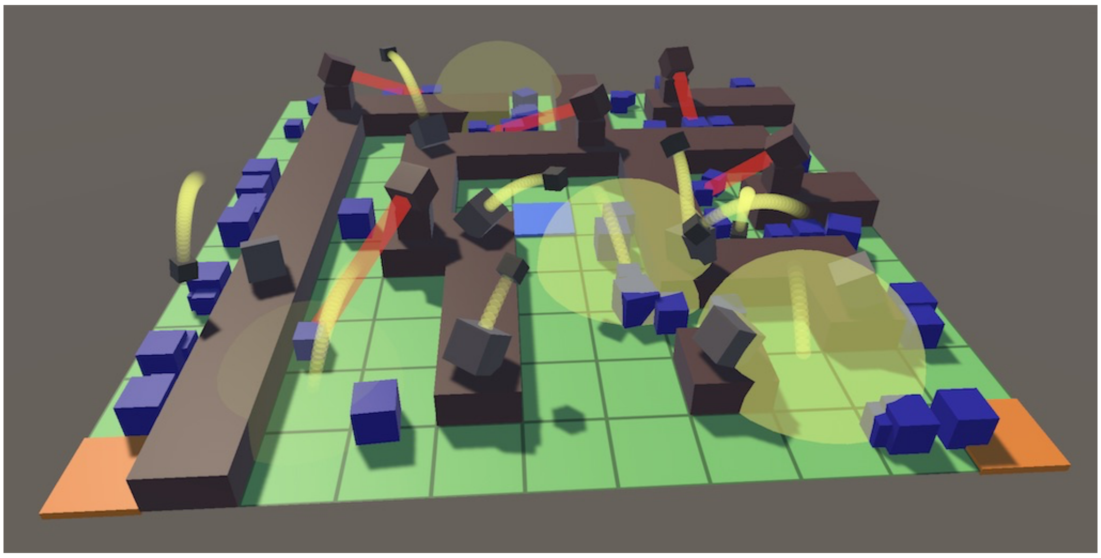
Ballistics Lobbing Explosives

Support more than one tower type.
Create a mortar tower.
Calculate parabolic trajectories.
Launch explosive shells.

This is the fourth installment of a tutorial series about creating a simple tower defense game. It adds a mortar tower that launches shells that detonate on impact.

This tutorial is made with Unity 2018.4.4f1.



Enemies getting shelled.

1 Tower Types

Laser aren't the only kind of weaponry that we could mount on a tower. In this tutorial we'll add a second tower type that lobs shells that explode on impact and damage all nearby enemies. To make that possible we have to support more than one type of tower.

1.1 Abstract Tower

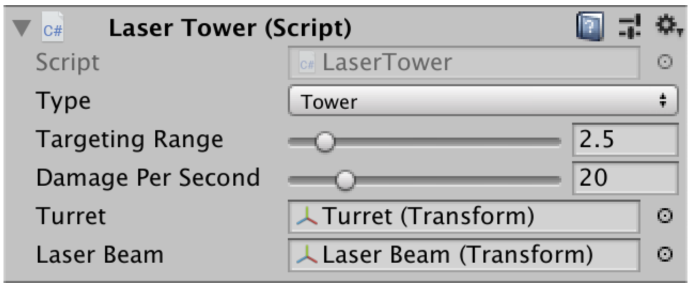
Acquiring and tracking a target is functionality that any tower could use, so we'll put that in an abstract base class for towers. We'll simply use `Tower` for that, but first duplicate it for later use as the concrete `LaserTower`. Then remove all code specific to the laser from `Tower`. A tower might not track a specific target, so also remove the `target` field and change `AcquireTarget` to use an output parameter and `TrackTarget` to use a reference parameter. Then remove the target visualization from `OnDrawGizmosSelected`, but keep the targeting range because that applies to all towers.

```
using UnityEngine;

public abstract class Tower : GameTileContent {

	const int enemyLayerMask = 1 << 9;

	static Collider[] targetsBuffer = new Collider[100];

	[SerializeField, Range(1.5f, 10.5f)]
	protected float targetingRange = 1.5f;

	~~//…~~

	protected bool AcquireTarget (out TargetPoint target) {
		…
	}

	protected bool TrackTarget (ref TargetPoint target) {
		…
	}

	void OnDrawGizmosSelected () {
		Gizmos.color = Color.yellow;
		Vector3 position = transform.localPosition;
		position.y += 0.01f;
		Gizmos.DrawWireSphere(position, targetingRange);
		~~//if (target != null) {~~
		~~//	Gizmos.DrawLine(position, target.Position);~~
		~~//}~~
	}
}
```

Adjust the duplicate class so it becomes `LaserTower` that extends `Tower` and uses the functionality of its base class, getting rid of the duplicate code.

```
using UnityEngine;

public class LaserTower : Tower {

    ~~//…~~

    [SerializeField, Range(1f, 100f)]
    float damagePerSecond = 10f;

    [SerializeField]
    Transform turret = default, laserBeam = default;

    TargetPoint target;

    Vector3 laserBeamScale;

    void Awake () {
        laserBeamScale = laserBeam.localScale;
    }

    public override void GameUpdate () {
        if (TrackTarget(ref target) || AcquireTarget(out target)) {
            Shoot();
        }
        else {
            laserBeam.localScale = Vector3.zero;
        }
    }

    void Shoot () {
        …
    }

    ~~//…~~
}
```

Then update the laser tower prefab so it uses the new specific component.



Laser Tower (Script)

Script	LaserTower
Type	Tower
Targeting Range	2.5
Damage Per Second	20
Turret	Turret (Transform)
Laser Beam	Laser Beam (Transform)

Laser tower component.

1.2 Fabricating a Specific Tower Type

To make it possible to select which kind of tower gets placed on the board we will introduce a `TowerType` enumeration, just like `GameTileContentType`. We'll support the existing laser type and the mortar type that we'll create later.

```
public enum TowerType {
	Laser, Mortar
}
```

As we'll create a class for each tower type, add an abstract getter property to `Tower` to indicate its type. This works the same as the shape behavior type from the Object Management series.

```
	public abstract TowerType TowerType { get; }
```

Override it in `LaserTower` to have it return the correct type.

```
	public override TowerType TowerType => TowerType.Laser;
```

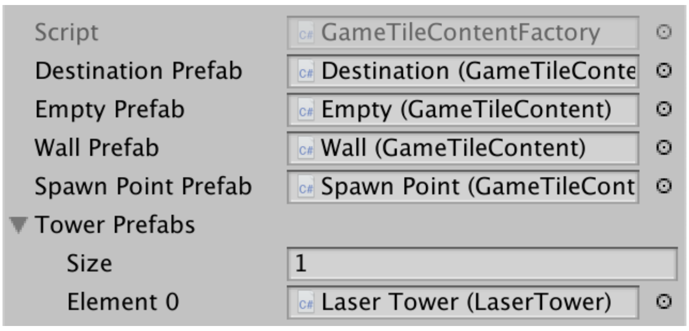
Next, adjust `GameTileContentFactory` so it can produce a tower of a desired type. We'll do that with a tower array and adding an alternative public `Get` method that has a `TowerType` parameter. We can use assertions to verify that the array is set up correctly. The other public `Get` method is now only for non-tower tile content.

```
	[SerializeField]
	Tower[] towerPrefabs = default;

	public GameTileContent Get (GameTileContentType type) {
		switch (type) {
			…
			//case GameTileContentType.Tower: return Get(towerPrefab);
		}
		Debug.Assert(false, "Unsupported non-tower type: " + type);
		return null;
	}

	public GameTileContent Get (TowerType type) {
		Debug.Assert((int)type < towerPrefabs.Length, "Unsupported tower type!");
		Tower prefab = towerPrefabs[(int)type];
		Debug.Assert(type == prefab.TowerType, "Tower prefab at wrong index!");
		return Get(prefab);
	}
```

It makes sense to return the most specific type, so ideally the new `Get` method's return type should be `Tower`. But the private `Get` method used to instantiate the prefab returns `GameTileContent`. We could either perform a cast here, or make the private `Get` method generic. Let's do the latter.

```
	public Tower Get (TowerType type) {
		…
	}

	T Get<T> (T prefab) where T : GameTileContent {
		T instance = CreateGameObjectInstance(prefab);
		instance.OriginFactory = this;
		return instance;
	}
```

As we only have the laser tower yet, make it the single element of the factory's tower array.



Script	GameTileContentFactory
Destination Prefab	Destination (GameTileConte
Empty Prefab	Empty (GameTileContent)
Wall Prefab	Wall (GameTileContent)
Spawn Point Prefab	Spawn Point (GameTileCont
▼ Tower Prefabs	
Size	1
Element 0	Laser Tower (LaserTower)

Tower prefabs array.

1.3 Spawning Specific Tower Types

To spawn a specific kind of tower, adjust `GameBoard.ToggleTower` so it requires a `TowerType` parameter and passes it on to the factory.

```
	public void ToggleTower (GameTile tile, TowerType towerType) {
		if (tile.Content.Type == GameTileContentType.Tower) {
			…
		}
		else if (tile.Content.Type == GameTileContentType.Empty) {
			tile.Content = contentFactory.Get(towerType);
			…
		}
		else if (tile.Content.Type == GameTileContentType.Wall) {
			tile.Content = contentFactory.Get(towerType);
			updatingContent.Add(tile.Content);
		}
	}
```

That introduces a new possibility: a tower gets toggled while one already exists, but they are of different types. Currently that just removes the existing tower, but it makes more sense that it gets replaced with the new type, so let's do that instead. As that keeps the tile occupied pathfinding isn't needed when this happens.

```
        if (tile.Content.Type == GameTileContentType.Tower) {
            updatingContent.Remove(tile.Content);
            if (((Tower)tile.Content).TowerType == towerType) {
                tile.Content = contentFactory.Get(GameTileContentType.Empty);
                FindPaths();
            }
            else {
                tile.Content = contentFactory.Get(towerType);
                updatingContent.Add(tile.Content);
            }
        }
```

Now `Game` has to keep track of what kind of tower should be toggled. We'll simply associate each tower type with a number. The laser tower is 1, which is also the default, while the mortar tower is 2. Pressing the number keys will select the corresponding tower type.

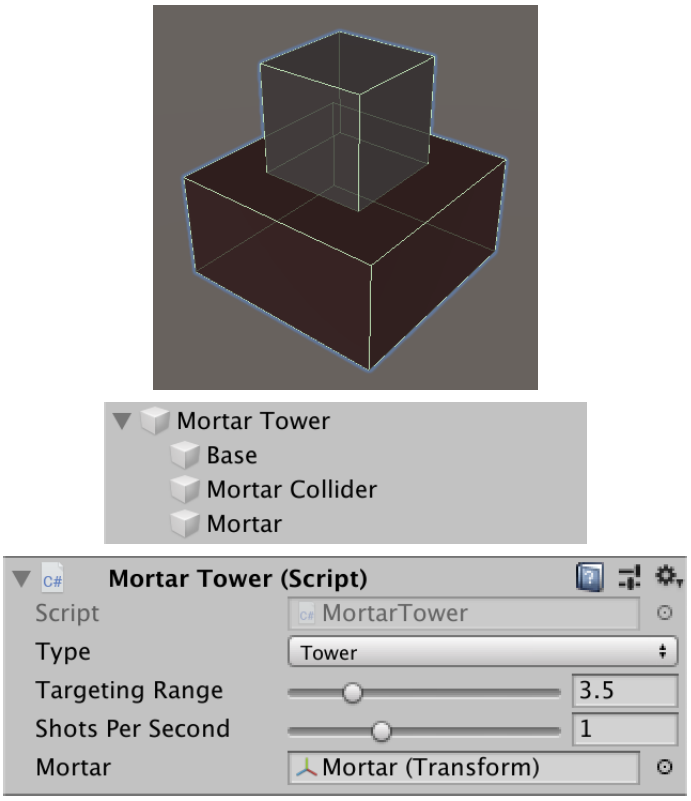
```
    TowerType selectedTowerType;

    …

    void Update () {
        …
        if (Input.GetKeyDown(KeyCode.G)) {
            board.ShowGrid = !board.ShowGrid;
        }

        if (Input.GetKeyDown(KeyCode.Alpha1)) {
            selectedTowerType = TowerType.Laser;
        }
        else if (Input.GetKeyDown(KeyCode.Alpha2)) {
            selectedTowerType = TowerType.Mortar;
        }

        …
    }

    …

    void HandleTouch () {
        GameTile tile = board.GetTile(TouchRay);
        if (tile != null) {
            if (Input.GetKey(KeyCode.LeftShift)) {
                board.ToggleTower(tile, selectedTowerType);
            }
            else {
                board.ToggleWall(tile);
            }
        }
    }
```

1.4 Mortar Tower

Placing a mortar tower currently fails, because we don't have a prefab for it yet. Begin by creating a minimal `MortarTower` type . Mortars have a fire rate, for which we can use a shots-per-second configuration field. Besides that we need a reference to the mortar so that we can aim it.

```
using UnityEngine;

public class MortarTower : Tower {

    [SerializeField, Range(0.5f, 2f)]
    float shotsPerSecond = 1f;

    [SerializeField]
    Transform mortar = default;

    public override TowerType TowerType => TowerType.Mortar;
}
```

Next, create a prefab for the mortar tower. You can do that by duplicating the laser tower prefab and replacing its tower component. Then get rid of the tower and laser beam objects. Rename the turret to mortar, move it down so it sits on top of the base, give it a slightly gray color, and hook it up. Again, we can keep the mortar's collider fixed, in this case by using a separate object with just the collider superimposed on the mortar's default orientation. I set its range to 3.5 and shots-per-second to 1.



```
▼ Mortar Tower
    Base
    Mortar Collider
    Mortar
```

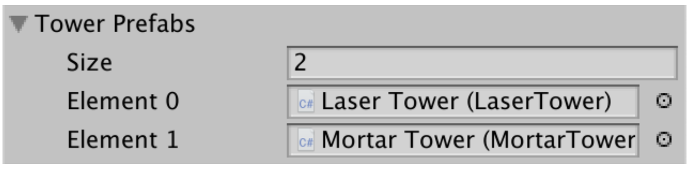
```
▼ Mortar Tower (Script)
  Script            MortarTower
  Type              Tower
  Targeting Range   3.5
  Shots Per Second  1
  Mortar            Mortar (Transform)
```

Mortar tower prefab.

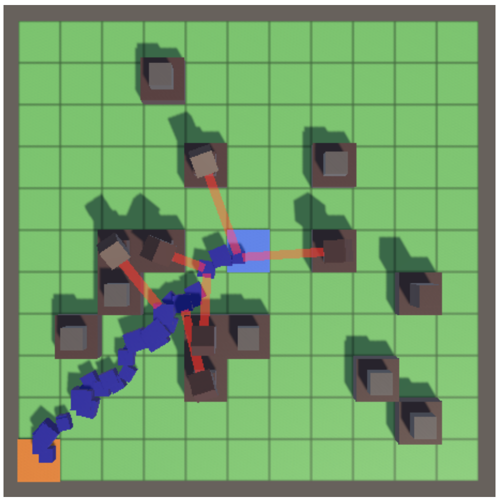
Why is it known as a mortar?

The earliest versions of the weapon were basically iron bowls, which looked like the mortars used to grind ingredients in with a pestle.

Add the mortar prefab to the factory's array, so it becomes possible to place mortar towers on the board. At this point they don't do anything yet though.



Tower Prefabs	
Size	2
Element 0	Laser Tower (LaserTower)
Element 1	Mortar Tower (MortarTower



Two tower types, one inactive

2 Calculating Trajectories

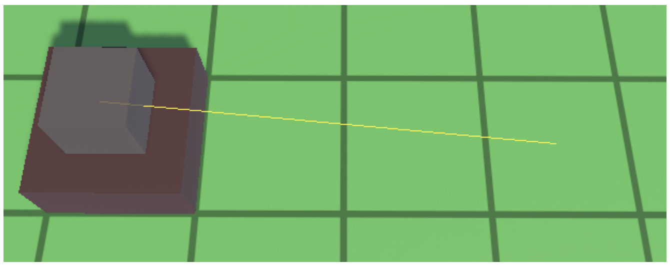
A mortar works by firing a projectile at an angle, so it gets lobbed over obstacles and hits its target from above. Typically, shells are used that detonate on impact or while they are still above their target. To keep it simple we'll always aim at the ground, so shells will detonate once their elevation has been reduced to zero.

2.1 Aiming Horizontally

To aim a mortar you have to both point it toward the target horizontally and then adjust its vertical orientation so the shell lands at the correct distance. We begin with the first step, initially using fixed relative points instead of moving targets to make it easy to verify that our calculations are correct.

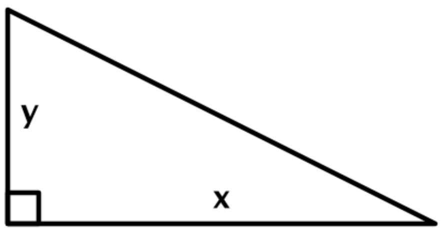
Add a `GameUpdate` method to `MortarTower` that always invokes a `Launch` method. Instead of launching an actual shell, for now we'll visualize the math involved. The launch point is the mortar's world position, which is a little above the ground. Place the target point three units further along the X axis, and set its Y component to zero as we always aim at the ground. Then show the points by drawing a yellow line between them, by invoking `Debug.DrawLine`. The line will be visible in the scene view for one frame, which is enough because we draw a new line every frame.

```
	public override void GameUpdate () {
		Launch();
	}

	public void Launch () {
		Vector3 launchPoint = mortar.position;
		Vector3 targetPoint = new Vector3(launchPoint.x + 3f, 0f, launchPoint.z);

		Debug.DrawLine(launchPoint, targetPoint, Color.yellow);
	}
```


Targeting a fixed relative point.

Using this line we can define a right triangle. Its top point sits at the mortar's position. Relative to the mortar that's $\begin{bmatrix}0\\0\end{bmatrix}$. The point below at the base of the tower is $\begin{bmatrix}0\\y\end{bmatrix}$ and the point at the target is $\begin{bmatrix}x\\y\end{bmatrix}$, where x is 3 and y is negative mortar's vertical position. We have to keep track of these two values.

```
		Vector3 launchPoint = mortar.position;
		Vector3 targetPoint = new Vector3(launchPoint.x + 3f, 0f, launchPoint.z);

		float x = 3f;
		float y = -launchPoint.y;
```



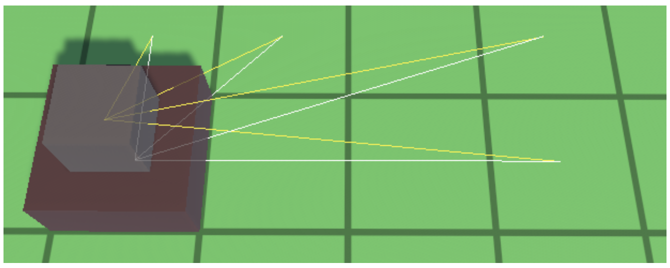
Targeting triangle.

In general the target can be anywhere within range, so the Z dimension also plays a role. However, the targeting triangle remains 2D, it just gets rotated around the Y axis. To illustrate this we'll add a relative offset vector parameter to `Launch` and invoke it with four XZ offsets: $\begin{bmatrix}3\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\end{bmatrix}$, $\begin{bmatrix}1\\1\end{bmatrix}$, and $\begin{bmatrix}3\\1\end{bmatrix}$. The the target point becomes equal to the launch point plus that offset, with its Y coordinate then set to zero.

```
	public override void GameUpdate () {
		Launch(new Vector3(3f, 0f, 0f));
		Launch(new Vector3(0f, 0f, 1f));
		Launch(new Vector3(1f, 0f, 1f));
		Launch(new Vector3(3f, 0f, 1f));
	}

	public void Launch (Vector3 offset) {
		Vector3 launchPoint = mortar.position;
		Vector3 targetPoint = launchPoint + offset;
		targetPoint.y = 0f;

		…
	}
```

Now the x of the targeting triangle is equal to the length of the 2D vector pointing from the base of the tower to the target point. Normalizing that vector also gives us an XZ direction vector that we can use to align the triangle. We can show that by drawing the bottom of the triangle with a white line, derived from the direction and x .

```
		Vector2 dir;
		dir.x = targetPoint.x - launchPoint.x;
		dir.y = targetPoint.z - launchPoint.z;
		float x = dir.magnitude;
		float y = -launchPoint.y;
		dir /= x;

		Debug.DrawLine(launchPoint, targetPoint, Color.yellow);
		Debug.DrawLine(
			new Vector3(launchPoint.x, 0.01f, launchPoint.z),
			new Vector3(
				launchPoint.x + dir.x * x, 0.01f, launchPoint.z + dir.y * x
			),
			Color.white
		);
```


Aligned targeting triangles.

2.2 Launch Angle

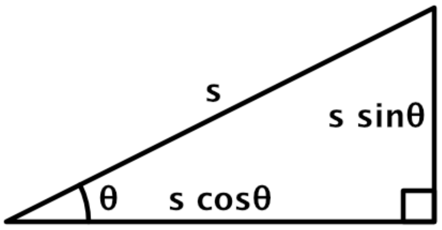
The next step is to figure out the angle at which the shell must be launched. We have to derive that from the physics of the shell's trajectory. We won't consider drag, wind, or any other kind of interference, only the launch velocity v and gravity $g = 9.81$.

The displacement d of the shell is aligned with the targeting triangle and can be described with two components. The horizontal displacement is straightforward $d_x = v_x t$ where t is the time since launch. The vertical component is similar but also subject to negative acceleration due to gravity, so $d_y = v_y t - \frac{gt^2}{2}$.

How does the displacement calculation work?

Velocity v defines distance per second, so multiplying a velocity and a duration t gives you a distance $d = vt$. When acceleration a is involved velocity isn't constant. Acceleration is velocity change per second, thus distance per squared second. The velocity at any given time is $v = at$. In our case we have constant acceleration $a = -g$, so we can halve that to get the average velocity and multiply that with the time to find the displacement $d = \frac{at^2}{2}$ caused by gravity.

We launch the shells with a fixed speed s that is independent of the launch angle θ (theta). So $v_x = s\cos\theta$ and $v_y = s\sin\theta$.



Launch velocity derivation.

Substituting, we arrive at $d_x = st\cos\theta$ and $d_y = st\sin\theta - \frac{gt^2}{2}$.

We launch the shell such that its flight time t is exactly long enough that it reaches its target. As the horizontal displacement is simplest, we can express the time using $t = \frac{d_x}{v_x}$. At the destination $d_x = x$, thus $t = \frac{x}{s\cos\theta}$. This means that $y = x\tan\theta - \frac{gx^2}{2s^2\cos^2\theta}$.

How does that derivation of y work?

$y = d_y = s\left(\frac{x}{s\cos\theta}\right)\sin\theta - \frac{g\left(\frac{x}{s\cos\theta}\right)^2}{2} = x\frac{\sin\theta}{\cos\theta} - \frac{gx^2}{2s^2\cos^2\theta}$ and $\tan\theta = \frac{\sin\theta}{\cos\theta}$.

Using that, we find $\tan\theta = \frac{s^2 \pm \sqrt{s^4 - g(gx^2 + 2ys^2)}}{gx}$.

How does that derivation of $\tan\theta$ work?

First, we use the trigonometric identities $\sec\theta = \frac{1}{\cos\theta}$ and $1+\tan^2\theta = \sec^2\theta$ to arrive at $y = x\tan\theta - \frac{gx^2}{2s^2}\left(1+\tan^2\theta\right) = -\frac{gx^2}{2s^2}\tan^2\theta + x\tan\theta - \frac{gx^2}{2s^2}$.

That's an expression of the form $au^2+bu+c=0$, with $u=\tan\theta$, $a=-\frac{gx^2}{2s^2}$, $b=x$, and $c=a-y$.

We can solve that with the quadratic formula $u = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$.

Substitution creates a mess, but we can simplify it by multiplying with $m=\frac{s^2}{x}$ such that we get $\tan\theta = \frac{-mb\pm m\sqrt{r}}{2ma}$ where $r=b^2-4ac$.

That leads to $\tan\theta = \frac{s^2\pm\sqrt{m^2r}}{gx}$.

Finally,

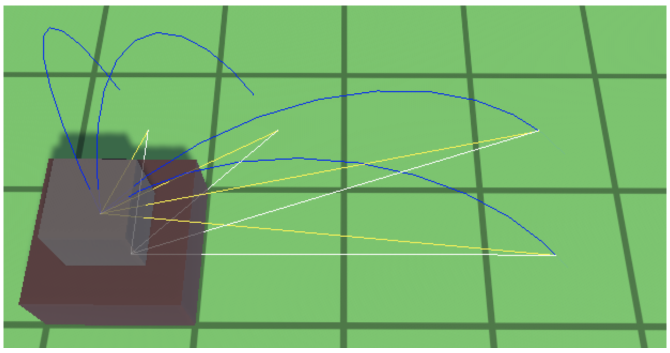
$$m^2r = \left(\frac{s^4}{x^2}\right)r = s^4+2gs^2c = s^4-g^2x^2-2gys^2 = s^4-g\left(gx^2+2ys^2\right).$$

There are two possible launch angles because it's possible to aim either low or high. A low trajectory is faster as it's closer to a straight line to the target. But a high trajectory is more visually interesting so we'll go for that. This means that we only need to use the largest solution, $\tan\theta = \frac{s^2+\sqrt{s^4-g(gx^2+2ys^2)}}{gx}$. Calculate that and also $\cos\theta$ and $\sin\theta$, because we need those to derive the launch velocity vector. We have to convert $\tan\theta$ to a radian angle for that, via `Mathf.Atan`. Let's initially use a fixed launch speed of 5.

```
		float x = dir.magnitude;
		float y = -launchPoint.y;
		dir /= x;

		float g = 9.81f;
		float s = 5f;
		float s2 = s * s;

		float r = s2 * s2 - g * (g * x * x + 2f * y * s2);
		float tanTheta = (s2 + Mathf.Sqrt(r)) / (g * x);
		float cosTheta = Mathf.Cos(Mathf.Atan(tanTheta));
		float sinTheta = cosTheta * tanTheta;
```

Let's visualize the trajectory by drawing ten blue line segments covering the first second of the flight.

```
		float sinTheta = cosTheta * tanTheta;

		Vector3 prev = launchPoint, next;
		for (int i = 1; i <= 10; i++) {
			float t = i / 10f;
			float dx = s * cosTheta * t;
			float dy = s * sinTheta * t - 0.5f * g * t * t;
			next = launchPoint + new Vector3(dir.x * dx, dy, dir.y * dx);
			Debug.DrawLine(prev, next, Color.blue);
			prev = next;
		}
```



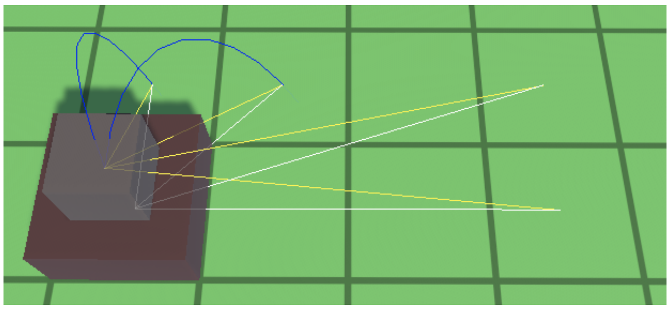
Parabolic flight trajectories up to one second.

The two farthest points can be reached in less than a second, so we see their entire trajectories and the lines continue a bit below the ground. The other two points require larger launch angles, which leads to longer trajectories that take more than a second to traverse.

2.3 Launch Speed

If we want to reach the nearest two points within a second then we'd have to reduce the launch speed. Let's set it to 4.

```
		float s = 4f;
```


Launch speed reduced to 4.

Their trajectories are now complete, but the other two have disappeared. That happened because the launch speed is now insufficient to reach those points. There are no solutions for $\tan\theta$ in those cases, which means that we end up with the square root of a negative number, leading to not-a-number values which cause our lines to disappear. We can detect that by checking whether r is negative.

```
		float r = s2 * s2 - g * (g * x * x + 2f * y * s2);
		Debug.Assert(r >= 0f, "Launch velocity insufficient for range!");
```

We can avoid this situation by using a high enough launch speed. But if it becomes too high then nearby targets would require very high trajectories and flight times to hit, so we want to keep the speed as low as possible. Our launch speed should be just enough to hit a target at maximum range.

At maximum range, $r = 0$ so there is only one solution for $\tan\theta$, which is a low trajectory. This means that we know the required launch speed $s = \sqrt{g\left(y + \sqrt{x^2 + y^2}\right)}$.

How does that derivation of s work?

We need to solve $s^4 - g(gx^2 + 2ys^2) = s^4 - 2gys^2 - g^2x^2 = 0$ for s.

That's an expression of the form $au^2 + bu + c = 0$, with $u = s^2$, $a = 1$, $b = -2gy$, and $c = -g^2x^2$.

We can solve that with the simplified quadratic formula $u = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$.

Substitution gives us $s^2 = \frac{2gy \pm \sqrt{4g^2y^2 + 4g^2x^2}}{2} = gy \pm g\sqrt{x^2 + y^2}$.

We require a positive solution, so we arrive at $s^2 = g\left(y + \sqrt{x^2 + y^2}\right)$.

We only need to figure out the required speed when the mortar awakens, or when we adjust its range while in play mode. So keep track of it with a field and calculate it in `Awake` and `OnValidate`.

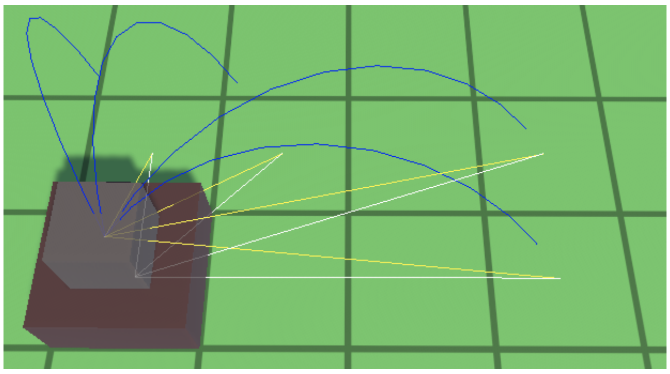
```
	float launchSpeed;

	void Awake () {
		OnValidate();
	}

	void OnValidate () {
		float x = targetingRange;
		float y = -mortar.position.y;
		launchSpeed = Mathf.Sqrt(9.81f * (y + Mathf.Sqrt(x * x + y * y)));
	}
```

However, due to floating-point precision a target very close to the maximum range might fail. So we should add a tiny amount to the range when calculating the required speed. Also, the enemy collider radius effectively extends the maximum tower range. We set it to 0.125 subject to at most doubling due to enemy scale, so increase the effective range by a further 0.25, to something like 0.25001.

```
		float x = targetingRange + 0.25001f;
```

Finally, use the derived launch speed in `Launch`.

```
		float s = launchSpeed;
```



Using derived speed for targeting range 3.5.

2.4 Barrage

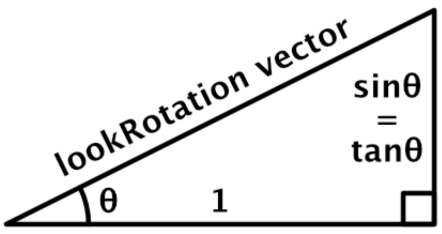
Now that our trajectory calculation is correct we can get rid of the relative test targets. Instead, `Launch` should be provided with a target point.

```
	public void Launch (TargetPoint target) {
		Vector3 launchPoint = mortar.position;
		Vector3 targetPoint = target.Position;
		targetPoint.y = 0f;

		…
	}
```

And we're not firing every frame. Keep track of the launch progress just like the spawn progress for enemies and acquire a random target when it's time to launch in `GameUpdate`. But there might be no target available at that time. In that case we keep the launch progress, but don't let it accumulate further. In fact, to prevent an infinite loop we should set it to slightly less than 1.

```
	float launchProgress;

	…

	public override void GameUpdate () {
		launchProgress += shotsPerSecond * Time.deltaTime;
		while (launchProgress >= 1f) {
			if (AcquireTarget(out TargetPoint target)) {
				Launch(target);
				launchProgress -= 1f;
			}
			else {
				launchProgress = 0.999f;
			}
		}
	}
```

We don't track targets in between launches, but we have to properly align the mortar when firing. We can use the horizontal launch direction vector to horizontally rotate the mortar by using `Quaternion.LookRotation`. We also have to incorporate the launch angle, by using $\tan\theta$ for the Y component of the direction vector. That works because the horizontal direction has a length of 1, thus $\tan\theta = \sin\theta$.

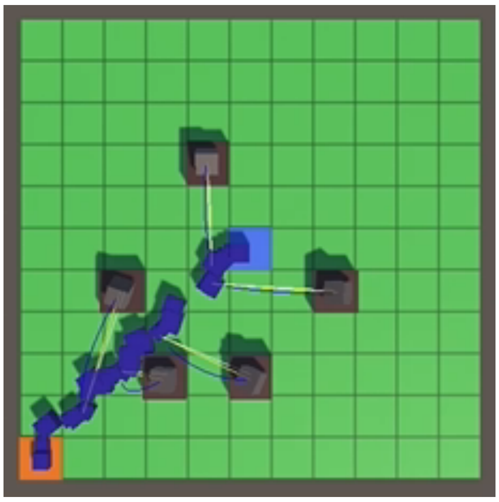


Decomposition of look-rotation vector.

```
        float tanTheta = (s2 + Mathf.Sqrt(r)) / (g * x);
        float cosTheta = Mathf.Cos(Mathf.Atan(tanTheta));
        float sinTheta = cosTheta * tanTheta;

        mortar.localRotation =
            Quaternion.LookRotation(new Vector3(dir.x, tanTheta, dir.y));
```

To still be able to see the launch trajectories, we can add a parameter to `Debug.DrawLine` to give them a duration.

```
        Vector3 prev = launchPoint, next;
        for (int i = 1; i <= 10; i++) {
            …
            Debug.DrawLine(prev, next, Color.blue, 1f);
            prev = next;
        }

        Debug.DrawLine(launchPoint, targetPoint, Color.yellow, 1f);
        Debug.DrawLine(
            …
            Color.white, 1f
        );
```


Barrage targeting.

3 Shells

The point of calculating trajectories is that we now know how to launch shells. The next step is to create and launch them.

3.1 War Factory

We need a factory to create instances of shell objects. Once in the air, shells exist on their own and no longer depend on the mortar that launched them. So mortar tower's shouldn't manage them and the game tile content factory also isn't a good fit. Let's create a new factory for everything related to weaponry, naming it the war factory. First, create an abstract `WarEntity` with an appropriate `OriginFactory` property and `Recycle` method.

```
using UnityEngine;

public abstract class WarEntity : MonoBehaviour {

	WarFactory originFactory;

	public WarFactory OriginFactory {
		get => originFactory;
		set {
			Debug.Assert(originFactory == null, "Redefined origin factory!");
			originFactory = value;
		}
	}

	public void Recycle () {
		originFactory.Reclaim(this);
	}
}
```

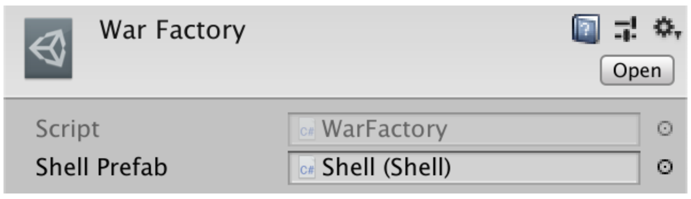
Then create a concrete `Shell` war entity of our shells.

```
using UnityEngine;

public class Shell : WarEntity { }
```

Followed by `WarFactory` itself, which can deliver a shell via a public getter property.

```
using UnityEngine;

[CreateAssetMenu]
public class WarFactory : GameObjectFactory {

    [SerializeField]
    Shell shellPrefab = default;

    public Shell Shell => Get(shellPrefab);

    T Get<T> (T prefab) where T : WarEntity {
        T instance = CreateGameObjectInstance(prefab);
        instance.OriginFactory = this;
        return instance;
    }

    public void Reclaim (WarEntity entity) {
        Debug.Assert(entity.OriginFactory == this, "Wrong factory reclaimed!");
        Destroy(entity.gameObject);
    }
}
```

Create a prefab for the shell. I simply used a cube with uniform scale 0.25 and a dark material, plus the `Shell` component. Then create the war factory asset and assign the shell prefab to it.



Script: WarFactory
Shell Prefab: Shell (Shell)

War factory.

3.2 Game Behavior

To move the shells we'll have to update them. We can use the same approach that `Game` uses to update the enemies. In fact, we can make this approach generic by introducing an abstract `GameBehavior` component that extends `MonoBehaviour` and adds a virtual `GameUpdate` method.

```
using UnityEngine;

public abstract class GameBehavior : MonoBehaviour {

    public virtual bool GameUpdate () => true;
}
```

Then refactor `EnemyCollection`, turning it into `GameBehaviorCollection`.

```
public class GameBehaviorCollection {

    List<GameBehavior> behaviors = new List<GameBehavior>();

    public void Add (GameBehavior behavior) {
        behaviors.Add(behavior);
    }

    public void GameUpdate () {
        for (int i = 0; i < behaviors.Count; i++) {
            if (!behaviors[i].GameUpdate()) {
                int lastIndex = behaviors.Count - 1;
                behaviors[i] = behaviors[lastIndex];
                behaviors.RemoveAt(lastIndex);
                i -= 1;
            }
        }
    }
}
```

Make `WarEntity` extend `GameBehavior` instead of `MonoBehavior`.

```
public abstract class WarEntity : GameBehavior { … }
```

And do the same for `Enemy`, now overriding the `GameUpdate` method.

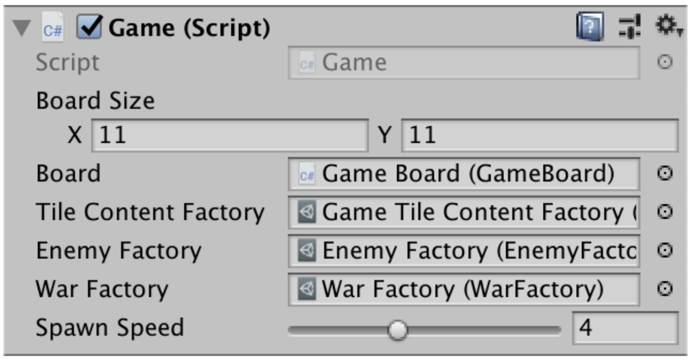
```
public class Enemy : GameBehavior {

    …

    public override bool GameUpdate () { … }

    …
}
```

From now on `Game` has to keep track of two collections, one for enemies and another for non-enemies. The non-enemies should be updated after everything else.

```
    GameBehaviorCollection enemies = new GameBehaviorCollection();
    GameBehaviorCollection nonEnemies = new GameBehaviorCollection();

    …

    void Update () {
        …
        enemies.GameUpdate();
        Physics.SyncTransforms();
        board.GameUpdate();
        nonEnemies.GameUpdate();
    }
```

The last step to get updating shells is to somehow add them to the collection of non-enemies. Let's do that by having `Game` function as a static facade for the war factory, so shells can be spawned by invoking `Game.SpawnShell()`. To make that work `Game` needs a reference to the war factory and has to keep track of its own instance.

```
	[SerializeField]
	WarFactory warFactory = default;

	…

	static Game instance;

	public static Shell SpawnShell () {
		Shell shell = instance.warFactory.Shell;
		instance.nonEnemies.Add(shell);
		return shell;
	}

	void OnEnable () {
		instance = this;
	}
```



```
Game (Script)
Script                 Game
Board Size
    X 11               Y 11
Board                  Game Board (GameBoard)
Tile Content Factory   Game Tile Content Factory (
Enemy Factory          Enemy Factory (EnemyFactc
War Factory            War Factory (WarFactory)
Spawn Speed                                     4
```

Game with war factory.

Is a static facade a good approach?

It's a convenient one for simple things that can be spawned anywhere by potentially lots of things, like shells.

3.3 Launching the Shell

Once a shell has been spawned, it must fly along its trajectory until it reaches its target. To make that possible, add an `Initialize` method to `Shell` and use it to set up its launch point, target point, and launch velocity.

```
	Vector3 launchPoint, targetPoint, launchVelocity;

	public void Initialize (
		Vector3 launchPoint, Vector3 targetPoint, Vector3 launchVelocity
	) {
		this.launchPoint = launchPoint;
		this.targetPoint = targetPoint;
		this.launchVelocity = launchVelocity;
	}
```

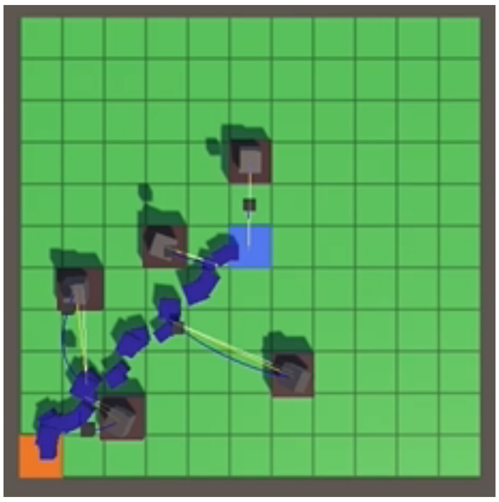
Now we can spawn a shell in `MortarTower.Launch` and send it on its way.

```
		mortar.localRotation =
			Quaternion.LookRotation(new Vector3(dir.x, tanTheta, dir.y));

		Game.SpawnShell().Initialize(
			launchPoint, targetPoint,
			new Vector3(s * cosTheta * dir.x, s * sinTheta, s * cosTheta * dir.y)
		);
```

3.4 Shell Motion

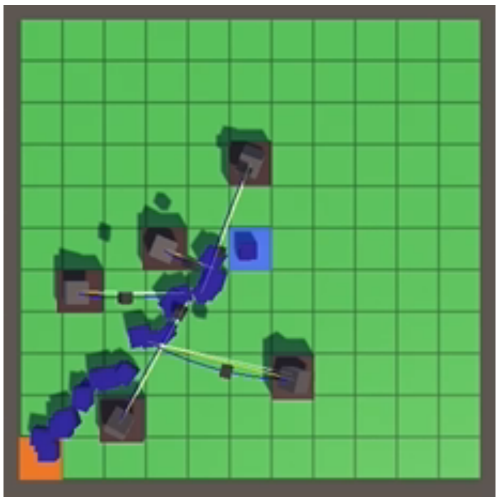
To make `Shell` move we have to keep track of its age, which is the time since launch. We can then calculate its position in `GameUpdate`. We always do this relative to its launch point, so it perfectly follows its trajectory, regardless of the update frequency.

```
	float age;

	…

	public override bool GameUpdate () {
		age += Time.deltaTime;
		Vector3 p = launchPoint + launchVelocity * age;
		p.y -= 0.5f * 9.81f * age * age;
		transform.localPosition = p;
		return true;
	}
```


Lobbing Shells.

To also align the shells with their trajectory we have to make them look along the derivate vector, which is simply the velocity at that time.

```
	public override bool GameUpdate () {
		…

		Vector3 d = launchVelocity;
		d.y -= 9.81f * age;
		transform.localRotation = Quaternion.LookRotation(d);
		return true;
	}
```


Rotating Shells.

3.5 Cleaning Up

Now that it is clear that shells are flying as they should, we can remove the trajectory visualization from `MortarTower.Launch`.

```
	public void Launch (TargetPoint target) {
		…

		Game.SpawnShell().Initialize(
			launchPoint, targetPoint,
			new Vector3(s * cosTheta * dir.x, s * sinTheta, s * cosTheta * dir.y)
		);

		~~//Vector3 prev = launchPoint, next;~~
		~~//…~~
	}
```

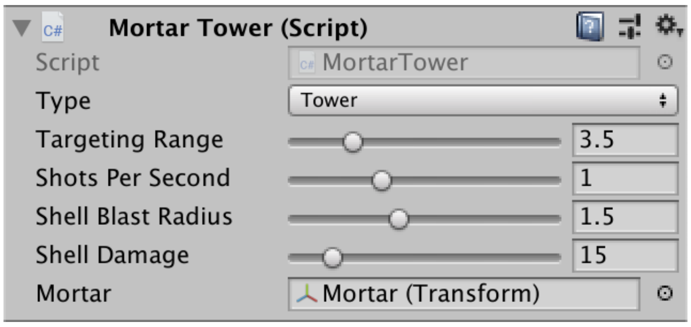
Also, we have to make sure that shells cease to exist once they've reached their target. As we always aim at the ground, we can do that by checking whether the vertical position has been reduced to zero or less in `Shell.GameUpdate`. We can do that directly after calculating it, before adjusting the shell's position and rotation.

```
	public override bool GameUpdate () {
		age += Time.deltaTime;
		Vector3 p = launchPoint + launchVelocity * age;
		p.y -= 0.5f * 9.81f * age * age;

		if (p.y <= 0f) {
			OriginFactory.Reclaim(this);
			return false;
		}

		transform.localPosition = p;
		…
	}
```

3.6 Detonation

We're launching shells because they're filled with explosives. When a shell reaches it target, it should detonate and damage all enemies in the blast zone. The blast radius and amount of damage depends on what shells are fired by the mortar, so add configuration options for them to `MortarTower`.

```
	[SerializeField, Range(0.5f, 3f)]
	float shellBlastRadius = 1f;

	[SerializeField, Range(1f, 100f)]
	float shellDamage = 10f;
```


Mortar Tower (Script)

Script	MortarTower
Type	Tower
Targeting Range	3.5
Shots Per Second	1
Shell Blast Radius	1.5
Shell Damage	15
Mortar	Mortar (Transform)

Shell blast radius 1.5 and damage 15.

This configuration only matters when the shell explodes, so must be added to `Shell` and its `Initialize` method as well.

```
	float age, blastRadius, damage;

	public void Initialize (
		Vector3 launchPoint, Vector3 targetPoint, Vector3 launchVelocity,
		float blastRadius, float damage
	) {
		…
		this.blastRadius = blastRadius;
		this.damage = damage;
	}
```

`MortarTower` only has to pass the data to the shell, after spawning it.

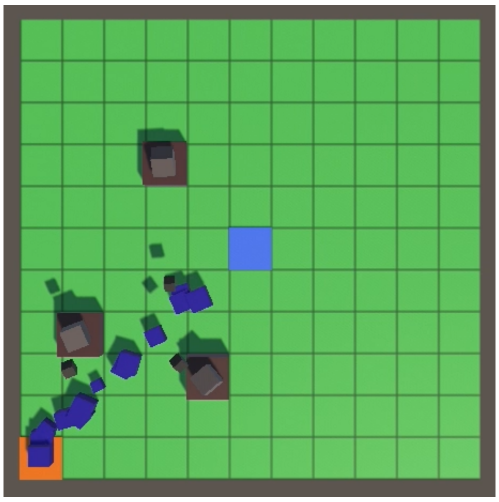
```
		Game.SpawnShell().Initialize(
			launchPoint, targetPoint,
			new Vector3(s * cosTheta * dir.x, s * sinTheta, s * cosTheta * dir.y),
			shellBlastRadius, shellDamage
		);
```

To hit enemies in range, the shell must acquire targets. We already have code for that, but it's in `Tower`. As it is useful for anything that needs a target, copy that functionality to `TargetPoint` and make it statically available. Add a method to fill the buffer, a property to get the buffered count, and a method to get a buffered target.

```
	const int enemyLayerMask = 1 << 9;

	static Collider[] buffer = new Collider[100];

	public static int BufferedCount { get; private set; }

	public static bool FillBuffer (Vector3 position, float range) {
		Vector3 top = position;
		top.y += 3f;
		BufferedCount = Physics.OverlapCapsuleNonAlloc(
			position, top, range, buffer, enemyLayerMask
		);
		return BufferedCount > 0;
	}

	public static TargetPoint GetBuffered (int index) {
		var target = buffer[index].GetComponent<TargetPoint>();
		Debug.Assert(target != null, "Targeted non-enemy!", buffer[0]);
		return target;
	}
```

Now we can acquire all targets in range—up to the maximum buffer size—and damage them when `Shell` detonates.

```
		if (p.y <= 0f) {
			TargetPoint.FillBuffer(targetPoint, blastRadius);
			for (int i = 0; i < TargetPoint.BufferedCount; i++) {
				TargetPoint.GetBuffered(i).Enemy.ApplyDamage(damage);
			}
			OriginFactory.Reclaim(this);
			return false;
		}
```



Detonating Shells.

We can also add a static property to `TargetPoint` to conveniently get a random buffered target.

```
	public static TargetPoint RandomBuffered =>
		GetBuffered(Random.Range(0, BufferedCount));
```

That allows us to simplify `Tower`, as it can now rely on `TargetPoint` to find a random target.

```
	~~//const int enemyLayerMask = 1 << 9;~~

	~~//static Collider[] targetsBuffer = new Collider[100];~~

	…

	protected bool AcquireTarget (out TargetPoint target) {
		~~//Vector3 a = transform.localPosition;~~
		~~//…~~
		~~//if (hits > 0) {~~
		~~//	…~~
		~~//}~~
		if (TargetPoint.FillBuffer(transform.localPosition, targetingRange)) {
			target = TargetPoint.RandomBuffered;
			return true;
		}
		target = null;
		return false;
	}
```

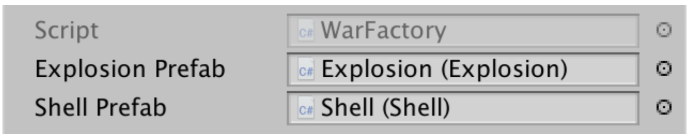
3.7 Explosions

Everything works, but it doesn't look very convincing yet. We can approve that by adding a visualization of the explosion blast when a shell detonates. Besides looking more interesting, it also provided useful visual feedback to the player. We'll do this by creating an explosion prefab that's just like the laser beam, except it's a sphere, it's more transparent, and has a brighter color. Give it a new `Explosion` war entity component with a configurable duration, with half a second as a reasonable default that is short but long enough to register clearly. Give it an `Initialize` method to set its position and blast radius. We have to double the radius when setting the scale, because the sphere mesh's radius is 0.5. This is also a good place to apply the damage to all enemies in range, so it should have a parameter for damage as well. Besides that, it needs a `GameUpdate` method that simply checks whether its time is up.

```
using UnityEngine;

public class Explosion : WarEntity {

    [SerializeField, Range(0f, 1f)]
    float duration = 0.5f;

    float age;

    public void Initialize (Vector3 position, float blastRadius, float damage) {
        TargetPoint.FillBuffer(position, blastRadius);
        for (int i = 0; i < TargetPoint.BufferedCount; i++) {
            TargetPoint.GetBuffered(i).Enemy.ApplyDamage(damage);
        }
        transform.localPosition = position;
        transform.localScale = Vector3.one * (2f * blastRadius);
    }

    public override bool GameUpdate () {
        age += Time.deltaTime;
        if (age >= duration) {
            OriginFactory.Reclaim(this);
            return false;
        }
        return true;
    }
}
```

Add the explosion to `WarFactory`.

```
    [SerializeField]
    Explosion explosionPrefab = default;

    [SerializeField]
    Shell shellPrefab = default;

    public Explosion Explosion => Get(explosionPrefab);

    public Shell Shell => Get(shellPrefab);
```



Script	WarFactory
Explosion Prefab	Explosion (Explosion)
Shell Prefab	Shell (Shell)

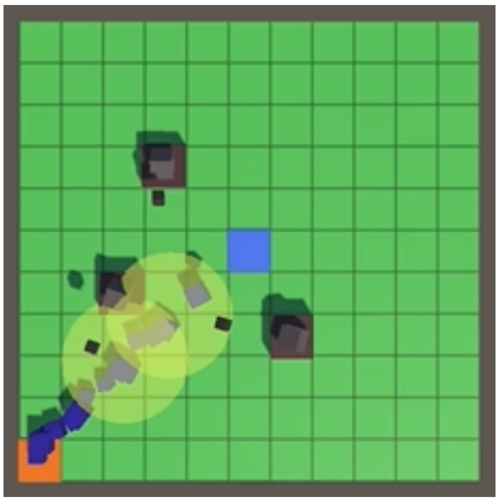
War factory with explosion.

And add a facade method for it to `Game`.

```
    public static Explosion SpawnExplosion () {
        Explosion explosion = instance.warFactory.Explosion;
        instance.nonEnemies.Add(explosion);
        return explosion;
    }
```

Now `Shell` can spawn and initialize an explosion when it reaches its target. The damage is taken care of by the explosion.

```
		if (p.y <= 0f) {
			~~//TargetPoint.FillBuffer(targetPoint, blastRadius);~~
			~~//for (int i = 0; i < TargetPoint.BufferedCount; i++) {~~
			~~//	TargetPoint.GetBuffered(i).Enemy.ApplyDamage(damage);~~
			~~//}~~
			Game.SpawnExplosion().Initialize(targetPoint, blastRadius, damage);
			OriginFactory.Reclaim(this);
			return false;
		}
```

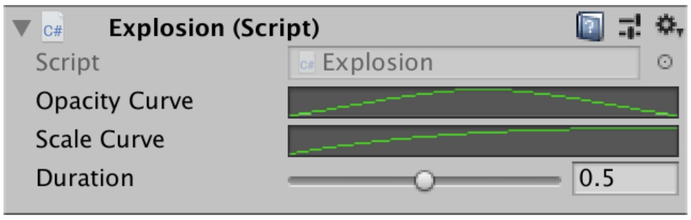

Shell Explosions.

3.8 Smoother Explosions

Using unchanging spheres for explosions looks bad. We can improve that by animating their opacity and scale. We could use simple formula for that, but let's use animation curves for both so it's easier to tweak them. Add two `AnimationCurve` configuration fields to `Explosion` for that. We'll use the curves to configure the values during the explosion's lifetime, with time 1 representing the end of the explosion, regardless of its actual duration. The same goes for the scale and blast radius. That makes it easier to configure.

```
	[SerializeField]
	AnimationCurve opacityCurve = default;

	[SerializeField]
	AnimationCurve scaleCurve = default;
```

I made opacity start and end at zero, smoothly scaling up to 0.3 at the halfway point. I made the scale start at 0.7 and quickly increase and then slowly approach 1.

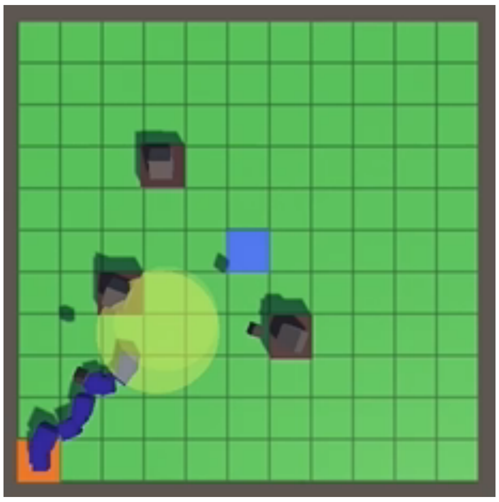

Explosion (Script)

Script	Explosion
Opacity Curve	
Scale Curve	
Duration	0.5

Explosion curves.

We'll use a material property block to set the material's color, which is black with variable opacity. The scale is now set in `GameUpdate`, but we have to keep track of the radius with a field. We can apply the scale doubling once in `Initialize`. The curve values are found by invoking `Evaluate` on them with the current age divided by the explosion duration as an argument.

```
	static int colorPropertyID = Shader.PropertyToID("_Color");

	static MaterialPropertyBlock propertyBlock;

	…

	float scale;

	MeshRenderer meshRenderer;

	void Awake () {
		meshRenderer = GetComponent<MeshRenderer>();
		Debug.Assert(meshRenderer != null, "Explosion without renderer!");
	}

	public void Initialize (Vector3 position, float blastRadius, float damage) {
		…
		transform.localPosition = position;
		~~//transform.localScale = Vector3.one * (2f * blastRadius);~~
		scale = 2f * blastRadius;
	}

	public override bool GameUpdate () {
		…

		if (propertyBlock == null) {
			propertyBlock = new MaterialPropertyBlock();
		}
		float t = age / duration;
		Color c = Color.clear;
		c.a = opacityCurve.Evaluate(t);
		propertyBlock.SetColor(colorPropertyID, c);
		meshRenderer.SetPropertyBlock(propertyBlock);
		transform.localScale = Vector3.one * (scale * scaleCurve.Evaluate(t));
		return true;
	}
```


Animated Explosions.

3.9 Tracer Shells

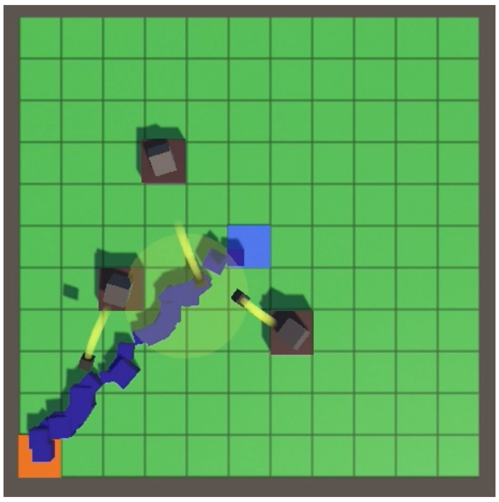
Because shells are small and have a relatively high velocity it can be hard to see them. And when looking at a screenshot of a single frame the trajectories aren't clear at all. We could make that more obvious by adding a trail effect to the shells. That isn't realistic for normal shells, but we can declare that they are tracer shells. Such projectiles are specifically made to leave a bright trail for the purpose of making their trajectories visible.

There are various ways to create trails, but we'll use a very simple approach here. We repurpose explosions, having `Shell` spawn a small one every frame. These are explosions that don't do any damage, so it would be a waste to acquire targets. Have `Explosion` support such cosmetic use by only applying damage that is larger than zero, then make the damage parameter of `Initialize` optional.

```
	public void Initialize (
		Vector3 position, float blastRadius, float damage = 0f
	) {
		if (damage > 0f) {
			TargetPoint.FillBuffer(position, blastRadius);
			for (int i = 0; i < TargetPoint.BufferedCount; i++) {
				TargetPoint.GetBuffered(i).Enemy.ApplyDamage(damage);
			}
		}
		transform.localPosition = position;
		radius = 2f * blastRadius;
	}
```

Spawn an explosion at the end up `Shell.GameUpdate` with a small radius, say 0.1, to turn them into tracer shells. Note that this approach spawns an explosion per frame, so is frame-rate dependent, which is fine for this simple effect.

```
	public override bool GameUpdate () {
		…

		Game.SpawnExplosion().Initialize(p, 0.1f);
		return true;
	}
```


Tracer Shells.

The next tutorial is Scenarios.

repository