Animation Lively Enemies

Record animations.
*Create playable graphs to animate enemies.*
Mix animations to transition between them.
Use existing models and animations.

This is the sixth and final installment of a tutorial series about creating a simple tower defense game. It is about animating enemies, covering both recording new animations and importing existing assets.

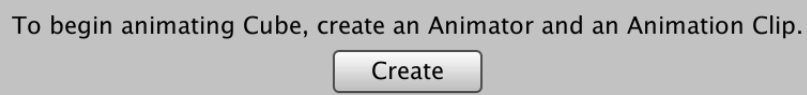This tutorial is made with Unity 2018.4.9f1.
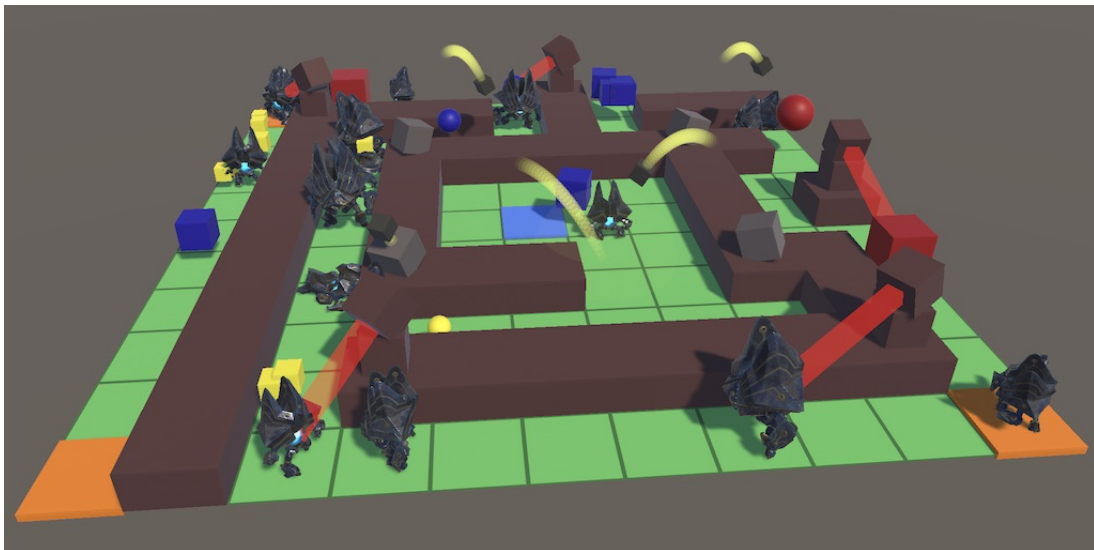
Enemies bouncing, walking, spinning, and falling apart.

1 Bouncing Enemies

Up to this point our enemies simply slide across the board. This can be fine for an abstract game that uses cubes and spheres for enemies, but even such enemies can be made more interesting by making them move in a more organic way. We could make them bounce by offsetting their vertical position with something like an absolute sine wave based on time, but the general approach is to use an animation clip. We'll use animations because that allows much more complex movement and also makes it possible to import existing animations.

1.1 Animation Clip

We can create animation clips in the Unity editor by recording adjustments to an object hierarchy. Drag a medium enemy cube prefab instance into the scene, or a separate scene dedicated to animation recording. Then select the *Cube* child of the enemy's *Model* and open the animation window via *Window / Animation / Animation*.
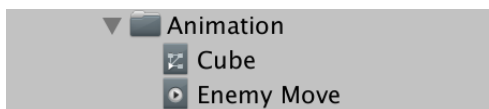

Create button.

Because we haven't already animated the cube, the *Animation* window displays a *Create* button. Pressing it will attach an *Animator* component to *Cube* and create two assets, an *Animation Controller* for the cube and an animation clip, which we'll name *Enemy Move*.


Animation and animator assets.

The animation clip asset contains the data for our animation, which is currently still empty. Selecting it will show a *Loop Time* toggle which is enabled by default, meaning that it represents a looping animation. This is correct as the movement animation should repeat as long as the enemy is in motion.
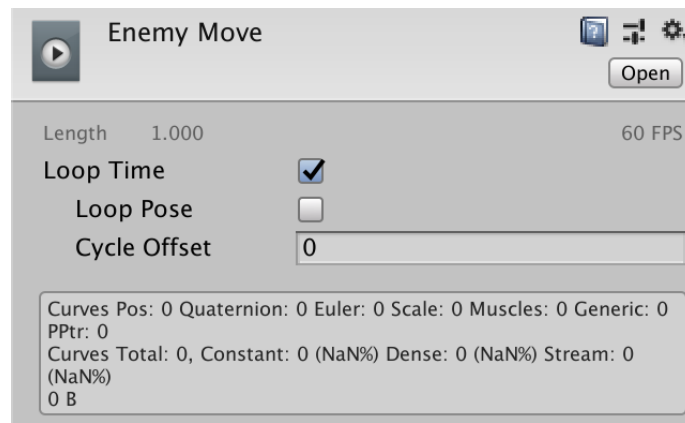
Move animation asset.

The *Animator* component that got added to *Cube* has a reference to the animation controller asset that also got created.
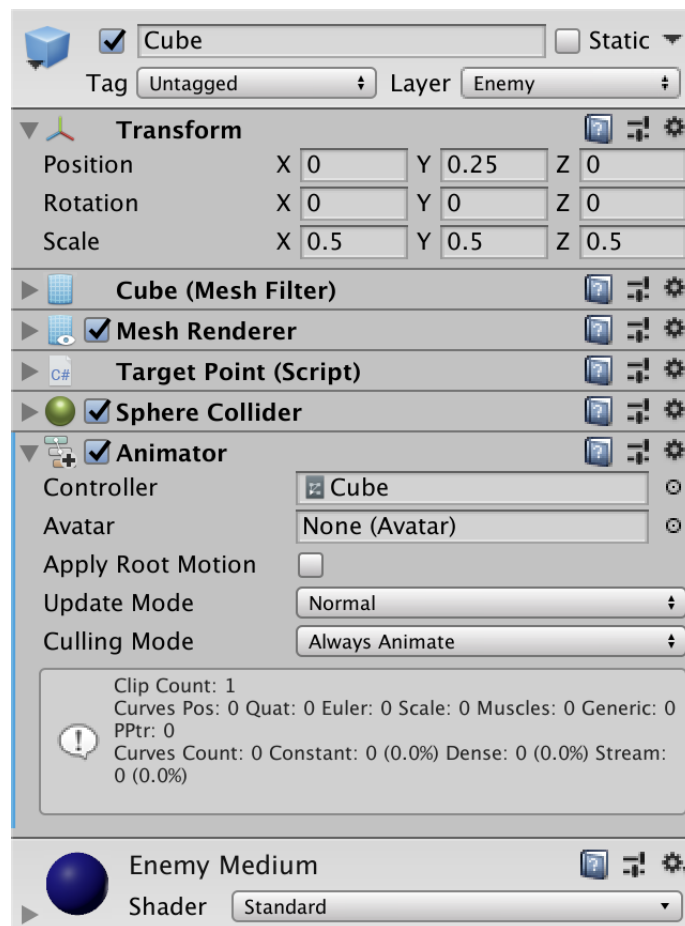
Cube with animator.

The controller is a state machine that can get very complex, but initially it only has an entry state that goes straight to the animation that we created, plus a catch-all any state. You can see those by opening the *Animator* window, which you can do by double-clicking the controller or pressing its *Open* button.
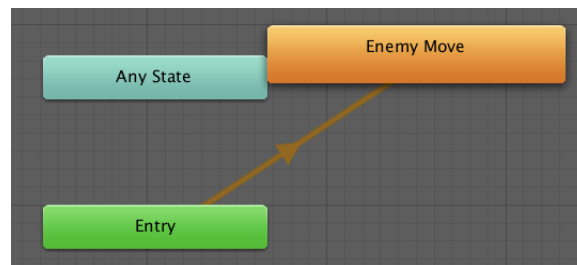
Animator graph.

1.2 Recording an Animation

To record the move animation, select *Cube* and press the red dot record button in the *Animation* window. We'll create a simple bounce with a duration of one second. Move the time line to 0:30, which represents half a second. Then increase the cube's vertical position from 0.25 to 0.75. That will create two key frames, one for the original position at 0:00 and the adjusted one at 0:30. After that, move the time line to 1:00, set the vertical position back to 0.25, and press the record button again to stop recording.
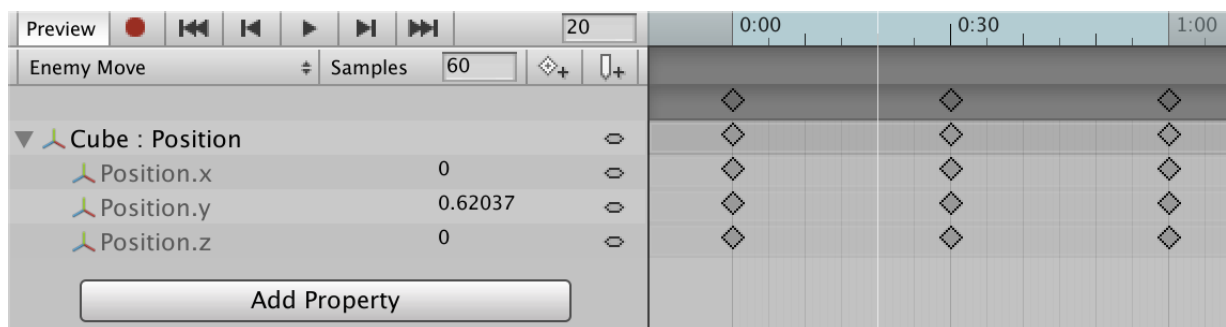
Three key frames for position.

You can preview the animation by pressing the play-animation button, a little to the right of the record button.
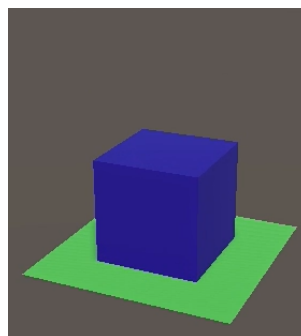
Moving up and down.

The cube's position interpolates between the key frames, causing it to move up and down between 0.25 and 0.75. We can make it a bit more organic-looking by changing the trajectory into a parabola. Switch from *Dopesheet* to *Cuves* via the buttons at the bottom of the *Animation* window. That shows us the curves used to interpolate between the key frames. You can zoom in by changing the size of the scroll bars. Then select the key point for Y at 0:00 and drag its tangent up until it looks nice. Do the same for the key at 1:00. As the movement is supposed to be organic the tangents don't have to exactly mirror each other.
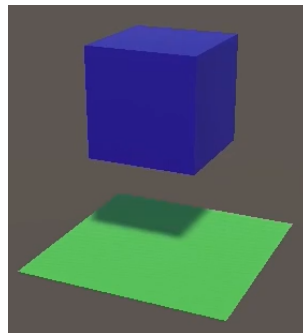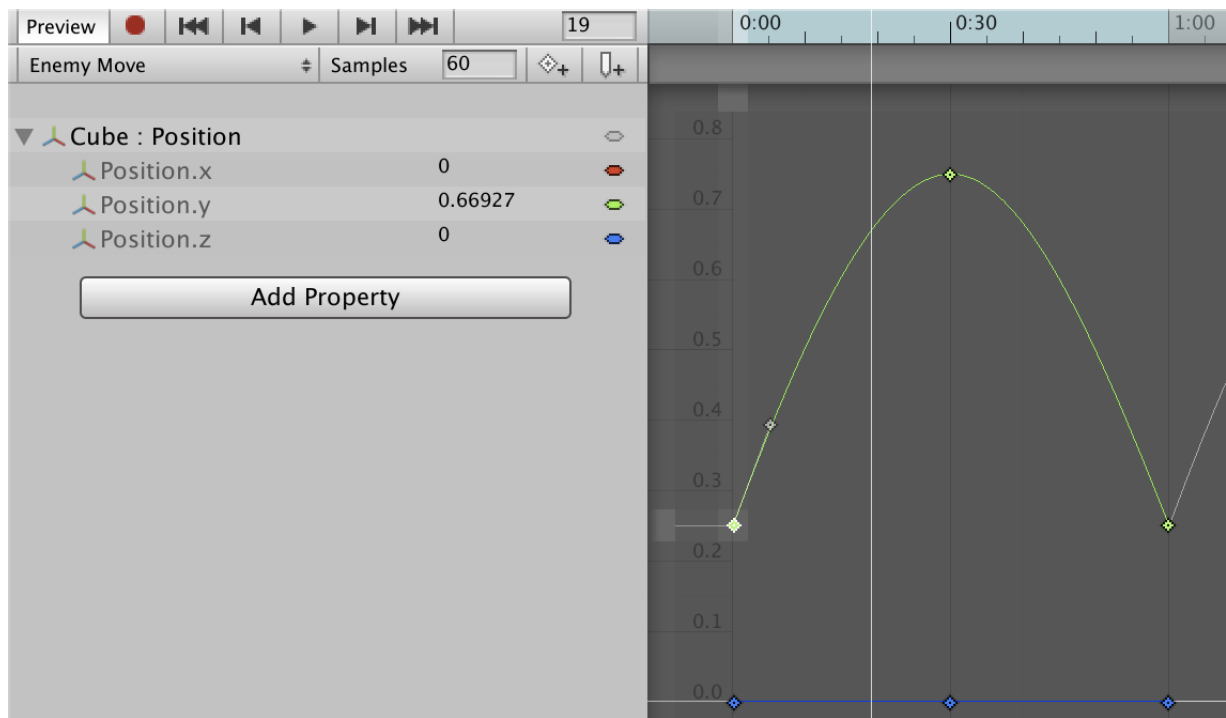
Bouncing with a parabola curve.

You could embellish the animation with scale adjustments and such, but this is good enough to make the enemies look alive.

1.3 Configuring Animations

Enemies can have different animations, even when using the same 3D model. Conversely, enemies with different models can have the same animations. So we'll make it possible to configure animation clips per enemy via a separate `EnemyAnimationConfig` asset type, so configurations can be easily shared. As we only have a move animation at this point, that's the only clip to store for now.

```csharp
using UnityEngine;

[CreateAssetMenu]
public class EnemyAnimationConfig : ScriptableObject {

    [SerializeField]
    AnimationClip move = default;

    public AnimationClip Move => move;
}
```

Create one animation config asset and assign the move animation to it.
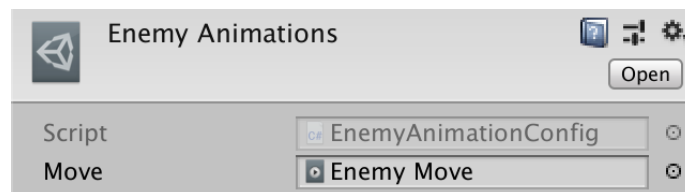
Enemy animation config asset.

Add a serializable field to `Enemy` for this configuration, then give all enemy prefabs a reference to our single animation config asset, as our animation works for both cubes and spheres.

```csharp
    [SerializeField]
    EnemyAnimationConfig animationConfig = default;
```


Enemy with animation config.

2 Playing Animations

The animation controller can be used to animate enemies, but it is a rather heavy-handed and rigid approach for our simple enemy behavior. Besides that, there can be many enemies alive at the same time and they would all need their own controller, so the logic to control animations should be as simple as possible. Finally, we want to use different animations per enemy while they all share the same logic. So rather than 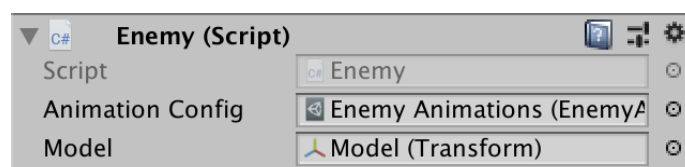rely on Unity's animation controller we'll create our own. Unity's animation controller is only needed to record animations.

2.1 Enemy Animator

`Enemy` can take care of animating itself, but the logic is still fairly complex, so we'll isolate it in a separate serializable `EnemyAnimator` struct type. It relies on types from the `UnityEngine.Animations` and `UnityEngine.Playables` namespaces, so we'll be using those.

```
using UnityEngine;
using UnityEngine.Animations;
using UnityEngine.Playables;

[System.Serializable]
public struct EnemyAnimator {}
```

Add a field for it to `Enemy`.

```
    EnemyAnimator animator;
```

To do its work `EnemyAnimator` needs three public methods. First `Configure` to set up the animation state, for which we need an `Animator` component and animation configuration. Second `Play` to start playing, and third `Stop` to stop playing.

```
    public void Configure (Animator animator, EnemyAnimationConfig config) {}

    public void Play () {}

    public void Stop () {}
```

In `Enemy`, stop playing in `Recycle` and start playing in `Initialize`.

```
    public override void Recycle () {
        animator.Stop();
        OriginFactory.Reclaim(this);
    }

    public void Initialize (
        float scale, float speed, float pathOffset, float health
    ) {
        …
        animator.Play();
    }
```

We could also configure the animator in `Initialize`, but we only need to do it once so let's do it in **Awake** instead. That way no unneeded extra configuration happens if enemies were to be reused at some point.

Rather than give all enemy prefabs an **Animator** component we'll create it programmatically here. Whatever we're animating has to be a child of the model, so add it to the first child of that object.

```
    void Awake () {
        animator.Configure(
            model.GetChild(0).gameObject.AddComponent<Animator>(),
            animationConfig
        );
    }
```

2.2 Playable Graphs

Controlling the animation state of an object is done via a playable graph, which exists in native code and not in C#. We can control it via a **PlayableGraph** struct, which contains a reference to the native data. A graph is created via the static **PlayableGraph**.Create method. All *Playables* are created in a similar way.

```
    PlayableGraph graph;

    public void Configure (Animator animator, EnemyAnimationConfig config) {
        graph = PlayableGraph.Create();
    }
```

Initially the graph is inert. We activate it by invoking `Play` on it and can stop it by invoking `Stop`. However, as we're not reusing enemies in this tutorial series we should invoke **Destroy** to get rid of the native graph data instead, otherwise it sticks around.

```
    public void Play () {
        graph.Play();
    }

    public void Stop () {
        graph.Destroy();
    }
```

Graphs can update themselves, but we should indicate how they should do this. We need animations linked to the game time, which is configured by invoking `SetTimeUpdateMode` with `DirectorUpdateMode.GameTime` after creating the graph.

```
        graph = PlayableGraph.Create();
        graph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);
```

To play an animation clip we first have to create a playable representation of it, via `AnimationClipPlayable.Create`. We have to provide the graph it belongs to and the animation clip as arguments.

```
        graph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);

        var clip = AnimationClipPlayable.Create(graph, config.Move);
```

Then we have to create an `AnimationPlayableOutput` for the graph, with an additional name and a reference to the animator component used for animation. Set the clip as the source for that output, via `SetSourcePlayable`. This will make our enemies bounce.

```
        var clip = AnimationClipPlayable.Create(graph, config.Move);

        var output = AnimationPlayableOutput.Create(graph, "Enemy", animator);
        output.SetSourcePlayable(clip);
```
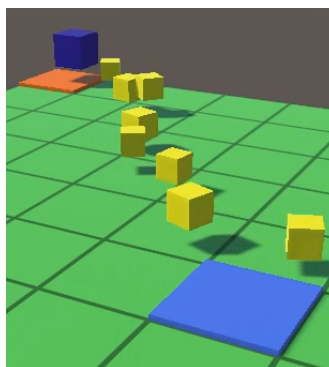

Bouncing Enemies.

The medium cubes end up animating in lockstep because they spawn once per second, which matches the animation duration.

Why doesn't code completion provide useful documentation for _Playables_?

The _Playables_ API mostly consists of extension methods that work for generic _Playable_ struct types. So there isn't much code documentation for specific types and methods. Because of this approach I'll also use `var` when storing _Playables_ in variables.

2.3 Adjusting Animation Speed

Our move animation loops every second, which isn't appropriate for all enemies. For some it should play faster and for others slower. We'll make that possible by adding a speed parameter to our `Play` method. Grab the graph's output at index zero via `GetOutput`, get its playable source via `GetSourcePlayable` and invoke `SetSpeed` on it with the provided speed.

```
public void Play (float speed) {
    graph.GetOutput(0).GetSourcePlayable().SetSpeed(speed);
    graph.Play();
}
```

Supply the required speed in `Enemy`.`Initialize`. Faster enemies require a faster animation because they cover more ground. Also, larger enemies make bigger steps and thus require a slower animation speed. So we make the move animation speed equal to the enemy's speed divided by its scale.

```
public void Initialize (
    float scale, float speed, float pathOffset, float health
) {
    …
    animator.Play(speed / scale);
}
```
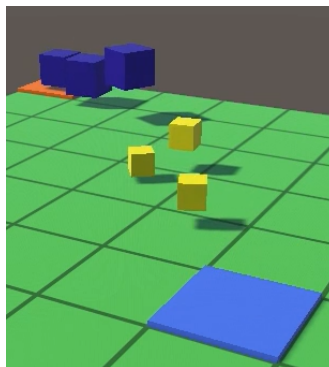

Adjusted speed.

As a bonus, because the medium cubes don't all have the exact same speed they no longer animate in exact lockstep.

2.4 Visualizing Playable Graphs

If you want to visually inspect the generated playable graphs you can do so by importing the *PlayableGraph Visualizer* package via *Window / Package Manager*. Its current version is 0.2.1, which is a preview version so you have to enable *Show preview packages* under *Advanced* to see it. After importing you can open the visualizer via *Window / Analysis / PlayableGraph Visualizer* and enter play mode. You won't be able to select specific graph instances because they don't have unique names, but it's enough to see the graph structure.
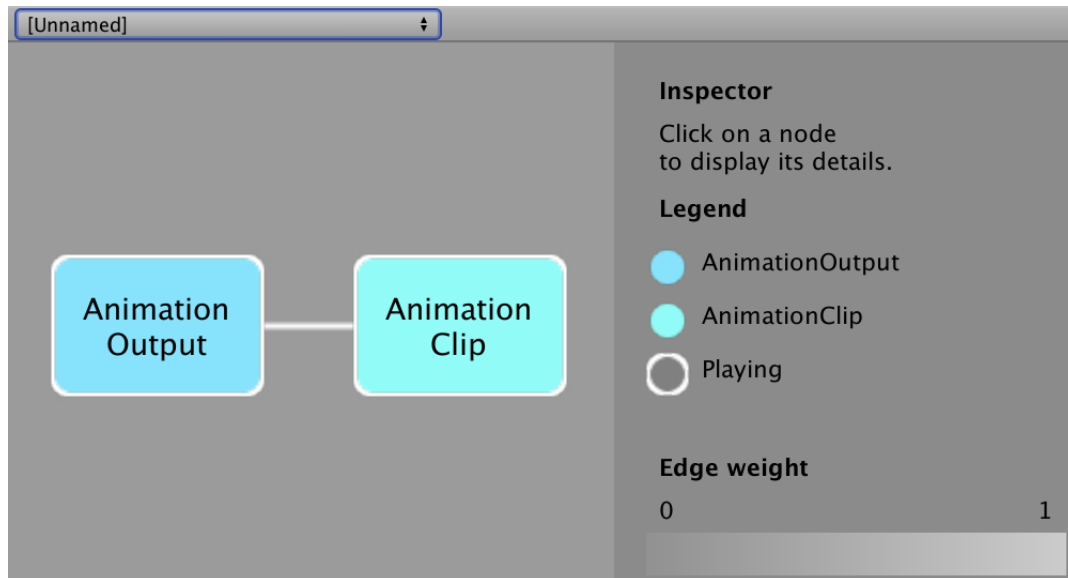
PlayableGraph visualizer.

One thing you'll notice is that while graphs are only created in play mode they stick around after play mode has been exited. That happens when enemies are destroyed on play exit. We can solve this by adding a public `Destroy` method to `EnemyAnimator` that destroys the graph. At this point we can also change `Stop` so it stops rather than destroys the graph, to support future reuse.

```
public void Stop () {
    graph.Stop();
}

public void Destroy () {
    graph.Destroy();
}
```

Add an `OnDestoy` method to `Enemy` that destroys the animator to always get rid of the graphs.

```
void OnDestroy () {
    animator.Destroy();
}
```
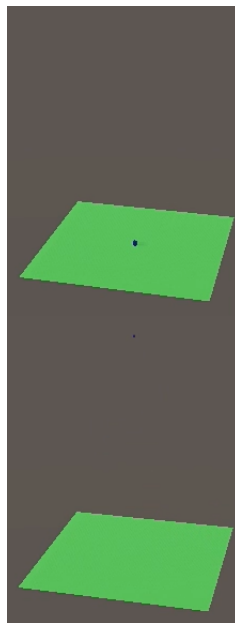
3 Intro and Outro

Bouncing enemies look more lively than sliding ones, but the animation clashes with the sudden appearance and disappearance of enemies when they are spawned and reach their destination. We can make that look much better by adding an intro and outro animation.

3.1 Animations

To create additional animations go back to the enemy instances set up for animation recording. Select *Cube* and then open the dropdown menu in the *Animation* window that's currently set to *Enemy Move*. Choose *Create New Clip...* twice, creating an *Enemy Intro* and *Enemy Outro* animation.

For the intro, set the scale and position at 0:00 to zero and to their original values at 0:30. Do it the other way around for the outro, but this time the duration is one second. Also, give it a lively twirl by increasing its vertical position to 1.25 and settings its Y rotation to 360°.

Intro and outro.

The animation control has also gained extra states for the new animations, which are disconnected from its graph. That's fine because we only use the animation controller to record animations.
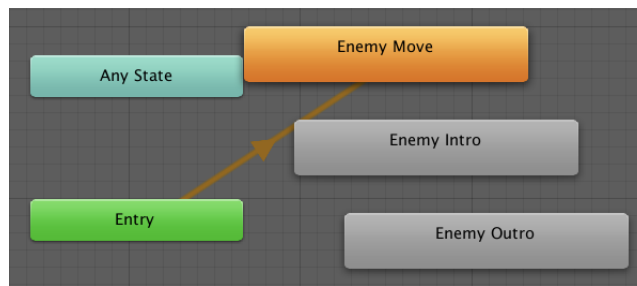
Animator with intro and outro states.

Add support for the intro and outro animations to `EnemyAnimationConfig`.

```
[SerializeField]
AnimationClip move = default, intro = default, outro = default;

public AnimationClip Move => move;

public AnimationClip Intro => intro;

public AnimationClip Outro => outro;
```
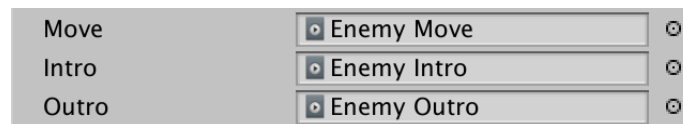

Configuration with intro and outro.

3.2 Mixing Animation Clips

To support multiple animations we have to add an animation mixer to `EnemyAnimator`. Give it an `AnimationMixerPlayable` field to keep track of it.

```
AnimationMixerPlayable mixer;
```

We now also have to create a mixer in `Configure`. Besides the graph, also provide the amount of animation clips—which is now 3—as an argument to its `Create` method. Then make the mixer the source for the output.

```
public void Configure (Animator animator, EnemyAnimationConfig config) {
    graph = PlayableGraph.Create();
    graph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);
    mixer = AnimationMixerPlayable.Create(graph, 3);

    var clip = AnimationClipPlayable.Create(graph, config.Move);

    var output = AnimationPlayableOutput.Create(graph, "Enemy", animator);
    output.SetSourcePlayable(mixer);
}
```

Each clip gets its own fixed index in the mixer. Let's define them with an enum type nested in `EnemyAnimator`. Make it public so `Enemy` can access it later.

```
public enum Clip { Move, Intro, Outro }
```

The easiest way to add a clip to a mixer is by invoking `ConnectInput` on the mixer with the clip's index and the playable clip as arguments. A third argument specifies the output index of the clip, which is always zero. Do this for all three clips in `Configure`.

```
public void Configure (Animator animator, EnemyAnimationConfig config) {
    graph = PlayableGraph.Create();
    graph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);
    mixer = AnimationMixerPlayable.Create(graph, 3);

    var clip = AnimationClipPlayable.Create(graph, config.Move);
    mixer.ConnectInput((int)Clip.Move, clip, 0);

    clip = AnimationClipPlayable.Create(graph, config.Intro);
    mixer.ConnectInput((int)Clip.Intro, clip, 0);

    clip = AnimationClipPlayable.Create(graph, config.Outro);
    mixer.ConnectInput((int)Clip.Outro, clip, 0);

    var output = AnimationPlayableOutput.Create(graph, "Enemy", animator);
    output.SetSourcePlayable(mixer);
}
```
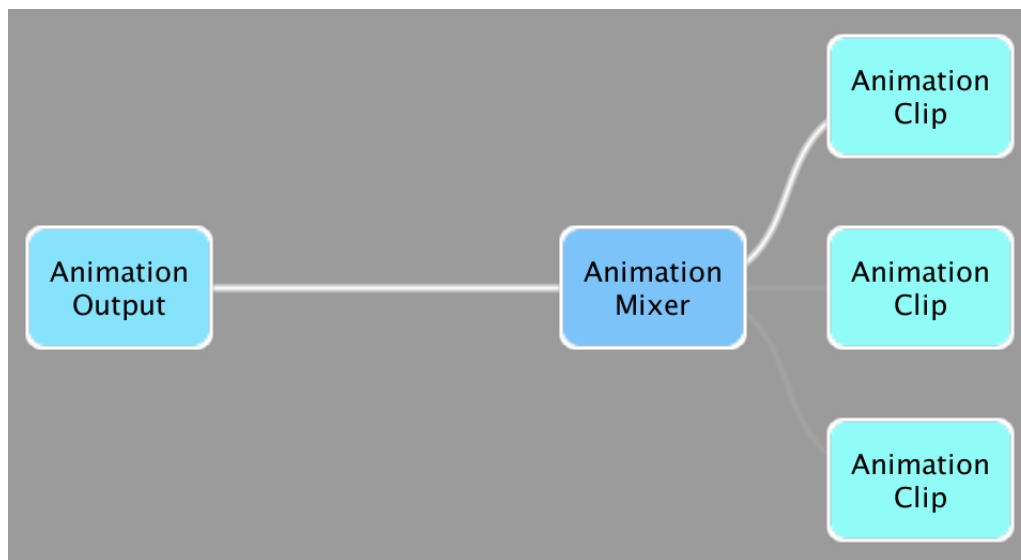

Visualizer showing a mixer with three clips.

3.3 Switching Between Clips

The mixer blends all its clips based on their weights, which are zero by default. We only need one active clip at a time, which we accomplish by setting its weight to 1 and all other weights to zero. It's handy to keep track of the currently active clip, so add a property for that. Make the getter public so `Enemy` can also access it.

```
public Clip CurrentClip { get; private set; }
```

Now replace the `Play` method with a more specific `PlayIntro` method. It doesn't need a speed, instead it invokes `SetInputWeight` on the mixer with the intro index to set the clip's weight to 1, sets the current clip, and plays the graph.

```
//public void Play (float speed) {
//	graph.GetOutput(0).GetSourcePlayable().SetSpeed(speed);
//	graph.Play();
//}

public void PlayIntro () {
	SetWeight(Clip.Intro, 1f);
	CurrentClip = Clip.Intro;
	graph.Play();
}

void SetWeight (Clip clip, float weight) {
	mixer.SetInputWeight((int)clip, weight);
}
```

Then add a `PlayMove` method with a speed parameter. It sets the current clip's weight to zero—in case you insert animations between intro and movement later—and the move clip's weight to 1, sets the speed, and updates the current clip. The playable handle for a specific clip can be retrieved by invoking `GetInput` on the mixer with the appropriate index.

```
public void PlayMove (float speed) {
	SetWeight(CurrentClip, 0f);
	SetWeight(Clip.Move, 1f);
	GetPlayable(Clip.Move).SetSpeed(speed);
	CurrentClip = Clip.Move;
}

Playable GetPlayable (Clip clip) {
	return mixer.GetInput((int)clip);
}
```

And also add a `PlayOutro` method that switches to the outro clip.

```
public void PlayOutro () {
    SetWeight(CurrentClip, 0f);
    SetWeight(Clip.Outro, 1f);
    CurrentClip = Clip.Outro;
}
```

3.4 Playing the Intro

Adjust Enemy.Initialize so it invokes PlayIntro instead of Play.

```
public void Initialize (
    float scale, float speed, float pathOffset, float health
) {
    …
    animator.PlayIntro();
}
```

We have to delay moving until the intro animation is done. EnemyAnimator can check this by grabbing the current clip of the mixer and invoking IsDone on it. Expose this via a property.

```
public bool IsDone => GetPlayable(CurrentClip).IsDone();
```

Now we have to check at the start of Enemy.GameUpdate whether we're currently playing the intro clip. If so and it's not done skip the rest of the method, otherwise invoke PlayMove with the speed and keep going.

```
public override bool GameUpdate () {
    if (animator.CurrentClip == EnemyAnimator.Clip.Intro) {
        if (!animator.IsDone) {
            return true;
        }
        animator.PlayMove(speed / Scale);
    }

    …
}
```

As this means that we delay updating the enemy position we have to make sure that it is set correctly in PrepareIntro.

```
void PrepareIntro () {
    positionFrom = tileFrom.transform.localPosition;
    transform.localPosition = positionFrom;
    …
}
```

We now get enemies that are stuck in a repeating intro animation. The first step to fix this is disabling the *Loop Time* option of the intro animation clip.
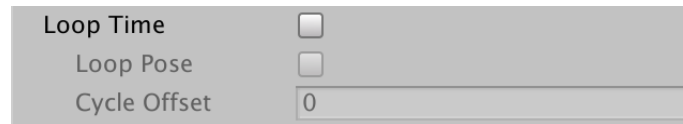
No looping.

But this is not enough. Because we're creating a playable graph ourselves we have to explicitly set the duration of non-looping clips if we need to detect when they're done. In **EnemyAnimator**.Configure, invoke SetDuration on the intro clip, providing the length of its clip as an argument.

```
clip = AnimationClipPlayable.Create(graph, config.Intro);
clip.SetDuration(config.Intro.length);
mixer.ConnectInput((int)Clip.Intro, clip, 0);
```
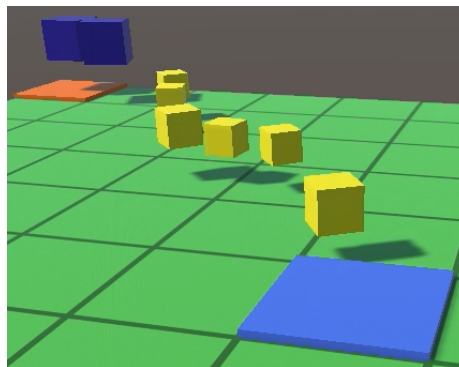

Intro then move.

Enemies now progress go from into to move, but there is a discontinuity as part of the movement gets skipped. That happens because time passes for all clips, no matter their weight. We can solve this by pausing the move clip when we create it in Configure and play it in PlayMove.

```
    public void Configure (Animator animator, EnemyAnimationConfig config) {
        …

        var clip = AnimationClipPlayable.Create(graph, config.Move);
        clip.Pause();
        mixer.ConnectInput((int)Clip.Move, clip, 0);


        …
    }

    …

    public void PlayMove (float speed) {
        SetWeight(CurrentClip, 0f);
        SetWeight(Clip.Move, 1f);
        //GetPlayable(Clip.Move).SetSpeed(speed);
        var clip = GetPlayable(Clip.Move);
        clip.SetSpeed(speed);
        clip.Play();
        CurrentClip = Clip.Move;
    }
```
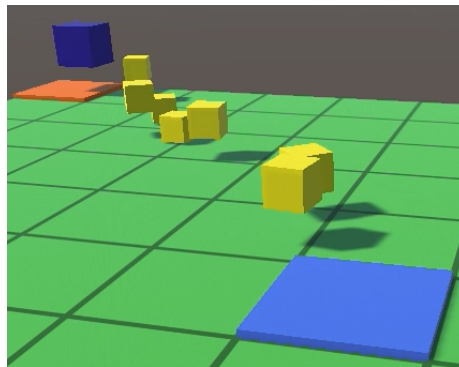

Move begins after intro.

3.5 **Playing the Outro**

The outro clip requires a similar treatment. Disable its *Loop Time* option and set its duration it in `Configure`. Also pause it initially, like the move animation.

```
        clip = AnimationClipPlayable.Create(graph, config.Outro);
        clip.SetDuration(config.Outro.length);
        clip.Pause();
        mixer.ConnectInput((int)Clip.Outro, clip, 0);
```

Play the clip in `PlayOutro`.

```
    public void PlayOutro () {
        SetWeight(CurrentClip, 0f);
        SetWeight(Clip.Outro, 1f);
        GetPlayable(Clip.Outro).Play()
        CurrentClip = Clip.Outro;
    }
```

To make the outro play, invoke `PlayOutro` in **Enemy**.GameUpdate instead if recycling it when the destination has been reached. Also return **true** so it keeps getting updated.

```
public override bool GameUpdate () {
    …

    progress += Time.deltaTime * progressFactor;
    while (progress >= 1f) {
        if (tileTo == null) {
            Game.EnemyReachedDestination();
            //Recycle();
            animator.PlayOutro();
            return true;
        }
        …
    }
    …
}
```

Now we also have to check whether the outro is playing at the start of GameUpdate. If so we're either done and can recycle or must keep playing and return **true**.

```
    if (animator.CurrentClip == EnemyAnimator.Clip.Intro) {
        if (!animator.IsDone) {
            return true;
        }
        animator.PlayMove(speed / Scale);
    }
    else if (animator.CurrentClip == EnemyAnimator.Clip.Outro) {
        if (animator.IsDone) {
            Recycle();
            return false;
        }
        return true;
    }
```
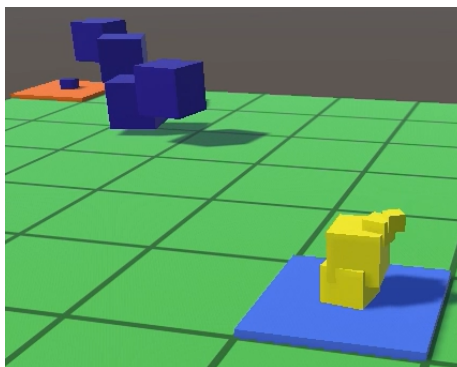

Intro, move, and outro.

4 Animation Transitions

The transition from intro to move is correct, but the transition from move to outro has problems. Whether move and outro animations align depends on the enemy's speed and distance traveled, which varies. The only way to get rid of the hard transition between these animations is to blend them.

4.1 Beginning a Transition

We blend between two animation by linearly interpolating their weights, the previous clip's decreasing from 1 and the current clip's increasing from zero. To keep track of this transition `EnemyAnimator` has to keep track of the previous clip and the progress of the transition.

```
Clip previousClip;

float transitionProgress;
```

Add a `BeginTransition` method with the next clip enum value as a parameter. It has to make the current clip the previous one, set the new current clip, set the transition progress to zero, and play the now current clip.

```
void BeginTransition (Clip nextClip) {
    previousClip = CurrentClip;
    CurrentClip = nextClip;
    transitionProgress = 0f;
    GetPlayable(nextClip).Play();
}
```

Invoke this method both in `PlayMove` and `PlayOutro` with the appropriate clip. Besides setting the move speed, that's all they now have to do.

```
public void PlayMove (float speed) {
    GetPlayable(Clip.Move).SetSpeed(speed);
    BeginTransition(Clip.Move);
}

public void PlayOutro () {
    BeginTransition(Clip.Outro);
}
```

We also blend from intro to move because they're not guaranteed to line up, that's just the case for our simple enemy animations.

4.2 Progressing a Transition

Progressing a transition needs to happen every game update, so add a public `GameUpdate` method to **EnemyAnimator**. Have it increase the progress by the time multiplied by some transition speed. The transition should be quick, so let's use 5 for a duration of 0.2 seconds.

If the transition is complete, set the current clip's weight to 1. Also set the previous clip's weight to zero and pause it. Otherwise make the weights equal to the progress and 1 minus the progress, respectively.

```
	const float transitionSpeed = 5f;

	…

	public void GameUpdate () {
		transitionProgress += Time.deltaTime * transitionSpeed;
		if (transitionProgress >= 1f) {
			SetWeight(CurrentClip, 1f);
			SetWeight(previousClip, 0f);
			GetPlayable(previousClip).Pause();
		}
		else {
			SetWeight(CurrentClip, transitionProgress);
			SetWeight(previousClip, 1f - transitionProgress);
		}
	}
```

This only has to be done when there is a transition in progress. We can use a progress value of −1 to indicate that there is no transition.

```
	public void GameUpdate () {
		if (transitionProgress >= 0f) {
			transitionProgress += Time.deltaTime * transitionSpeed;
			if (transitionProgress >= 1f) {
				transitionProgress = -1f;
				SetWeight(CurrentClip, 1f);
				SetWeight(previousClip, 0f);
			}
			else {
				SetWeight(CurrentClip, transitionProgress);
				SetWeight(previousClip, 1f - transitionProgress);
			}
		}
	}

	public void PlayIntro () {
		…
		transitionProgress = -1f;
	}
```
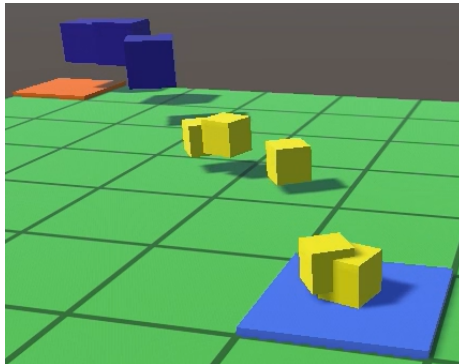
Invoke the animator's `GameUpdate` method at the start of **Enemy**.`GameUpdate` to enable transitions.

```
public override bool GameUpdate () {
    animator.GameUpdate();

    …
}
```


With animation transitions.

```
public override bool GameUpdate () {
    animator.GameUpdate();

    …
}
```

5 Dying Enemies

Intro, move, and outro animation now work and blend correctly. The next step is to add an animation for when an enemy dies.

5.1 Dying Animation

Create a new animation for a dying enemy. Like the outro animation, the dying animation can get rid of the enemy by reducing its scale to zero. Rather then adding a levitating whirl, give it a more appropriate animation, like rolling over. Let's increase the Z position to 0.5 in half a second while increasing the X rotation to 90° at the same time. Then drop the scale and Y position to zero in the next half second. When finished, add it to `EnemyAnimationConfig`.

```
[SerializeField]
AnimationClip
    move = default, intro = default, outro = default, dying = default;

public AnimationClip Move => move;

public AnimationClip Intro => intro;

public AnimationClip Outro => outro;

public AnimationClip Dying => dying;
```
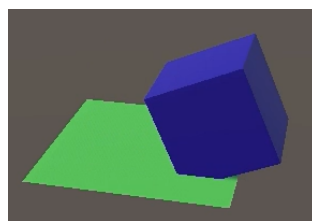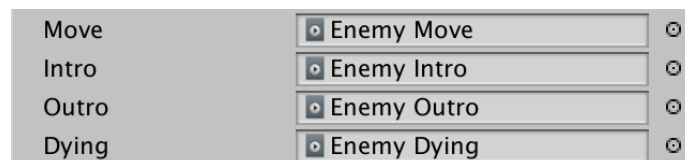

Dying animation.

Add support for it to `EnemyAnimator` as well, by adding a fourth value to the enum, creating its clip in `Configure`, and adding a `PlayDying` method that begins the appropriate transition.

```csharp
public enum Clip { Move, Intro, Outro, Dying }

…

public void Configure (Animator animator, EnemyAnimationConfig config) {
    graph = PlayableGraph.Create();
    graph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);
    mixer = AnimationMixerPlayable.Create(graph, 4);

    …

    clip = AnimationClipPlayable.Create(graph, config.Dying);
    clip.SetDuration(config.Dying.length);
    clip.Pause();
    mixer.ConnectInput((int)Clip.Dying, clip, 0);

    var output = AnimationPlayableOutput.Create(graph, "Enemy", animator);
    output.SetSourcePlayable(mixer);
}

…

public void PlayDying () {
    BeginTransition(Clip.Dying);
}
```

5.2 No More Instantaneous Death

Instead of immediately recycling in `Enemy`.GameUpdate when the health has been dropped to zero, invoke `PlayDying` and return **true**. As the dying clip comes after the outro clip, we can catch both cases by checking whether the current clip is at least the outro clip instead of an exact match.

```
public override bool GameUpdate () {
    animator.GameUpdate();

    if (animator.CurrentClip == EnemyAnimator.Clip.Intro) {
        …
    }
    else if (animator.CurrentClip >= EnemyAnimator.Clip.Outro) {
        if (animator.IsDone) {
            Recycle();
            return false;
        }
        return true;
    }

    if (Health <= 0f) {
        //Recycle();
        animator.PlayDying();
        return true;
    }

    …
}
```
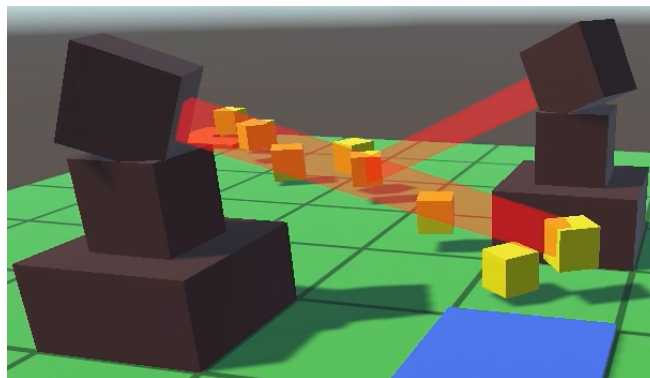

Enemy dying en route.

5.3 Only Target Moving Enemies

Towers are not aware of the state of the enemy, so will keep targeting it even though it is already dying. This is also the case for enemies that are playing their outro, even through they will no longer die. And enemies playing their intro also won't immediately die, though they could as soon as they start moving. To keep this simple and also efficient from a gameplay perspective, let's enforce that towers only target and damage enemies that are moving.

We can make it impossible to target an enemy by disabling its collider. Add a collider field to **Enemy** for this purpose. We could make it configurable via the editor, but let's give it a public setter property instead, which should only be invoked once.

```
Collider targetPointCollider;

public Collider TargetPointCollider {
    set {
        Debug.Assert(targetPointCollider == null, "Redefined collider!");
        targetPointCollider = value;
    }
}
```

TargetPoint is attached to the same game object that has the collider, so when it awakens grab the collider and assign it to the enemy.

```
void Awake () {
    …
    Enemy.TargetPointCollider = GetComponent<Collider>();
}
```

Disable the collider in **Enemy**.Initialize, as we begin with playing the intro.

```
public void Initialize (
    float scale, float speed, float pathOffset, float health
) {
    …
    animator.PlayIntro();
    targetPointCollider.enabled = false;
}
```

Also disable the collider in GameUpdate when playing the dying or outro animations, and enable it when playing the move animation.

```
        public override bool GameUpdate () {
            animator.GameUpdate();

            if (animator.CurrentClip == EnemyAnimator.Clip.Intro) {
                if (!animator.IsDone) {
                    return true;
                }
                animator.PlayMove(speed / Scale);
                targetPointCollider.enabled = true;
            }
            else if (animator.CurrentClip >= EnemyAnimator.Clip.Outro) {
                …
            }

            if (Health <= 0f) {
                animator.PlayDying();
                targetPointCollider.enabled = false;
                return true;
            }

            progress += Time.deltaTime * progressFactor;
            while (progress >= 1f) {
                if (tileTo == null) {
                    Game.EnemyReachedDestination();
                    animator.PlayOutro();
                    targetPointCollider.enabled = false;
                    return true;
                }
                …
            }
            …
        }
```

We also have to make sure that towers stop tracking targets that are no longer valid. Give **Enemy** a property that indicates whether it is a valid target, which is the case when it's moving.

```
        public bool IsValidTarget => animator.CurrentClip == EnemyAnimator.Clip.Move;
```
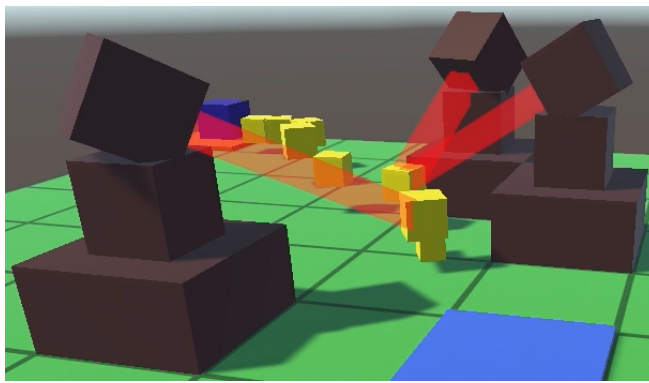
If this is not the case then **Tower**.TrackTarget must return **false**.

```
        protected bool TrackTarget (ref TargetPoint target) {
            if (target == null || !target.Enemy.IsValidTarget) {
                return false;
            }
            …
        }
```

Switching targets immediately.

6 Importing Models and Animations

While it is possible to create simple animations in the Unity editor, they're usually imported along with 3D models. You either created them yourself in a separate program or got them from somewhere else, like the asset store. As an example, I'll import the grenadier from Unity's *3D Game Kit*.
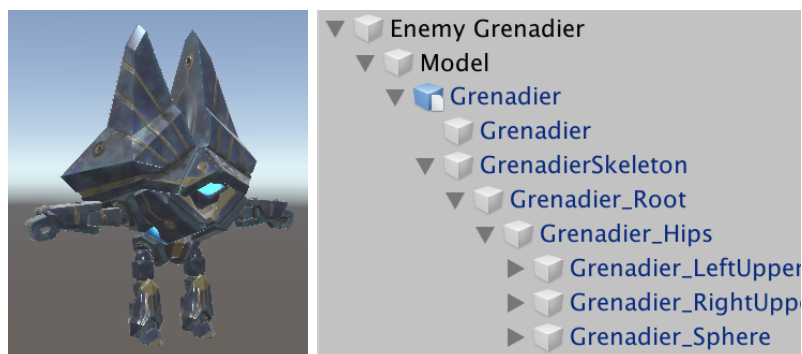
6.1 Grenadier

Go to the asset store and search for *3D Game Kit – Character Pack* from *Unity Technologies*. Download and then import it. You can suffice with importing only the grenadier model and its dependencies. Do not get the entire *3D Game Kit* package as it is huge and it will mess up your project.

The grenadier is far too large for our game. Scale it down by selecting the model, going to the *Model* tab and reducing its *Scale Factor* to 0.25. You also have to do this for all animations that we end up using, because the model will break apart otherwise.
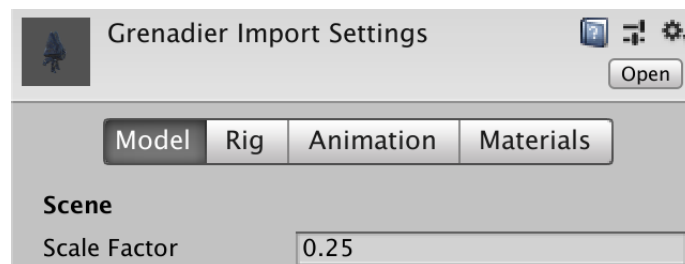
Grenadier scale factor set to 0.25.

Create an enemy prefab with the grenadier as its model, instead of a cube or sphere. Add the `TargetPoint` and collider to the *Grenadier_Sphere* object in the skeleton hierarchy, as that's its center of mass. Set the collider's scale to 0.125, because we haven't scaled the model as we already did that when importing.

Grenadier enemy.

At this point we can already use the grenadier enemy, for example by simply replacing the cube enemies in an existing scenario wave. But it looks rather silly, as the grenadiers bounce around in their default T pose and are levitating.
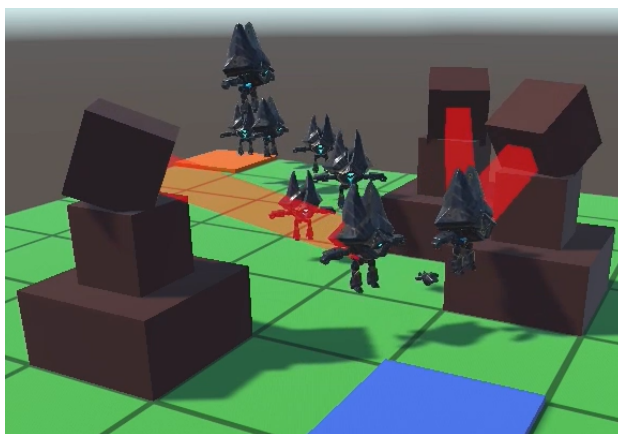
Grenadier with standard animations.

6.2 **Animation Selection**

Give the grenadier its own animation configuration. We can use the *GrenadierWalk* animation for movement, *GrenadierCloseRangeAttack* for both intro and outro, and *GrenadierDeath* for dying. All are found under the *AnimationClips* folder inside assets with an @ in front of their name. Make sure that the scale factor for all these assets is set to 0.25. Also, go to their *Animation* tab and remove all entries under *Events* as leaving them in will cause errors.

Unfortunately we cannot directly use the *GrenadierWalk* animation, because it has forward movement baked in, while we need an animation that walks in place. So duplicate that animation clip and select it. All we have to do it find the *Grenadier_Root : Position* row in the left part of the *Animation* window and delete it, via the *Remove Properties* option in its context menu.
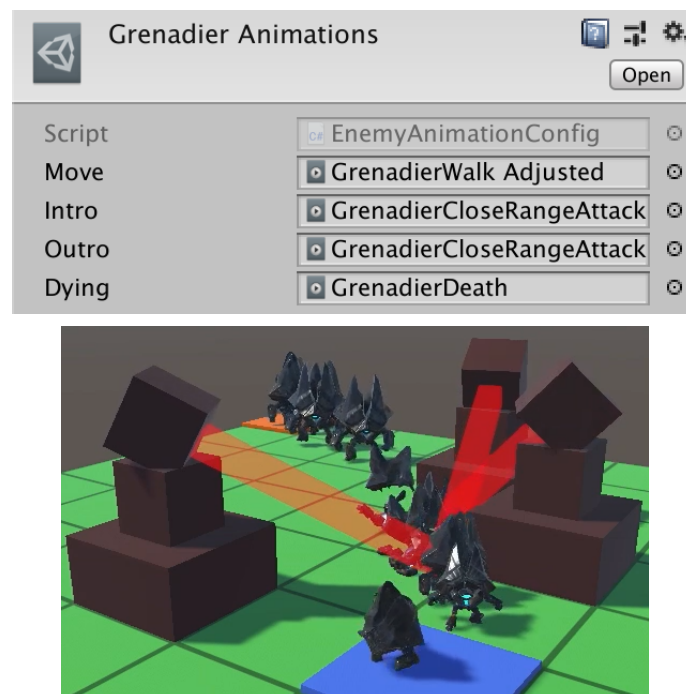
Grenadier animations.

6.3 Adjusting Walk Speed

The grenadier's walk speed doesn't match its in-game speed, which causes sliding feet even when moving straight ahead. This happens because the animation clip doesn't cover one unit per second. We'll compensate for this by adding a move animation speed configuration option to **EnemyAnimationConfig**, set to 1 by default.

```
[SerializeField]
float moveAnimationSpeed = 1f;

…

public float MoveAnimationSpeed => moveAnimationSpeed;
```

Factor this value into the move speed in **Enemy**.GameUpdate.

```
animator.PlayMove(animationConfig.MoveAnimationSpeed * speed / Scale);
```

In case of the grenadier we have to double the animation speed to make it line up.
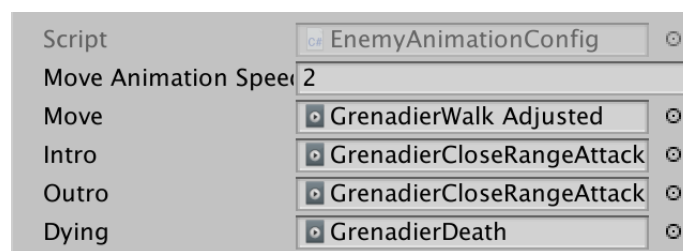
Move animation speed set to 2.

Note that the grenadier also has an animation for running. You could create a separate enemy prefab with a running animation for fast grenadiers.

6.4 Appearing and Disappearing

The grenadier doesn't have any animations that shrink or grow it. While it is possible to edit the existing animations to incorporate scaling, this is annoying work and needs to be redone each time new animations are imported. It's more convenient to create separate animations for appearing and disappearing and mix those with the existing ones.

Create two new animation, one that scales from 0 to 1 for and another that does the reverse, both in half a second. You can use the cube enemy set up for animation recording for this. Then add configuration options for them to `EnemyAnimationConfig`.

```
	[SerializeField]
	AnimationClip appear = default, disappear = default;
```

Select these animations for the grenadier config. Don't do this for the cubes and spheres, as they already appear and disappear on their own.

Dying	▣ GrenadierDeath	⊗
Appear	▣ Enemy Appear	⊗
Disappear	▣ Enemy Disappear	⊗
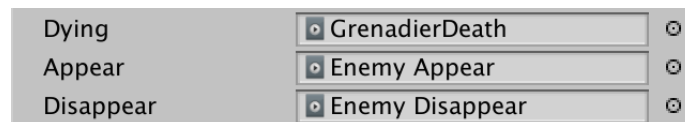
Appear and disappear animations configured.

Have `EnemyAnimator` keep track of whether it has an appear and disappear clips, separately for most flexibility. Also add them to the enum.

```
	public enum Clip { Move, Intro, Outro, Dying, Appear, Disappear }

	…

	bool hasAppearClip, hasDisappearClip;
```

In `Config`, increase the amount of clips to six if we have at least one of them. Then create the appropriate playable clips.

```
public void Configure (Animator animator, EnemyAnimationConfig config) {
    hasAppearClip = config.Appear;
    hasDisappearClip = config.Disappear;

    graph = PlayableGraph.Create();
    graph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);
    mixer = AnimationMixerPlayable.Create(
        graph, hasAppearClip || hasDisappearClip ? 6 : 4
    );

    …

    if (hasAppearClip) {
        clip = AnimationClipPlayable.Create(graph, config.Appear);
        clip.SetDuration(config.Appear.length);
        clip.Pause();
        mixer.ConnectInput((int)Clip.Appear, clip, 0);
    }

    if (hasDisappearClip) {
        clip = AnimationClipPlayable.Create(graph, config.Disappear);
        clip.SetDuration(config.Disappear.length);
        clip.Pause();
        mixer.ConnectInput((int)Clip.Disappear, clip, 0);
    }

    var output = AnimationPlayableOutput.Create(graph, "Enemy", animator);
    output.SetSourcePlayable(mixer);
}
```

When playing the intro, also play the appear clip at full weight if it exists. That means two clips have weight 1, which works fine as long as they don't both animate the same properties. So it only works correctly if the imported animation doesn't scale its root, which it typically doesn't.

```
public void PlayIntro () {
    …

    if (hasAppearClip) {
        GetPlayable(Clip.Appear).Play();
        SetWeight(Clip.Appear, 1f);
    }
}
```

When movement begins we no longer need the appear clip, so set its weight to zero in `PlayMove` if needed.

```
public void PlayMove (float speed) {
    GetPlayable(Clip.Move).SetSpeed(speed);
    BeginTransition(Clip.Move);

    if (hasAppearClip) {
        SetWeight(Clip.Appear, 0f);
    }
}
```

When playing the outro or dying animation we now also have to play the disappear clip if it exists. But we have to delay that clip—we assume that the disappear clip is the shortest—so both end at the same time. That's done by invoking `SetDelay` on the clip with a duration equal to the other clip's duration minus the disappear duration.

```
public void PlayOutro () {
    BeginTransition(Clip.Outro);

    if (hasDisappearClip) {
        PlayDisappearFor(Clip.Outro);
    }
}

public void PlayDying () {
    BeginTransition(Clip.Dying);

    if (hasDisappearClip) {
        PlayDisappearFor(Clip.Dying);
    }
}

…

void PlayDisappearFor (Clip otherClip) {
    var clip = GetPlayable(Clip.Disappear);
    clip.Play();
    clip.SetDelay(GetPlayable(otherClip).GetDuration() - clip.GetDuration());
    SetWeight(Clip.Disappear, 1f);
}
```
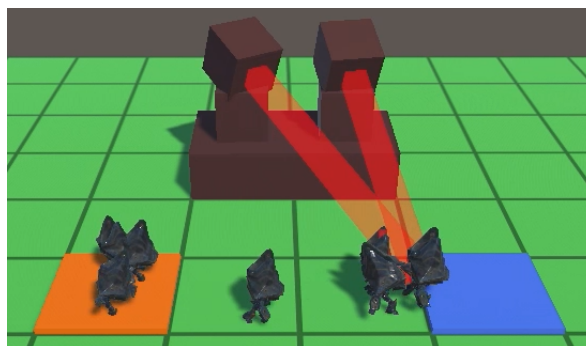

Appearing and disappearing.

Surviving a Hot Reload

The only problem with using `PlayableGraph` is that is isn't serializable. This isn't an issue in a build, but animations will stop in the editor when a hot reload happens. As enemies rely on detecting the end of animations to progress, they can become stuck. So it's not just a visual glitch. We have to recover from a hot reload to keep the game functional.

7.1 Recreating the Playable Graph

`EnemyAnimator` is serializable, but its graph becomes nonfunctional after the native data is lost during it hot reload. We can detect this by invoking `IsValid` on the graph. Wrap that in a public property so the enemy can also detect it. We only need this in the editor, so we can make the code conditional on that.

```
#if UNITY_EDITOR
    public bool IsValid => graph.IsValid();
#endif
```

To restore the animation state after a hot reload we have to create a new graph. Add a `RestorAfterHotReload` method for that, which invokes `Configure`, sets the move speed, sets the current clip's weight to 1, and plays that clip and the graph. This doesn't recover transitions, but those are purely cosmetic and the game freezes during a hot reload anyway.

```
#if UNITY_EDITOR
    public void RestoreAfterHotReload (
        Animator animator, EnemyAnimationConfig config, float speed
    ) {
        Configure(animator, config);
        GetPlayable(Clip.Move).SetSpeed(speed);
        var clip = GetPlayable(CurrentClip);
        clip.Play();
        SetWeight(CurrentClip, 1f);
        graph.Play();
    }
#endif
```

`Enemy`.`GameUpdate` now has to restore the animator if it isn't valid before it does anything else.

```
    public override bool GameUpdate () {
#if UNITY_EDITOR
        if (!animator.IsValid) {
            animator.RestoreAfterHotReload(
                model.GetChild(0).GetComponent<Animator>(),
                animationConfig,
                animationConfig.MoveAnimationSpeed * speed / Scale
            );
        }
#endif
        animator.GameUpdate();

        …
    }
```

7.2 Restoring Clip Time

Enemies now keep their animation, but its time gets set back to zero. To keep the time **EnemyAnimator** has to keep track of it and set it when restoring. The time is kept track of with a double instead of a float, for higher precision.

```
#if UNITY_EDITOR
    double clipTime;
#endif
```

```
    public void RestoreAfterHotReload (
        Animator animator, EnemyAnimationConfig config, float speed
    ) {
        Configure(animator, config);
        GetPlayable(Clip.Move).SetSpeed(speed);
        SetWeight(CurrentClip, 1f);
        var clip = GetPlayable(CurrentClip);
        clip.SetTime(clipTime);
        clip.Play();
        graph.Play();
    }
```

To keep the time up to date it has to be retrieved at the end of `GameUpdate`.

```
    public void GameUpdate () {
        …
#if UNITY_EDITOR
        clipTime = GetPlayable(CurrentClip).GetTime();
#endif
    }
```

7.3 Appear and Disappear Restoration

We can also restore the appear animation. If we're restoring the intro clip and the appear clip exists, then activate the appear clip with the same time as the current clip.

```
    public void RestoreAfterHotReload (
        Animator animator, EnemyAnimationConfig config, float speed
    ) {
        …
        if (CurrentClip == Clip.Intro && hasAppearClip) {
            clip = GetPlayable(Clip.Appear);
            clip.SetTime(clipTime);
            clip.Play();
            SetWeight(Clip.Appear, 1f);
        }
    }
```

The disappear clip works the same, but when we're restoring the outro and dying animation. In this case the disappear delay has to be reduced by the current clip time. If the delay is still positive then that's the remaining delay. If it's negative then that means the disappear animation was already playing and its time is equal to the negated delay.

```
if (CurrentClip == Clip.Intro && hasAppearClip) {
    …
}
else if (CurrentClip >= Clip.Outro && hasDisappearClip) {
    clip = GetPlayable(Clip.Disappear);
    clip.Play();
    double delay =
        GetPlayable(CurrentClip).GetDuration() -
        clip.GetDuration() -
        clipTime;
    if (delay >= 0f) {
        clip.SetDelay(delay);
    }
    else {
        clip.SetTime(-delay);
    }
    SetWeight(Clip.Disappear, 1f);
}
```

This concludes the Tower Defense tutorial series. You can use it as a starting point for your own game or turn it into something else. You could add sound, a GUI, save/load functionality, more tower types, other game tile content, and so on.

How do I make the graph work with enemy reuse?

Before playing the graph again you'll have to set the time all clips to zero and pause them. The weights of the last active clips must also become zero. Finally, the done state of non-looping clips has to be reset, by invoking `SetDone(false)` on them.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick