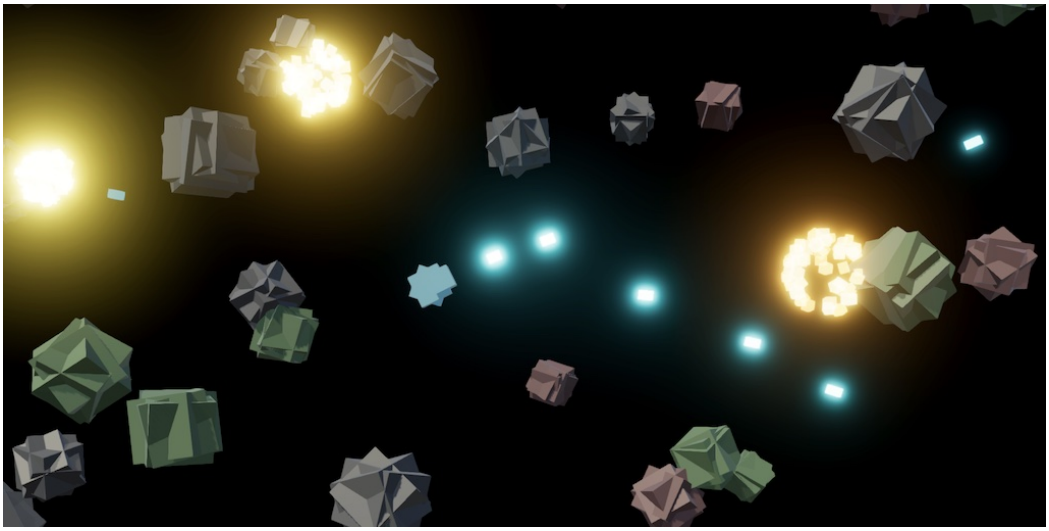
published 2023-07-31

Bouncy Ball Shooter Bouncing in a Wrapping 2D World

Spawn balls and bullets in a wrapping 2D world.
Synchronize native data and prefab instances.
Implement wrapping 2D physics with soft bounces.
Smooth animation with time extrapolation and interpolation.
Make the player follow the cursor and shoot backwards.

This is the seventh tutorial in a series about prototypes. In it we will create a simple 2D ball-shooting action game.

This tutorial is made with Unity 2022.3.4f1.



Navigating through bouncing and exploding balls.

1 The Game

The game prototype that we'll make in this tutorial is a hybrid of the classical *Asteroids* game and a simple modern bullet-hell shooter. It's something like a clone of *Death vs Monstars* but with a wrapping world. As an extra twist we replace the asteroids with soft bouncing balls, which requires custom 2D physics.

As usual we begin with duplicating the Paddle Square project and removing everything that we don't need. In this case we keep the text and particles prefabs and materials. We'll put the 2D game on the XY plane, so clear the camera rotation and set its position to (0, 0, −20). Its clipping planes can be narrowed to something like 16 and 24. I set the main light's rotation to (50, 45, 0).

Remove all the scripts, only keeping an empty `Game` component type attached to a game object to control our game. Note that I do not mark the initial contents of a script, which would be fully marked otherwise.

```
using Unity.Jobs;
using TMPro;
using UnityEngine;

public class Game : MonoBehaviour { }
```

We're going to use prefab instances for both balls and bullets, so we also reuse the `PrefabInstancePool` from Match 3. Even though it can deal with hot reloads we won't support it in this tutorial, as the games sessions are short and we rely on native data.

```
using System.Collections.Generic;
using UnityEngine;

public struct PrefabInstancePool<T> where T : MonoBehaviour
{
	Stack<T> pool;

	public T GetInstance (T prefab)
	{
		if (pool == null)
		{
			pool = new();
		}
#if UNITY_EDITOR
		else if (pool.TryPeek(out T i) && !i)
		{
			// Instances destroyed, assuming due to exiting play mode.
			pool.Clear();
		}
#endif

		if (pool.TryPop(out T instance))
		{
			instance.gameObject.SetActive(true);
		}
		else
		{
			instance = Object.Instantiate(prefab);
		}
		return instance;
	}

	public void Recycle (T instance)
	{
#if UNITY_EDITOR
		if (pool == null)
		{
			// Pool lost, assuming due to hot reload.
			Object.Destroy(instance.gameObject);
			return;
		}
#endif
		pool.Push(instance);
		instance.gameObject.SetActive(false);
	}
}
```

We'll use the particle system for explosions. I adjusted it a bit to create effects that work with the different view distance and made them more varied. Its shape module should be set to sphere mode. Also increase the intensity of the particles material to 20.

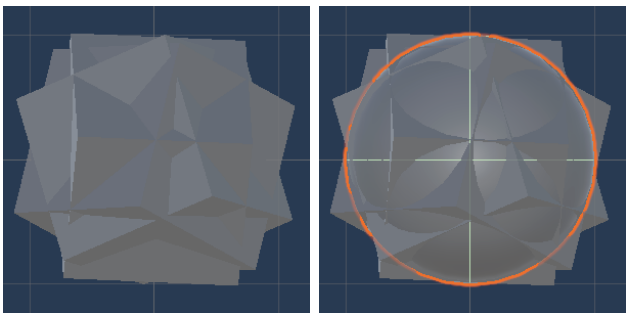
Explosion Particle System.

Also create a new unlit opaque shader graph with an HDR color property. We'll use it for a dark red world edge and a bright yellow health bar later. We don't use the standard unlit material because it doesn't support HDR colors.

Finally, clear the rotation of the text prefab and reduce its font size to 8.

1.1 Balls

Our game uses balls, but I'm still limiting myself to only using cubes, quads, and text. So our bouncy balls will look like roughly-hewn foam blocks. Create an empty game object prefab and give it three cube children—without colliders—that each have an arbitrary rotation. To make it roughly match the size of a sphere with radius ½ reduce the cube scales to $\sqrt{½}$, roughly 0.7071. That makes the cube edges touch the sphere surface while their corners will poke out. Note that this is the scale of the cild cubes, the root object's scale should be 1.

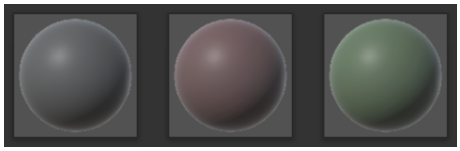


Ball prefab; without and with default ½ radius sphere.

This prefab only represents a visualization of a ball. It won't contain essential game logic nor state. Our physics system will not be aware of it. Attach a `BallVisualization` component to its root game object, with methods to spawn and despawn a pooled instance.

```
using Unity.Mathematics;
using UnityEngine;

using Random = UnityEngine.Random;

public class BallVisualization : MonoBehaviour
{
	PrefabInstancePool<BallVisualization> pool;

	public BallVisualization Spawn()
	{
		BallVisualization instance = pool.GetInstance(this);
		instance.pool = pool;
		return instance;
	}

	public void Despawn() => pool.Recycle(this);
}
```

Duplicate the prefab a few times and give each a different material to provide a little visual variety. I made gray, red, and green variants, all with low saturation to not be distracting.



Ball materials.

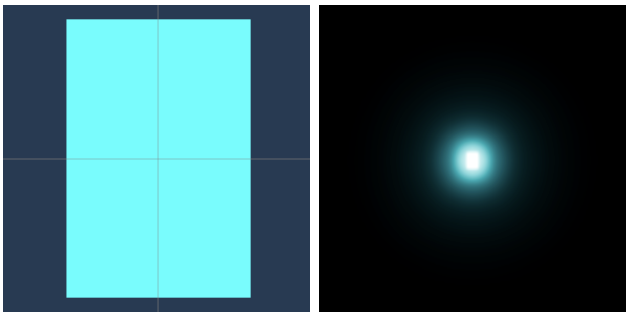
1.2 Bullet

We'll also use prefabs to visualize bullets, in this case using a single cube, without collider. Give it a duplicate of the ball's component with its type changed to `BulletVisualization`.

```
using UnityEngine;

public class BulletVisualization : MonoBehaviour
{
	PrefabInstancePool<BulletVisualization> pool;

	public BulletVisualization Spawn()
	{
		BulletVisualization instance = pool.GetInstance(this);
		instance.pool = pool;
		return instance;
	}

	public void Despawn() => pool.Recycle(this);
}
```

Make the bullet small and a little elongated, using (0.2, 0.3, 0.2) for its scale.

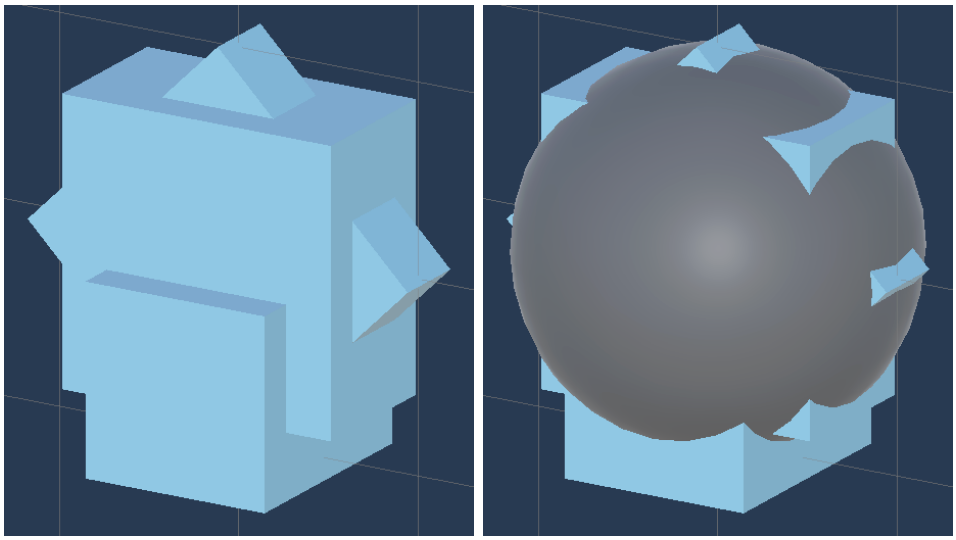
We'll change the bullet's color based on its age and will use instanced rendering to make this simple and efficient. Create an unlit transparent additive shader graph—like for the particles—with properties for a bright HDR cyan color and an 0—1 life factor that gets squared and used for alpha. The life factor should have `_LifeFactor` as its *Reference*. Enable *Override Property Declaration* for it and set its *Shader Declaration* to *Hybrid Per Instance*. Keep it exposed with a default of 1 so the material is visible in the editor.



Bullet prefab and instance in the scene.

1.3 Player

Create a small visualization for the player, roughly fitting a sphere with radius ½. It can be either a prefab or made directly in the scene as we only need a single instance of it. I designed it as a small cyan pointy thing with an extrusion at the bottom, which is where bullets will be fired from. Once again we don't need any colliders. It also doesn't need to be pooled so we don't give it a component yet. If you put it directly in the scene deactivate it for now.



Player prefab; without and with radius ½ sphere.

1.4 Wrapping 2D World

Our game will wrap its 2D world as if it were a torus, like *Asteroids* does. Create an `Area2D` struct type for this with a `float2` extents field to store the world extents. Give it a `RandomVector2` property to retrieve a random position as a `Vector2`, which we'll use to spawn ball instances. Also give it a `Wrap` method that wraps a position, teleporting it to the other side of the area when needed. This method will be used by our physics system, for which we'll use Burst jobs, so make it use the `Mathematics` library. End with a static `FromView` method that produces an area that fills XY plane for a perspective camera, assuming it sits on and looks down the Z axis.

```
using Unity.Mathematics;
using UnityEngine;

using Random = UnityEngine.Random;

using static Unity.Mathematics.math;

public struct Area2D
{
	public float2 extents;

	public Vector2 RandomVector2 =>
		new(Random.Range(-extents.x, extents.x), Random.Range(-extents.y, extents.y));

	public float2 Wrap(float2 position) => position + extents *
		select(select(0f, 2f, position < -extents), -2f, position > extents);

	public static Area2D FromView(Camera camera)
	{
		Area2D area;
		area.extents.y =
			-Mathf.Tan(camera.fieldOfView * 0.5f * Mathf.Deg2Rad) *
			camera.transform.localPosition.z;
		area.extents.x = area.extents.y * camera.aspect;
		return area;
	}
}
```

Now we can get the world area when `Game` awakens and store it in a field. We could show duplicate balls when they partially cross an edge, but we'll keep thing simple and show only a single instance, teleporting it when needed. To hide this sudden teleportation we introduce an opaque boundary region. It needs to contain the largest ball's visualization. We'll use 1 as the maximum radius, so its child cubes can extend up to $\sqrt{\frac{3}{2}}$, roughly 1.24. We use that as the world bounds radius, but make it configurable as well. Then we position simple cube prefab instances on the edges scaled to fill the view.

```
    [SerializeField]
    Transform worldBoundsPrefab;

    [SerializeField, Min(0f)]
    float worldBoundsRadius = 1.24f;

    Area2D worldArea;

    void Awake()
    {
        worldArea = Area2D.FromView(Camera.main);
        Transform
            boundsT = Instantiate(worldBoundsPrefab),
            boundsB = Instantiate(worldBoundsPrefab),
            boundsL = Instantiate(worldBoundsPrefab),
            boundsR = Instantiate(worldBoundsPrefab);
        boundsT.localPosition = new Vector3(0f, worldArea.extents.y);
        boundsB.localPosition = new Vector3(0f, -worldArea.extents.y);
        boundsL.localPosition = new Vector3(-worldArea.extents.x, 0f);
        boundsR.localPosition = new Vector3(worldArea.extents.x, 0f);
        boundsT.localScale = boundsB.localScale =
            new Vector3(worldArea.extents.x, worldBoundsRadius, worldBoundsRadius) * 2f;
        boundsL.localScale = boundsR.localScale =
            new Vector3(worldBoundsRadius, worldArea.extents.y, worldBoundsRadius) * 2f;
    }
```

World boundary; bright red.

I used bright red in the screenshot above to make the boundary clearly visible, but use a dark red value of 16/255 for the rest of this tutorial. You could also make the boundary black, but a visible boundary act as a warning sign that balls might be hiding there.

2 Balls

Our game will have a lot of balls. We visualize them with prefab instances, but we also need a physical representation of them for our physics engine. We keep the two separate, so it's possible to easily change their representation. For example, instead of prefabs we could use procedural drawing or 2D sprites instead of 3D models. Also, to make our physics engine fast we will just jobs and thus need to store the ball state in structs that we want to keep as small as possible.

2.1 Ball Manager

To keep track of the balls and keep their physics states and visualizations synchronized we introduce a `BallManager` component type. Give it an `Initialize` method with a parameter for the world area, a `Dispose` method, and a `StartNewGame` method. Follow that with an `UpdateBalls` job that takes a time delta and returns a job handle, and a `ResolveBalls` that takes a job handle dependency. The first method will take care of linear motion, after which other things like ball destruction could happen, after which we'll resolve the balls to take care of bounces and spawning. The last method that we need is `UpdateVisualization`, which has a parameter for an extrapolated time delta.

```
using System.Collections.Generic;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

public class BallManager : MonoBehaviour
{
	public void Initialize(Area2D worldArea) { }

	public void Dispose() { }

	public void StartNewGame() { }

	public JobHandle UpdateBalls(float dt)
	{
		return default;
	}

	public void ResolveBalls(JobHandle dependency)
	{
		dependency.Complete();
	}

	public void UpdateVisualization(float dtExtrapolated) { }
}
```

Add a configuration field for the ball manager to `Game`. Initialize it when the game awakens and dispose it when the game is disabled. We won't invoke `StartNewGame` at this time, instead letting the game immediately run from its initial state.

```
	[SerializeField]
	BallManager ballManager;

	…

	void Awake()
	{
		…

		ballManager.Initialize(worldArea);
	}

	void OnDisable()
	{
		ballManager.Dispose();
	}
```

Like Unity's physics we'll use a fixed delta time for our simulation to make it independent of the frame rate. Make the fixed delta time configurable and let's set it to 0.01, half Unity's default. That makes our physics system update 100 times per second.

Keep track of the accumulated time delta in a field, increase it each update, invoke a new `UpdateGameState` method as often as needed, and then update the ball visualization with the remaining time delta.

`UpdateGameState` updates and then immediately resolves the balls with a given time delta. We make the time delta a parameter of the method to make it simple to switch to a variable time delta, so you could compare both approaches.

```
	[SerializeField, Min(0.001f)]
	float fixedDeltaTime = 0.01f;

	float dt;

	…

	void Update()
	{
		dt += Time.deltaTime;
		while (dt > fixedDeltaTime)
		{
			UpdateGameState(fixedDeltaTime);
			dt -= fixedDeltaTime;
		}

		ballManager.UpdateVisualization(dt);
	}

	void UpdateGameState(float dt)
	{
		JobHandle handle = ballManager.UpdateBalls(dt);
		ballManager.ResolveBalls(handle);
	}
```

You could add the `BallManager` component to the same object that has the `Game` component or give it its own game object.

2.2 Ball State

Create a `BallState` struct to contain the state of a ball. It needs a 2D position and velocity, a mass, a current radius, and a target radius. Also give it integers to define a stage and type for it, plus a boolean to indicate whether the ball is alive.

We'll support three ball states, from small to large their masses are ¼, ½, and 1, with accompanying radii ½, √½, and 1. Store these in separate static readonly arrays that can be used by Burst jobs, along with a constant integer that designates the largest as the initial stage. The idea is that when balls are hit by a bullet they split in two balls that have the next smaller stage.

```
using Unity.Mathematics;

public struct BallState
{
    public const int initialStage = 2;

    public static readonly float[] masses =
    {
        0.25f, 0.5f, 1f
    };

    public static readonly float[] radii =
    {
        0.5f, 0.7071067812f, 1f
    };

    public float2 position, velocity;

    public float mass, radius, targetRadius;

    public int stage, type;

    public bool alive;
}
```

How do those masses and radii relate?

As our balls are 2D they are effectively discs, so I made their mass relative to their surface area, which is πr^2. I chose a mass of 1 for radius 1, so $m = r^2$ and $r = \sqrt{m}$.

Can Burst jobs access static arrays from different types?

It doesn't matter where the readonly static arrays are defined, but the Burst compiler will hard-code them in the job at compile time. So changing the arrays via code won't affect the jobs.

As the amount of balls is highly variable `BallManager` will store their states in a native list, with initial capacity set to 100. Make sure that you have the *Collections* package imported and restart your IDE if it doesn't detect it immediately. The manager also tracks the visualization instances with a regular list. When a new game is started all existing instances need to be despawned before clearing the lists.

```
	NativeList<BallState> states;

	List<BallVisualization> visualizations;

	public void Initialize(Area2D worldArea) {
		states = new(100, Allocator.Persistent);
		visualizations = new(states.Capacity);
	}

	public void Dispose() {
		states.Dispose();
	}

	public void StartNewGame() {
		for (int i = 0; i < visualizations.Count; i++)
		{
			visualizations[i].Despawn();
		}
		visualizations.Clear();
		states.Clear();
	}
```

2.3 Spawning Balls

The manager will spawn balls based on a cooldown. Give it a configurable starting cooldown of 4 with at configurable persistence set to 0.96. That will speed up the spawning as the game progresses. We decrease the cooldown in `UpdateBalls` but will delay spawning a new ball until the end of `ResolveBalls`, when all jobs are completed. At that point we add a single new randomized ball state if needed, with mass and target radius matching the initial stage and a random type, which is simply the index of a configurable prefab array, which will contain references to our prefabs. Make sure that the ball is alive and omit its position and velocity.

```
	[SerializeField]
	BallVisualization[] ballPrefabs;

	[SerializeField, Min(0f)]
	float startingCooldown = 4f;

	[SerializeField, Range(0.1f, 1f)]
	float cooldownPersistence = 0.96f;

	float cooldown, cooldownDuration;

	public void Initialize(Area2D worldArea) {
		…
		cooldown = cooldownDuration = startingCooldown;
	}

	public void StartNewGame() {
		…
		cooldown = cooldownDuration = startingCooldown;
	}

	public JobHandle UpdateBalls(float dt)
	{
		cooldown -= dt;
		updateBallJob.dt = dt;
		return updateBallJob.Schedule(states.Length, default);
	}

	public void ResolveBalls(JobHandle dependency)
	{
		dependency.Complete();

		if (cooldown <= 0f)
		{
			cooldown += cooldownDuration;
			cooldownDuration *= cooldownPersistence;
			states.Add(new BallState
			{
				mass = BallState.masses[BallState.initialStage],
				targetRadius = BallState.radii[BallState.initialStage],
				stage = BallState.initialStage,
				type = Random.Range(0, ballPrefabs.Length),
				alive = true
			});
		}
	}
```

The spawning of the visualization is done in `UpdateVisualization`. Add as many visualizations as needed, using the ball type to select the appropriate prefab.

```
	public void UpdateVisualization(float dtExtrapolated)
	{
		for (int i = visualizations.Count; i < states.Length; i++)
		{
			visualizations.Add(ballPrefabs[states[i].type].Spawn());
		}
	}
```

At this point we will see overlapping balls spawn at the origin after we enter play mode. To speed this up you can temporarily reduce the starting cooldown to 0.1. These balls are small, because the prefab represents a ball with radius ½.

To make the visualization match the ball state, add an `UpdateVisualization` method to `BallVisualization` with parameters for its 2D position and target radius. Setting the position is straightforward, but the starting radius of our balls is zero. We'll grow it to the target radius by simply adding the time since it was spawned. To make the visualization's radius match this we have to set its scale to double the indicated radius. We can avoid adjusting the scale each update by keeping track of the visualization's own radius and only rescaling when needed.

```
	float radius;

	public BallVisualization Spawn()
	{
		BallVisualization instance = pool.GetInstance(this);
		instance.pool = pool;
		instance.radius = -1f;
		return instance;
	}

	public void Despawn() => pool.Recycle(this);

	public void UpdateVisualization(float2 position, float targetRadius)
	{
		transform.localPosition = new Vector3(position.x, position.y);
		if (radius != targetRadius)
		{
			radius = targetRadius;
			transform.localScale = Vector3.one * (2f * targetRadius);
		}
	}
```

Now `BallManager.UpdateVisualization` has to update all its visualizations. We could directly pass through the state's position and radius, but that would introduce visual stuttering when the actual and fixed time deltas don't match. So we use the extrapolated time delta to move the visualization forward in time relative to its simulated state to match the actual elapsed time. This is just a linear guess that fails when balls are about to bounce, but that mistake will only be obvious at such low frame rates that the game would be unplayable anyway.

```
	public void UpdateVisualization(float dtExtrapolated)
	{
		…

		for (int i = 0; i < visualizations.Count; i++)
		{
			BallState state = states[i];
			visualizations[i].UpdateVisualization(
				state.position + state.velocity * dtExtrapolated,
				Mathf.Min(state.radius + dtExtrapolated, state.targetRadius)
			);
		}
	}
```

Now the balls will no longer be visible, because their radii are nearly zero.

2.4 Updating Balls

To update the balls introduce an `UpdateBallJob` Burst job that takes care of updating a single ball. It performs simple linear motion and grows the radius by adding the time delta to it. The new position has to be wrapped to keep it inside the world area. Also enforce a max speed when needed to ensure that balls won't move too fast to keep the game playable.

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;

using static Unity.Mathematics.math;

[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
public struct UpdateBallJob : IJobFor
{
    public NativeList<BallState> balls;

    public Area2D worldArea;

    public float dt, maxSpeed;

    public void Execute(int i)
    {
        BallState ball = balls[i];
        if (dot(ball.velocity, ball.velocity) > maxSpeed * maxSpeed)
        {
            ball.velocity = normalize(ball.velocity) * maxSpeed;
        }
        ball.position = worldArea.Wrap(ball.position + ball.velocity * dt);
        ball.radius = min(ball.targetRadius, ball.radius + dt);
        balls[i] = ball;
    }
}
```

Shouldn't `UpdateBallJob` only update living balls?

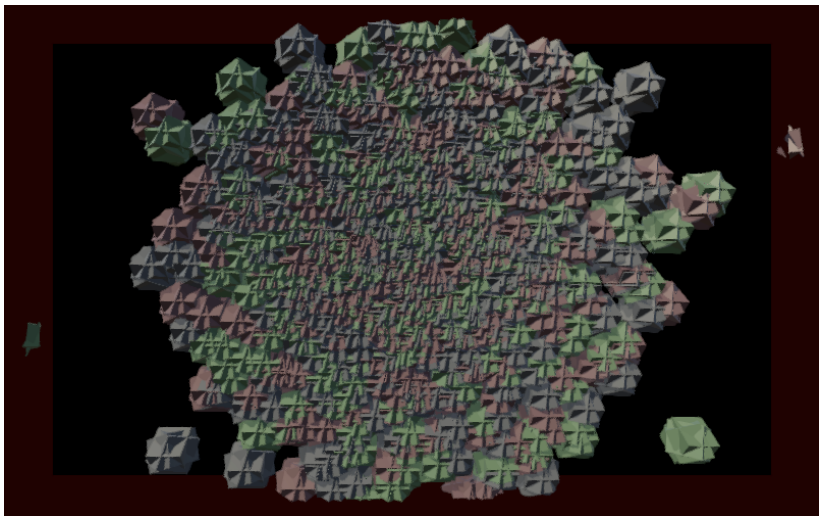
Technically yes, but dead balls will be rare and updating dead balls has no side effects. I only check whether a ball is alive when needed for correctness.

We'll initialize this job once when the ball manager initializes and schedule it when the balls should be updated. Also add a configurable max speed set to 12.5 by default. That ensures that two of the smallest balls won't pass though each other when they would collide head-on at max speed. Also give the new balls a random start speed with a configurable max, set to 4 by default.

```
[SerializeField, Min(0f)]
float maxSpeed = 12.5f, maxStartSpeed = 4f;

…

UpdateBallJob updateBallJob;

public void Initialize(Area2D worldArea) {
	…

	updateBallJob = new UpdateBallJob
	{
		balls = states,
		worldArea = worldArea,
		maxSpeed = maxSpeed
	};
}

…

public JobHandle UpdateBalls(float dt)
{
	updateBallJob.dt = dt;
	return updateBallJob.Schedule(states.Length, default);
}

public void ResolveBalls(Vector2 avoidSpawnPosition, JobHandle dependency)
{
	…
		states.Add(new BallState
		{
			velocity = Random.insideUnitCircle * maxStartSpeed,
			…
		});
	…
}
```

Why not schedule parallel?

There will be around 200 balls at maximum, which guarantees a game-over state. It isn't worth trying to spread so little work over multiple threads.



Many balls; starting cooldown 0.1.

2.5 Spawning in Empty Space

All balls currently spawn at the origin and a low cooldown shows that they overlap. Let's make it so new balls are only spawned in an empty region that could contain them at their full size. Create a `VerifySpawnPositionJob` job for this that loops through all living balls to check whether they overlap a given spawn position and radius. This can be done by performing a point-inside-circle check relative to the spawn position, using the sum of both radii for the check. There is an overlap when $||p||^2 < r^2$ where p is the spawn position relative to the ball and r is the radius sum. The relative position has to be wrapped to make this work correctly near the world boundary. Store the indication of success in a `NativeReference`.

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

using static Unity.Mathematics.math;

[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
public struct VerifySpawnPositionJob : IJob
{
    [ReadOnly]
    public NativeList<BallState> balls;

    [WriteOnly]
    public NativeReference<bool> success;

    public Area2D worldArea;

    public float2 position;

    public float radius;

    public void Execute()
    {
        for (int i = 0; i < balls.Length; i++)
        {
            BallState ball = balls[i];
            if (ball.alive)
            {
                float2 p = worldArea.Wrap(position - ball.position);
                float r = ball.radius + radius;
                if (dot(p, p) <= r * r)
                {
                    success.Value = false;
                    return;
                }
            }
        }

        success.Value = true;
    }
}
```

What's a `NativeReference`?

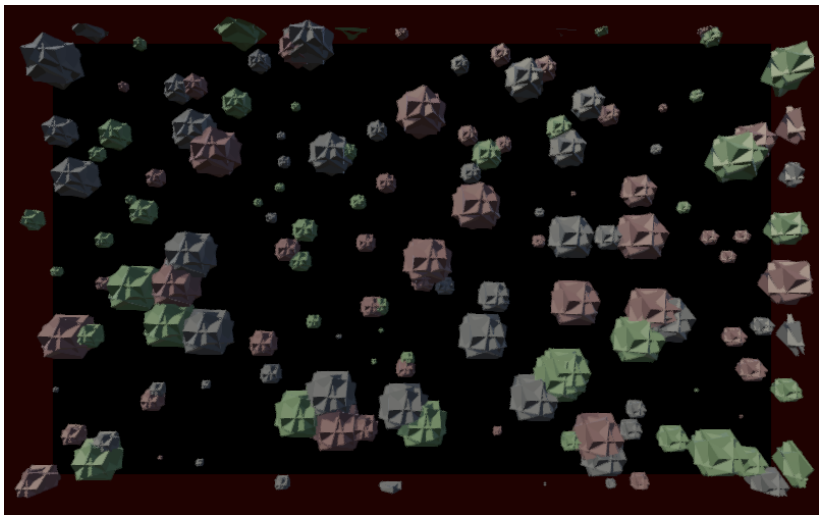
It is a convenient type from the *Collections* package that functions like a native array of length one.

Schedule the verification job in `ResolveBalls` if needed and check whether it was successful before spawning. If not we simply try again next physics step. Because we only access the success state directly after scheduling the job I didn't bother with a separate field for it and directly use the job's field.

```
    VerifySpawnPositionJob verifySpawnPositionJob;

    public void Initialize(Area2D worldArea) {
        …
        verifySpawnPositionJob = new VerifySpawnPositionJob
        {
            balls = states,
            success = new NativeReference<bool>(Allocator.Persistent),
            worldArea = worldArea,
            radius = BallState.radii[BallState.initialStage]
        };
    }

    public void Dispose() {
        states.Dispose();
        verifySpawnPositionJob.success.Dispose();
    }

    public void ResolveBalls(float dt, JobHandle dependency)
    {
        if (cooldown <= 0f)
        {
            dependency = verifySpawnPositionJob.Schedule(dependency);
        }
        dependency.Complete();

        if (cooldown <= 0f && verifySpawnPositionJob.success.Value)
        {
            cooldown += cooldownDuration;
            …
        }
    }
```

The game now only spawns a new ball once the previous one has moved far enough away from the origin. Change this so the balls are randomly spawning in the entire world.

```
		if (cooldown <= 0f)
		{
			verifySpawnPositionJob.position = updateBallJob.worldArea.RandomVector2;
			dependency = verifySpawnPositionJob.Schedule(dependency);
		}
		dependency.Complete();

		if (cooldown <= 0f && verifySpawnPositionJob.success.Value)
		{
			cooldown += cooldownDuration;
			cooldownDuration *= cooldownPersistence;
			states.Add(new BallState
			{
				position = verifySpawnPositionJob.position,
				…
			});
		}
```

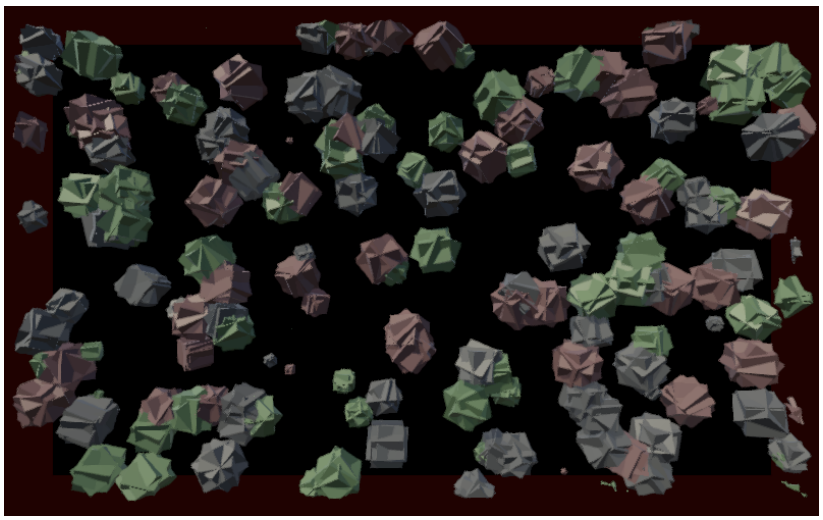


Spawning everywhere.

2.6 Visual Ball Variety

We already have some visual variety by spawning different prefab versions, but all balls still have the same shape. To change this assign random rotations to all the visualization's children when it spawns.

```
	public BallVisualization Spawn()
	{
		…
		for (int i = 0; i < instance.transform.childCount; i++)
		{
			instance.transform.GetChild(i).localRotation = Random.rotation;
		}
	}
```



Different balls.

To add even more variety and make it more lively let's also give each visualization a random rotation, with a configurable spin speed range set to 20–60 by default. Note that this is purely visual and thus bases on the regular time delta. The simulated balls don't spin.

```
	[SerializeField, Min(0f)]
	float minSpinSpeed = 20f, maxSpinSpeed = 60f;

	PrefabInstancePool<BallVisualization> pool;

	Vector3 rotationAxis;

	float radius, rotationSpeed, rotationAngle;

	public BallVisualization Spawn()
	{
		BallVisualization instance = pool.GetInstance(this);
		instance.pool = pool;
		instance.radius = -1f;
		instance.rotationAxis = Random.onUnitSphere;
		instance.rotationSpeed = Random.Range(minSpinSpeed, maxSpinSpeed);
		…
	}

	public void Despawn() => pool.Recycle(this);

	public void UpdateVisualization(float2 position, float targetRadius)
	{
		rotationAngle += rotationSpeed * Time.deltaTime;
		if (rotationAngle > 360f)
		{
			rotationAngle -= 360f;
		}
		transform.SetLocalPositionAndRotation(
			new Vector3(position.x, position.y),
			Quaternion.AngleAxis(rotationAngle, rotationAxis)
		);
		//transform.localPosition = new Vector3(position.x, position.y);
		…
	}
```

2.7 Bouncing

The last thing that the balls need is to bounce, for which we'll create a `BounceBallsJob` that loops though all pairs of balls. For each pair of living balls we perform a point-inside-circle check to see whether they overlap. Besides the balls and world area we also need to know the bounce strength and time delta to push them apart.

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

using static Unity.Mathematics.math;

[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
public struct BounceBallsJob : IJob
{
    public NativeList<BallState> balls;

    public Area2D worldArea;

    public float bounceStrength, dt;

    public void Execute()
    {
        for (int i = 0; i < balls.Length; i++)
        {
            BallState a = balls[i];
            if (!a.alive)
            {
                continue;
            }

            for (int j = i + 1; j < balls.Length; j++)
            {
                BallState b = balls[j];
                if (!b.alive)
                {
                    continue;
                }

                float2 p = worldArea.Wrap(b.position - a.position);
                float r = a.radius + b.radius;
                if (dot(p, p) < r * r) { }
            }
        }
    }
}
```

Could we parallelize `BounceBallsJob`?

Yes, by having every ball pair itself with all other balls and only adjusting itself. That would require n^2 checks, where n is one less than the amount of living balls. That's 40,000 checks for 201 living balls and each bounce would have to be resolved twice.

What we're doing instead is $1 + 2 + ... + n = \sum_{k=1}^{n} k = (n + 1)\frac{n}{2}$ checks, resolving each bounce only once. That's 20,010 checks for 201 living balls, roughly half the work. It would require at least three threads running in parallel to outperform that, over which we have no direct control. In the worst case it all runs on the same thread.

We implement a simple soft deforming bounce, which is done by making the balls repel each other the more they overlap. We use the formula $a = s(||p|| - r)\hat{p}$ where a is the bounce acceleration for the first ball and s is the bounce strength. The second ball accelerates in the opposite direction. This formula ignores the ball mass so all balls bounce the same.

```
if (dot(p, p) < r * r)
{
	float2 v =
		(1f - r * rsqrt(max(dot(p, p), 0.0001f))) *
		bounceStrength * dt * p;
	a.velocity += v;
	b.velocity -= v;
	balls[i] = a;
	balls[j] = b;
}
```

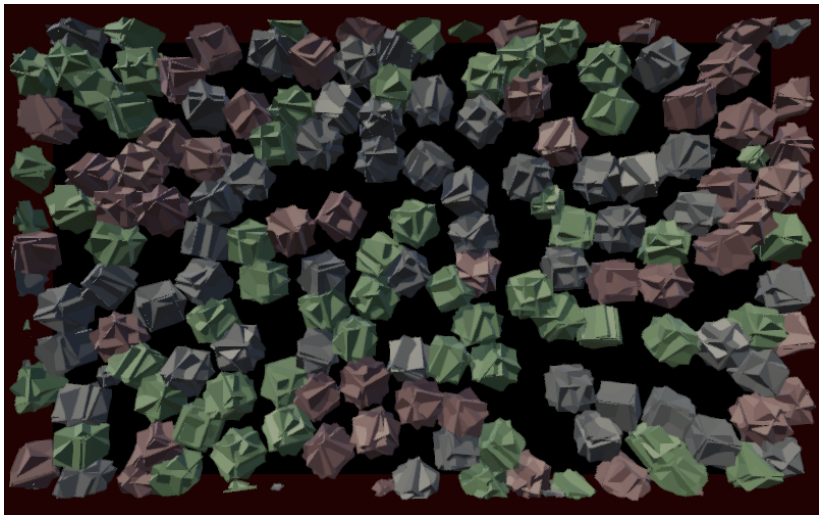
How did you derive that computation from the formula?

$$a = s(||p|| - r)\hat{p} = s(||p|| - r)\frac{p}{||p||} = s\left(1 - \frac{r}{||p||}\right)p$$

Factor the time delta into that to get the velocity change.

Add this job to the ball manager with a configurable bounce strength set to 100 by default. Schedule it at the start of `ResolveBalls`.

```
	[SerializeField, Min(0f)]
	float bounceStrength = 100f;

	…

	BounceBallsJob bounceBallsJob;

	VerifySpawnPositionJob verifySpawnPositionJob;

	public void Initialize(Area2D worldArea) {
		…
		bounceBallsJob = new BounceBallsJob
		{
			balls = states,
			worldArea = worldArea,
			bounceStrength = bounceStrength
		};
		…
	}

	…

	public JobHandle UpdateBalls(float dt)
	{
		cooldown -= dt;
		bounceBallsJob.dt = updateBallJob.dt = dt;
		return updateBallJob.Schedule(states.Length, default);
	}

	public void ResolveBalls(JobHandle dependency)
	{
		dependency = bounceBallsJob.Schedule(dependency);
		…
	}
```



Bouncing balls, around 200.

This currently works correctly because all balls have the same mass. To also support different masses we have to change the bounce formula slightly, multiplying it with $\frac{2m_b}{m_a + m_b}$ for the first ball and $\frac{2m_b}{m_a + m_b}$ for the second ball, where m_a and m_b are their masses. This distributes the bounce strength according to relative mass, while balls of the same mass always bounce with the same strength, no matter how massive are.

```
                float2 v =
                    (2f / (a.mass + b.mass)) *
                    (1f - r * rsqrt(max(dot(p, p), 0.0001f))) *
                    bounceStrength * dt * p;
                a.velocity += b.mass * v;
                b.velocity -= a.mass * v;
```

3 Player

Now that the balls are functional we move on to the player.

3.1 Following the Cursor

Create a `Player` component and attach it to the player game object or prefab and activate it in the scene. Give the player a configurable radius set to 0.5, which should be publicly available. We make the player follow the cursor. Do this by giving it a `Vector2 TargetPosition` property that is publicly set and a `Position` property that is privately set. Set both to a given position in a `StartNewGame` method. It needs an `UpdateState` method that initially directly sets the current position to the target. Also give it `Initialize` and `Dispose` methods.

As the player also gets updated using the fixed time step it will visually stutter. In this case extrapolation won't work because the player's movement is unpredictable and can be very fast. We'll instead keep track of its previous position and interpolate to its current position given an interpolator based on the time delta, provided via a parameter of `UpdateVisualization`. Thus the player will visually lag one step behind, but that isn't a problem with a small fixed time delta.

```
using TMPro;
using Unity.Collections;
using UnityEngine;

public class Player : MonoBehaviour
{
	[SerializeField, Min(0f)]
	float radius = 0.5f;

	Vector2 previousPosition;

	public float Radius => radius;

	public Vector2 Position
	{ get; private set; }

	public Vector2 TargetPosition
	{ private get; set; }

	public void Initialize() { }

	public void Dispose() { }

	public void StartNewGame(Vector2 position)
	{
		Position = TargetPosition = previousPosition = position;
	}

	public void UpdateState(float dt)
	{
		previousPosition = Position;
		Position = TargetPosition;
	}

	public void UpdateVisualization(float dtInterpolator)
	{
		transform.localPosition =
			Vector2.LerpUnclamped(previousPosition, Position, dtInterpolator);
	}
}
```

Add a configuration field for the player to `Game`, initialize and dispose it, and also update it and its visualization. The interpolator is the remaining time delta divided by the fixed time delta.

We also clamp the player position to the world area, because as it follows the cursor it won't wrap. To keep it visible we use a reduced player area, shrinking it by the world bound radius and the player radius. Set the target position at the start of `Update`, by using a new `GetTargetPoint` method that converts the mouse position to a clamped world position.

```
	[SerializeField]
	Player player;

	…

	Area2D worldArea, playerArea;

	void Awake()
	{
		…
		player.Initialize();
		playerArea.extents = worldArea.extents - worldBoundsRadius - player.Radius;
	}

	void OnDisable()
	{
		ballManager.Dispose();
		player.Dispose();
	}

	void Update()
	{
		player.TargetPosition = GetTargetPoint();
		float dt = Time.deltaTime;
		…

		ballManager.UpdateVisualization();
		player.UpdateVisualization(dt / fixedDeltaTime);
	}

	void UpdateGameState(float dt)
	{
		player.UpdateState(dt);
		…
	}

	Vector2 GetTargetPoint()
	{
		Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
		Vector2 p = ray.origin - ray.direction * (ray.origin.z / ray.direction.z);
		p.x = Mathf.Clamp(p.x, -playerArea.extents.x, playerArea.extents.x);
		p.y = Mathf.Clamp(p.y, -playerArea.extents.y, playerArea.extents.y);
		return p;
	}
```

This makes the player teleport each time its position changes, although its visualization linearly interpolates. To prevent teleportation through balls we slow down the player's movement by using `Vector2.SmoothDamp` in `Player.UpdateState`, but only when its distance to the target is greater than 0.01. We need to keep track of its velocity and give it a configurable cursor follow speed and cursor snap duration, set to 40 and 0.05 by default.

```
	[SerializeField, Min(0.01f)]
	float cursorFollowSpeed = 40f, cursorSnapDuration = 0.05f;

	Vector2 previousPosition, velocity;

	…

	public void StartNewGame(Vector2 position)
	{
		…
		velocity = Vector2.zero;
	}

	public void UpdateState(float dt)
	{
		previousPosition = Position;
		//Position = TargetPosition;
		Vector2 targetVector = TargetPosition - Position;
		float squareTargetDistance = targetVector.sqrMagnitude;
		if (squareTargetDistance > 0.0001f)
		{
			Position = Vector2.SmoothDamp(
				Position, TargetPosition, ref velocity,
				cursorSnapDuration, cursorFollowSpeed, dt
			);
		}
	}
```

This makes the player smoothly stick close to the cursor if it changes a little, while if the cursor is moved at great speed the player will start to lag behind. The last step is to hide the cursor, by setting `Cursor.visible` to `false` when the game awakens. Note that this only affects the editor game window after you click or touch it.

```
	void Awake()
	{
		…
		Cursor.visible = false;
	}
```

3.2 Avoiding Player When Spawning

The player should avoid touching balls, so we should also avoid spawning new balls that immediately overlap the player. So add a position and radius to avoid to `VerifySpawnPositionJob` and abort if the spawn position is too close.

```
	public float2 avoidPosition, position;

	public float avoidRadius, radius;

	public void Execute()
	{
		float2 p = worldArea.Wrap(position - avoidPosition);
		float r = avoidRadius + radius;
		if (dot(p, p) <= r * r)
		{
			success.Value = false;
			return;
		}

		for (int i = 0; i < balls.Length; i++)
		{
			BallState ball = balls[i];
			if (ball.alive)
			{
				p = worldArea.Wrap(position - ball.position);
				r = radius + ball.radius;
				if (dot(p, p) <= r * r)
				{
					success.Value = false;
					return;
				}
			}
		}

		success.Value = true;
	}
```

Add a configurable spawn radius for avoidance to `BallManager`, set to 2 by default. Also add the position to avoid as a parameter to `ResolveBalls`.

```
	[SerializeField, Min(0f)]
	float avoidSpawnRadius = 2f, startingCooldown = 4f;

	…

	public void Initialize(Area2D worldArea) {
		…
		verifySpawnPositionJob = new VerifySpawnPositionJob
		{
			…
			avoidRadius = avoidSpawnRadius,
			radius = BallState.radii[BallState.initialStage]
		};
	}

	public void ResolveBalls(Vector2 avoidSpawnPosition, JobHandle dependency)
	{
		dependency = bounceBallsJob.Schedule(dependency);
		if (cooldown <= 0f)
		{
			verifySpawnPositionJob.avoidPosition = avoidSpawnPosition;
			verifySpawnPositionJob.position = updateBallJob.worldArea.RandomVector2;
			dependency = verifySpawnPositionJob.Schedule(dependency);
		}
		…
	}
```

Now `Game` can pass the player position when it asks the manager to resolve the balls.

```
        ballManager.ResolveBalls(player.Position, handle);
```

3.3 Aiming

Make `Player` automatically aim itself based on its last movement delta, converting it to a direction angle. To make this appear smooth also apply visual interpolation to it.

```
    float directionAngle, previousDirectionAngle;

    …

    public void StartNewGame(Vector2 position)
    {
        Position = TargetPosition = position;
        directionAngle = previousDirectionAngle = 0f;
    }

    public void UpdateState(float dt)
    {
        previousPosition = Position;
        previousDirectionAngle = directionAngle;
        …
        if (squareTargetDistance > 0.0001f)
        {
            Position = Vector2.SmoothDamp(
                Position, TargetPosition, ref velocity,
                cursorSnapDuration, cursorFollowSpeed, dt
            );
            directionAngle = Mathf.Atan2(targetVector.x, targetVector.y) * -Mathf.Rad2Deg;
        }
    }

    public void UpdateVisualization(float dtInterpolator)
    {
        //transform.localPosition =
        transform.SetLocalPositionAndRotation(
            Vector2.LerpUnclamped(previousPosition, Position, dtInterpolator),
            Quaternion.Euler(
                0f, 0f,
                Mathf.LerpAngle(previousDirectionAngle, directionAngle, dtInterpolator)
            )
        );
    }
```

Death vs Monstars allows locking the aim angle and we will support this as well, by only adjusting the aim angle when a public `FreeAim` property is set to `true`.

```
	public bool FreeAim
	{ get; set; }

	…

	public void UpdateState(float dt)
	{
		…
		if (squareTargetDistance > 0.0001f)
		{
			…
			if (FreeAim)
			{
				directionAngle =
					Mathf.Atan2(targetVector.x, targetVector.y) * -Mathf.Rad2Deg;
			}
		}
	}
```

Lock the aim at the start of `Game.Update` while either the primary mouse button or the space bar is held down.

```
	void Update()
	{
		player.FreeAim = !Input.GetMouseButton(0) && !Input.GetKey(KeyCode.Space);
		player.TargetPosition = GetTargetPoint();
		…
	}
```

3.4 Firing

Now that we can aim, the player should be able to fire bullets, for which we need add a configurable fire cooldown duration set to 0.1 and also a fire spread angle set to 5, which randomizes the bullet direction a little. Also introduce a fire offset vector that makes bullets spawn on the player's radius in the appropriate direction instead of its center. Update the cooldown and fire at the end of `UpdateState`, but just log that we would do so for now because we don't have bullets to spawn yet. Finally, like *Death vs Monstars* we will shoot backwards, not forwards. So to aim at a ball you have to move away from it.

```
    [SerializeField, Min(0f)]
    float fireCooldown = 0.1f, fireSpreadAngle = 5f;

    Vector2 fireOffset, previousPosition, velocity;

    float cooldown, directionAngle, previousDirectionAngle;

    …

    public void StartNewGame(Vector2 position)
    {
        …
        fireOffset = new Vector2(0f, radius);
    }

    public void UpdateState(float dt)
    {
        …
        if (squareTargetDistance > 0.0001f)
        {
            …
            if (FreeAim)
            {
                fireOffset = targetVector * (radius / Mathf.Sqrt(squareTargetDistance));
                directionAngle =
                    Mathf.Atan2(targetVector.x, targetVector.y) * -Mathf.Rad2Deg;
            }
        }

        cooldown -= dt;
        if (cooldown <= 0f)
        {
            cooldown += fireCooldown;
            Vector2 firePosition = Position - fireOffset;
            float fireAngle =
                directionAngle + 180f + Random.Range(-fireSpreadAngle, fireSpreadAngle);
            Debug.Log($"fire position {firePosition} angle {fireAngle}");
        }
    }
```

How does that string work?

Using a string prefixed with `$` is known as string interpolation and allows us to directly reference fields by putting them inside curly brackets inside the string.

We could've also written `"fire position " + firePower + " angle " + fireAngle`.

4 Bullets

To make the player fire for real we have to manage bullets.

4.1 Bullet State

Introduce a `BulletState` struct with a position and a velocity. Bullets have limited lifetime so include a field for the time they have left. A bullet counts as alive while its remaining time is greater than zero, so add a convenient property to check that.

```
using Unity.Mathematics;

public struct BulletState
{
    public float2 position, velocity;

    public float timeRemaining;

    public bool Alive => timeRemaining > 0f;
}
```

Create an `UpdateBulletJob` that both moves and decreases the time remaining of a single bullet. Like balls bullets also wrap, so the player could shoot from one side of the world to the opposite side.

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;

[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
public struct UpdateBulletJob : IJobFor
{
    public NativeList<BulletState> bullets;

    public Area2D worldArea;

    public float dt;

    public void Execute(int i)
    {
        BulletState bullet = bullets[i];
        bullet.timeRemaining -= dt;
        bullet.position = worldArea.Wrap(bullet.position + bullet.velocity * dt);
        bullets[i] = bullet;
    }
}
```

4.2 Bullet Visualization

Give `BulletVisualization` an `UpdateVisualization` method with parameters for a `Vector2` position and a life factor. We need to pass the life factor to the GPU, for which we use a material property block, so we also need to keep track of the visualization's `MeshRenderer`. Finally, to correctly orientate the bullet add a `Quaternion` parameter to its `Spawn` method.

```
using UnityEngine;

public class BulletVisualization : MonoBehaviour
{
	static int lifeFactorID = Shader.PropertyToID("_LifeFactor");

	static MaterialPropertyBlock materialPropertyBlock;

	PrefabInstancePool<BulletVisualization> pool;

	MeshRenderer meshRenderer;

	void Awake()
	{
		materialPropertyBlock ??= new MaterialPropertyBlock();
		meshRenderer = GetComponent<MeshRenderer>();
	}

	public BulletVisualization Spawn(Quaternion rotation)
	{
		BulletVisualization instance = pool.GetInstance(this);
		instance.pool = pool;
		instance.transform.localRotation = rotation;
		return instance;
	}

	public void Despawn() => pool.Recycle(this);

	public void UpdateVisualization(Vector2 position, float lifeFactor)
	{
		transform.localPosition = position;
		materialPropertyBlock.SetFloat(lifeFactorID, lifeFactor);
		meshRenderer.SetPropertyBlock(materialPropertyBlock);
	}
}
```

What does `??=` do?

It's a the null-coalescing operator that only performs the assignment in case of `null`. It is a shorter way of writing

```
		if (materialPropertyBlock == null)
		{
			materialPropertyBlock = new MaterialPropertyBlock();
		}
```

Note that this won't work as expected with `UnityEngine.Object` types because it bypasses its overloaded equality check, but material property blocks are regular class types.

4.3 Bullet Manager

Create a `BulletManager` that works like `BallManager` but adjusted to work with bullets. It only needs an `UpdateBullets` method to schedule its single job and no resolve method. The player takes care of the fire cooldown so the manger has nothing to check and won't spawn bullets on its own. We do give it configuration options for the bullet speed and start lifetime, set to 12 and 1 by default.

```
using System.Collections.Generic;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

public class BulletManager : MonoBehaviour
{
    [SerializeField]
    BulletVisualization bulletPrefab;

    [SerializeField, Min(0f)]
    float speed = 12f, startLifetime = 1f;

    NativeList<BulletState> states;

    List<BulletVisualization> visualizations;

    UpdateBulletJob updateBulletJob;

    public void Initialize(Area2D worldArea) {
        states = new(100, Allocator.Persistent);
        visualizations = new(states.Capacity);
        updateBulletJob = new UpdateBulletJob
        {
            bullets = states,
            worldArea = worldArea
        };
    }

    public void Dispose() {
        states.Dispose();
    }

    public void StartNewGame() {
        for (int i = 0; i < visualizations.Count; i++)
        {
            visualizations[i].Despawn();
        }
        visualizations.Clear();
        states.Clear();
    }

    public JobHandle UpdateBullets(float dt)
    {
        updateBulletJob.dt = dt;
        return updateBulletJob.Schedule(states.Length, default);
    }

    public void UpdateVisualization(float dtExtrapolated)
    {
        for (int i = 0; i < visualizations.Count; i++)
        {
            BulletState state = states[i];
            if (state.Alive)
            {
                visualizations[i].UpdateVisualization(
                    state.position + state.velocity * dtExtrapolated,
                    Mathf.Max(0f, state.timeRemaining - dtExtrapolated) / startLifetime
                );
            }
        }
    }
}
```

Add a configurable bullets manager to `Game` and treat it the same as the ball manager. As the balls and bullets don't depend on each other when updating we can schedule them in parallel. Whether they are indeed executed on separate threads and in parallel is up to Unity.

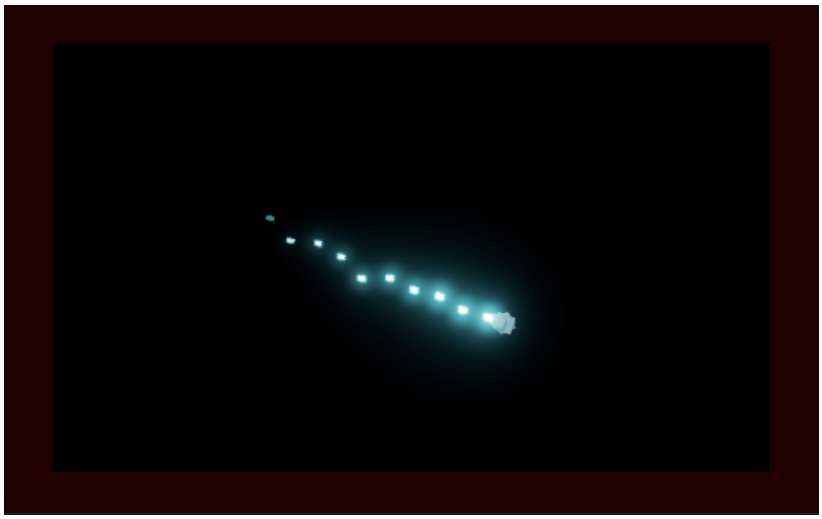
```
	[SerializeField]
	BulletManager bulletManager;

	…

	void Awake()
	{
		…

		ballManager.Initialize(worldArea);
		bulletManager.Initialize(worldArea);
		…
	}

	void OnDisable()
	{
		ballManager.Dispose();
		bulletManager.Dispose();
		player.Dispose();
	}

	void Update()
	{
		…

		ballManager.UpdateVisualization(dt);
		bulletManager.UpdateVisualization(dt);
		player.UpdateVisualization(dt / fixedDeltaTime);
	}

	void UpdateGameState(float dt)
	{
		player.UpdateState(dt);
		JobHandle handle = JobHandle.CombineDependencies(
			ballManager.UpdateBalls(dt),
			bulletManager.UpdateBullets(dt)
		);
		ballManager.ResolveBalls(handle);
	}
```

To create bullets give `BulletManager` an `Add` method with position and angle parameters. This is the only place where bullets will be added, so we create both its state and visualization at the same time.

```
	public void Add(Vector2 position, float angle)
	{
		Quaternion rotation = Quaternion.Euler(0f, 0f, angle);
		states.Add(new BulletState
		{
			position = position,
			velocity = (Vector2)(rotation * new Vector3(0f, speed)),
			timeRemaining = startLifetime
		});
		visualizations.Add(bulletPrefab.Spawn(rotation));
	}
```

Now `Player` can fire bullets for real.

```
    [SerializeField]
    BulletManager bulletManager;

    …

    public void UpdateState(float dt)
    {
        …
        if (cooldown <= 0f)
        {
            cooldown += fireCooldown;
            bulletManager.Add(
                Position - fireOffset,
                directionAngle + 180f + Random.Range(-fireSpreadAngle, fireSpreadAngle)
            );
            //Debug.Log($"fire position {firePosition} angle {fireAngle}");
        }
    }
```



Firing bullets.

At this point dead bullets will become invisible but still stick around. We get rid of the dead ones in `BulletManager.UpdateVisualization`.

```
            if (state.Alive)
            {
                …
            }
            else
            {
                int lastIndex = states.Length - 1;
                states[i] = states[lastIndex];
                states.Length -= 1;

                visualizations[i].Despawn();
                visualizations[i] = visualizations[lastIndex];
                visualizations.RemoveAt(lastIndex);
                i -= 1;
            }
```

5 Gameplay

We have bouncing balls and a shooting player, but they do not interact yet. To make the game playable we have to add this interaction and also track the player's health and score.

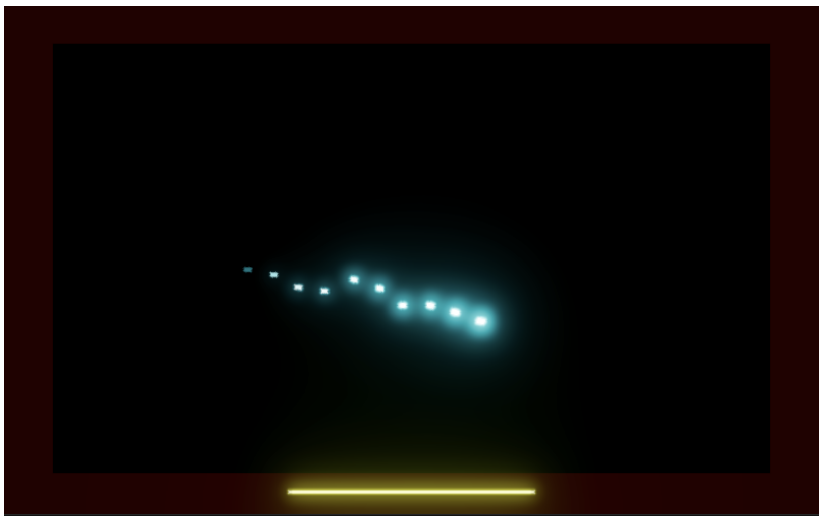
5.1 Health

We show the player's health by displaying a health bar at the bottom of the window. Create a cube with position (0,−9.5, −2) and scale (10, 0.1, 0.1), giving it the bright yellow material. Create a `HealthBar` component for it with a `Show` method that adjusts its X scale based on a given health percentage. Sets its with to zero when it awakens.

```
using UnityEngine;

public class HealthBar : MonoBehaviour
{
	Vector3 scale;

	float maxSize;

	void Awake()
	{
		scale = transform.localScale;
		maxSize = scale.x;
		transform.localScale = Vector3.zero;
	}

	public void Show(float healthPercentage)
	{
		scale.x = Mathf.Max(maxSize * healthPercentage, 0f);
		transform.localScale = scale;
	}
}
```

Give `Player` a configurable reference to the health bar and also a configurable max health set to 10. Track its current health via a `NativeReference` and also remember the last checked health value. Set everything to max in `StartNewGame` and update the last checked health in `UpdateVisualization` if needed. Have that method return whether the player's health has run out and if so also deactivate it. The player should also deactivate itself during initialization and activate itself again when a new game starts.

```
    [SerializeField]
    HealthBar healthBar;

    [SerializeField, Min(1)]
    int maxHealth = 10;

    int lastCheckedHealth;

    NativeReference<int> health;

    …

    public void Initialize() {
        health = new(Allocator.Persistent);
        gameObject.SetActive(false);
    }

    public void Dispose() {
        health.Dispose();
    }

    public void StartNewGame(Vector2 position)
    {
        …
        health.Value = lastCheckedHealth = maxHealth;
        healthBar.Show(1f);
        gameObject.SetActive(true);
    }

    …

    public bool UpdateVisualization(float dtInterpolator)
    {
        …

        if (lastCheckedHealth == health.Value)
        {
            return false;
        }

        lastCheckedHealth = health.Value;
        healthBar.Show((float)lastCheckedHealth / maxHealth);
        bool isDestroyed = lastCheckedHealth <= 0;
        if (isDestroyed)
        {
            gameObject.SetActive(false);
        }
        return isDestroyed;
    }
```



Health bar at full size; invisible player.

We now see a health bar but the player is invisible. To make this work correctly have `Game` track whether the game is currently playing, which is initially not the case. Give it a `StartNewGame` method that starts a new game and invoke it in update if we're not playing and the space bar is pressed. Only update the player is we're playing but update everything else always, so the initial game state will spawn balls for show and balls and bullets will keep moving even after the player has been destroyed.

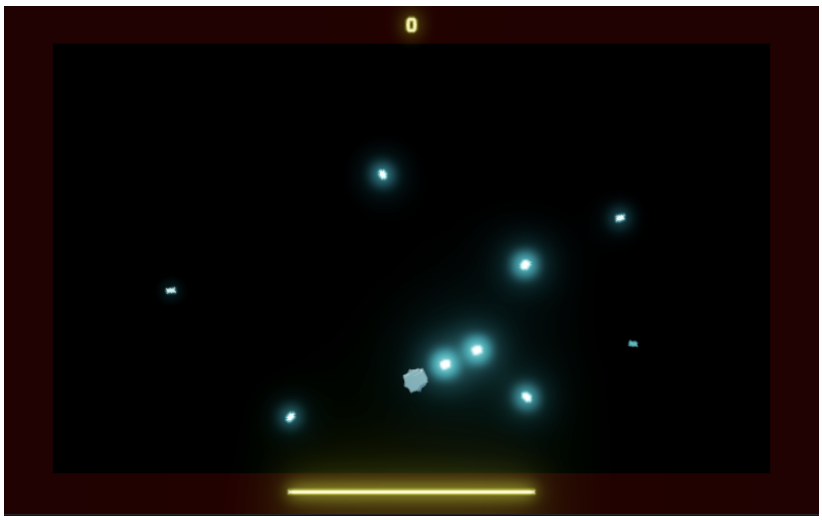
```
	bool isPlaying;

	…

	void StartNewGame()
	{
		isPlaying = true;
		ballManager.StartNewGame();
		bulletManager.StartNewGame();
		player.StartNewGame(GetTargetPoint());
	}

	void Update()
	{
		if (isPlaying)
		{
			player.FreeAim = !Input.GetMouseButton(0) && !Input.GetKey(KeyCode.Space);
			player.TargetPosition = GetTargetPoint();
		}
		else if (Input.GetKeyDown(KeyCode.Space))
		{
			StartNewGame();
		}

		…
		if (isPlaying && player.UpdateVisualization(dt / fixedDeltaTime))
		{
			isPlaying = false;
		}
	}

	void UpdateGameState(float dt)
	{
		if (isPlaying)
		{
			player.UpdateState(dt);
		}
		…
	}
```

The game still starts spawning balls immediately but the player won't be active yet. A proper game begins after the space bar is pressed, which clears all the balls and bullets, then shows the player and its health bar.

5.2 Score

We'll also show the player's score, via a text game object positioned at (0, 9.5, −2). Add a configuration field for it to `Player` and have it track its score via a `NativeReference`. Make the score initially score empty and update its text when needed in `UpdateVisualization`.

```
    [SerializeField]
    TextMeshPro scoreDisplay;

    …

    int lastCheckedHealth, lastCheckedScore;

    NativeReference<int> health, score;

    …

    public void Initialize() {
        health = new(Allocator.Persistent);
        score = new(Allocator.Persistent);
        scoreDisplay.SetText("");
        gameObject.SetActive(false);
    }

    public void Dispose() {
        health.Dispose();
        score.Dispose();
    }

    public void StartNewGame(Vector2 position)
    {
        …
        score.Value = lastCheckedScore = 0;
        scoreDisplay.SetText("0");
        gameObject.SetActive(true);
    }

    …

    public bool UpdateVisualization(float dtInterpolator)
    {
        …

        if (lastCheckedScore != score.Value)
        {
            lastCheckedScore = score.Value;
            scoreDisplay.SetText("{0}", lastCheckedScore);
        }

        if (lastCheckedHealth == health.Value)
        {
            return false;
        }

        …
    }
```



Score display at top of window.

This shows a small score value at the top of the window while playing, which is currenty always zero. The idea is that each time a ball gets destroyed we score one point.

5.3 Hit Job

To bring everything together we introduce a `HitJob` that takes care of resolving all hits of both bullets and the player. To do this it needs access to the balls, bullets, health, score, world area, player position, bullet radius, and player radius. It will check every living ball, first checking whether any of the bullets hit it and if not whether the player hits it. This is done via a `CheckHit` method that takes the ball and a position and radius to compare with as parameters, then performs the usual point-inside-circle check.

If a bullet hits the ball then kill both the bullet and the ball and increment the score. If the player hits the ball then kill the ball, increment the score, and decrement the health. We cannot easily parallelize this job because we only want either a single bullet or only the player to hit a ball, so just run through it sequentially.

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

using static Unity.Mathematics.math;

[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
public struct HitJob : IJob
{
    public NativeList<BallState> balls;

    public NativeList<BulletState> bullets;

    public NativeReference<int> health, score;

    public Area2D worldArea;

    public float2 playerPosition;
```

```
    public float bulletRadius, playerRadius;

    public void Execute()
    {
        for (int i = 0; i < balls.Length; i++)
        {
            CheckBall(i);
        }
    }

    void CheckBall (int i)
    {
        BallState ball = balls[i];
        if (!ball.alive)
        {
            return;
        }

        for (int b = 0; b < bullets.Length; b++)
        {
            BulletState bullet = bullets[b];
            if (bullet.Alive && CheckHit(ball, bullet.position, bulletRadius))
            {
                bullet.timeRemaining = 0f;
                bullets[b] = bullet;
                ball.alive = false;
                balls[i] = ball;
                score.Value += 1;
                return;
            }
        }

        if (health.Value > 0 && CheckHit(ball, playerPosition, playerRadius))
        {
            ball.alive = false;
            balls[i] = ball;
            health.Value -= 1;
            score.Value += 1;
        }
    }

    bool CheckHit (BallState ball, float2 position, float radius)
    {
        float2 p = worldArea.Wrap(position - ball.position);
        float r = ball.radius + radius;
        if (dot(p, p) >= r)
        {
            return false;
        }

        return true;
    }
}
```

This jobs needs access to data stored in various places. To makes its configuration easy while hiding the details from `Game` we add a reference to it as an extra parameter to the `Initialize` methods. First, `BallManager` sets its balls.

```
    public void Initialize(Area2D worldArea, ref HitJob hitJob) {
        …
        hitJob.balls = states;
    }
```

Second, `BulletManager` sets both the bullets and their radius. Give it a configuration field for the bullet radius, set to 0.15.

```
    [SerializeField, Min(0f)]
    float speed = 12f, startLifetime = 1f, radius = 0.15f;

    …

    public void Initialize(Area2D worldArea, ref HitJob hitJob) {
        …
        hitJob.bullets = states;
        hitJob.bulletRadius = radius;
    }
```

Third, `Player` sets the health, score, and its radius. It doesn't set its position during initialization because it will change while playing.

```
    public void Initialize(ref HitJob hitJob) {
        …
        hitJob.health = health;
        hitJob.score = score;
        hitJob.playerRadius = radius;
    }
```

Now we can initialize a hit job in `Game` and schedule it after updating the balls and bullets and before resolving the balls, setting the player's position is we're playing.

```
    HitJob hitJob;

    void Awake()
    {
        …

        hitJob.worldArea = worldArea;
        ballManager.Initialize(worldArea, ref hitJob);
        bulletManager.Initialize(worldArea, ref hitJob);
        player.Initialize(ref hitJob);
        …
    }

    …

    void UpdateGameState(float dt)
    {
        if (isPlaying)
        {
            player.UpdateState(dt);
            hitJob.playerPosition = player.Position;
        }
        JobHandle handle = JobHandle.CombineDependencies(
            ballManager.UpdateBalls(dt),
            bulletManager.UpdateBullets(dt)
        );
        handle = hitJob.Schedule(handle);
        ballManager.ResolveBalls(player.Position, handle);
    }
```

From now on we can have dead balls, so `BallManager.UpdateVisualization` should remove them, just like `BulletManager` does.

```
		if (state.alive)
		{
			…
		}
		else
		{
			int lastIndex = states.Length - 1;
			states[i] = states[lastIndex];
			states.Length -= 1;

			visualizations[i].Despawn();
			visualizations[i] = visualizations[lastIndex];
			visualizations.RemoveAt(lastIndex);
			i -= 1;
		}
```

While playing we can now shoot balls for real and the player can also collide with balls, eventually causing the game to end. When the game ends balls will keep spawning and bouncing, just like before the first game begins. Remaining bullets could also cause more hits, but they won't show up in the score.

5.4 Splitting Balls

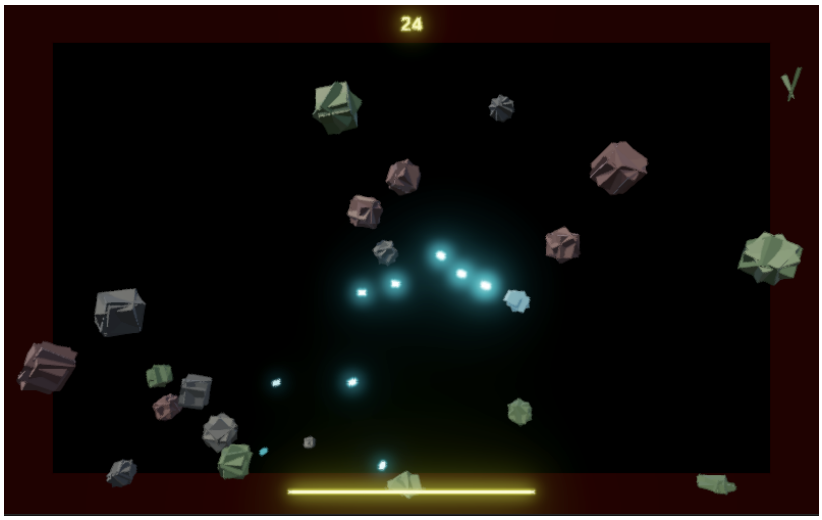
Because we kept the addition of a new ball state and spawning its visualization separate it is possible to spawn balls in jobs. We will use this to spawn ball fragments when a ball gets hit, adding two new balls of the same type with their stage reduced by one step, unless the stage was already zero. This is done in `HitJob.CheckHit`.

The fragments must not be spawned on top of each other, so we push them apart perpendicular to the hit direction. The displacement distance is equal to their radius scaled by a fragment separation factor that we will pass to the job.

```
    public float bulletRadius, playerRadius, fragmentSeparation;

    …

    bool CheckHit (BallState ball, float2 position, float radius)
    {
        …

        if (ball.stage > 0)
        {
            float2 direction = p * rsqrt(max(dot(p, p), 0.0001f));
            float2 displacement =
                fragmentSeparation * ball.radius * float2(direction.y, -direction.x);

            ball.stage -= 1;
            ball.radius = ball.targetRadius = BallState.radii[ball.stage];
            ball.mass = BallState.masses[ball.stage];
            float2 originalPosition = ball.position;
            ball.position = originalPosition + displacement;
            balls.Add(ball);
            ball.position = originalPosition - displacement;
            balls.Add(ball);
        }

        return true;
    }
```

Add a configuration field for the fragment separation to `BallManager` and pass it to the job. Set it to 0.6 so the fragments overlap a significant amount. Our physics engine will make them bounce apart.

```
    [SerializeField, Range(0.01f, 1f)]
    float fragmentSeparation = 0.6f;

    …

    public void Initialize(Area2D worldArea, ref HitJob hitJob) {
        …
        hitJob.balls = states;
        hitJob.fragmentSeparation = fragmentSeparation;
    }
```



Ball fragments.

The fragments bounce apart, but they also inherit the velocity of the original ball, which might've been moving toward the player. Let's also add an explosion strength to `HitJob` and use it to push the original ball away from either the bullet of the player. This is an instantaneous explosion force so time doesn't factor into it, only the ball's mass.

```
	public float bulletRadius, playerRadius, fragmentSeparation, explosionStrength;

	bool CheckHit (BallState ball, float2 position, float radius)
	{
		…
			ball.velocity -= direction * explosionStrength / ball.mass;
			ball.stage -= 1;
			ball.radius = ball.targetRadius = BallState.radii[ball.stage];
			ball.mass = BallState.masses[ball.stage];
		…
	}
```

As this force only applies to ball behavior configure it in `BallManager`, setting it to 2.

```
	[SerializeField, Min(0f)]
	float bounceStrength = 100f, explosionStrength = 2f;

	…

	public void Initialize(Area2D worldArea, ref HitJob hitJob) {
		…
		hitJob.fragmentSeparation = fragmentSeparation;
		hitJob.explosionStrength = explosionStrength;
	}
```

When balls explode their fragments are now pushed a bit away from the explosion point, which usually biases them to move away from the player or slows down their approach.

5.5 Explosions

As we treat hits as explosions we should show them as well, which is what we have a particle system for. First, add an indication whether a bullet exploded to `BulletState`. Also give it an `Explode` method that sets it and also kills the bullet by settings its remaining time to zero.

```
	public bool exploded;

	public bool Alive => timeRemaining > 0f;

	public void Explode()
	{
		exploded = true;
		timeRemaining = 0f;
	}
```

Only `HitJob` explodes bullets.

```
                //bullet.timeRemaining = 0f;
                bullet.Explode();
```

A configuration fields for the explosion particle system and how many particles to spawn to `BulletManager`, for which 50 is a decent amount. Emit those particles for each dead bullet that exploded and not just ran out of time.

```
    [SerializeField]
    ParticleSystem explosionParticleSystem;

    [SerializeField, Min(0)]
    int explosionParticleCount = 50;

    …

    public void UpdateVisualization(float dtExtrapolated)
    {
        for (int i = 0; i < visualizations.Count; i++)
        {
            BulletState state = states[i];
            if (state.Alive)
            {
                …
            }
            else
            {
                if (state.exploded)
                {
                    explosionParticleSystem.Emit(
                        new ParticleSystem.EmitParams
                        {
                            position = new Vector3(state.position.x, state.position.y),
                            applyShapeToPosition = true
                        },
                        explosionParticleCount
                    );
                }

                …
            }
        }
    }
```

Explosions.

Let's make `Player` emit explosions as well, spawning more particles when destroyed than when merely damaged, configured to 100 and 400. We could use a different particle system but I use the same one as for bullet explosions.

```
	[SerializeField]
	ParticleSystem explosionParticleSystem;

	[SerializeField, Min(1)]
	int hitParticleCount = 100, destructionParticleCount = 400;

	…

	public bool UpdateVisualization(float dtInterpolator)
	{
		…
		bool isDestroyed = lastCheckedHealth <= 0;
		explosionParticleSystem.Emit(
			new ParticleSystem.EmitParams
			{
				position = Position,
				applyShapeToPosition = true
			},
			isDestroyed ? destructionParticleCount : hitParticleCount
		);
		…
	}
```

5.6 Game Instructions

We wrap up this prototype by adding game instructions that also act as a game-over indicator. Create a text game object for this with position (0, 0, −2) and set its text:

```
Press SPACE to start.
Hold SPACE or mouse button to lock fire direction.
```

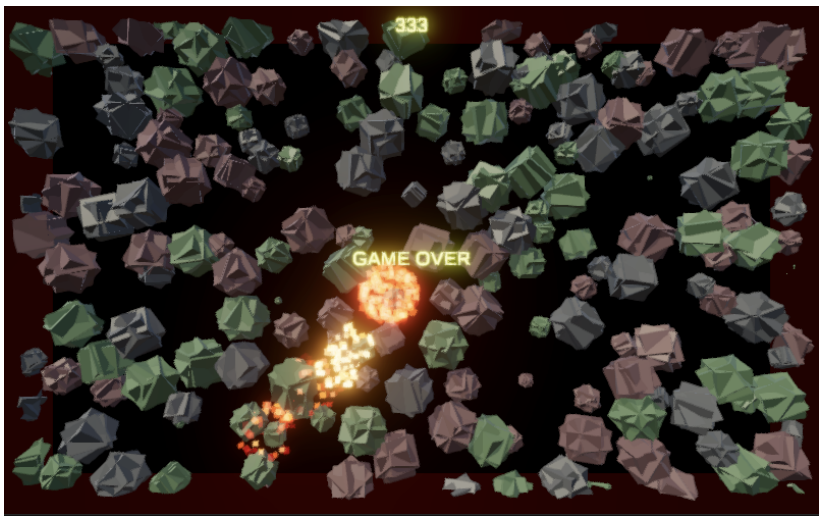
Give `Game` a configurable reference to it, deactivating it when starting a new game.

```
[SerializeField]
TextMeshPro instructionsDisplay;

…

void StartNewGame()
{
    …
    instructionsDisplay.SetActive(false);
}
```

Game instructions.

Use the same text to display `GAME OVER` when the player is destroyed.

```
        if (isPlaying && player.UpdateVisualization(dt / fixedDeltaTime))
        {
            isPlaying = false;
            instructionsDisplay.SetText("GAME OVER");
            instructionsDisplay.gameObject.SetActive(true);
        }
```



Game over.

Want to know when another tutorial gets released? Keep tabs on my Patreon page!

license

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick