**Stellar Tower Defense**

Documentation

Voinea Ioan-Andrei

**Part 1.** *Game Tutorial*

This is a tower defense! Play the game by placing and upgrading towers on the map. Enemies will spawn at the edge of the map and advance towards the player base. Defeat all the enemies to receive credits (in-game currency) and progress further into the game. Use the credits to upgrade and place more towers. The game ends when the enemies manage to defeat you.

**Part 2.** *Concept*

The idea for this game came from trying to add something new to the idea of a tower defense. The map is not linear, it’s made from rotating circles with certain entry points. I used polar coordinates to create, move and align everything in a circular pattern.

Seeing the rotating circular pattern around the center, made me instantly think about galaxies and space, so I chose a theme based around that. The map is made out of asteroids, while the enemies are spaceships. The towers are hidden between the asteroid fields, and the base is a space station.

**Part 3.** *Map*

The map is made out of 9 circular Asteroid Fields centered around the Base with radiuses increasing from 3 to 11. Each field rotates around the center depending on how far it is from it ( further circles rotate faster) . The map is creating by creating “gaps” in the field, places where asteroids don’t spawn, that serve as entry points for the enemies. The further you progress, there are more gaps present in each field, meaning enemies have faster routes to your Base.

**Part 4.** *Entities*

There are 3 times of entities in Stellar Tower Defense: EnemyEntity, FriendlyEntity and PlayerEntity.

**Part 4.1** *EnemyEntity*

This is (as the name implies ) the instance for an Enemy type of Object. It is spawned on the adege of the map and follows an open path to the main base. It can only attack the main base after it finishes it’s journey. It inherits the stats for fire rate and damage, as well as a few key features from the main Entity class like the ability to shoot targets.

**Part 4.2** *FriendlyEntity*

This represents a tower instance. A tower is created with one of the buttons on the left side of the screen. It can only be placed on an asteroid belt, and when it’s placed it aligns and attaches with the belt. In addition to fire rate and damage the towers have range that can be increased to attack enemies further away.

**Part 4.3** *PlayerEntity*

This is the player Singleton. It contains almost no similarities with the other two components. The only things they share are fire Rate, damage, range and the ability to shoot, as well as HitPoints.

**Part 5.** *Movement & Math*

As I have mentioned before this game uses polar coordinates to dictate where exactly we are on the game world. By knowing that the y will stay constant, this being an isometric game ( transform.position.y = 0f ), we can determine a point's coordinates from a radius R and an angle theta like this :

transform.position.x = R \* cos (theta);

transform.position.z = R \* sin (theta);

This also means that knowing only 2 of the 3 variables, we can solve for the third. That leaves me with the ability to move an object along a circle with radius R by simply increasing or decreasing theta, because if R is a constant that means the transform.position is in direct correlation with theta. That also means by decreasing or increasing R while keeping theta constant, one can move towards or away from the center. This can be seen when enemies pass through a gap.

The whole calculations have their base in the simple class Point, that represents a point ln the 2D plane. Selecting a random point on a radius is very simple, simply select a random value between 0 and 2π and use that In the equations above.

The Asteroid Fields are created by first determining the maximum number of asteroids from the total circumference of a circle divided by the minimum difference between asteroids. I’ve set the minimum difference between asteroids to be one of their faces’ diagonal and given thag we know the radius of each belt it can be calculated from the start.

The second step is determining the gaps. I’ve chosen the gap length to be an arbitrary size of ten asteroids + 1 (the gap center). Then I select a random point from the possible gap locations and mark the asteroids around it as no longer selectable. To remove the possibility of gaps overlapping or gaps being too close to each other. From the center I mark both to the left and to the : gapLength/2 (half of the current gap) + gapLenght (the distance between gaps) + gapLeght/2 (half of the next possible gap) = 2\* gapLenght. So In total I mark : 4\* gapLength +1 points, 2\*gapLength to the left and to the right and 1 being the center point, as impossible to be selected as a next gap center. And then I repeat the proccess until I find all the gaps. The method also has a failsafe, by trying to find a possible gap center more than 10000 times, it quits. That prevents the method from trying to build, for example, 8 gaps on the AsteroidField with radius 3, if there are only 4 possible gaps to be created (given the gapLenght, the length between gaps and the total number of asteroids).

The method for determining the next closest gap is based around finding the minimum distance from the enemy to any possible gap.

Each entity moves with a velocity that is calculated from the start, being based around speed which is unchangeable during gameplay. The velocity is calculated the first time when the entity’s theta has been changed (when it has moved at least along it’s radius). The formula for the velocity is a rather complex one yet with a simple calculation. Given I wanted to find the rate of change between thetas, I turned to the formula of distance between 2 points:

A ( X1 , Y1) ; B ( X2, Y2 );

AB = √ ((X1 – X2)2 + (Y1 – Y2)2)

Or

AB2 = (X1 – X2)2 + (Y1 – Y2)2

And as it turns out in fact we know all coordinates of these points in terms of R and theta from the previous formulas. Just substituting the first set of formulas into this formula we obtain

AB2 = R2( (cos(θ1) – cos(θ2))2 + (sin(θ1) – sin(θ2))2 )

And expanding the brackets whilst substituting for cos(θ)2 + sin(θ)2 = 1 gives us:

AB2 = R2( 2 - (2cos(θ1) cos(θ2) + 2sin(θ1) sin(θ2)) )

Applying the next trigonometric formulas:

2cos(θ1) cos(θ2) = cos(θ1 - θ2) + cos(θ1 + θ2)

2sin(θ1) sin(θ2) = cos(θ1 – θ2) - cos(θ1 + θ2)

And reducing the common term, we obtain:

AB2 = R2( 2 - 2 cos(θ1 - θ2))

Knowing that AB means the distance between two points on a theta change, that means that the velocity or rate of change is equal to AB and we obtain the final formula for the velocity:

Velocity = R \* √2 \* √(1- cos(θ1 - θ2))

The last bit of more complex mathematic that I used for the project is involved in the placing of the tower on the belts.

The way I wanted to do it was using Camera.ScreenToWorldPoint, but given that the game is an isometric game and therefore a 3D one, the positions I was getting while using this formula were almost never on y = 0f.

Therefore I had to create a method that gets the line passing through the Camera position and the Camera.ScreenToWorldPoint and determines it’s intersection with the y = 0f plane

I achieved this using the equation for a vector form of a line in 3D space:

A ( X0 , Y0, Z0 ) ; B ( X1, Y1, Z1 );

X = X0 + t \* (X1 - X0)

Y = Y0 + t \* (Y1 - Y0)

Z = Z0 + t \* (Z1 - Z0)

(assuming the line goes through A and in the direction of B)

From this equation I chose t = 1 when the line goes through B (t being an arbitrary step function it can be set as anything) and then I calculated the intersection when Y = 0f, and the final formula is:

C(X2, 0f, Z2);

t = (0f – Y0)/ (Y1- Y0)

X2 = X0 + t \* ( X1 – X0 )

Z2 = Z0 + t \* ( Z1 – Z0 )

(where C is the point that we are looking for)

**Part 6.** *Modularity*

I planed this game with the idea of modularity in mind. That means the main components of the game such as: level layout, enemy structure, enemy or tower types, upgrades, etc… can easily be created and upgraded. I chose a JSON file structure for the modularity of the game and I achieved it using 4 JSON files:

* *EnemyInformation.json*

Holds the information for every type of enemy. Here you can create a new type of enemy just by adjusting the numbers and name prefabs of this file.

* *LevelInformation.json*

Holds the information for the level structure, how many gaps per asteroid field, how many enemies and in what order will they appear. Tweaking with the numbers can balance one level or create another with just a few edits.

* *UpgradeInformation.json*

Holds the information for the available upgrades for the base and for the towers. Simple modifications here can change the amount o e upgrade differs from the other.

* *PlayerInformation.json*

Finally this holds Information about a default player, one that just pressed new game. From this script you can edit the new player experience , and it serves as a base on top of which a player could save it’s progress in the future.

**Part 7.** *Plans for the future*

The game is just a prototype and nowhere near a publishing state, and in development, the game’s idea evolves. This being said I reserved the last chapter to talk about future prospects of the supposed game. One idea would be adding a set of “ super powers” for the player to use, to encourage more active gameplay. These powers could deal damage, stun, slow or even scatter enemies along the belts. A few other things that could be improved are:

* Adding the ability to save and continue a game.
* Adding a tutorial for new players
* Improve the balance of the game
* Improve on the pause button functionality
* Add the ability to change the number of asteroid fields per level from the modularity part
* Add shooting animations and different projectiles

*Thank you for your time and I hope you enjoy the game!*