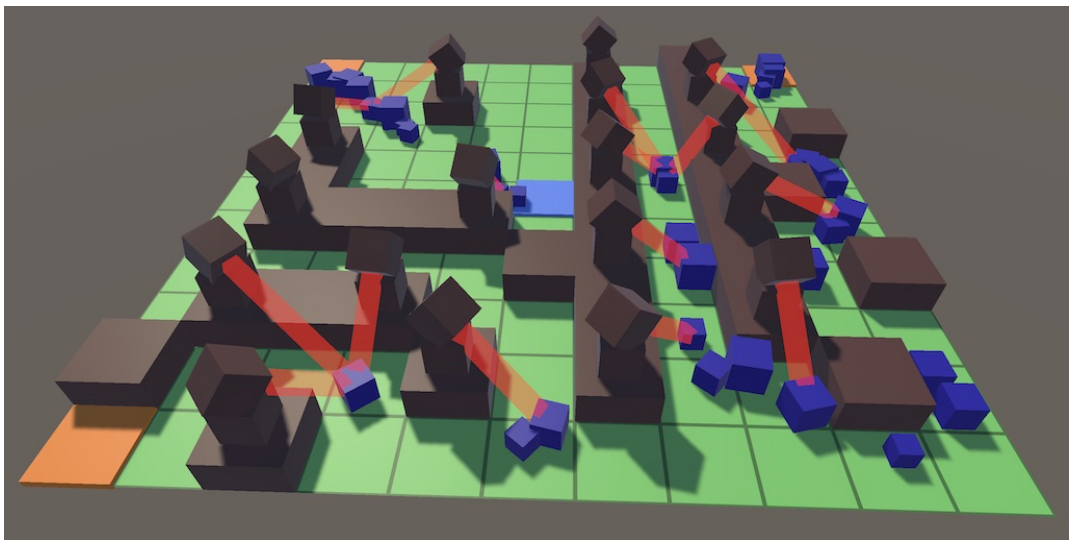
Towers Shooting Enemies

Put towers on the board.
Target enemies with the aid of physics.
Keep track of them for as long as possible.
Shoot them with a laser beam.

This is the third installment of a tutorial series about creating a simple tower defense game. It covers the creations of towers and how they target and shoot enemies.

This tutorial is made with Unity 2018.3.0f2.



Enemies are feeling the heat.

1 Building a Tower

Walls only slow enemies down by increasing the length of the path that they have to travel. But the goal of the game is to eliminate the enemies before they reach their destination. That's done by placing towers on the board that shoot them.

1.1 Tile Content

Towers are yet another type of tile content, so add an entry for them to `GameTileContent`.

```
public enum GameTileContentType {
	Empty, Destination, Wall, SpawnPoint, Tower
}
```

We'll only support one kind of tower in this tutorial, so we can make do by giving `GameTileContentFactory` one reference to a tower prefab, which can also be instantiated via `Get`.

```
	[SerializeField]
	GameTileContent towerPrefab = default;

	public GameTileContent Get (GameTileContentType type) {
		switch (type) {
			…
			case GameTileContentType.Tower: return Get(towerPrefab);
		}
		…
	}
```

But towers need to shoot, so they will need to get updated and require their own code. Create a `Tower` class for this purpose that extends `GameTileContent`.

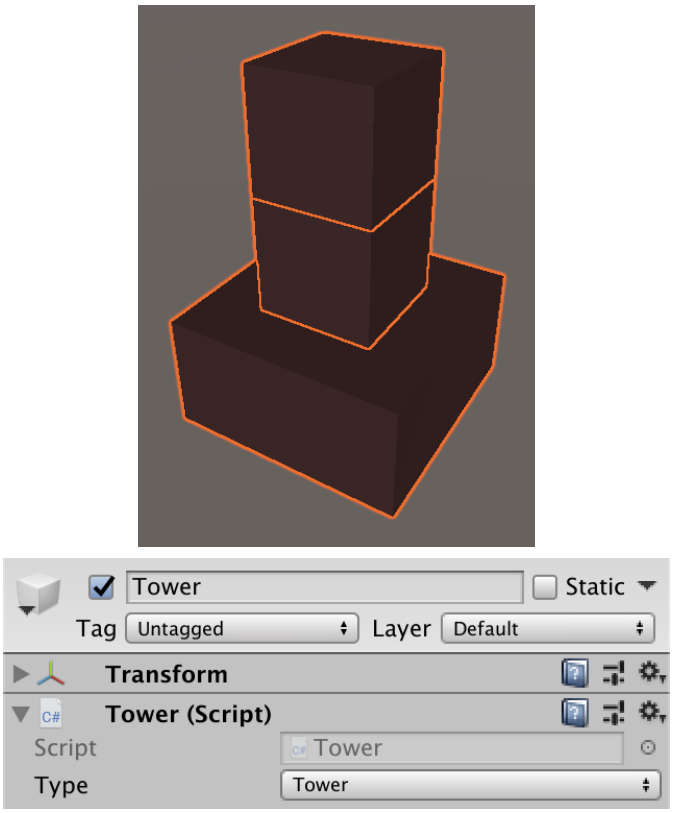
```
using UnityEngine;

public class Tower : GameTileContent {}
```

We can enforce that the tower prefab has this component by chancing the type of the factory's field to `Tower`. As it still counts as `GameTileContent` we don't need to change anything else.

```
	Tower towerPrefab = default;
```

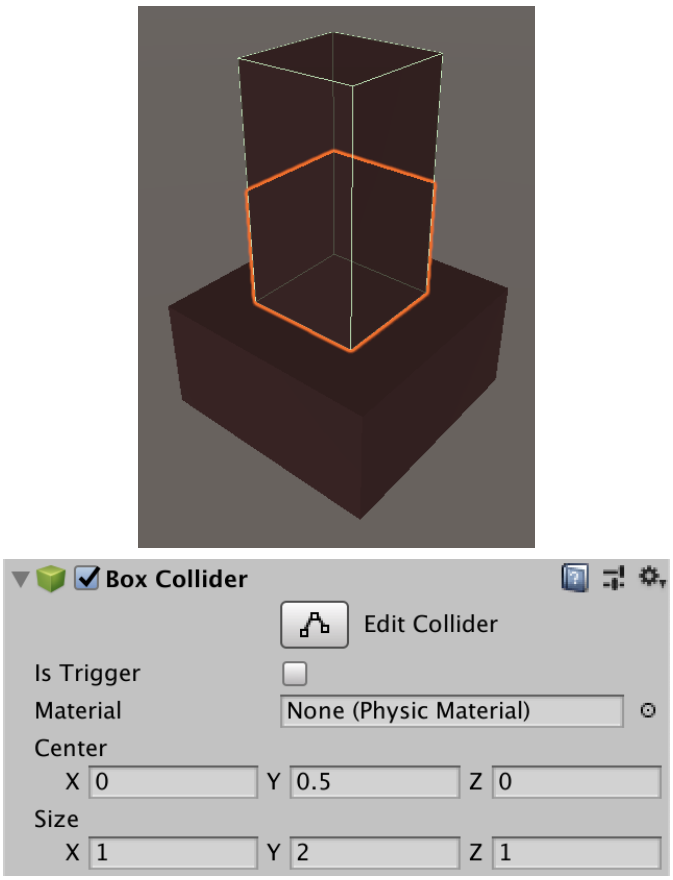
1.2 Prefab

Create a prefab for the tower. You can begin by duplicating the wall prefab and replacing its `GameTileContent` component with a `Tower` component and settings its type to *Tower*. To make the tower fit in with the walls, keep the existing wall cube as the tower's base. Then place another cube on top of it to represent the tower. I set its scale to 0.5. Put yet another cube of the same size on top of that, to represent the turret, which is the part that aims and shoots.



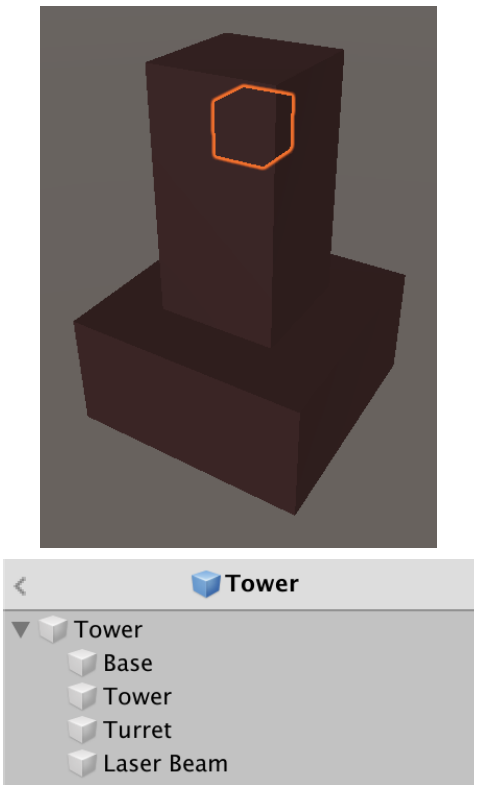
Three cubes form a tower.

The turret will rotate, and because it has a collider the physics engine will have to keep track of it. But we don't really need to be so precise, because all we use the tower colliders for is selecting cells. We can make do with an approximation. Remove the collider of the turret cube and adjust the collider of the tower cube so it covers both cubes.



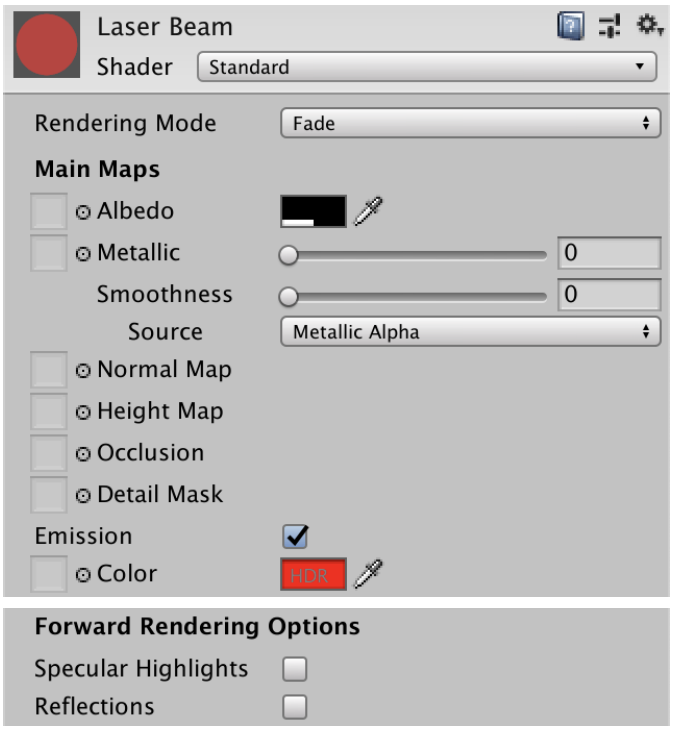
Tower cube collider.

Our tower will shoot a laser beam. There are many ways to visualize that, but we'll simply use a semitransparent cube that we stretch to form the beam. Each tower will need one of its own, so add it to the tower prefab. Place it inside the turret so it's hidden by default and give it a smaller scale, like 0.2. Make it a child of the prefab root, not of the turret cube.



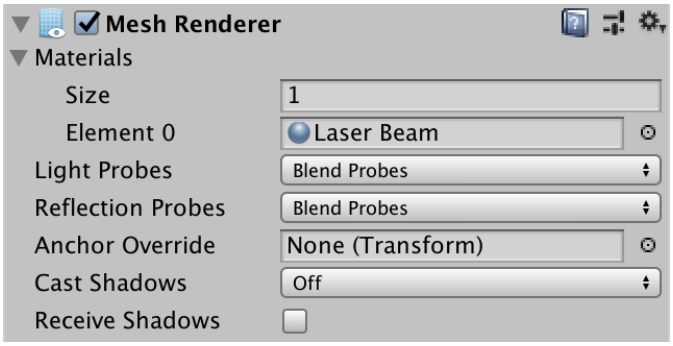
Hidden laser beam cube.

Give the laser beam an appropriate material. I simply used a standard semitransparent black material and turned off all reflections while giving it a red emissive color.



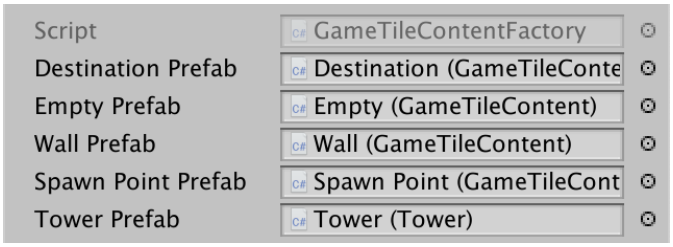
Laser beam material.

Make sure that the laser beam cube doesn't have a collider, and also turn off shadow casting and receiving for it.



Laser beam doesn't interact with shadows.

Once the tower prefab is finished, add it to the factory.



Factory with tower.

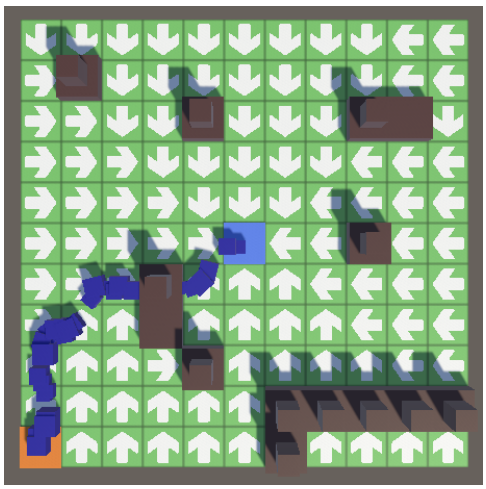
1.3 Placing Towers

We'll add and remove towers via another toggle method. We can simply duplicate `GameBoard.ToggleWall` and change the method's name and content type.

```
	public void ToggleTower (GameTile tile) {
		if (tile.Content.Type == GameTileContentType.Tower) {
			tile.Content = contentFactory.Get(GameTileContentType.Empty);
			FindPaths();
		}
		else if (tile.Content.Type == GameTileContentType.Empty) {
			tile.Content = contentFactory.Get(GameTileContentType.Tower);
			if (!FindPaths()) {
				tile.Content = contentFactory.Get(GameTileContentType.Empty);
				FindPaths();
			}
		}
	}
```

In `Game.HandleTouch`, toggle a tower instead of a wall if the player is holding down the shift key.

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



Towers on the board.

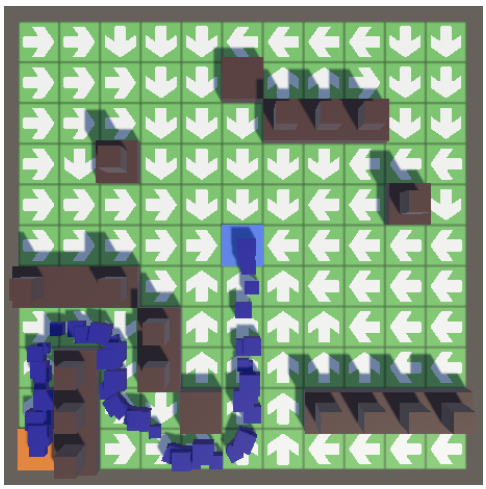
1.4 Blocking the Path

Currently only walls block pathfinding, so enemies move through towers. Let's add a convenient property to `GameTileContent` that indicates whether it blocks the path or not. It does so if it's either a wall or a tower.

```
	public bool BlocksPath =>
		Type == GameTileContentType.Wall || Type == GameTileContentType.Tower;
```

Use this property in `GameTile.GrowPathTo`, instead of checking the exact content type.

```
	GameTile GrowPathTo (GameTile neighbor, Direction direction) {
		…
		return
			//neighbor.Content.Type != GameTileContentType.Wall ? neighbor : null;
			neighbor.Content.BlocksPath ? null : neighbor;
	}
```



Both towers and walls block the path.

1.5 Replacing Walls

It is likely that a player will end up replacing walls with towers a lot. Having to remove the wall first is inconvenient, and it's possible for enemies to sneak through the temporary gap. We can make a direct replacement possible by having `GameBoard.ToggleTower` also check whether the tile currently has a wall in it. If so, directly replace it with a tower. In this case we do not have to find new paths, since the tile is still blocking them.

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

2 Targeting Enemies

A tower can only do its job if it can find an enemy. Once an enemy is found, it must also decide which part of the enemy to aim at.

2.1 Target Point

We'll use the physics engine to detect targets. Just like with the tower collider, we don't need the enemy's collider to exactly match its shape. We can make do with the simplest collider, which is a sphere. Once detected, we'll use the position of the game object with the collider attached to it as the point to aim at.

We cannot attach a collider to the enemy's root object, because it doesn't match its model's position all the time and would make the tower aim at the ground. So we have to put the collider somewhere in the model. The physics engine will give us a reference to that object, which we can use for targeting, but we'll also need to access the `Enemy` component on the root object. Let's create a `TargetPoint` component to make this easy. Give it a property to privately set and publicly get the `Enemy` component, and another property to get its world position.

```
using UnityEngine;

public class TargetPoint : MonoBehaviour {

	public Enemy Enemy { get; private set; }

	public Vector3 Position => transform.position;
}
```

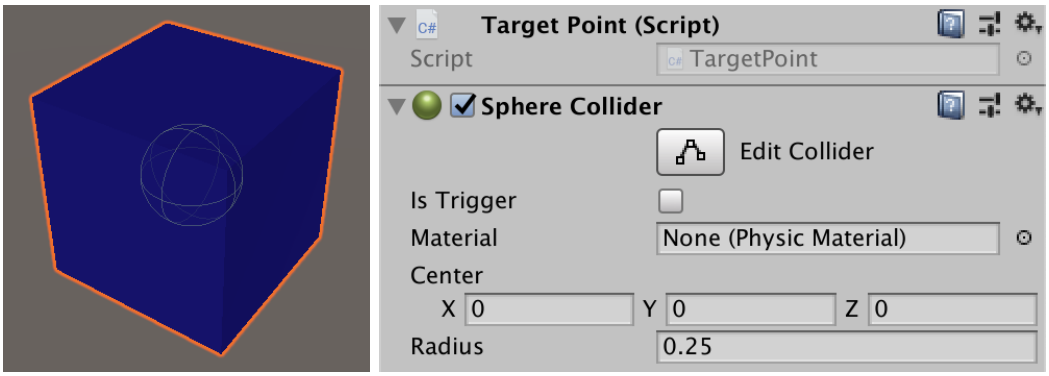
Give it an `Awake` method that sets the reference to its `Enemy` component. We can go directly to its root object via `transform.root`. If the `Enemy` component doesn't exist then we made a design mistake, so let's add an assertion for that.

```
	void Awake () {
		Enemy = transform.root.GetComponent<Enemy>();
		Debug.Assert(Enemy != null, "Target point without Enemy root!", this);
	}
```

Also, the collider should be attached to the same game object that `TargetPoint` is attached to.

```
		Debug.Assert(Enemy != null, "Target point without Enemy root!", this);
		Debug.Assert(
			GetComponent<SphereCollider>() != null,
			"Target point without sphere collider!", this
		);
```

Add the component and collider to the cube of our enemy prefab. That will make towers aim at the center of the cube. Use a sphere collider with a radius of 0.25. As the cube has a scale of 0.5, the collider's effective radius is 0.125. That will make it so the enemy must have visually penentrated a tower's range at bit before it becomes a valid target. The collider's size is also affected by the random scale of the enemy, so its in-game size will vary as well.



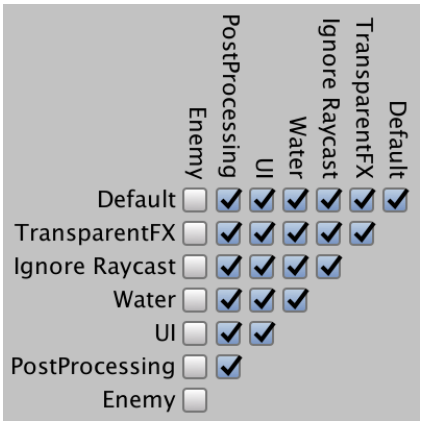
Enemy with target point and collider on its cube.

2.2 Enemy Layer

Towers only care about enemies and shouldn't target anything else, so we'll put all enemies on a dedicated layer. We'll use layer 9. Set its named to *Enemy* via the *Layers & Tags* window, which can be opened via the *Edit Layers* option in the *Layers* dropdown menu at the top right corner of the editor.

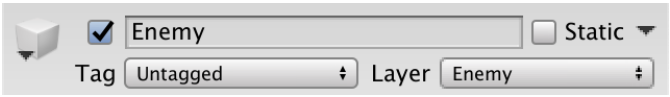
Layer 9 is for enemies.

This layer is only for detecting enemies, not for physics interactions. Let's indicate that by disabling it in the *Layer Collision Matrix*, which you can find under the *Physics* panel in the project settings.



Layer collision matrix.

Make sure that the target point's game object is on the correct layer. The rest of the enemy prefab can be on other layers, but it's easiest to be consistent and place the entire prefab on the enemy layer. If you'll change the root object's layer you get the option to change all its child objects as well.



Enemy on the correct layer.

Let's assert that `TargetPoint` is indeed on the correct layer.

```
	void Awake () {
		…
		Debug.Assert(gameObject.layer == 9, "Target point on wrong layer!", this);
	}
```

Also, the player interaction should ignore enemy colliders. We can do that by adding a layer mask argument to `Physics.Raycast` in `GameBoard.GetTile`. It has a variant that takes the ray distance and the layer mask as additional arguments. Provide the maximum range and the layer mask for the default layer, which is 1.

```
	public GameTile GetTile (Ray ray) {
		if (Physics.Raycast(ray, out RaycastHit hit, float.MaxValue, 1)) {
			…
		}
		return null;
	}
```

Shouldn't the layer mask be zero?

The default layer's index is zero, but we're providing a layer mask. The mask works by setting individual bits of an integer to 1 if the layer should be included. In this case, only the first bit must be set, which means its least significant bit, which defines the number 2^0, which is 1.

2.3 Updating Tile Content

Towers can only do their work if they get updated. This is also true for tile content in general, even though our other content currently does nothing. So let's add a virtual `GameUpdate` method to `GameTileContent` that does nothing by default.

```
	public virtual void GameUpdate () {}
```

Have `Tower` override it, initially just logging that it's looking for a target.

```
	public override void GameUpdate () {
		Debug.Log("Searching for target...");
	}
```

`GameBoard` is in charge of the tiles and their content, so it will also keep track of which content needs to get updated. Give it a list for that purpose, plus a public `GameUpdate` method that updates everything in that list.

```
	List<GameTileContent> updatingContent = new List<GameTileContent>();

	…

	public void GameUpdate () {
		for (int i = 0; i < updatingContent.Count; i++) {
			updatingContent[i].GameUpdate();
		}
	}
```

In this tutorial only towers need to be updated. Adjust `ToggleTower` so it adds and removes the content as appropriate. If other content would require updating as well then we'd need a more general approach, but for now this suffices.

```
	public void ToggleTower (GameTile tile) {
		if (tile.Content.Type == GameTileContentType.Tower) {
			updatingContent.Remove(tile.Content);
			tile.Content = contentFactory.Get(GameTileContentType.Empty);
			FindPaths();
		}
		else if (tile.Content.Type == GameTileContentType.Empty) {
			tile.Content = contentFactory.Get(GameTileContentType.Tower);
			//if (!FindPaths()) {
			if (FindPaths()) {
				updatingContent.Add(tile.Content);
			}
			else {
				tile.Content = contentFactory.Get(GameTileContentType.Empty);
				FindPaths();
			}
		}
		else if (tile.Content.Type == GameTileContentType.Wall) {
			tile.Content = contentFactory.Get(GameTileContentType.Tower);
			updatingContent.Add(tile.Content);
		}
	}
```

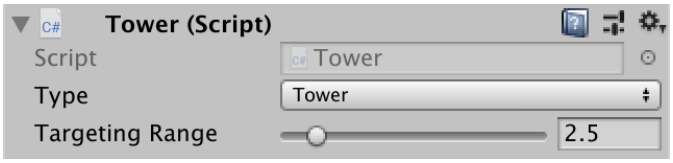
To make this work we now also have to update the board in `Game.Update`. Update the board after the enemies. That way the towers will aim where the enemies currently are. If we did it the other way around then towers would aim where their targets were one frame earlier.

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

2.4 Targeting Range

Towers only have a limiting targeting range. Make that configurable by adding a field to `Tower`. Distance is measured from the center of the tower's tile, so a range of 0.5 only covers its own tile. Thus a reasonable minimum and default range would be 1.5, covering most of the neighboring tiles.

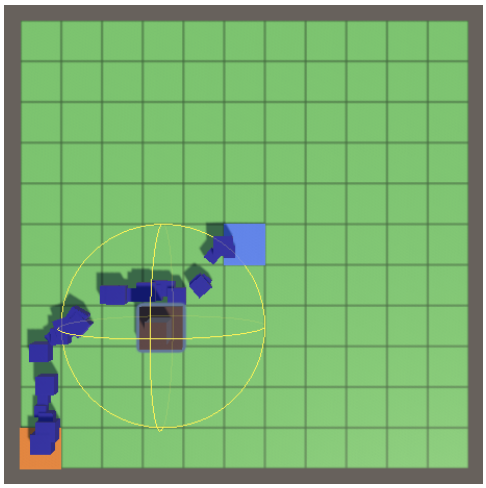
```
	[SerializeField, Range(1.5f, 10.5f)]
	float targetingRange = 1.5f;
```



Targeting range set to 2.5.

Let's visualize the range with a gizmo. We don't need to see it all the time, so let's create an `OnDrawGizmosSelected` method, which only gets invoked for selected objects. Draw a yellow wire sphere with the range as its radius, centered on the tower. Position it a bit above the ground so it's always clearly visible.

```
	void OnDrawGizmosSelected () {
		Gizmos.color = Color.yellow;
		Vector3 position = transform.localPosition;
		position.y += 0.01f;
		Gizmos.DrawWireSphere(position, targetingRange);
	}
```



Targeting range gizmo.

We can now see which enemies are valid targets for each tower. But selecting towers in the scene window is inconvenient, because we end up selecting one of the child cubes and then have to change the selection to the tower root object. Other tile content suffers from the same problem. We can enforce selection of the content root in the scene window by adding the `SelectionBase` attribute to `GameTileContent`.

```
[SelectionBase]
public class GameTileContent : MonoBehaviour { … }
```

2.5 Acquiring a Target

Add a `TargetPoint` field to `Tower` so it can keep track of its acquired target. Then change `GameUpdate` so it invokes a new `AquireTarget` method that returns whether it found a target. If so, log this fact.

```
	TargetPoint target;

	public override void GameUpdate () {
		if (AcquireTarget()) {
			Debug.Log("Acquired target!");
		}
	}
```

In `AcquireTarget`, retrieve all valid targets by invoking `Physics.OverlapSphere` with the tower's position and range as arguments. The result is a `Collider` array containing all colliders that overlap the described sphere. If the array's length is positive then there is at least a single target point and we'll simply pick the first one. Grab its `TargetPoint` component which should always exist, assign it to the target field, and indicate success. Otherwise clear the target and indicate failure.

```
	bool AcquireTarget () {
		Collider[] targets = Physics.OverlapSphere(
			transform.localPosition, targetingRange
		);
		if (targets.Length > 0) {
			target = targets[0].GetComponent<TargetPoint>();
			Debug.Assert(target != null, "Targeted non-enemy!", targets[0]);
			return true;
		}
		target = null;
		return false;
	}
```

We're only guaranteed to get a valid target point if we only consider colliders on the enemy layer. That's layer 9, so provide the corresponding layer mask.

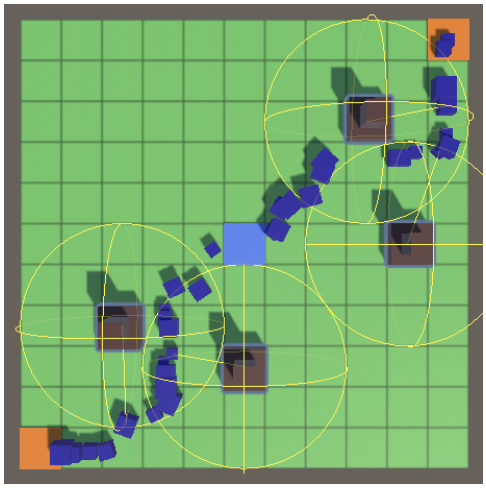
```
	const int enemyLayerMask = 1 << 9;

	…

	bool AcquireTarget () {
		Collider[] targets = Physics.OverlapSphere(
			transform.localPosition, targetingRange, enemyLayerMask
		);
		…
	}
```

How does that bit mask work?

As the enemy layer has index nine, the bit mask must have its tenth bit set to 1. The corresponding integer is 2^9, which is 512. But that isn't an intuitive way to write a bit mask. We could also write a binary literal, like `0b10_0000_0000` but then we have to count zeros. In this case the most convenient notation is to use the left-shift operator << to shift bits leftwards, representing a power-of-two number if we apply it to `1`.

We can visualize the acquired target by drawing a gizmo line between the positions of the tower and the target.

```
	void OnDrawGizmosSelected () {
		…
		if (target != null) {
			Gizmos.DrawLine(position, target.Position);
		}
	}
```



Visualizing targets.

Why not use methods like `OnTriggerEnter`?

The advantage of manually checking for overlapping targets is that we only have to do it when necessary. There is no reason to check for targets if a tower already has one. Also, by fetching all potential targets at once we don't have to manage a list of potential targets per tower, which change all the time.

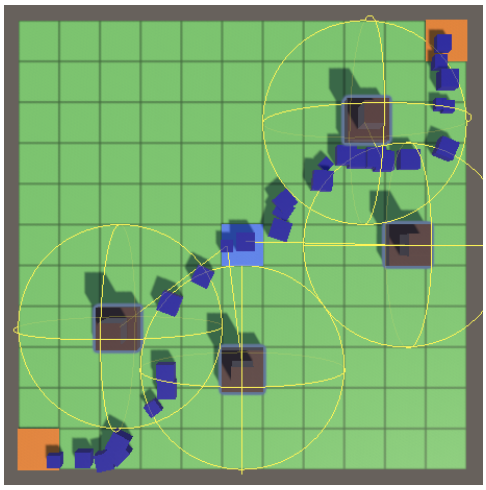
2.6 Target Locking

Which target gets acquired depends on the order in which the physics engine presents them, which is effectively arbitrary. As a result the acquired target seems to change without reason. Once a tower has a target, it makes sense that it keeps tracking that one instead of switching to another. Add a `TrackTarget` method that does this tracking and returns whether it was successful. Begin by only indicating whether a target has already been acquired.

```
	bool TrackTarget () {
		if (target == null) {
			return false;
		}
		return true;
	}
```

Invoke this method in `GameUpdate` and only if it fails invoke `AcquireTarget`. If either succeeds then we have a target. We can do that by putting both method invocations in the `if` check with an OR operator, because if the first operand yields `true` then the second operand is not evaluated, so its invocation is skipped. The AND operator behaves in a similar way.

```
	public override void GameUpdate () {
		if (TrackTarget() || AcquireTarget()) {
			Debug.Log("Locked on target!");
		}
	}
```



Tracking targets.

The result is that towers lock on to a target until it reaches a destination and gets destroyed. If you're reusing enemies then you'd have to check for a valid reference instead, like how shape references handled in the Object Management series.

To only track targets while they are in range, `TrackTarget` has to check the distance between the tower and target. If it goes out of range then clear the target and return failure. We can use the `Vector3.Distance` method for the check.

```
	bool TrackTarget () {
		if (target == null) {
			return false;
		}
		Vector3 a = transform.localPosition;
		Vector3 b = target.Position;
		if (Vector3.Distance(a, b) > targetingRange) {
			target = null;
			return false;
		}
		return true;
	}
```

However, this doesn't take the collider's radius into consideration. So a tower can end up failing to track a target, then immediately acquiring it again, only to stop tracking it the next frame, and so on. We can prevent that by adding the radius of the collider to the range.

```
		if (Vector3.Distance(a, b) > targetingRange + 0.125f) { … }
```

That gives us the correct results, but only when the enemy's scale is unchanged. As we give each enemy a random scale, we should factor that into the range adjustment. To do so we must remember the scale we gave `Enemy` and expose it via a getter property.

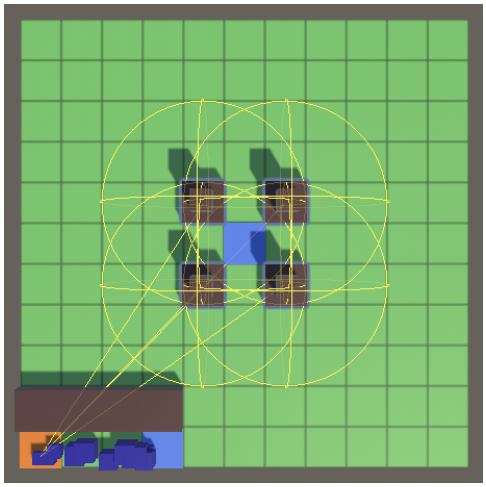
```
	public float Scale { get; private set; }

	…

	public void Initialize (float scale, float speed, float pathOffset) {
		Scale = scale;
		…
	}
```

Now we can check the appropriate distance in `Tower.TrackTarget`.

```
		if (Vector3.Distance(a, b) > targetingRange + 0.125f * target.Enemy.Scale) { … }
```

2.7 Synchronizing Physics

All seems to work fine, except towers that can target the center of the board are able to acquire targets that should be out of range. They will fail to track those targets, so they only lock on for a single frame per target.



Incorrect targeting.

This happens because the state of the physics engine is not perfectly synchronized with our game state. All enemies are instantiated at the world origin, which coincides with the center of the board. We then move them to their spawn point, but the physics engine isn't immediately aware of that.

It's possible to force immediate synchronization as soon as an object's transformation changes, by setting `Physics.autoSyncTransforms` to `true`. But it's turned off by default because it's much more efficient to only synchronize everything at once, when needed. In our case, we only need to be synchronized when updating the towers. We can enfore that by invoking `Physics.SyncTransforms` in between updating the enemies and board in `Game.Update`.

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

2.8 Ignoring Elevation

Our gameplay is essentially 2D. So let's change `Tower` so it only takes the X and Z dimensions into consideration when targeting and tracking. The physics engine works in 3D space, but we can make the check in `AcquireTarget` effectively 2D by extruding the sphere upward so it should cover all colliders regardless of their vertical position. This can be done by using a capsule instead, with its second point a few units above the ground, let's say three.

```
	bool AcquireTarget () {
		Vector3 a = transform.localPosition;
		Vector3 b = a;
		b.y += 3f;
		Collider[] targets = Physics.OverlapCapsule(
			a, b, targetingRange, enemyLayerMask
		);
		…
	}
```

Can't we use the 2D physics engine?

The problem is that our game is defined in the XZ plane, while the 2D physics engine works in the XY plane. You could make it work, by either reorienting the entire game or creating a separate 2D representation for physics purposes only. But it's simpler to just use 3D physics.

We have to adjust `TrackTarget` as well. While we could create 2D vectors and use `Vector2.Distance`, let's do the math ourselves and compare square distances instead, which is all that we need. That eliminates a square root operation.

```
	bool TrackTarget () {
		if (target == null) {
			return false;
		}
		Vector3 a = transform.localPosition;
		Vector3 b = target.Position;
		float x = a.x - b.x;
		float z = a.z - b.z;
		float r = targetingRange + 0.125f * target.Enemy.Scale;
		if (x * x + z * z > r * r) {
			target = null;
			return false;
		}
		return true;
	}
```

How does that math work?

It relies on the Pythagorean theorem to calculate the 2D distance, but leaves out the square root. Instead it squares the radius so we end up comparing square lengths. That suffices because we only need to check for relative length and don't need an exact difference.

2.9 Avoiding Memory Allocations

A downside of using `Physics.OverlapCapsule` is that it allocates a new array per invocation. That can be avoided by allocating an array once and invoking the alternative `OverlapCapsuleNonAlloc` method, with the array as an extra argument, after the radius. The length of the provided array limits how many results we get. Any potential targets beyond the limit are omitted. As we're only using the first element anyway, we can make do with an array of length 1.

Instead of an array, `OverlapCapsuleNonAlloc` returns how many hits occured—up to the maximum allowed—which we have to check instead of the array's length.

```
	static Collider[] targetsBuffer = new Collider[1];

	…

	bool AcquireTarget () {
		Vector3 a = transform.localPosition;
		Vector3 b = a;
		b.y += 2f;
		int hits = Physics.OverlapCapsuleNonAlloc(
			a, b, targetingRange, targetsBuffer, enemyLayerMask
		);
		if (hits > 0) {
			target = targetsBuffer[0].GetComponent<TargetPoint>();
			Debug.Assert(target != null, "Targeted non-enemy!", targetsBuffer[0]);
			return true;
		}
		target = null;
		return false;
	}
```

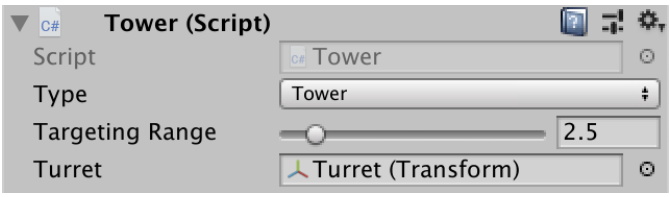
3 Shooting Enemies

Now that we have a valid target, it's time to shoot it. This involves aiming the turret, firing the laser, and dealing damage.

3.1 Aiming the Turret

In order to point the turret at the target, `Tower` needs to have a reference to the turret's `Transform` component. Add a configuration field for that and hook it up in the tower prefab.

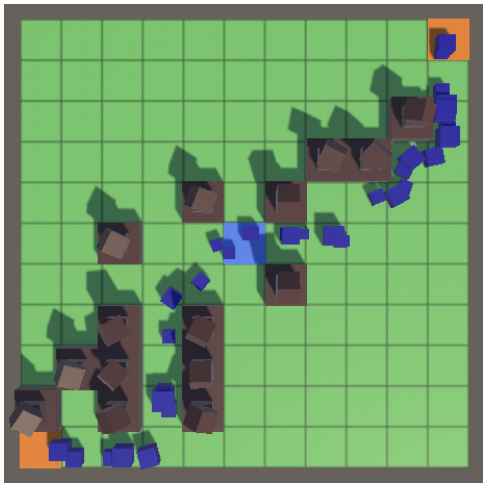
```
	[SerializeField]
	Transform turret = default;
```



Turret hooked up.

In `GameUpdate`, if we have a valid target we should shoot it. Put the code for that in a separate method. Have it rotate the turret to face the target by invoking its `Transform.LookAt` method with the target point as an argument.

```
	public override void GameUpdate () {
		if (TrackTarget() || AcquireTarget()) {
			//Debug.Log("Locked on target!");
			Shoot();
		}
	}

	void Shoot () {
		Vector3 point = target.Position;
		turret.LookAt(point);
	}
```

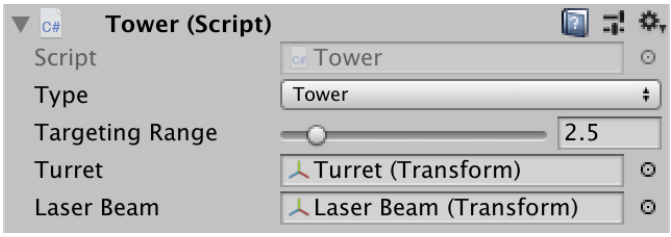


Just aiming.

3.2 Shining the Laser

To position the laser beam `Tower` needs a reference to it as well.

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



Laser beam connected.

Turning a cube into a correct laser beam requires three steps. First, its orientation must match the turret's. We can do that by copying the rotation.

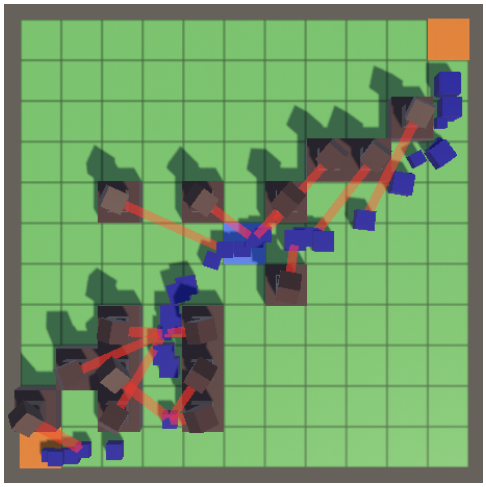
```
	void Shoot () {
		Vector3 point = target.Position;
		turret.LookAt(point);
		laserBeam.localRotation = turret.localRotation;
	}
```

Second, we scale the laser beam so it's as long as the distance between the turret's local origin and the target point. We scale its Z dimension, which is the local axis that's pointed at the target. To keep the original XY scale, store the original scale when the turret awakens.

```
	Vector3 laserBeamScale;

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

	…

	void Shoot () {
		Vector3 point = target.Position;
		turret.LookAt(point);
		laserBeam.localRotation = turret.localRotation;

		float d = Vector3.Distance(turret.position, point);
		laserBeamScale.z = d;
		laserBeam.localScale = laserBeamScale;
	}
```

Third, position the laser beam halfway between the turret and target point.

```
		laserBeam.localScale = laserBeamScale;
		laserBeam.localPosition =
			turret.localPosition + 0.5f * d * laserBeam.forward;
```



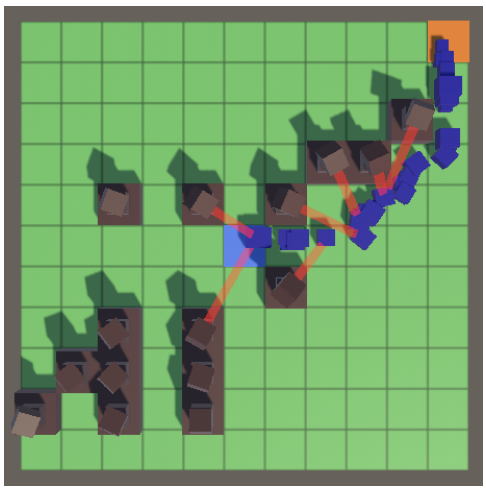
Shooting laser beams.

Can't we make the laser beam a child of the turret?

If we did that then we wouldn't need to rotate the laser beam separately and also wouldn't need its forward vector. However, the turret's scale would also affect it, so we'd have to compensate for that. It's easier to keep them separate.

This works as long as a turret remains locked on a target. But when no target is available the laser remains active. We can visually turn off the laser by setting its scale to zero in `GameUpdate` if we're not shooting.

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



Idle towers don't shoot.

3.3 Enemy Health

Currently our laser beams are just pointing out enemies and have no effect beyond that. The idea is that the enemies are hurt by the laser beams. We don't want to instantaneously destroy enemies, so give `Enemy` a health property. We can use an arbitrary amount to represent a healthy enemy, so let's use 100. But it makes sense that bigger enemies should be able to take more punishment, so let's factor in the scale.

```
	float Health { get; set; }

	…

	public void Initialize (float scale, float speed, float pathOffset) {
		…
		Health = 100f * scale;
	}
```

To support taking damage, add a public `ApplyDamage` method that subtracts its parameter from health. We assume that the damage isn't negative, so assert that.

```
	public void ApplyDamage (float damage) {
		Debug.Assert(damage >= 0f, "Negative damage applied.");
		Health -= damage;
	}
```

We won't immediately get rid of an enemy when its health reaches zero. Instead, we check whether health has run out at the start of `GameUpdate` and terminate if so.

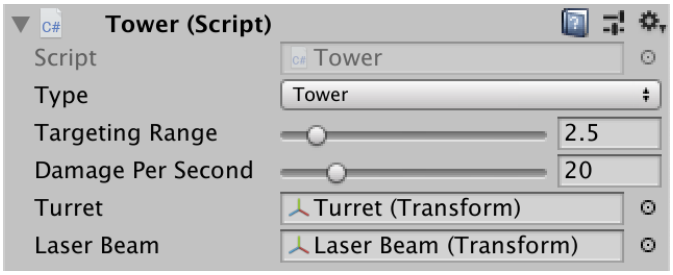
```
	public bool GameUpdate () {
		if (Health <= 0f) {
			OriginFactory.Reclaim(this);
			return false;
		}

		…
	}
```

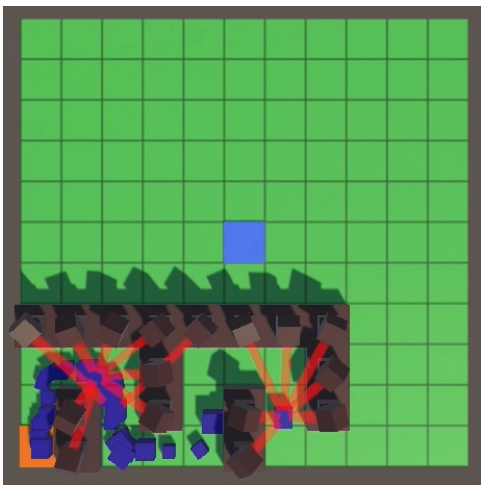
Doing it like that makes it so all towers effectively fire at the same time, instead of in a sequence that allows them to switch targets in case a previous tower destroyed an enemy they were also targeting.

3.4 Damage per Second

Now we must determine how much damage the laser beam deals. Add a configuration field to `Tower` for that. As the laser beam causes continuous damage, we express it as damage per second. In `Shoot`, apply it to the target's `Enemy` component, multiplied by the time delta.

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

	…

	void Shoot () {
		…

		target.Enemy.ApplyDamage(damagePerSecond * Time.deltaTime);
	}
```





20 damage per second per tower.

3.5 Targeting at Random

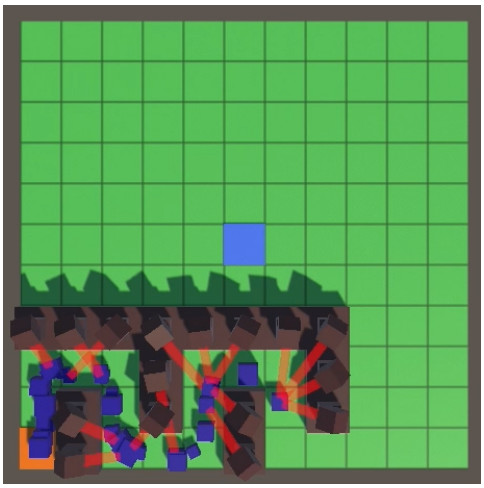
Because we're always picking the first available target per tower, the targeting behavior depends on the order in which the physics engine checks for overlapping colliders. This dependency isn't good because we don't know the details, have no control over it, and it can also look weird and inconsistent. It often results in focused fire, but not always.

Instead of completely being at the mercy of the physics engine, let's add some randomness to it. We do this by increasing the amount of hits that we can receive to a large number, let's say 100. That might not be enough to get all potential targets on a very crowded board, but should give us more than enough room to improve targeting behavior.

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

Now instead of always picking the first potential target, pick a random element from the array.

```
	bool AcquireTarget () {
		…
		if (hits > 0) {
			target =
				targetsBuffer[Random.Range(0, hits)].GetComponent<TargetPoint>();
			…
		}
		target = null;
		return false;
	}
```



Random targeting.

Can we also use other target selection criteria?

Yes, you could for example select the one with the lowest or highest health. Or Keep track of how many towers target each enemy, to either focus fire or spread out. Or combine multiple criteria. However, it is hard to come up with good targeting criteria that consistently beat simply picking a random target per tower.

The next tutorial is Ballistics.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick