed above, but the removal of an enemy also triggers an update to the scoreboard, which will be discussed when UI elements are discussed.

The final sprite object that needed to be implemented is the enemy bullet. The enemy bullet seems similar to the player bullet object, but has a bit more complexity to it. Because shot patterns can drastically vary in speed, direction, angular speed, bullet type, starting position, and bullet size, all these properties needed to be passed in. In the update() method, the bullet follows the path given by its speed, direction, and angular speed. Rather than removing bullets after they travel a set range, enemy bullets are despawned based on active time (since curved bullets may never be a certain distance away from their starting point, forcing a distance traveled parameter to be tracked as well). Player - enemy bullet collisions are also performed here. Since there is only one player, every update cycle each enemy bullet checks if its bounding box intersects with the player's bounding box, and if so, ends the game. In addition, instead of having enemy bullets also only be circles, there are additional bullet types, such as arrows and triangles. These bullet types, since they aren't symmetric, require their sprites to be rotated in the direction they are facing as well as an approximation for a rectangular hitbox. Rather than implement detailed polygonal hitboxes, the entities in this game are reasonably approximated by rectangles due to their small size. Even though player - enemy bullet collisions are handled within each enemy bullet, ending the game is handled by the scene. The scene does this by removing every object from the update list, as well as changing a flag to indicate the game is done. This flag prevents actions from occurring in the various objects, but still allows additional input to reset the game, which is better than just preventing all updates outright. One notable element that was not implemented was bullet acceleration (either non-constant acceleration or constant acceleration). This was mostly left out because of time constraints, but the increased flexibility of accelerating / decelerating bullets would've given much room for creative expression.

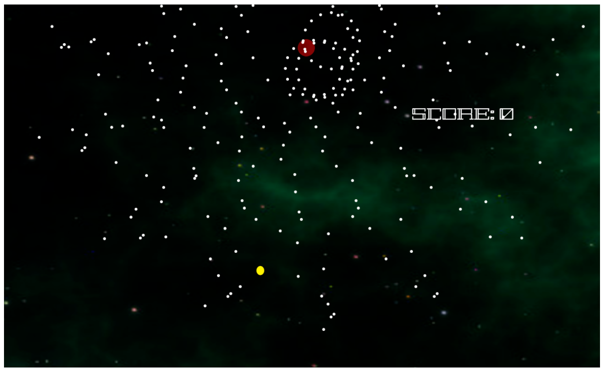
As mentioned above, a separate object, the enemyPattern object, was created in order to handle the creation of shot patterns. Because shot patterns could have various triggers, and triggers could be different between patterns, it was better to implement a new object rather than bloat the enemy object with flags. Each enemyPattern object follows the same basic principle: every time the timer hits a certain value (done through modulus), a bullet is fired with a given set of properties. Patterns may have multiple timers to trigger overlapping bullet patterns, as well as randomness in order to allow for some on the spot dodging rather than pattern memorization. The approach we used for this object was to hard code in each pattern, which allows greater flexibility of what patterns can be expressed. However, a more standardized type of input (such as json) might allow for easier creation of patterns at the cost of flexibility. Because of that, our method seems better suited for this project, because we aren't mass producing patterns. Examples of patterns include:



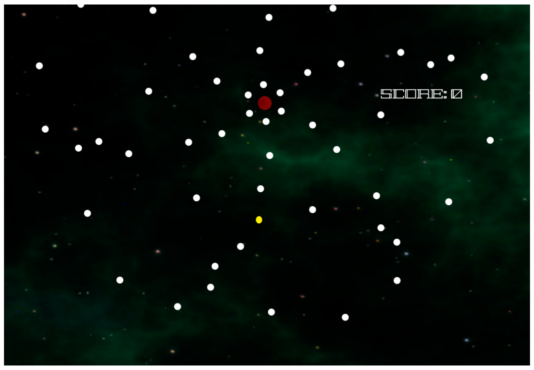
Spiral shots that flip direction periodically, forcing players to follow the openings



Aimed triangular shots with circular shots around the player to constrain movement



Bullets fired radially from different starting points



Bullets fired radially that follow angular paths that swap direction every other shot

Finally, there are UI elements for the scoreboard, the starting menu, as well as the game over screen. Upon game start, the scene will begin as the main menu, and then if space is pressed, the scene switches to the actual game scene. The space button reset will only work when the game is not active, which is either before or after the game. The game over screen, rather than a separate scene like the main menu, is a text overlay over the game scene that just says "Game Over". The scoreboard is another text object located in the top right of the screen. However, unlike the game over text, the position of the score must be able to change due to the player's position and the camera's position being dynamic, and the value of the score must also change. The scoreboard's position was updated in the same way the camera is updated: within the player's update() method, since the player's movement is what triggers repositioning of the scoreboard. In order to update the score, since it is hard to change the material (which changes the text of the Three.Js object under the hood), a new mesh is created when scores need to update, and the reference is changed to this new mesh. Since the score is not updated very frequently, this is an effective solution.

Results

The primary way in which we tested our project was by playing it and ensuring that the various functions, such as the player's motion and shooting, the spawning of enemies and their bullet patterns, collision detection, and score updating, were performed correctly, especially with an increasing number of enemies and more complex bullet patterns. Because of the modular nature of the code, experiments were first done to ensure player movement and shooting worked well in a vacuum, then experiments were done to ensure enemies shot well in a vacuum, and then the interacting methods such as collision detection were tested. After running through various scenarios and fixing bugs, we were able to determine that our game would be able to run smoothly without any issues.

Discussion and Conclusion

Overall, our approach was promising as we were able to meet our goals and implement a fully functional bullet hell game. To follow up our project, we would want to look into implementing more UI elements into the game, such as background music, sounds for player-enemy collision, and buttons to control properties such as speed or difficulty. It would also be interesting to create more animated features, such as the background or the players and enemies themselves. We would also look to implement power ups to increase the player's engagement with the game and incentive to continue playing. Additional features regarding score would be having a more complex algorithm that rewards more actions than merely killing enemies. Examples include rewarding enemy kills while other enemies are present on the field (overlapping patterns are more difficult, so it should be rewarded accordingly), rewarding more difficult enemies with more points, and rewarding grazing (when a player grazes a bullet without colliding with it). Expanding off of this, a local or global leaderboard can be implemented to give more incentive to play and improve. Through completing this project, we have gotten a better grasp on how to use the various objects and functions in Three.Js as well as how to implement more interactive features with the user.

Works Cited

Davison, Pete. “Curtains for You: The History of Bullet Hell.” *US Gamer*, Gamer Network Limited, 9 June 2013, https://www.usgamer.net/articles/curtains-for-you-the-history-of-bullet-hell.

Three.js – JavaScript 3D Library, https://threejs.org/.