#Space O(ddity)
#Herman Lin, Ricky Chen, Victor Teoh
#APCS2 - pd3
#2017-05-16

Proposal Idea:
A rendition of the classic game of Asteroids combined with extra features such as 3D graphics. The game will play out as if the world were two-dimensional, the graphics themselves are simply 3D.

Topics We Can Engage:
- Binary Search Trees
- Queues
- Arrays/ArrayLists
- Nested Classes
- Linked Lists

How We Can Implement Said Topics:
Nested Classes:

We can use nested classes for our main ship (player) and enemy classes. It is simply more efficient to return a nested class for a shot (based on the main class's current location vector and rotation angle) upon a player shooting and calling a shoot() method than to create a completely different class which will need to access another class's private variables. The colliders too are more efficient to create as a nested class because a call to the collider's destruction method can easily flag the main class's destruction state. The main classes will return colliders and shots to be placed within data structures such as a Binary Search Tree and a Linked List respectively. For shots, they can be spawned by looking at the current angle, comparing it to the main class's location vector, and adding to it with the model's length adjusted for the angle, a process more easily done with access to the main class's private variables. The same logic may be applied to colliders.

Binary Search Trees and Linked Lists:

We can use Binary Search Trees to store colliders so that they may be searched for in collisions with shots. The shots will be stored in a linked list for ease of removal, with the root being the player object (as it too will need to check itself against the colliders) and each element will check against the tree. Then each node of the tree will wrap a collider and have extra variables such as parent and left/right node for ease of use. The tree will be sorted by the x variable of the location vector and upon finding the node closest to the shot's x position, the shot

will compare its location vector to the collider's location vector to see if they both should be removed. The tree will be updated every frame, which will have a runtime of nlogn or n^2 as we will find every node in the tree (nlogn) then either not swap them (1), swap them with every parent and child not in order with it (n / 2), and everything in between. the runtime will be nlogn for the most part, and thus this will be a better runtime than simply storing it in an array (n^2). By using Binary Search Trees, we can efficiently look for things close to the shots being made.

Arrays:

The main classes will have to be rendered, so putting them into an array is quite useful. Whether we choose to use an Array or an ArrayList will depend on how many things we will permit to be in the play area at the same time. For objects that will need 3D models, we will cap the amount of objects with an array. For objects that are simply an image on a 2D plane (VFX class), we will store them in an ArrayList. When rendering the array, if the an index has been destroyed, then the index will be swapped with the last index and the index to insert new objects for will be reduced by one. Thus, all the dead will be moved out before rendering. The ArrayList does not need to take this extra step and should instead simply use the remove() method once an element's time is up.

Queues (Priority Queues):

We can uses queues for spawning new entities. Every time an object is removed from the main array, whether it be from player destruction or outside camera boundary cleanup, an enemy will be dequeued and spawned into the play area while another enemy made with the specifications of the current difficulty level (perhaps it will scale with velocity and size) will be enqueued.

Technologies We Will Need

- 3D Shapes
- Cameras
- Vectors