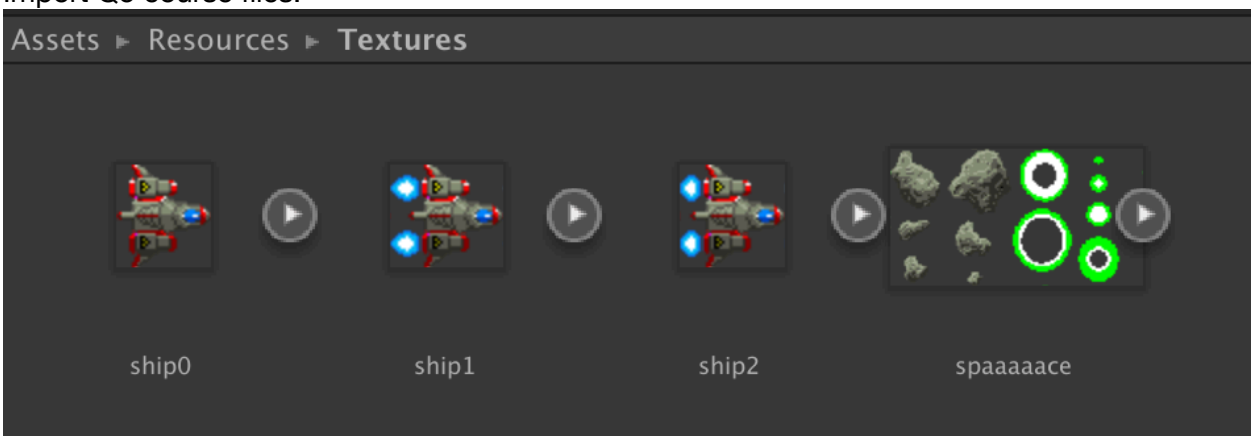
Quest 5 - Steps

1) Import and Slice Graphics

Import Q5 course files.



For all 4 graphics, set:
- Filter = Point
- Pixels Per Unit = 36

For spaaaaace, set Sprite Mode = Multiple and use the Sprite Editor to Slice with Type = Automatic. Within the Sprite Editor, set the Pivot point for the Missile sprites to the Right.

For ship[0-2], set Pivot = Right.

Click Apply as prompted.

2) Create the player's Ship

Create a Ship scene object with the following components and settings:

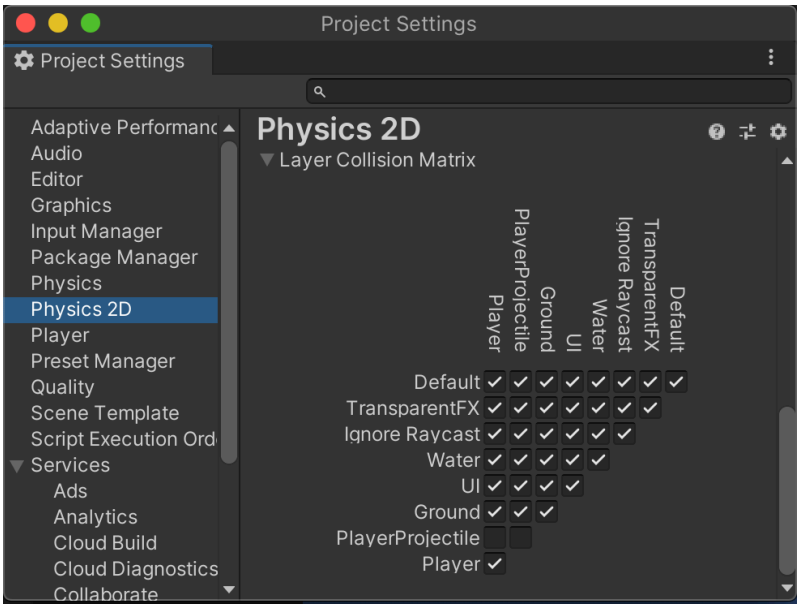
- Layer = Player
- Transform
  - Position = 0, 0, 0
- SpriteRenderer
  - Sprite = ship0
- Rigidbody2D
  - BodyType = Kinematic
  - Use Full Kinematic Contacts = True
- PolygonCollider2D
- Animator (not Animation)

3) Create a Projectile prefab

Create a Projectile scene object with the following components and settings. Prefab it.

Layer = PlayerProjectile
SpriteRenderer
Sprite = one of your two rocket sprites
Rigidbody2D
Gravity = 0
BoxCollider2D
Animator (not animation)

From Edit->Project Settings->Physics2D, make sure the Player and PlayerProjectile intersection is unchecked, so that Players cannot collider with their own PlayerProjectiles. We also do not want PlayerProjectiles to hit themselves.



4) Create SIX Asteroid prefabs

Create 6 Asteroid scene objects with the following components and settings. Prefab each of them separately.

SpriteRenderer
Sprite = one for each of your asteroid sprites
Rigidbody2D
Gravity = 0
PolygonCollider2D

5) Create an Explosion prefab

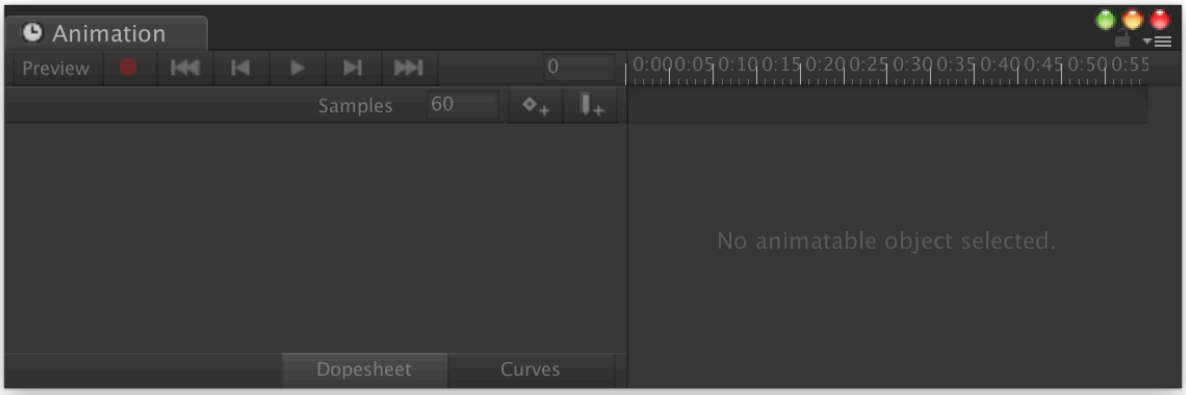
Create an Explosion scene object with the following components and settings. Prefab it.

SpriteRenderer
Sprite = the smallest green circle
Animator (not animation)

6) Create the Ship Animation

Create a folder Assets/Resources/Animations/Ship/

Open Window->Animation->Animation



Select the Ship scene object

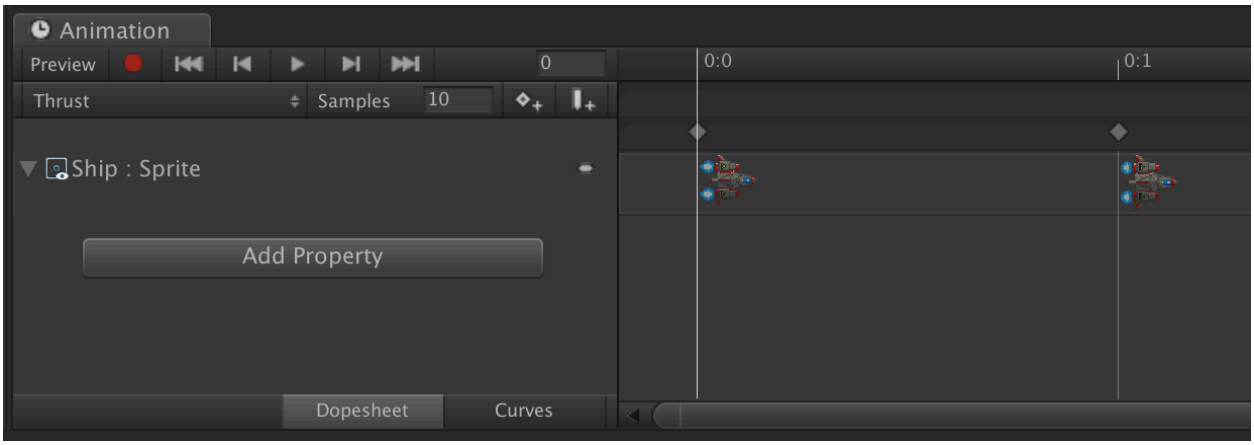
Click Create in the Animation window and save the file Animations/Ship/Thrust.anim

Change Sample Rate to 10 frames per second

Click Add Property->Sprite Renderer->Sprite

Select and Delete the Existing Keyframes

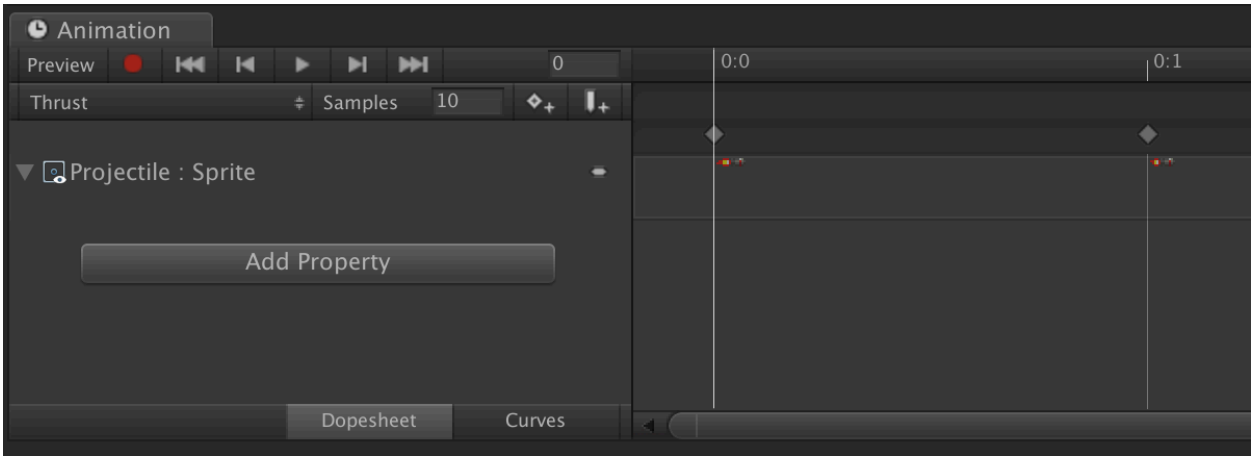
Drag the two Ship Thrust textures from your Project onto different parts of the Animation timeline



7) Create the Projectile Animation

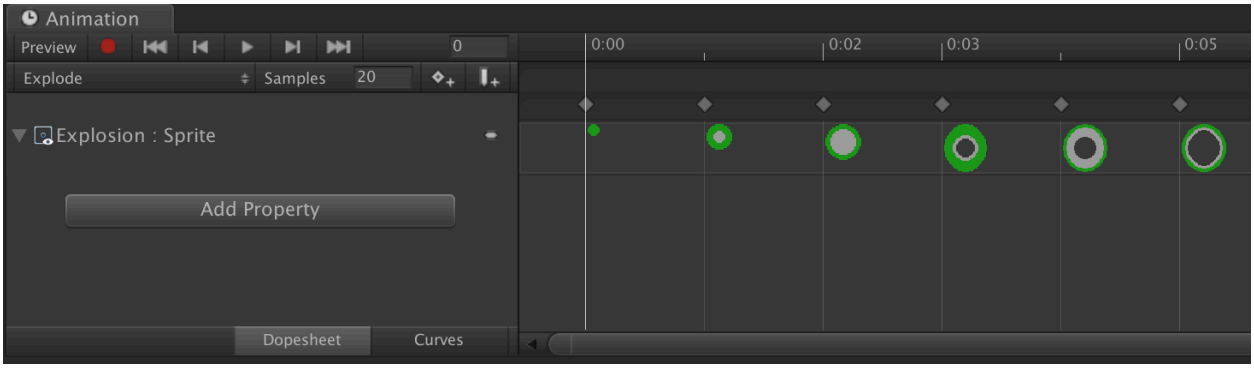
Create a folder Assets/Resources/Animations/Projectile/
Open the Animation window
Select the Projectile scene object

Click Create in the Animation window and save the file Animations/Projectile/Thrust.anim
Change Sample Rate to 10 frames per second
Click Add Property->Sprite Renderer->Sprite
Delete the Existing Keyframes
Drag the two Projectile textures onto different parts of the timeline



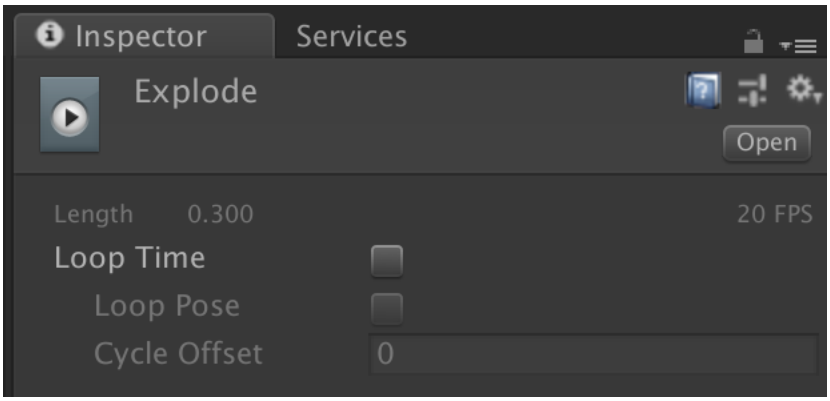
8) Create the Explosion Animation

Create a folder Assets/Resources/Animations/Explosion/
Open the Animation window
Select the Explosion scene object
Click Create in the Animation window and save the file Animations/Explosion/Explode.anim
Change Sample Rate to 20 frames per second
Click Add Property->Sprite Renderer->Sprite
Delete the Existing Keyframes
Drag the various Explosion textures in sequence onto different parts of the timeline



Select your Explode animation in your Project library

Uncheck "Loop Time”



9) Prefab cleanup

Apply all updates to your prefabs and Remove all prefabs from the scene

- 6 Asteroids
- Projectile
- Explosion

10) GameController setup

Create a GameController scene object

Create a GameController c# script and attach it to the GameController scene object

We will use a technique known as Static Instance Referencing to write more efficient code. This technique is similar to the Singletons programming methodology.

```
public class GameController : MonoBehaviour {

    public static GameController instance;

    // Methods
    void Awake() {
        instance = this;
    }
```

This allows us to reference GameController purely through code without having to drag-and-drop a fill-in-the-blank on every other object. Static Instance Referencing ONLY works if there is exactly one of an object. There will always be only one GameController.

11) Ship movement

We will keep track of time within GameController using a public property timeElapsed.

Time.deltaTime accurately represents the passage of time between game frames even if your game lags.

```
public class GameController : MonoBehaviour {

    public static GameController instance;

    // State Tracking
    public float timeElapsed;

    // Methods
    void Awake() {
        instance = this;
    }

    void Update() {
        // Increment passage of time for each frame of the game
        timeElapsed += Time.deltaTime;
    }
```

Create a Ship c# script and attach to the Ship.

Using our Static Reference to GameController, we are able to read the timeElapsed from the Ship without creating a Public Outlet.

We will feed the passage of time into a Sine Wave to get an automatic back-and-forth motion for our ship.

```
public class Ship : MonoBehaviour {

    void Update() {
        transform.position = new Vector2(0, Mathf.Sin(GameController.instance.timeElapsed) * 3f);
    }
```

12) Asteroid movement

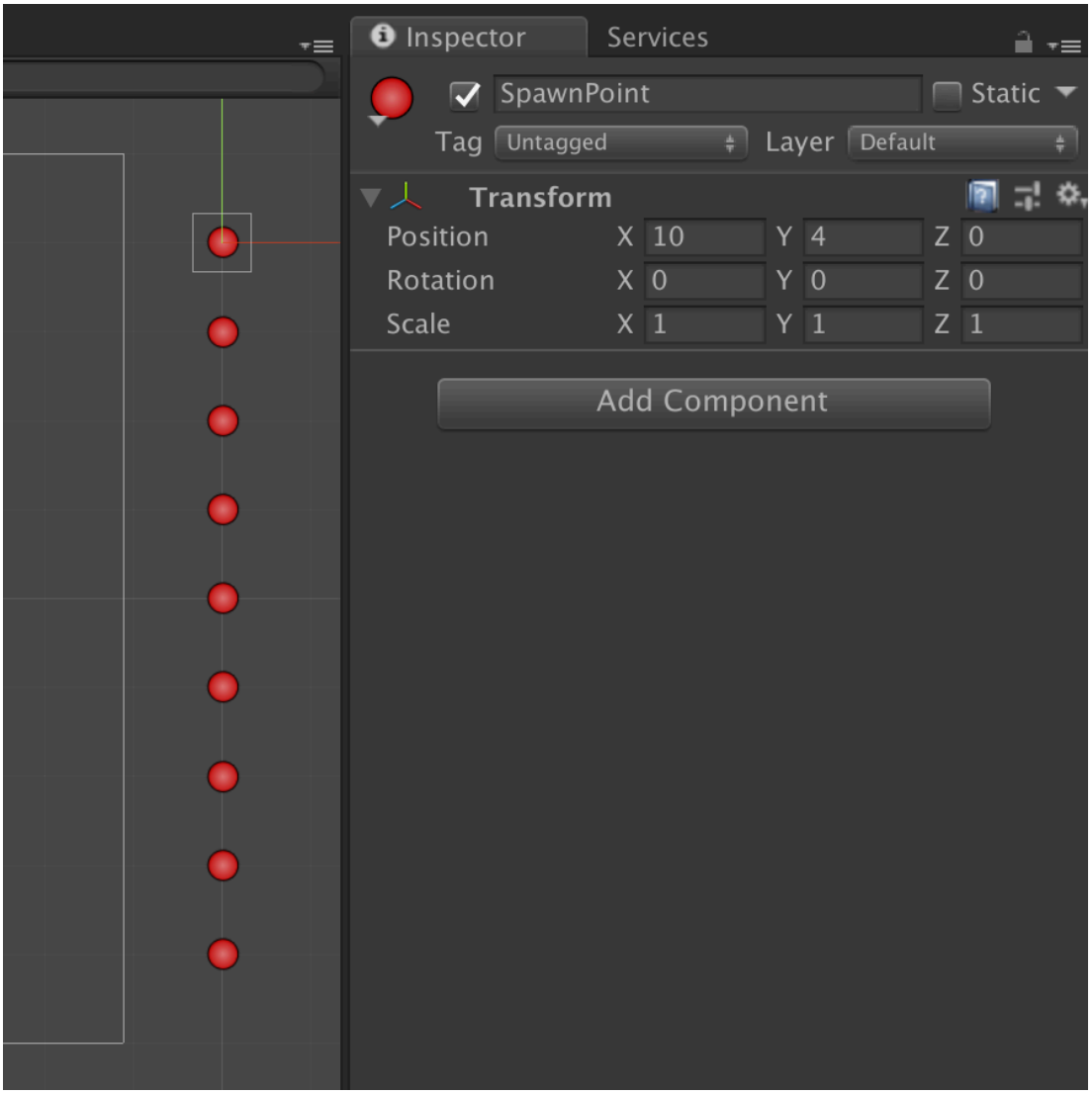
Create an Asteroid C# script and attach to your Asteroid prefab.

```
public class Asteroid : MonoBehaviour
{
    // Outlet
    Rigidbody2D rigidbody;

    // State Tracking
    float randomSpeed;

    // Start is called before the first frame update
    void Start() {
        rigidbody = GetComponent<Rigidbody2D>();
        randomSpeed = Random.Range(0.5f, 3f);
    }

    void Update() {
        rigidbody.velocity = Vector2.left * randomSpeed;
    }

    void OnBecameInvisible() {
        Destroy(gameObject);
    }
}
```

13) Spawn Points setup

Create 9 SpawnPoint gameObjects.

Position them just off screen spread vertically with position spacings {10, 4, 0}, {10, 3, 0}, {10, 2, 0}, {10, 4, 0} ... {10, -4, 0}.

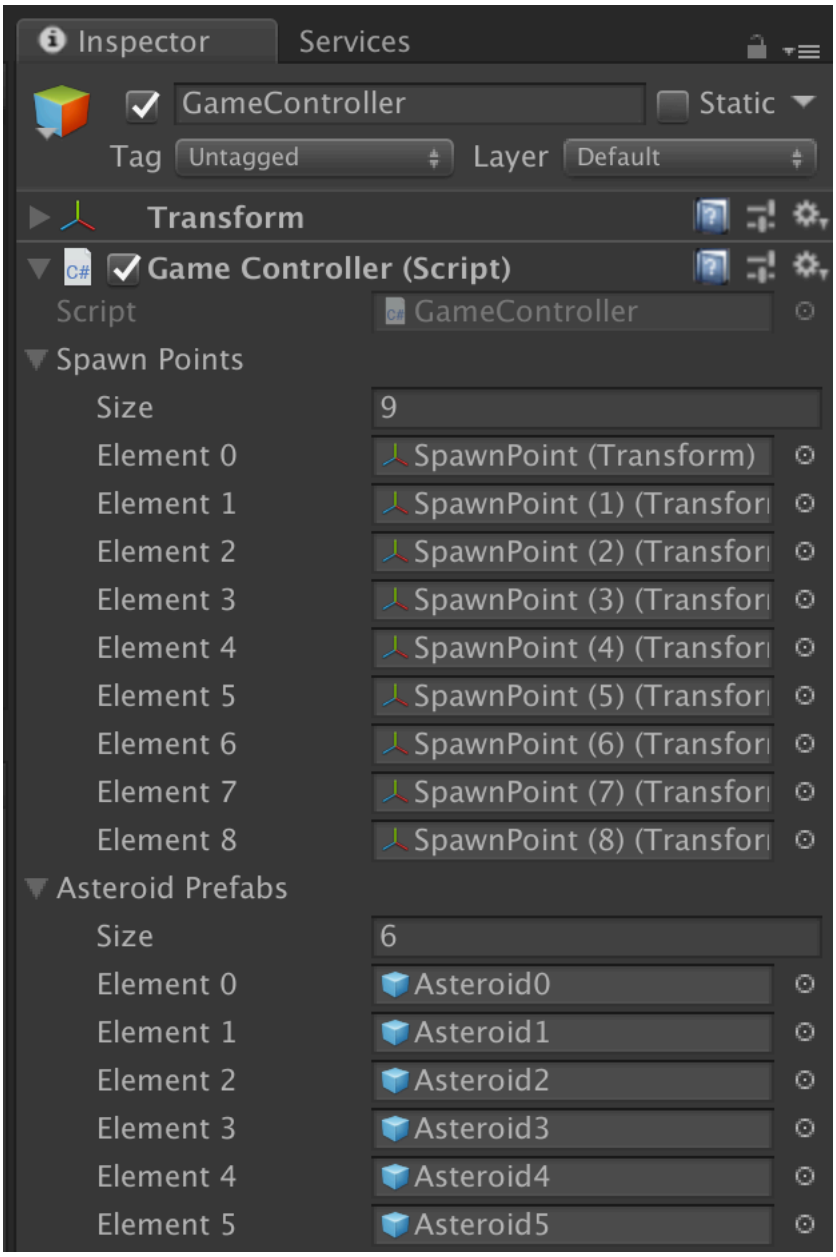


14) Randomized Asteroid Spawning and Timer

Create Public Outlets for what Asteroid prefabs we will use and where they can potential spawn

```
public class GameController : MonoBehaviour {

    public static GameController instance;

    // Outlets
    public Transform[] spawnPoints;
    public GameObject[] asteroidPrefabs;
```

Fill in these blanks in the inspector. spawnPoints are in your Scene. asteroidPrefabs must be Prefabs from your Library.



Create a function for spawning a random Asteroid Prefab at a Random Spawn Point.

```
void SpawnAsteroid() {
    // Pick random spawn points and prefabs
    Transform randomSpawnPoint = spawnPoints[Random.Range(0, spawnPoints.Length)];
    GameObject randomAsteroidPrefab = asteroidPrefabs[Random.Range(0, asteroidPrefabs.Length)];

    // Spawn
    Instantiate(randomAsteroidPrefab, randomSpawnPoint.position, Quaternion.identity);
}
```

To put this on a timer, we need a property to keep track of the delay between asteroid spawnings. We will also have a configurable minimum and maximum delay.

```
// Configuration
public float minAsteroidDelay = 0.2f;
public float maxAsteroidDelay = 2f;

// State Tracking
public float timeElapsed;
public float asteroidDelay;
```

We will dynamically decrease this delay as time progresses. There is no best equation for this computation. It needs to be Playtested for Game Feel.

```
void Update() {
    // Increment passage of time for each frame of the game
    timeElapsed += Time.deltaTime;

    // Compute Asteroid delay
    float decreaseDelayOverTime = maxAsteroidDelay - ((maxAsteroidDelay - minAsteroidDelay) / 30f * timeElapsed);
    asteroidDelay = Mathf.Clamp(decreaseDelayOverTime, minAsteroidDelay, maxAsteroidDelay);
}
```

We will create a Coroutine using an IEnumerator to act as a delay timer between asteroid spawns. Coroutines ARE NOT true multithreading or background threads, but are a technique for setting up “Side Events” in addition to Unity’s main loop.

This Coroutine calls itself at the end of itself, creating a repeating timer. IEnumerators MUST have a yield in order to prevent them from becoming an Infinite Loop that freezes your game.

```
IEnumerator AsteroidSpawnTimer() {
    // Wait
    yield return new WaitForSeconds(asteroidDelay);

    // Spawn
    SpawnAsteroid();

    // Repeat
    StartCoroutine("AsteroidSpawnTimer");
}
```

While the Coroutine loop restarts itself, we must start it in the first place. We want to start spawning asteroids at the Start of our game:

```
void Start() {
    StartCoroutine("AsteroidSpawnTimer");
}
```

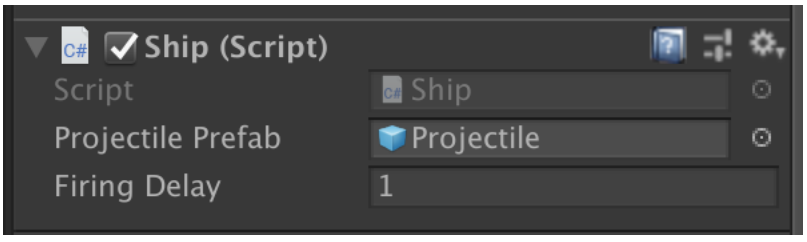
15) Ship Firing Logic

Our ship firing logic is very similar to the asteroid spawning because both are automated.

We start by creating a Public Outlet for the projectile prefab we want to shoot, and creating a firingDelay property so we can dynamically change the frequency of our shooting.

```
public class Ship : MonoBehaviour {

    // Outlet
    public GameObject projectilePrefab;

    // State Tracking
    public float firingDelay = 1f;
```

The projectile prefab must be assigned in the fill-in-the-blank.



We create a FireProjectile function to call whenever we want our ship to shoot.

```
void FireProjectile() {
    Instantiate(projectilePrefab, transform.position, Quaternion.identity);
}
```

A Coroutine built using an IEnumerator yields for the appropriate delay time, fires a projectile, and then starts the timer all over again.

```
IEnumerator FiringTimer() {
    yield return new WaitForSeconds(firingDelay);

    FireProjectile();

    StartCoroutine("FiringTimer");
}
```

The Timer still has to be started in the first place.

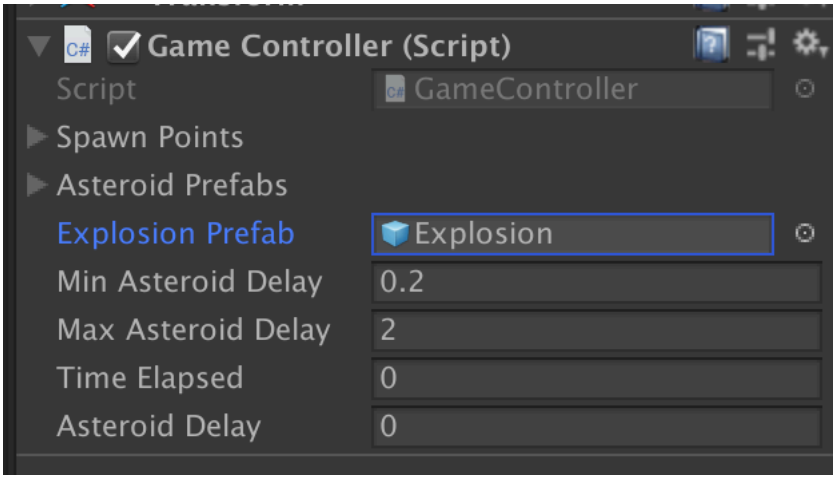
```
void Start() {
    StartCoroutine("FiringTimer");
}
```

16) Explosion prep

We want to use Explosions throughout our game, so we will keep a reference to our Explosion prefab in the GameController where it can be easily references by everything else in the game.

```
public class GameController : MonoBehaviour {

    public static GameController instance;

    // Outlets
    public Transform[] spawnPoints;
    public GameObject[] asteroidPrefabs;
    public GameObject explosionPrefab;
```

Public Outlets must have their blanks filled.



17) Projectile Logic

Create a C# script for the projectile and attach it to the prefab.

We will start off with a standard Rigidbody2D Reference Outlet. We will also create a target property for keeping track of our projectile's target.

```
public class Projectile : MonoBehaviour {

    // Outlets
    Rigidbody2D rigidbody;

    // State Tracking
    Transform target;

    // Methods
    void Start () {
        rigidbody = GetComponent<Rigidbody2D>();
    }
```

A ChooseNearestTarget function will look at every asteroid in the scene and choose a target that meets the following criteria:

- It is to the right of the projectile because we don't want to target asteroids behind the player.
- It is the closest asteroid of all the ones that we have checked.

```
void ChooseNearestTarget() {
    float closestDistance = 9999f; // Pick a really high number as default
    Asteroid[] asteroids = FindObjectsOfType<Asteroid>();
    for(int i = 0; i < asteroids.Length; i++) {
        Asteroid asteroid = asteroids[i];

        // Asteroid must be to our right
        if(asteroid.transform.position.x > transform.position.x) {
            Vector2 directionToTarget = asteroid.transform.position - transform.position;

            // Filter for the closest target we've seen so far
            if(directionToTarget.sqrMagnitude < closestDistance) {
                // Update closest distance for future comparisons
                closestDistance = directionToTarget.sqrMagnitude;

                // Store reference to closest target we've seen so far
                target = asteroid.transform;
            }
        }
    }
}
```

We will set up two variables for acceleration and maxSpeed. We will make these dynamic in a future step.

Every frame of our game, we will point the projectile toward our target.

Our projectile will continually thrust forward (which happens to be towards the right based on how our sprite was drawn) clamped by the maxSpeed set in our variables.

```
void Update() {
    // Some dynamic projectile attributes
    float acceleration = 1f;
    float maxSpeed = 2f;

    // Home in on target
    ChooseNearestTarget();
    if(target != null) {
        // Rotate towards target
        Vector2 directionToTarget = target.position - transform.position;
        float angle = Mathf.Atan2(directionToTarget.y, directionToTarget.x) * Mathf.Rad2Deg;

        rigidbody.MoveRotation(angle);
    }

    // Accelerate forward
    rigidbody.AddForce(transform.right * acceleration);

    // Cap max speed
    rigidbody.velocity = Vector2.ClampMagnitude(rigidbody.velocity, maxSpeed);
}
```

Last, on collision, we check if we have impacted an asteroid. Our projectile will destroy the asteroid, destroy itself, and create an explosion where the asteroid was. That explosion game object will destroy itself soon after.

```
void OnCollisionEnter2D(Collision2D other) {
    // Only explode on Asteroids
    if(other.gameObject.GetComponent<Asteroid>()) {
        Destroy(other.gameObject);
        Destroy(gameObject);

        // Create an explosion and destroy it soon after
        GameObject explosion = Instantiate(GameController.instance.explosionPrefab, transform.position,
            Quaternion.identity);
        Destroy(explosion, 0.25f);
    }
}
```

18) To Be Continued…