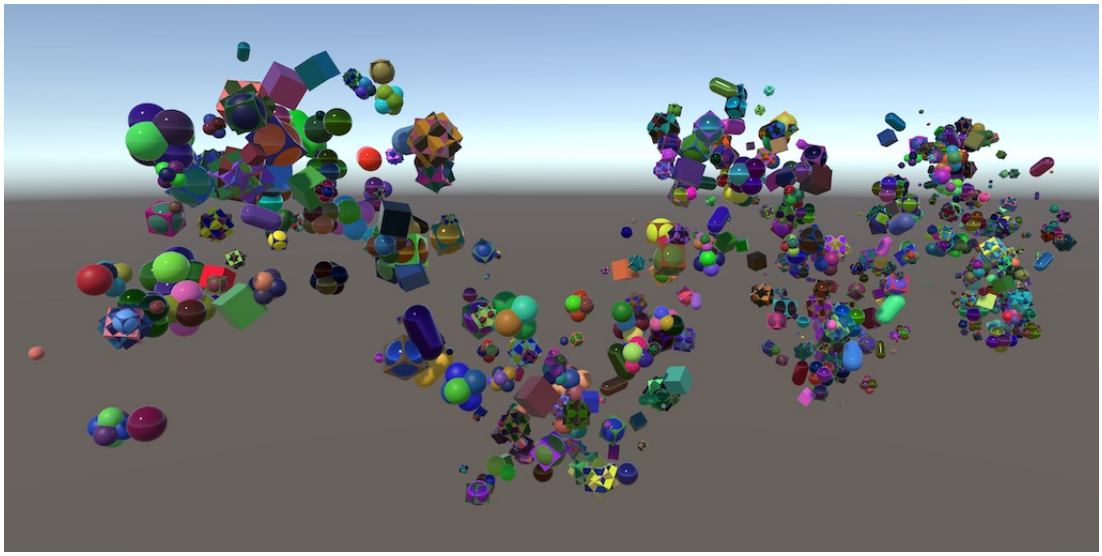
Shape Behavior Modular Functionality

Define abstract and concrete behavior for shapes.
Only include behavior when needed.
Create a generic method and class.
Use conditional compilation.
Add a method to an enumeration.
Make shapes oscillate.

This is the ninth tutorial in a series about Object Management. It adds support for modular behavior to shapes.

This tutorial is made with Unity 2017.4.12f1.



Shapes doing their thing.

1 Behavior Components

Currently, all shapes move and rotate, but that's not the only thing that they could do. We could come up with different behavior that we'd like shapes to exhibit. To make shapes do something else, we just have to add code for it to `Shape.GameUpdate`. But if we define lots of behavior, then that method would become quite large. Also, we might not want all shapes to behave the same. We could use toggles to control what a shape does, but that would bloat `Shape` with toggles and configuration options for all possible behavior. Ideally, the behavior is modular and can be defined in isolation. That's exactly what Unity's `MonoBehaviour` offers, so it makes sense to implement each behavior pattern as its own Unity component.

1.1 Abstract Behavior

Create a new `ShapeBehavior` component script and have it extend `MonoBehaviour`, as usual. This will be the base class for our behavior, which we'll extend with concrete behavior, like movement. The base `ShapeBehavior` type shouldn't be instantiated, because it doesn't do anything on its own. To enforce this, mark the class as `abstract`.

> **Why not name it `ShapeBehaviour`?**
>
> Unity uses the British spelling for its `MonoBehaviour` class, which deviates from the otherwise consistent usage of American spelling. We're defining our own behavior base, so I stick to the American spelling.

```
using UnityEngine;

public abstract class ShapeBehavior : MonoBehaviour {}
```

Just like with `Shape`, we won't rely on separate `Update` methods but instead use our own `GameUpdate` method, so add it to `ShapeBehavior`. But `ShapeBehavior` just defines common functionality, not an actual implementation. So we'll only define the method signature, followed by a semicolon instead of a code block. That defines an abstract method, which has to be implemented by classes that extend `ShapeBehavior`.

```
	public void GameUpdate ();
```

Abstract methods must be defined as such explicitly, with the `abstract` keyword.

```
	public abstract void GameUpdate ();
```

Also, the behavior acts on a shape, so we'll add one as a parameter. That way we don't have to keep track of it with a field.

```
	public abstract void GameUpdate (Shape shape);
```

Besides that, each shape behavior will probably have configuration and state, which we'll have to save and load. So add abstract `Save` and `Load` methods too.

```
	public abstract void Save (GameDataWriter writer);

	public abstract void Load (GameDataReader reader);
```

1.2 Movement

Our first concrete shape behavior component will be about simple linear movement. It'll function exactly like the movement that we currently have, just implemented in a separate class. Create a `MovementShapeBehavior` script that extends `ShapeBehavior`. It needs a `Velocity` vector property that it uses in `GameUpdate` to adjust the shape's position, and it must save and load it too.

```
using UnityEngine;

public class MovementShapeBehavior : ShapeBehavior {

	public Vector3 Velocity { get; set; }

	public override void GameUpdate (Shape shape) {
		shape.transform.localPosition += Velocity * Time.deltaTime;
	}

	public override void Save (GameDataWriter writer) {
		writer.Write(Velocity);
	}

	public override void Load (GameDataReader reader) {
		Velocity = reader.ReadVector3();
	}
}
```

1.3 Rotation

Do the same for rotation, creating a `RotationShapeBehavior` class that rotates with an `AngularVelocity` vector property.

```
using UnityEngine;

public class RotationShapeBehavior : ShapeBehavior {

	public Vector3 AngularVelocity { get; set; }

	public override void GameUpdate (Shape shape) {
		shape.transform.Rotate(AngularVelocity * Time.deltaTime);
	}

	public override void Save (GameDataWriter writer) {
		writer.Write(AngularVelocity);
	}

	public override void Load (GameDataReader reader) {
		AngularVelocity = reader.ReadVector3();
	}
}
```

1.4 Adding Behavior When Needed

In `SpawnZone.SpawnShape`, add these behavior components to the shape and set their properties, instead of the properties of the shape itself.

```
	public virtual Shape SpawnShape () {
		…

		var rotation = shape.gameObject.AddComponent<RotationShapeBehavior>();
		rotation.AngularVelocity =
			Random.onUnitSphere * spawnConfig.angularSpeed.RandomValueInRange;

		Vector3 direction;
		switch (spawnConfig.movementDirection) {
			…
		}
		var movement = shape.gameObject.AddComponent<MovementShapeBehavior>();
		movement.Velocity = direction * spawnConfig.speed.RandomValueInRange;
		return shape;
	}
```

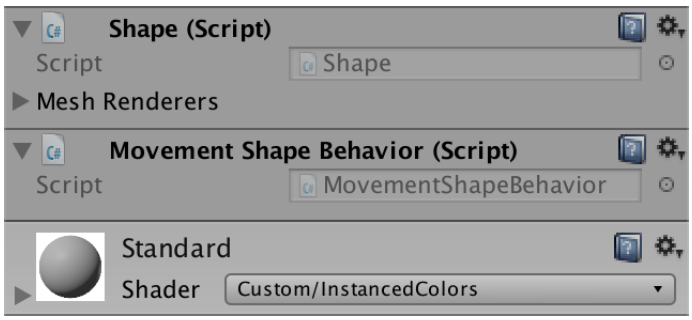
Is it acceptable to use `var` here?

There are no hard rules about when to use `var` instead of an explicit variable type, as long as the compiler can figure it out. My rule of thumb is that the type should be mentioned explicitly somewhere in the assignment. A constructor method invocation is the best example, but I also consider `AddComponent<RotationShapeBehavior>` explicit enough.

A benefit of using components for isolated bits of behavior is that we can omit them when they aren't needed. That way we can avoid some unnecessary work. In the case of movement and rotation, we only have to add their behavior if they would have a nonzero speed.

```
		float angularSpeed = spawnConfig.angularSpeed.RandomValueInRange;
		if (angularSpeed != 0f) {
			var rotation = shape.gameObject.AddBehavior<RotationShapeBehavior>();
			rotation.AngularVelocity = Random.onUnitSphere * angularSpeed;
		}

		float speed = spawnConfig.speed.RandomValueInRange;
		if (speed != 0f) {
			Vector3 direction;
			switch (spawnConfig.movementDirection) {
				…
			}
			var movement = shape.gameObject.AddBehavior<MovementShapeBehavior>();
			movement.Velocity = direction * speed;
		}
```

If the spawn zone has a speed range from zero to some nonzero value, then it is extremely unlike that we'd end up with a speed of zero. But if the spawn zone's speed range is set to zero—because we didn't want any movement or rotation at all—then the behavior will always be omitted.



Shape with movement but without rotation.

1.5 Adding Behavior

We're now adding the required components to shapes, but they've stopped moving and rotating. That's because we're not invoking the required `GameUpdate` methods yet. That's the responsibility of `Shape`, and to do so it will need to keep track of its behavior components. Give it a list field for that purpose.

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

public class Shape : PersistableObject {

	…

	List<ShapeBehavior> behaviorList = new List<ShapeBehavior>();

	…
}
```

Next, we need a method to add a behavior instance to the shape. The most straightforward approach is a public `AddBehavior` method with the behavior as a parameter, which adds it to the list. That method has to be invoked either before or after adding the component to the shape's game object.

```
	public void AddBehavior (ShapeBehavior behavior) {
		behaviorList.Add(behavior);
	}
```

We can make this more convenient by moving the `AddComponent` invocation inside the `AddBehavior` method, having it return the new behavior. To make that work, we have to turn `AddBehavior` into a generic method, just like `AddComponent`. That's done by attaching a type placeholder to the method name, between angle brackets. The placeholder name doesn't matter but is usually named `T` as a shorthand for template type.

```
	public T AddBehavior<T> () {
		T behavior = gameObject.AddComponent<T>();
		behaviorList.Add(behavior);
		return behavior;
	}
```

However, it only works when `AddBehavior` is used with a type that extends `ShapeBehavior`. To enforce that constraint, write `where T : ShapeBehavior` after the method name.

```
	public T AddBehavior<T> () where T : ShapeBehavior {
		…
	}
```

Now we can simply replace `AddComponent` with `AddBehavior` in `SpawnZone.SpawnShape`.

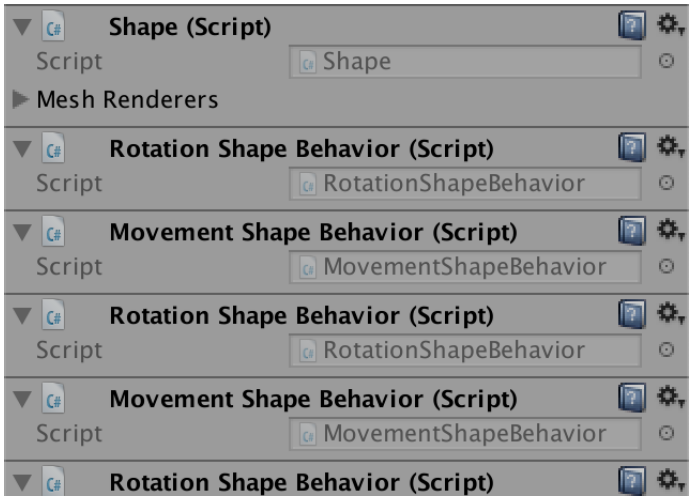
```
		var rotation = shape.AddBehavior<RotationShapeBehavior>();
		…
		var movement = shape.AddBehavior<MovementShapeBehavior>();
```

Finally, we can remove the old code from `Shape.GameUpdate` and instead invoke the `GameUpdate` method of all its behavior, with itself as the argument. That will make the shapes move and rotate again.

```
	public void GameUpdate () {
		//transform.Rotate(AngularVelocity * Time.deltaTime);
		//transform.localPosition += Velocity * Time.deltaTime;
		for (int i = 0; i < behaviorList.Count; i++) {
			behaviorList[i].GameUpdate(this);
		}
	}
```

1.6 Removing Behavior

Adding behavior each time we spawn a shape work fine when instantiating new shapes, but leads to duplicate behavior components when shapes get recycled.



Behavior duplicates.

The quickest way to fix this is to simply destroy all behavior and clear the list when a shape is recycled. That means that we'll be allocating memory even when reusing shapes, but we'll deal with that later.

```
	public void Recycle () {
		for (int i = 0; i < behaviorList.Count; i++) {
			Destroy(behaviorList[i]);
		}
		behaviorList.Clear();
		OriginFactory.Reclaim(this);
	}
```

1.7 Saving

When saving a shape, we now also have to save all its behavior. That means that we change our save file format, so increase `Game.saveVersion` to 6.

```
	const int saveVersion = 6;
```

Just like with the list of shapes, we have to save the type of each behavior in the list. Once again, we can use an identifier number for that. But this time we're dealing with class types, not prefab array indices. We have a fixed amount of behavior types, two at the moment. Let's define a `ShapeBehaviorType` enumeration to identify movement and rotation, put in its own script file.

```
public enum ShapeBehaviorType {
	Movement,
	Rotation
}
```

Next, add an abstract `BehaviorType` getter property to `ShapeBehavior`, so we can get a hold of the correct enumeration value.

```
	public abstract ShapeBehaviorType BehaviorType { get; }
```

The implementation of the property is simple. `MovementShapeBehavior` always returns `ShapeBehaviorType.Movement`.

```
	public override ShapeBehaviorType BehaviorType {
		get {
			return ShapeBehaviorType.Movement;
		}
	}
```

And `RotationShapeBehavior` always returns `ShapeBehaviorType.Rotation`.

```
	public override ShapeBehaviorType BehaviorType {
		get {
			return ShapeBehaviorType.Rotation;
		}
	}
```

Now we can write the behavior list in `Shape.Save`. For each behavior, first write its type, cast to an integer, then invoke its own `Save` method. That replaces the writing of the old movement and rotation data.

```
	public override void Save (GameDataWriter writer) {
		…
		//writer.Write(AngularVelocity);
		//writer.Write(Velocity);
		writer.Write(behaviorList.Count);
		for (int i = 0; i < behaviorList.Count; i++) {
			writer.Write((int)behaviorList[i].BehaviorType);
			behaviorList[i].Save(writer);
		}
	}
```

1.8 Loading

When loading shape behavior, we now have to read and enumeration value and then add the correct behavior component to the shape. Add a private `AddBehavior` method to `Shape` for that, with a `ShapeBehaviorType` parameter. Have it use a `switch` statement to add the correct behavior component. Also have it return `null` when we fail to add the correct type. If we ever end up with a null-reference exception after invoking this method, it means that we forgot to include a behavior type in the switch.

```
	ShapeBehavior AddBehavior (ShapeBehaviorType type) {
		switch (type) {
			case ShapeBehaviorType.Movement:
				return AddBehavior<MovementShapeBehavior>();
			case ShapeBehaviorType.Rotation:
				return AddBehavior<RotationShapeBehavior>();
		}
		Debug.LogError("Forgot to support " + type);
		return null;
	}
```

Replace the old code for reading the movement and rotation data with reading the behavior list. For each behavior, read its identifier integer, cast it to `ShapeBehaviorType`, invoke `AddBehavior` with it, and then load the rest of the behavior's data.

```
	public override void Load (GameDataReader reader) {
		…
		//AngularVelocity =
		//	reader.Version >= 4 ? reader.ReadVector3() : Vector3.zero;
		//Velocity = reader.Version >= 4 ? reader.ReadVector3() : Vector3.zero;
		if (reader.Version >= 6) {
			int behaviorCount = reader.ReadInt();
			for (int i = 0; i < behaviorCount; i++) {
				AddBehavior((ShapeBehaviorType)reader.ReadInt()).Load(reader);
			}
		}
	}
```

That works for file version 6 and newer, but file versions 4 and 5 still contain the old movement and rotation data. To remain backwards compatible, read that data when it exists and add the necessary behavior. We don't have to do this for even older versions, because those only contain motionless shapes.

```
		if (reader.Version >= 6) {
			…
		}
		else if (reader.Version >= 4) {
			AddBehavior<RotationShapeBehavior>().AngularVelocity =
				reader.ReadVector3();
			AddBehavior<MovementShapeBehavior>().Velocity = reader.ReadVector3();
		}
```

The `AngularVelocity` and `Velocity` properties of **`Shape`** are no longer used at this point, so should be removed.

```
        //public Vector3 AngularVelocity { get; set; }

        //public Vector3 Velocity { get; set; }
```

2 Recycling Behavior

Because we add shape behavior components each time we spawn a shape and later destroy the behavior, we end up allocating memory all the time. The whole point of recycling shapes was to minimize memory allocations, so we have to find a way to recycle shape behavior too.

Unity components cannot be detached from their game object, thus they cannot be placed in a pool to be attached to a different game object later. If we want to keep using Unity components, then once we add a behavior to a shape it cannot be removed. It's possible to work with that restriction, for example by just not destroying unused components and checking whether they already exist before adding them when needed later. Or by making factories aware of shape behavior, requiring complex pooling. Those solutions are not ideal because we end up fighting against Unity's component architecture, instead of taking advantage of it. The simple solution is to just not use Unity components for shape behavior.

2.1 No Longer Unity Components

To not have `ShapeBehavior` be a Unity component, simply have it not extend `MonoBehaviour`. It doesn't need to extend anything.

```
//public abstract class ShapeBehavior : MonoBehaviour {
public abstract class ShapeBehavior {
	…
}
```

Now we can no longer use `AddComponent` in `Shape.AddBehavior<T>`. Instead, we have to create a regular object instance, by invoking the type's default constructor method.

```
	public T AddBehavior<T> () where T : ShapeBehavior {
		T behavior = new T();
		behaviorList.Add(behavior);
		return behavior;
	}
```

Although classes implicitly have a public default constructor method when no explicit construct method is defined, their existence isn't guaranteed. So we have to constrain our template type further by explicitly requiring the existence of a construct method without parameters. That's done by adding `new()` to the list of constraints for `T`.

```
	public T AddBehavior<T> () where T : ShapeBehavior, new() {
		…
	}
```

We can also no longer destroy behavior in `Shape.Recycle`. Instead, we'll only clear the list. The unused objects will be cleaned up by the garbage collector at some point. But the idea is that we'll recycle the behavior, so keep the loop even though it does nothing right now.

```
	public void Recycle () {
		for (int i = 0; i < behaviorList.Count; i++) {
			//Destroy(behaviorList[i]);
		}
		behaviorList.Clear();
		OriginFactory.Reclaim(this);
	}
```

2.2 Behavior Pools

To recycle behavior, we have to put it in pools. Each behavior has its own type, so should get its own pool. We'll create a generic `ShapeBehaviorPool<T>` class for this purpose. The type restriction is the same as before. As these pools exist per type, we don't have bother with creating instances of them. Instead, we can make do with a static class. That means that the pools won't survive hot reloads, but that's fine.

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

public static class ShapeBehaviorPool<T> where T : ShapeBehavior, new() {}
```

This time, we'll use a stack to keep track of the unused behavior, so add a static `Stack<T>` field to the class, immediately initializing it.

```
	static Stack<T> stack = new Stack<T>();
```

> **What's a stack?**
>
> It's like a list, except you can only add to and remove from the top, via pushing and popping. Unity doesn't serialize stacks, but that's fine in this case.

Give the pool a `Get` and a `Reclaim` method. They work just like those of `ShapeFactory`, except that they're a lot simpler. When getting a behavior, pop it from the stack if it's not empty, otherwise return a new instance. When reclaiming, push the behavior on the stack.

```
	public static T Get () {
		if (stack.Count > 0) {
			return stack.Pop();
		}
		return new T();
	}

	public static void Reclaim (T behavior) {
		stack.Push(behavior);
	}
```

2.3 Returning to the Correct Pool

Add an abstract `Recycle` method to `ShapeBehavior` to make recycling possible.

```
	public abstract void Recycle ();
```

In the case of `MovementShapeBehavior`, have the pool with the correct template type reclaim it.

```
	public override void Recycle () {
		ShapeBehaviorPool<MovementShapeBehavior>.Reclaim(this);
	}
```

Do the same for `RotationShapeBehavior`.

```
	public override void Recycle () {
		ShapeBehaviorPool<RotationShapeBehavior>.Reclaim(this);
	}
```

2.4 Sealed Classes

Unlike shape prefabs, each shape behavior has its own type, thus all code is strongly-typed. It is not possible for a behavior to be added to the wrong pool. However, that is only true when each behavior only extends `ShapeBehavior`. Technically, it is possible to extend another behavior, for example some weird movement type that extends `MovementShapeBehavior`. Then it would be possible to add an instance of that behavior to the `ShapeBehaviorPool<MovementShapeBehavior>` pool, instead of its own type's pool. To prevent that, we can make it impossible to extend `MovementShapeBehavior`, by adding the `sealed` keyword to it.

```
public sealed class MovementShapeBehavior : ShapeBehavior { … }
```

Do the same for `RotationShapeBehavior`.

```
public sealed class RotationShapeBehavior : ShapeBehavior { … }
```

2.5 Using the Pools

To use the pools, invoke `ShapeBehaviorPool<T>.Get` in `Shape.AddBehavior<T>` instead of always creating a new object instance.

```
	public T AddBehavior<T> () where T : ShapeBehavior, new() {
		T behavior = ShapeBehaviorPool<T>.Get();
		behaviorList.Add(behavior);
		return behavior;
	}
```

And to finally enable behavior reuse, recycle them in `Shape.Recycle`.

```
	public void Recycle () {
		for (int i = 0; i < behaviorList.Count; i++) {
			behaviorList[i].Recycle();
		}
		behaviorList.Clear();
		OriginFactory.Reclaim(this);
	}
```

2.6 Surviving a Hot Reload

A downside of not using Unity components is that our shape behavior no longer survives hot reloads. When the recompilation is finished, all behavior is gone. This isn't an issue for builds, but can be annoying while working in the editor.

Making the behavior serializable is not enough, because Unity will try to deserialize a list of abstract `ShapeBehavior` instances per shape, because the list's type is `List<ShapeBehavior>`.

What we can do is have `ShapeBehavior` extend `ScriptableObject`. That effectively turns our behavior instances into runtime-only assets, which Unity can serialize correctly.

```
public abstract class ShapeBehavior : ScriptableObject { … }
```

That appears to work, but Unity will complain about us directly invoking the constructor method to create new asset instances, instead of using `ScriptableObject.CreateInstance`. Adjust `ShapeBehaviorPool.Get` to do it the correct way.

```
	public static T Get () {
		if (stack.Count > 0) {
			return stack.Pop();
		}
		return ScriptableObject.CreateInstance<T>();
	}
```

Now behavior that is in use by shapes survives hot reloads. But the pools don't survive and references to reclaimed behavior are lost. That's isn't a big problem, but it is possible to recreate the pools.

First, add a public boolean `IsReclaimed` property to `ShapeBehavior`.

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

Second, set this property to true in `ShapeBehaviorPool.Reclaim` and to false in `Get` after popping.

```
	public static T Get () {
		if (stack.Count > 0) {
			T behavior = stack.Pop();
			behavior.IsReclaimed = false;
			return behavior;
		}
		return ScriptableObject.CreateInstance<T>();
	}

	public static void Reclaim (T behavior) {
		behavior.IsReclaimed = true;
		stack.Push(behavior);
	}
```

Finally, add an `OnEnable` method to `ShapeBehavior` that checks whether it is reclaimed. If so, have it recycle itself. This method is invoked when the asset gets created via `ScriptableObject.CreateInstance` and after each hot reload, so the pools will be regenerated.

```
	void OnEnable () {
		if (IsReclaimed) {
			Recycle();
		}
	}
```

2.7 Conditional Compilation

Extending `ScriptableObject` is only needed while working in the editor. The overhead of creating runtime assets is not needed in builds. We can use conditional compilation to only have `ShapeBehavior` extend `ScriptableObject` when our code is compiled for use in the editor. That's done by putting the `: ScriptableObject` code on a separate line, in between `#if UNITY_EDITOR` and `#endif` compiler directives.

```
public abstract class ShapeBehavior
#if UNITY_EDITOR
	: ScriptableObject
#endif
{
	…
}
```

How does `#if UNITY_EDITOR` work?

The `#if` directive is used by the compiler to decide whether to include or skip a section of code during compilation. That means that there are two ways in which the code can be compiled: either `ShapeBehavior` extends `ScriptableObject`, or it doesn't.

The decision is made based on whether the symbol written after `#if` is defined. Symbols can be defined via the `#define` directive, but can also be passed to the compiler by the code editor or another application. In this case, Unity makes sure that the UNITY_EDITOR symbol is defined when it compiles our code for usage in the editor. The same approach can also be used to check the Unity version and for which target platform the code is compiled.

Likewise, we only need the `IsReclaimed` and `OnEnable` code in the editor, so make that bit conditional too.

```
#if UNITY_EDITOR
	public bool IsReclaimed { get; set; }

	void OnEnable () {
		if (IsReclaimed) {
			Recycle();
		}
	}
#endif
```

The usage of `IsReclaimed` by `ShapeBehaviorPool` must also become conditional.

```
    public static T Get () {
        if (stack.Count > 0) {
            T behavior = stack.Pop();
#if UNITY_EDITOR
            behavior.IsReclaimed = false;
#endif
            return behavior;
        }
        return ScriptableObject.CreateInstance<T>();
    }

    public static void Reclaim (T behavior) {
#if UNITY_EDITOR
        behavior.IsReclaimed = true;
#endif
        stack.Push(behavior);
    }
```

Finally, we must only use `ScriptableObject.CreateInstance` in the editor. Otherwise, we have to use the constructor method. That can be done with the help of an `#else` directive.

```
    public static T Get () {
        if (stack.Count > 0) {
            T behavior = stack.Pop();
#if UNITY_EDITOR
            behavior.IsReclaimed = false;
#endif
            return behavior;
        }
#if UNITY_EDITOR
        return ScriptableObject.CreateInstance<T>();
#else
        return new T();
#endif
    }
```

3 Oscillation

Our new approach for shape behavior is pointless if all we're doing is moving and rotating shapes. It's only useful if we have a sizable selection of behavior that shapes could exhibit. So let's add third type of behavior. We'll add support for oscillating shapes, moving back and forth along a straight line, relative to its original position.

3.1 Minimal Behavior

To support another behavior type, we first have to add an element for it to the `ShapeBehaviorType` enumeration. We must not change the order of existing elements, so append it to the list.

```
public enum ShapeBehaviorType {
	Movement,
	Rotation,
	Oscillation
}
```

Then we can create a minimal behavior class, in this case `OscillationShapeBehavior`, with minimal implementations of all required methods and properties. We'll add the code responsible for oscillation later.

```
using UnityEngine;

public sealed class OscillationShapeBehavior : ShapeBehavior {

	public override ShapeBehaviorType BehaviorType {
		get {
			return ShapeBehaviorType.Oscillation;
		}
	}

	public override void GameUpdate (Shape shape) {}

	public override void Save (GameDataWriter writer) {}

	public override void Load (GameDataReader reader) {}
]
	public override void Recycle () {
		ShapeBehaviorPool<OscillationShapeBehavior>.Reclaim(this);
	}
}
```

3.2 From Enum to Instance

To support loading, we also have to add a case for oscillation to the non-generic `Shape.AddBehavior` method. But it's more convenient if we didn't have to edit `Shape` each time we add a behavior type. So let's move the conversion from enumeration to behavior instance to `ShapeBehaviorType`.

While we cannot directly put methods inside an enumeration type, we can use extension methods to do it indirectly. An extension method can be defined in any class or struct, so we'll use a dedicated static `ShapeBehaviorTypeMethods` class, which we can put in the same file as the enumeration.

```
public enum ShapeBehaviorType {
	Movement,
	Rotation,
	Oscillation
}

public static class ShapeBehaviorTypeMethods {}
```

What's an extension method?

An extension method is a static method inside a static class that behaves like an instance method of some type. That type could be anything, a class, an interface, a struct, a primitive value, or an enum. The first argument of an extension method defines the type and instance value that the method will operate on.

Does this allow us to add methods to everything? Yes, just like you could write any static method that has any type as its argument. Is this a good idea? When used in moderation, it can be. It is a tool that has its uses, but wielding it with abandon will produce an unstructured mess.

Give this class a public static `GetInstance` method with a `ShapeBehaviorType` parameter. Then put the code from `Shape.AddShapeBehavior` in it, adjust it to use the pools, and add a new case for oscillation.

```
public static class ShapeBehaviorTypeMethods {

	public static ShapeBehavior GetInstance (ShapeBehaviorType type) {
		switch (type) {
			case ShapeBehaviorType.Movement:
				return ShapeBehaviorPool<MovementShapeBehavior>.Get();
			 case ShapeBehaviorType.Rotation:
				return ShapeBehaviorPool<RotationShapeBehavior>.Get();
			case ShapeBehaviorType.Oscillation:
				return ShapeBehaviorPool<OscillationShapeBehavior>.Get();
		}
		UnityEngine.Debug.Log("Forgot to support " + type);
		return null;
	}
}
```

To turn it into an extension method for `ShapeBehaviorType`, add the `this` keyword before the `ShapeBehaviorType` parameter.

```
public static ShapeBehavior GetInstance (this ShapeBehaviorType type) { … }
```

Now it's possible to write code like `ShapeBehaviorType.Movement.GetInstance()` and get a `MovementShapeBehavior` instance out of it. Use this approach in `Shape.Load` to get a behavior instance, add it to the list, and then load it.

```
		if (reader.Version >= 6) {
			int behaviorCount = reader.ReadInt();
			for (int i = 0; i < behaviorCount; i++) {
				//AddBehavior((ShapeBehaviorType)reader.ReadInt()).Load(reader);
				ShapeBehavior behavior =
					((ShapeBehaviorType)reader.ReadInt()).GetInstance();
				behaviorList.Add(behavior);
				behavior.Load(reader);
			}
		}
```

Delete the non-generic `AddBehavior` method, as we no longer need it.

```
	//ShapeBehavior AddBehavior (ShapeBehaviorType type) { … }
```

3.3 Oscillation Implementation

We'll implement the oscillation behavior by moving the shape with a sine wave along an offset vector. This vector defines the maximum offset in the positive direction. We also need a frequency to control the oscillation speed, defined in oscillations per second. Add properties for both to `OscillationShapeBehavior`.

```
	public Vector3 Offset { get; set; }

	public float Frequency { get; set; }
```

The oscillation curve is simply the sine of 2π multiplied by the frequency and current time. That's used to scale the configured offset, which is then used to set the shape's position.

```
	public override void GameUpdate (Shape shape) {
		float oscillation = Mathf.Sin(2f * Mathf.PI * Frequency * Time.time);
		shape.transform.localPosition = oscillation * Offset;
	}
```

But that would make all shapes oscillate around the origin, instead of their spawn position. Even worse, it wouldn't work in combination with the movement behavior. So we have to add the oscillation to the position instead of replacing it.

```
		shape.transform.localPosition += oscillation * Offset;
```

However, if we add the oscillation offset to the position each update, then we end up accumulating offsets instead of using a new offset each update. To compensate for our previous oscillation, we have to remember it and subtract it before determining the final offset, and also set it to zero when recycling.

```
	float previousOscillation;

	public override void GameUpdate (Shape shape) {
		float oscillation = Mathf.Sin(2f * Mathf.PI * Frequency * Time.time);
		shape.transform.localPosition +=
			(oscillation - previousOscillation) * Offset;
		previousOscillation = oscillation;
	}

	…

	public override void Recycle () {
		previousOscillation = 0f;
		ShapeBehaviorPool<OscillationShapeBehavior>.Reclaim(this);
	}
```

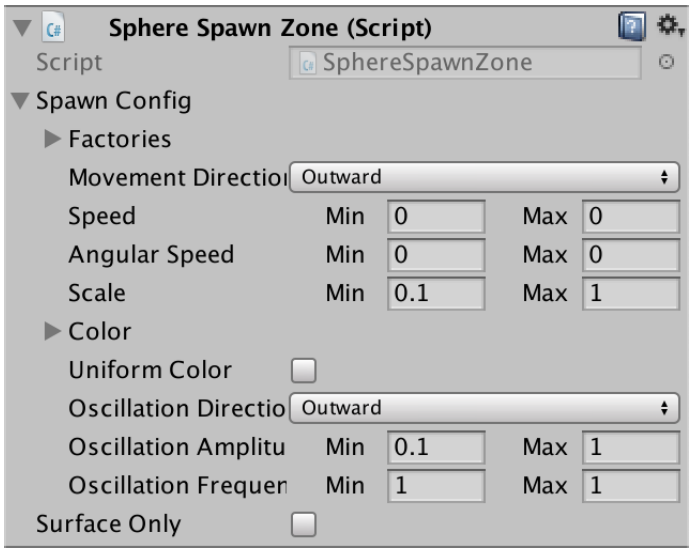
Now we also know what state has to be saved and loaded: both properties and the previous oscillation value.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(Offset);
		writer.Write(Frequency);
		writer.Write(previousOscillation);
	}

	public override void Load (GameDataReader reader) {
		Offset = reader.ReadVector3();
		Frequency = reader.ReadFloat();
		previousOscillation = reader.ReadFloat();
	}
```

3.4 Oscillation Configuration

Like movement and rotation, we'll configure oscillation per spawn zone, by adding fields to `SpawnConfiguration`. Use `MovementDirection` for the direction and `FloatRange` to control the amplitude and frequency of the oscillation.

```
	public struct SpawnConfiguration {

		…

		public MovementDirection oscillationDirection;

		public FloatRange oscillationAmplitude;

		public FloatRange oscillationFrequency;
	}
```



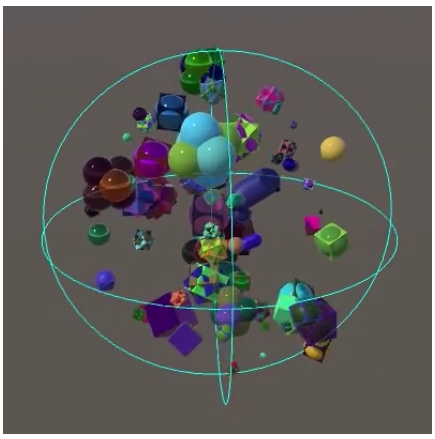
Spawn zone with oscillation.

We now have two cases in `SpawnZone` where we need to convert `MovementDirection` to a vector, so move the relevant code to its own method.

```
	public virtual Shape SpawnShape () {
		…

		float speed = spawnConfig.speed.RandomValueInRange;
		if (speed != 0f) {
			//Vector3 direction;
			//switch (spawnConfig.movementDirection) {
			//	…
			//}
			var movement = shape.AddBehavior<MovementShapeBehavior>();
			movement.Velocity =
				GetDirectionVector(spawnConfig.movementDirection, t) * speed;
		}

		return shape;
	}

	Vector3 GetDirectionVector (
		SpawnConfiguration.MovementDirection direction, Transform t
	) {
		switch (direction) {
			case SpawnConfiguration.MovementDirection.Upward:
				return transform.up;
			case SpawnConfiguration.MovementDirection.Outward:
				return (t.localPosition - transform.position).normalized;
			case SpawnConfiguration.MovementDirection.Random:
				return Random.onUnitSphere;
			default:
				return transform.forward;
		}
	}
```

Because the `SpawnShape` method is getting large, put the code to add an oscillation behavior in its own method too. In this case, we can skip adding the behavior if either the amplitude or the frequency ends up zero.

```
	public virtual Shape SpawnShape () {
		…

		SetupOscillation(shape);
		return shape;
	}

	void SetupOscillation (Shape shape) {
		float amplitude = spawnConfig.oscillationAmplitude.RandomValueInRange;
		float frequency = spawnConfig.oscillationFrequency.RandomValueInRange;
		if (amplitude == 0f || frequency == 0f) {
			return;
		}
		var oscillation = shape.AddBehavior<OscillationShapeBehavior>();
		oscillation.Offset = GetDirectionVector(
			spawnConfig.oscillationDirection, shape.transform
		) * amplitude;
		oscillation.Frequency = frequency;
	}
```



Oscillating in lockstep.

3.5 Oscillating Based on Shape Age

Because we're oscillating based on the current game time, all shapes oscillate in lockstep. Worse, because we don't save the game time oscillation isn't saved correctly. We can solve both problems by oscillating based on the shape's age instead, and saving the age.

First, add an `Age` property to `Shape`. It's publicly accessible, but the shape controls its own age, so its setter should be private.

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

In `GameUpdate`, increase the age by the time delta. And set the age back to zero when recycling.

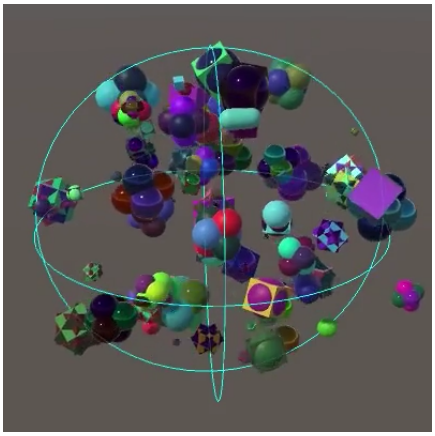
```
	public void GameUpdate () {
		Age += Time.deltaTime;
		for (int i = 0; i < behaviorList.Count; i++) {
			behaviorList[i].GameUpdate(this);
		}
	}

	public void Recycle () {
		Age = 0f;
		…
	}
```

The age should be saved and loaded too. Write it directly before the behavior list.

```
	public override void Save (GameDataWriter writer) {
		…
		writer.Write(Age);
		writer.Write(behaviorList.Count);
		…
	}

	public override void Load (GameDataReader reader) {
		…
		if (reader.Version >= 6) {
			Age = reader.ReadFloat();
			int behaviorCount = reader.ReadInt();
			…
		}
		…
	}
```

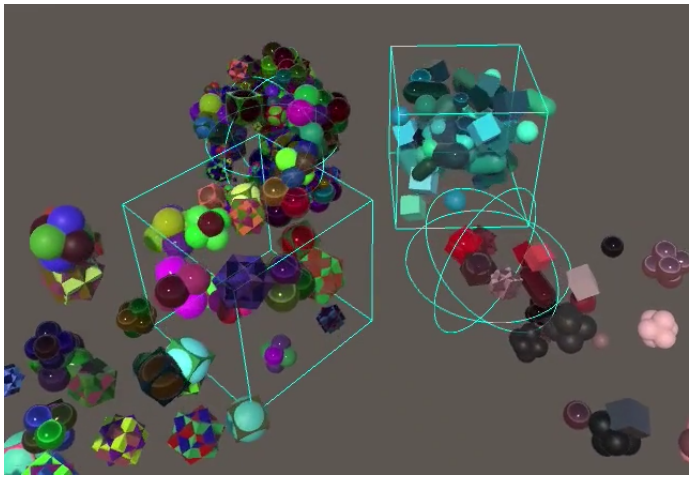
Finally, adjust `OscillationShapeBehavior` so it uses the shape's age instead of the current time.

```
	public override void GameUpdate (Shape shape) {
		float oscillation = Mathf.Sin(2f * Mathf.PI * Frequency * shape.Age);
		…
	}
```



Oscillating based on shape age.

We now have a framework for adding modular behavior to shapes. The current approach is overkill for just three simple behavior types, but we'll add more complex behavior in the next tutorial, Satellites.



Varied shape behavior.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

BECOME A PATRON

Or make a direct donation!

made by Jasper Flick