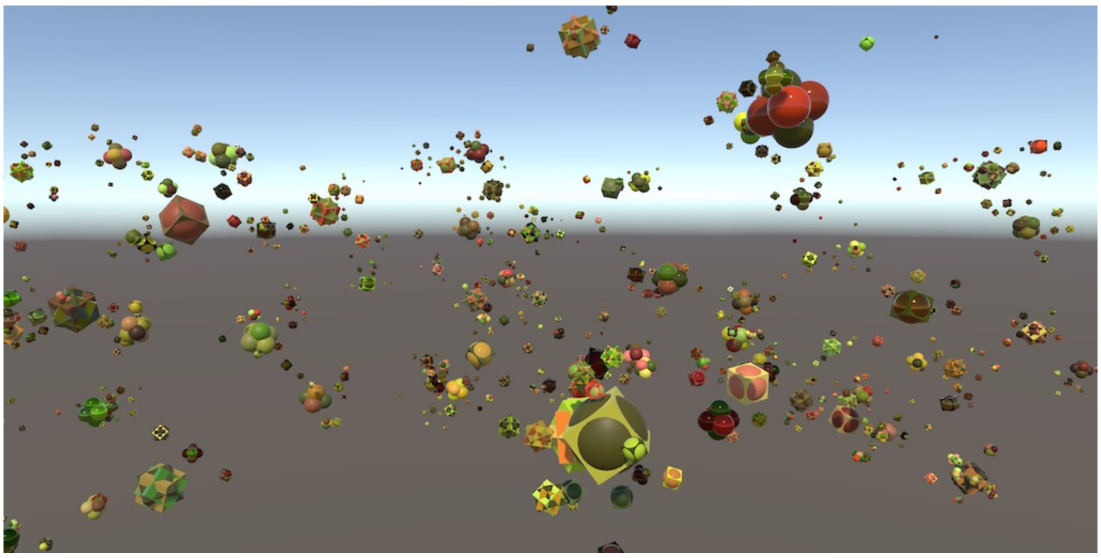
Satellites Shape Relationships

Spawn multiple shapes at once.
Make a shape orbit another shape.
Keep track of references to specific shape instances.
Enforce a shape population limit.

This is the tenth tutorial in a series about Object Management. It adds a satellite behavior, which introduces relationships between shapes.

This tutorial is made with Unity 2017.4.12f1.


The big shapes are popular; they have orbiters.

1 Spawning Multiple Shapes

In this tutorial we're going to create a behavior that lets a shape orbit another shape, like a satellite. We'll decide whether a shape has a satellite when it is spawned. If so, then we'll also spawn its satellite. This means that each time that we spawn a shape we might end up with two new shapes instead of always one.

1.1 Spawning a Satellite Per Shape

To spawn a satellite we'll add a `CreateSatelliteFor` method to **`SpawnZone`**, with a parameter for the focal shape. The focal shape is what the satellite will orbit. We could use separate factories for the satellites, but we'll simply use the same factories for the regular shape to spawn a random one and give it an arbitrary rotation.

```
	void CreateSatelliteFor (Shape focalShape) {
		int factoryIndex = Random.Range(0, spawnConfig.factories.Length);
		Shape shape = spawnConfig.factories[factoryIndex].GetRandom();
		Transform t = shape.transform;
		t.localRotation = Random.rotation;
	}
```

We won't create a real satellite just yet. For now, we make the satellite half the size of the focus shape, position it one unit above it, and make it move upwards.

```
		t.localRotation = Random.rotation;
		t.localScale = focalShape.transform.localScale * 0.5f;
		t.localPosition = focalShape.transform.localPosition + Vector3.up;
		shape.AddBehavior<MovementShapeBehavior>().Velocity = Vector3.up;
```

We'll also have to give the satellite a color, the same way we give the focus shape a color. Put the relevant code in a `SetupColor` method so we can reuse it.

```
	public virtual Shape SpawnShape () {
		…
		~~//if (spawnConfig.uniformColor) {~~
		~~//	shape.SetColor(spawnConfig.color.RandomInRange);~~
		~~//}~~
		~~//else {~~
		~~//	…~~
		~~//}~~
		SetupColor(shape);

		…
	}

	void CreateSatelliteFor (Shape focalShape) {
		…
		SetupColor(shape);
	}

	void SetupColor (Shape shape) {
		if (spawnConfig.uniformColor) {
			shape.SetColor(spawnConfig.color.RandomInRange);
		}
		else {
			for (int i = 0; i < shape.ColorCount; i++) {
				shape.SetColor(spawnConfig.color.RandomInRange, i);
			}
		}
	}
```

Give each new shape a satellite companion by invoking `CreateSatteliteFor` at the end of `SpawnShape`, so we spawn shapes in pairs.

```
	public virtual Shape SpawnShape () {
		…

		CreateSatelliteFor(shape);
		return shape;
	}
```

1.2 Adding Shapes to the Game

The idea of `SpawnShape` was that it brings a new shape into the game, which is returned so that `Game` can add it to its list of shapes. That still happens for the regular shape, but the satellite shape isn't added to the list, which means that it doesn't get updated and remains frozen.

We could have `SpawnShape` return a list of shapes, but the goal is to get the shapes added to the shape list when they are spawned, no matter when or where that happens. We can do that by turning the responsibility around, and once again make `Game` available via a static `Instance` property. Then it's up to whoever spawns a shape to pass it to `Game`.

```
	public static Game Instance { get; private set; }

	…

	void OnEnable () {
		Instance = this;
		…
	}
```

In order to receive new shapes, give `Game` a public `AddShape` method that simply adds a shape to its list.

```
	public void AddShape (Shape shape) {
		shapes.Add(shape);
	}
```

We'll make `ShapeFactory.Get` responsible for adding each shape to `Game`. This makes `ShapeFactory` aware of `Game`, but also makes it so that we never need to worry about whether a shape has been added to `Game` yet, assuming that all shapes are retrieved via a factory.

```
	public Shape Get (int shapeId = 0, int materialId = 0) {
		…
		Game.Instance.AddShape(instance);
		return instance;
	}
```

This is also true in `Game.LoadGame`, so we must no longer explicitly add loaded shapes to the list. To do so would result in duplicate entries.

```
	IEnumerator LoadGame (GameDataReader reader) {
		…
		
		for (int i = 0; i < count; i++) {
			int factoryId = version >= 5 ? reader.ReadInt() : 0;
			int shapeId = version > 0 ? reader.ReadInt() : 0;
			int materialId = version > 0 ? reader.ReadInt() : 0;
			Shape instance = shapeFactories[factoryId].Get(shapeId, materialId);
			instance.Load(reader);
			~~//shapes.Add(instance);~~
		}
	}
```

1.3 Spawning Any Amount of Shapes

At this point the old design of the **SpawnZone**.SpawnShape method is no longer appropriate. First, it no longer needs to return a shape. Second, it no longer needs to be restricted to spawning a single shape, as it currently spawns two per invocation. So change it to **void** SpawnShapes.

```
	~~//public virtual Shape SpawnShape () {~~
	public virtual void SpawnShapes () {
	int factoryIndex = Random.Range(0, spawnConfig.factories.Length);
		…

		CreateSatelliteFor(shape);
		~~//return shape;~~
	}
```

This also requires an update of **CompositeSpawnZone**.

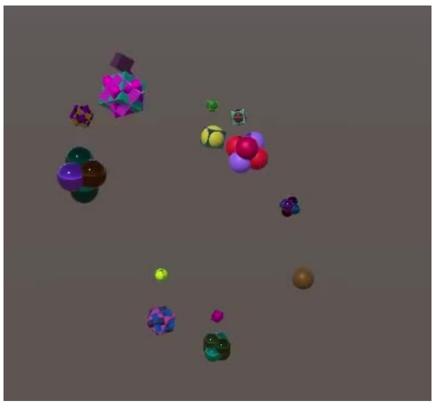
```
	~~//public override Shape SpawnShape () {~~
	public override void SpawnShapes () {
		if (overrideConfig) {
			~~//return~~
			base.SpawnShapes();
		}
		else {
			…
			~~//return~~
			spawnZones[index].SpawnShapes();
		}
	}
```

Also adjust the SpawnShape method of **GameLevel** to match.

```
	~~//public Shape SpawnShape () {~~
	public void SpawnShapes () {
		~~//return~~
		spawnZone.SpawnShapes();
	}
```

We can now remove the **Game**.CreateShape method. Instead, directly invoke **GameLevel**.Current.SpawnShapes in **Update** and **FixedUpdate**.

```
	void Update () {
		if (Input.GetKeyDown(createKey)) {
			~~//CreateShape();~~
			GameLevel.Current.SpawnShapes();
		}
		…
	}

	void FixedUpdate () {
		…
		while (creationProgress >= 1f) {
			creationProgress -= 1f;
			~~//CreateShape();~~
			GameLevel.Current.SpawnShapes();
		}

		…
	}

	…

	~~//void CreateShape () {~~
	~~//	shapes.Add(GameLevel.Current.SpawnShape());~~
	~~//}~~
```


Spawning pairs of shapes.

2 Satellite Behavior

To turn the companion shape into an actual satellite we have to make it behave like one, which requires us to create a new behavior type.

2.1 New Shape Behavior

Add a `Satellite` option to the `ShapeBehaviorType` enum, along with an accompanying case in the `GetInstance` method that returns a `SatelliteShapeBehavior` instance.

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

public static class ShapeBehaviorTypeMethods {

    public static ShapeBehavior GetInstance (this ShapeBehaviorType type) {
        switch (type) {
            …
            case ShapeBehaviorType.Satellite:
                return ShapeBehaviorPool<SatelliteShapeBehavior>.Get();
        }
        UnityEngine.Debug.Log("Forgot to support " + type);
        return null;
    }
}
```

Create a minimal `SatelliteShapeBehavior` component that does nothing, for now.

```
using UnityEngine;

public sealed class SatelliteShapeBehavior : ShapeBehavior {

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

    public override void GameUpdate (Shape shape) {}

    public override void Save (GameDataWriter writer) {}

    public override void Load (GameDataReader reader) {}

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

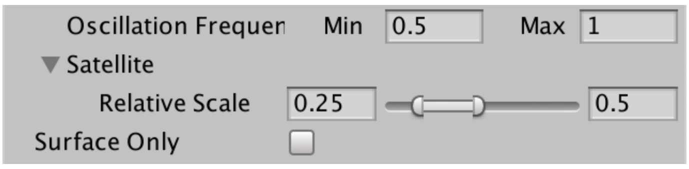
Add this behavior to the shape in SpawnZone.CreateSatelliteFor and remove the test position and movement behavior.

```
	void CreateSatelliteFor (Shape focalShape) {
		…
		~~//t.localPosition = focalShape.transform.localPosition + Vector3.up;~~
		~~//shape.AddBehavior<MovementShapeBehavior>().Velocity = Vector3.up;~~
		SetupColor(shape);
		shape.AddBehavior<SatelliteShapeBehavior>();
	}
```

2.2 Satellite Configuration

Like for regular spawn, we'll also make it possible to configure satellites via the inspector of the spawn zone. We'll support a few options, so define a SatelliteConfiguration struct inside SpawnConfiguration that we'll use to group them. Begin with a single range to control the scale of the satellite. Make it relative to its focal shape, from 0.1 to 1.

```
	[System.Serializable]
	public struct SpawnConfiguration {

		…

		[System.Serializable]
		public struct SatelliteConfiguration {

			[FloatRangeSlider(0.1f, 1f)]
			public FloatRange relativeScale;
		}

		public SatelliteConfiguration satellite;
	}
```


Oscillation Frequen Min 0.5 Max 1
▼ Satellite
Relative Scale 0.25 0.5
Surface Only ☐

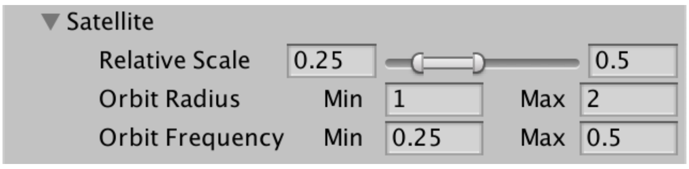
Satellite configuration, relative scale.

Use a random value in this range instead of the fixed 0.5 that we used up to this point.

```
		t.localScale =
			focalShape.transform.localScale *
			spawnConfig.satellite.relativeScale.RandomValueInRange;
```

We also need an orbit radius to control the distance between the satellite and its focus, plus an orbit frequency to control how quickly it orbits around its focus.

```
	public struct SatelliteConfiguration {

		[FloatRangeSlider(0.1f, 1f)]
		public FloatRange relativeScale;

		public FloatRange orbitRadius;

		public FloatRange orbitFrequency;
	}
```


▼ Satellite

Relative Scale	0.25			0.5
Orbit Radius	Min	1	Max	2
Orbit Frequency	Min	0.25	Max	0.5

Satellite orbit radius and frequency.

Turning those configuration values into orbital motion requires math specific to satellites, so we won't put the code for that in `SpawnZone`. Instead, we'll add a public `Initialize` method to `SatelliteShapeBehavior`. Like `GameUpdate`, give it its own shape as a parameter. Besides that, also add parameters for the focal shape, radius, and frequency.

```
	public void Initialize (
		Shape shape, Shape focalShape, float radius, float frequency
	) {}
```

Now we can initialize the satellite's behavior with a randomized orbit radius and frequency.

```
			shape.AddBehavior<SatelliteShapeBehavior>().Initialize(
				shape, focalShape,
				spawnConfig.satellite.orbitRadius.RandomValueInRange,
				spawnConfig.satellite.orbitFrequency.RandomValueInRange
			);
```

2.3 Orbiting

To make the satellite shape orbit its focus, we have to move it in a circle around it. We can do that with trigonometry, offsetting the satellite's position along two orthogonal vectors scaled by the cosine and sine of the shape's age. This requires `SatelliteShapeBehavior` to keep track of the focal shape, the frequency, and the two offset vectors. The radius can be factored into the offsets.

We'll begin by always using the X axis for the cosine offset and the Z axis for the sine offset. When looking from above, it will result in the satellite starting at the right of the focal shape and circling around it counter-clockwise.

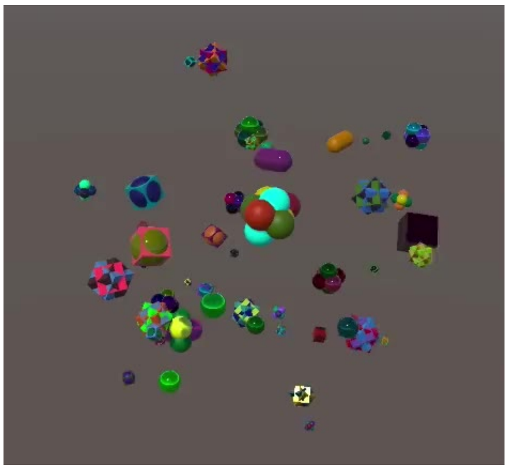
```
	Shape focalShape;

	float frequency;

	Vector3 cosOffset, sinOffset;

	public void Initialize (
		Shape shape, Shape focalShape, float radius, float frequency
	) {
		this.focalShape = focalShape;
		this.frequency = frequency;
		cosOffset = Vector3.right;
		sinOffset = Vector3.forward;
		cosOffset *= radius;
		sinOffset *= radius;
	}
```

To make the satellite move, we have to adjust its position in `GameUpdate`. Set it to the focal position plus both offsets, each scaled by either the cosine or sine of 2π times the frequency times its age.

```
	public override void GameUpdate (Shape shape) {
		float t = 2f * Mathf.PI * frequency * shape.Age;
		shape.transform.localPosition =
			focalShape.transform.localPosition +
			cosOffset * Mathf.Cos(t) + sinOffset * Mathf.Sin(t);
	}
```

To make sure that the satellite's initial position is valid, invoke `GameUpdate` once at the end of `Initialize`. This is necessary because `GameUpdate` won't be invoked during the same frame that a shape is spawned.

```
	public void Initialize (
		Shape shape, Shape focalShape, float radius, float frequency
	) {
		…

		GameUpdate(shape);
	}
```


Orbiting satellites.

What about elliptic orbits?

Elliptic orbits are also possible, but are more complex than circular orbits. You can turn the orbit into an ellipse by using a different radius for each offset. Besides that, the orbit must be offset such that the focal shape ends up in one of the two foci of the ellipse. Also, the orbital velocity should no longer be constant, but depend on the distance between the satellite and its focus.

2.4 Randomized Orbit Axis

Our satellites are currently always orbiting in the XZ plane, rotating around an Y axis centered on the focal shape. We can randomize the orbit axis, by retrieving one via `Random.onUnitSphere`.

```
		Vector3 orbitAxis = Random.onUnitSphere;
		cosOffset = Vector3.right;
		sinOffset = Vector3.forward;
```

That gives us an axis. The next step it to find an arbitrary offset vector that lies in the plane defined by the axis. We can do that by taking the cross product of the orbit axis and another random vector. That gives us a random vector projected onto the orbit plane, which is most likely not of unit length, so we should normalize it.

```
		Vector3 orbitAxis = Random.onUnitSphere;
		cosOffset = Vector3.Cross(orbitAxis, Random.onUnitSphere).normalized;
```

What's a cross product?

See Rendering 6, Bumpiness for an explanation.

This works, unless the second random vector ends up being the same as or the negative version of the orbit axis. That would result in a zero vector, which cannot be normalized. Or specifically, `Vector3.normalized` will return the zero vector when the vector ends up too short to normalize. We can detect that by checking whether the offset vector's square magnitude is less than one. But because of numerical precision we should check for a smaller value, so let's use 0.1 instead. It's going to be either very close to 1 or exactly zero.

Getting an invalid offset is very unlikely, so when it happens we'll just try again. We can do that with a `do while` loop.

```
		do {
			cosOffset = Vector3.Cross(orbitAxis, Random.onUnitSphere).normalized;
		}
		while (cosOffset.sqrMagnitude < 0.1f);
```

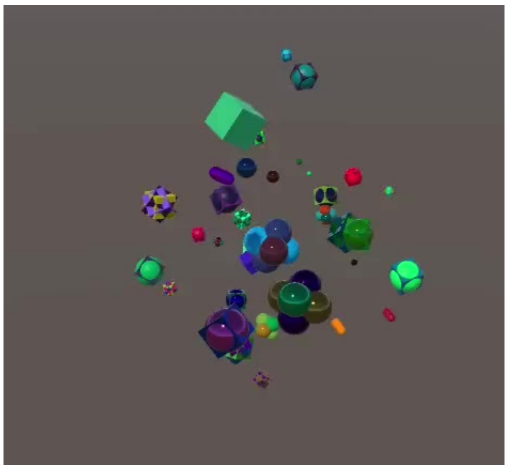
How does a `do while` loop work?

It is an alternative `while` loop, for when the code block of the loop has to be executed at least once. You begin with `do` instead of `while`, followed by the code block, after which comes the `while` statement, terminated with a semicolon. Had you used a regular `while` loop instead, you'd have to repeat the code block once before the loop to get the same effect.

`do { Work(); } while (Condition());` is the same as `Work(); while (Condition()) { Work(); }`.

The second offset can be found by taking the cross product of the first offset and the orbit axis. After that the offset are scaled.

```
		sinOffset = Vector3.Cross(cosOffset, orbitAxis);
		cosOffset *= radius;
		sinOffset *= radius;
```



Random orbits.

2.5 Tidal Locking

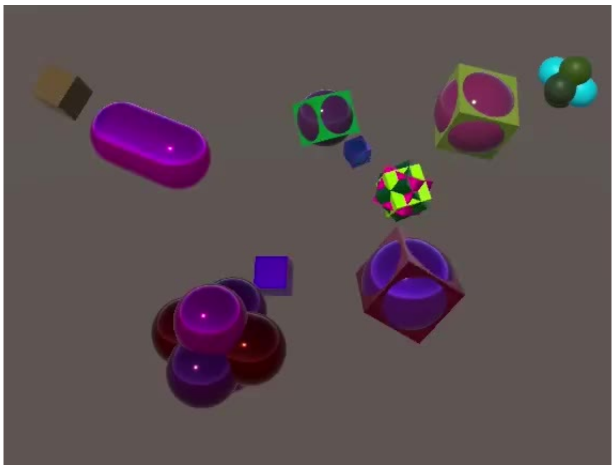
Although our satellites orbit, they currently do not rotate. We could give then arbitrary rotation velocities, but we could also tidally lock them. This means that their rotation matches their orbit, so they always face their focal shape with the same side.

To give each satellite a matching rotation, add a rotation behavior with an angular velocity equal to the orbit axis scaled by the frequency, multiplied by 360 degrees. Also, because of the way that we construct the orbit, we have to negate the velocity.

```
			cosOffset *= radius;
			sinOffset *= radius;

			shape.AddBehavior<RotationShapeBehavior>().AngularVelocity =
				-360f * frequency * orbitAxis;
```

However, because the angular velocity is applied in the shape's local space, directly using the orbit axis only works when the satellite has no rotation of its own. As we gave it an arbitrary rotation, we have to convert the orbit axis to the shape's local space, by invoking `InverseTransformDirection` on its transformation.

```
			shape.AddBehavior<RotationShapeBehavior>().AngularVelocity =
				-360f * frequency *
				shape.transform.InverseTransformDirection(orbitAxis);
```


Tidally locked satellites.

If you wanted to, you could also make the tidal locking imperfect by slightly randomizing the velocity. The satellite could either rotate too fast, too slow, have a retrograde rotation, and could have axial tilt.

3 Shape References

Satellites function correctly as long as their focus shape exists, but things get weird when the focus is recycled while the satellite is still around. Initially, the satellite will keep orbiting the last position of its focus. When the focus shape gets reused for a new spawn, the satellite still orbits it, teleporting to its new position.

We have to sever the connection between a satellite and its focus when the focus is recycled. If we destroyed the focus, then all we needed to do was check whether the `focusShape` reference has become null. But we recycle shapes, so the reference remains intact even though the shape isn't part of the game anymore. So we have to find a way to determine whether a shape reference is still valid.

3.1 Instance Identification

We can distinguish between different incarnations of the same shape by adding an instance identifier property to `Shape`. Just like `Age`, it has to be publicly accessible but will only be modified by the shape itself.

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

	public int InstanceId { get; private set; }
```

Each time a shape is recycled, increment its instance identifier. That way we can tell whether we're dealing with the same or a recycled shape instance.

```
	public void Recycle () {
		Age = 0f;
		InstanceId += 1;
		…
	}
```

By keeping track of both a reference to the shape and the correct instance identifier, we're able to check whether the shape's identifier is still the same each update. If not, it got recycled and is no longer valid.

3.2 Indirect References

Rather than explicitly add an identifier field each time we need a `Shape` reference, lets combine both in a new `ShapeInstance` struct. We'll make this a serializable struct with a `Shape` and an instance identifier field. The shape has to be publicly accessible, but the instance identifier is a technicality that doesn't have to be public.

```
[System.Serializable]
public struct ShapeInstance {

	public Shape Shape { get; private set; }

	int instanceId;
}
```

The idea is that a `ShapeInstance` struct is immutable, representing a reference to a specific shape instance that's only valid until that shape is recycled. The only way to create a valid instance reference is via a constructor method that has a single shape parameter, which we use to set the reference and copy its current instance identifier.

```
	public ShapeInstance (Shape shape) {
		Shape = shape;
		instanceId = shape.InstanceId;
	}
```

To verify whether the instance reference is valid, add an `IsValid` getter property that checks whether the shape's instance identifier is still the same.

```
	public bool IsValid {
		get {
			return instanceId == Shape.InstanceId;
		}
	}
```

But there is still a default constructor, which is used for example when a `ShapeInstance` array is created. That would result in null references, so we should also check whether we have a shape reference at all. That also guarantees that instances become invalid if for some reason a shape object is destroyed instead of recycled.

```
			return Shape && instanceId == Shape.InstanceId;
```

3.3 Casting from Shape to Instance

Converting a `Shape` `shape` reference to a `ShapeInstance` value can now be done by via `new ShapeInstance(shape)`. But we can made the code even shorter by adding a casting operator to `ShapeInstance`. An operator is defined like a method, except that it is static, includes the `operator` keyword, and doesn't have a method name. In the case of an explicit cast, we have to add the `explicit` keyword in front of `operator`.

```
	public static explicit operator ShapeInstance (Shape shape) {
		return new ShapeInstance(shape);
	}
```

Now the conversion can be done via `(ShapeInstance)shape`. But it can become even shorter, by making the cast implicit instead of explicit. Then a direct assignment of `shape` to a `ShapeInstance` field or variable is enough. That's also how Unity supports implicit conversions between `Vector2` and `Vector3` and other struct types.

```
	public static implicit operator ShapeInstance (Shape shape) {
		return new ShapeInstance(shape);
	}
```

3.4 Focal Shape Instance

Change the `focalShape` reference in `SatelliteShapeBehavior` into a `ShapeInstance` value. Because of the implicit cast, we don't have to change the code in `Initialize`.

```
	ShapeInstance focalShape;
```

We do have to change `GameUpdate`, because we now have to indirectly access the focal shape via `focalShape.Shape`. Also, we must only do this if the focal shape is still valid.

```
	public override void GameUpdate (Shape shape) {
		if (focalShape.IsValid) {
			float t = 2f * Mathf.PI * frequency * shape.Age;
			shape.transform.localPosition =
				focalShape.Shape.transform.localPosition +
				cosOffset * Mathf.Cos(t) + sinOffset * Mathf.Sin(t);
		}
	}
```

4 Free Satellites

From now on, satellites orbit their focus as long as it is still in the game and stop moving when the focus is recycled. At that point the link between them has become invalid and is no longer used to update the satellite. But the `SatelliteShapeBehavior` is still attached to the satellite shape. Its `GameUpdate` method still gets invoked each update, even though that is now pointless. Ideally, the behavior is recycled too.

4.1 Removing Behavior

It is possible for satellite behavior to become useless, and we could create many other kinds of temporary behavior. So let's make it possible for shapes to rid themselves of behavior that is no longer useful. We'll do that by having the behavior tell their shape whether they're still needed. We'll have `GameUpdate` return a boolean to indicate this, so adjust the method definition in `ShapeBehavior`.

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

Adjust the `GameUpdate` overrides in all shape behaviors too, always returning `true` at the end.

```
	public override bool GameUpdate (Shape shape) {
		…
		return true;
	}
```

Except for `SatelliteShapeBehavior`, which should return `true` only when the focus shape is valid. Otherwise, it returns `false`, indicating that it is no longer useful and can be removed.

```
	public override bool GameUpdate (Shape shape) {
		if (focalShape.IsValid) {
			…
			return true;
		}

		return false;
	}
```

In `Shape.GameUpdate`, we must now check each iteration whether the behavior is still needed. If not, recycle it, remove it from the behavior list, and then decrement the iterator so we won't skip any behavior. We can simply invoke `RemoveAt` on the list, so the order of behavior isn't changed. The behavior list should be short, so we don't need to worry about optimizing the removal by shuffling the order like we do when deleting from the shape list.

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

4.2 Conservation of Momentum

Satellites now become regular shapes when their focus shape ceases to exist. Without their satellite behavior, they no longer move, but they keep their rotation because that's a separate behavior. But it is both more interesting and more realistic if the shapes keep moving in whatever direction they were going when the focus shape disappeared. It would be as if the satellites got ejected from their system.

To make continued motion possible, we have to know the satellite's velocity at all times, which depends on both its orbital motion and the movement of its focus. Rather than figure that out, we'll simply keep track of the satellite's position before its last update. We can use that to determine the last position delta and convert that to a velocity when we need it.

Add a `previousPosition` vector field to `SatelliteShapeBehavior`, copy the current position to it before calculating the new position, and add a movement behavior to the shape when the satellite behavior is no longer needed.

```
	Vector3 previousPosition;

	…

	public override bool GameUpdate (Shape shape) {
		if (focalShape.IsValid) {
			float t = 2f * Mathf.PI * frequency * shape.Age;
			previousPosition = shape.transform.localPosition;
			shape.transform.localPosition =
				focalShape.Shape.transform.localPosition +
				cosOffset * Mathf.Cos(t) + sinOffset * Mathf.Sin(t);
			return true;
		}

		shape.AddBehavior<MovementShapeBehavior>().Velocity =
			(shape.transform.localPosition - previousPosition);
		return false;
	}
```

To arrive at a correct velocity, we have to divide the position delta by the time delta of the previous frame. We'll simply assume that the delta is the same as for the current frame, which is true because we're using a fixed time step.

```
		shape.AddBehavior<MovementShapeBehavior>().Velocity =
			(shape.transform.localPosition - previousPosition) / Time.deltaTime;
```

This works, except when the focal shape ends up invalid before the first game update of the satellite, which is unlikely but possible. In that case, the previous position vector is arbitrary, either zero for a new behavior or still containing the value of a recycled satellite behavior. At this point the satellite hasn't moved yet, so initially set the previous position to its current position, at the end of `Initialize`.

```
	public void Initialize (
		Shape shape, Shape focalShape, float radius, float frequency
	) {
		…

		GameUpdate(shape);
		previousPosition = shape.transform.localPosition;
	}
```

Escaping satellite.

5 Saving and Loading

Satellites are now fully functional, can deal with recycled focus shapes, and can even survive recompilation. However, we haven't supported saving and loading them yet.

We know what that needs to be stored to persist satellite behavior. The frequency, both offset vectors, and the previous position are straightforward. We can save and load them as usual.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(frequency);
		writer.Write(cosOffset);
		writer.Write(sinOffset);
		writer.Write(previousPosition);
	}

	public override void Load (GameDataReader reader) {
		frequency = reader.ReadFloat();
		cosOffset = reader.ReadVector3();
		sinOffset = reader.ReadVector3();
		previousPosition = reader.ReadVector3();
	}
```

But saving the focus shape instance requires more work. We somehow have to persist a relationship between shapes.

5.1 Shape Index

Because all shapes that are currently in the game are stored in the game's shape list, we can use the indices of this list to uniquely identify shapes. So we can suffice with writing the shape's index when saving the shape instance. That means that we have to know the shape's index when saving the focus shape, so let's add add a `SaveIndex` property to `Shape` for that.

```
	public int SaveIndex { get; set; }
```

This property is set in `Game.AddShape` and is only useful when saving shape references.

```
	public void AddShape (Shape shape) {
		shape.SaveIndex = shapes.Count;
		shapes.Add(shape);
	}
```

We also have to make sure that the index remains correct when we shuffle the order of the shapes in `DestroyShape`.

```
	void DestroyShape () {
		if (shapes.Count > 0) {
			int index = Random.Range(0, shapes.Count);
			shapes[index].Recycle();
			int lastIndex = shapes.Count - 1;
			shapes[lastIndex].SaveIndex = index;
			shapes[index] = shapes[lastIndex];
			shapes.RemoveAt(lastIndex);
		}
	}
```

5.2 Saving a Shape Instance

Because shape instances represent a low-level fundamental part of our game and because we want to keep them as easy to work with as possible, we'll add support for directly saving them to `GameDataWriter`. It only has to write the save index of the shape.

```
	public void Write (ShapeInstance value) {
		writer.Write(value.Shape.SaveIndex);
	}
```

Now we can write the focal shape just like the other state in `SatelliteShapeBehavior.Save`.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(focalShape);
		…
	}
```

5.3 Loading a Shape Instance

When loading a shape instance, we will end up reading a save index. We need to be able to convert that to an actual shape reference. Add a public `GetShape` method to `Game` for that, with an index parameter. It simply returns a reference to the corresponding shape.

```
	public Shape GetShape (int index) {
		return shapes[index];
	}
```

To convert directly from a save index to a shape instance, let's add an alternative constructor method to `ShapeInstance` that has an index parameter instead of a `Shape` parameter. It can use the new `GetShape` method to retrieve the shape and then set its instance identifier.

```
	public ShapeInstance (int saveIndex) {
		Shape = Game.Instance.GetShape(saveIndex);
		instanceId = Shape.InstanceId;
	}
```

Add a `ReadShapeInstance` method to `GameDataReader` that reads an integer and uses it to construct a new shape instance.

```
	public ShapeInstance ReadShapeInstance () {
		return new ShapeInstance(reader.ReadInt32());
	}
```

That allows us to read the shape instance in `SatelliteShapeBehavior.Load`.

```
	public override void Load (GameDataReader reader) {
		focalShape = reader.ReadShapeInstance();
		…
	}
```

5.4 Resolving Shape Instances

Saving and loading satellite data now works, but only if no shapes have been removed during the game before saving. If shapes have been destroyed, the order of the shape list changed and it is possible that satellite shapes end up with a lower index than their focus shape. If a satellite is loaded before its focus shape, it makes no sense to immediately retrieve a reference to its focus. We have to postpone retrieving the shapes until after all shapes have been loaded.

We can still load the shape instances, but delay resolving the shape references until later. This requires us to temporarily store the save index in the shape instance. Rather than using a separate field for that and increase the size of `ShapeInstance`, we can have the instance identifier field perform double duty as a save index too. Rename the field accordingly.

```
	int instanceIdOrSaveIndex;
```

The constructor with a save index parameter will now store the index and set the shape reference to null instead of immediately resolving it.

```
	public ShapeInstance (int saveIndex) {
		Shape = null;
		instanceIdOrSaveIndex = saveIndex;
	}
```

Resolving the shape reference becomes an explicit separate step, for which we'll add a public `Resolve` method. This approach breaks the immutability principle of the struct, but we'll only use it once, after loading a game.

```
	public void Resolve () {
		Shape = Game.Instance.GetShape(instanceIdOrSaveIndex);
		instanceIdOrSaveIndex = Shape.InstanceId;
	}
```

Next, we need a way to signal behavior that it is time to resolve any shape instances that they might have. Add a `ResolveShapeInstances` method to `ShapeBehavior` for that purpose. Because only one behavior so far has need for this, we'll provide a default empty implementation of the method, by marking it as `virtual` instead of `abstract` and giving it an empty code block.

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

Only `SatelliteShapeBehavior` needs to override this method, in which it invokes `Resolve` on its focal shape instance.

```
	public override void ResolveShapeInstances () {
		focalShape.Resolve();
	}
```

We also have to add a `ResolveShapeInstances` method to `Shape`, which forwards the request to all its behavior.

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

Finally, at the end of `Game.LoadGame`, we'll resolve the shape instances of all shapes.

```
	IEnumerator LoadGame (GameDataReader reader) {
		…

		for (int i = 0; i < shapes.Count; i++) {
			shapes[i].ResolveShapeInstances();
		}
	}
```

5.5 Dealing with Invalid Instances

Up to this point we have assumed that all shape instances are valid at the moment that the game is saved, but this is not guaranteed. We have to be able to cope with the saving and loading of invalid instances. We can indicate an invalid shape instance by writing -1.

```
	public void Write (ShapeInstance value) {
		writer.Write(value.IsValid ? value.Shape.SaveIndex : -1);
	}
```

Reading a shape instance doesn't require extra attention, but `ShapeInstance.Resolve` can only do its job when it has a valid save index. If not, its shape reference has to remain null and thus invalid.

```
	public void Resolve () {
		if (instanceIdOrSaveIndex >= 0) {
			Shape = Game.Instance.GetShape(instanceIdOrSaveIndex);
			instanceIdOrSaveIndex = Shape.InstanceId;
		}
	}
```

6 Shape Population Explosion

A side effect of spawning satellites along with regular shapes is that we have increased the rate at which new shapes are spawn. Currently each shape gets a satellite, thus to keep the amount of shapes stable the destruction speed has to be set to double the creation speed.

6.1 Multiple Satellites Per Shape

We don't have to limit ourselves to exactly one satellite per regular shape. Let's make it configurable by adding a range for the amount of satellites per shape. We need an `IntRange` struct value for that, which we can create by duplicating `FloatRange` and changing the types used from `float` to `int`. Also, to keep the random range inclusive on both ends, we have to add one to the maximum when invoking the integer variant of `Random.Range`.

```
using UnityEngine;

[System.Serializable]
public struct IntRange {

	public int min, max;

	public int RandomValueInRange {
		get {
			return Random.Range(min, max + 1);
		}
	}
}
```

We can also duplicate `FloatRangeDrawer` to create a variant for the new integer range, but we don't need to do that. The code in `FloatRangeDrawer` doesn't care about the type of the minimum and maximum values, only that they exist. So we can use the same drawer for both `FloatRange` and `IntRange`. All we have to do is add a second `CustomPropertyDrawer` attribute to it. Let's also rename the drawer to `FloatOrIntRangeDrawer`, renaming its asset file too.

```
[CustomPropertyDrawer(typeof(FloatRange)), CustomPropertyDrawer(typeof(IntRange))]
public class FloatOrIntRangeDrawer : PropertyDrawer { … }
```

Can we do the same for `FloatRangeSliderDrawer`?

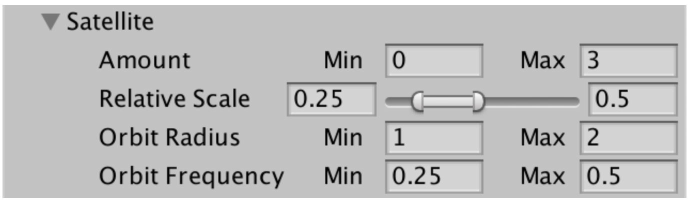
Yes, but the code of that drawer specifically works with floats, so needs to be adapted to work for either floats or integers. I'm not bothering with that here, because I don't have a limited range for the satellite amount in mind. Even negative values are useful, as they can be used to make it less likely for satellites to appear.

Add an integer range option to `SatelliteConfiguration` to configure the amount of satellites spawned per shape.

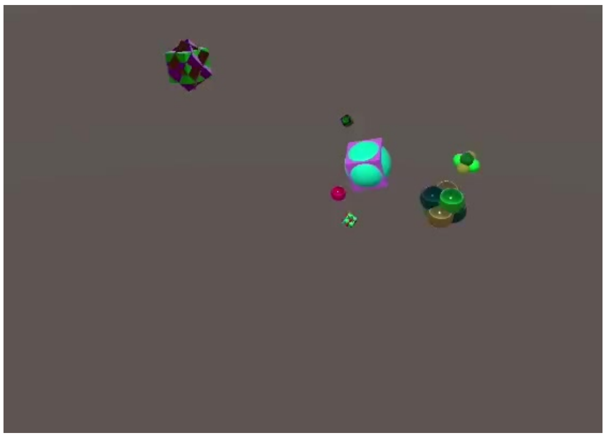
```
	public struct SatelliteConfiguration {

		public IntRange amount;

		…
	}
```

In `SpawnShapes`, determine a random count and invoke `CreateSatelliteFor` that many times.

```
		int factoryIndex = Random.Range(0, spawnConfig.factories.Length);
		…

		int satelliteCount = spawnConfig.satellite.amount.RandomValueInRange;
		for (int i = 0; i < satelliteCount; i++) {
			CreateSatelliteFor(shape);
		}
	}
```



```
▼ Satellite
    Amount              Min  0       Max  3
    Relative Scale      0.25              0.5
    Orbit Radius        Min  1       Max  2
    Orbit Frequency     Min  0.25    Max  0.5
```



Between zero and three satellites per shape.

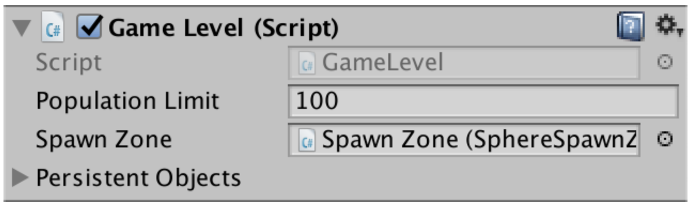
Could we create satellites for other satellites?

Yes, besides their behavior a satellite is like any other shape and can act as the focus for another satellite. Nested satellite relationships can quickly produce cluttered movement though, and you have to avoid creating an infinite chain of satellites.

6.2 Population Limit

With the amount of satellites per shape no longer constant, we cannot rely on a fixed creation and destruction speed to keep the amount of shapes constant. The destruction speed is still useful, but if we want to limit the amount of shapes then we have no choice but to add a hard limit. Let's define a shape population limit and make it configurable per level, so add a field for it to `GameLevel`.

```
	[SerializeField]
	int populationLimit;
```


Game Level (Script)

Script	GameLevel
Population Limit	100
Spawn Zone	Spawn Zone (SphereSpawnZ
Persistent Objects	

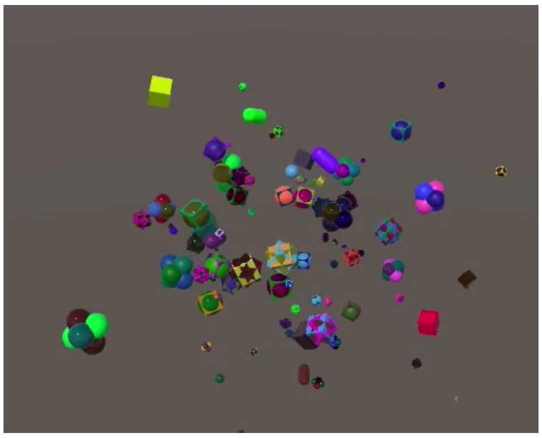
Population limited to 100.

Make the limit available via a public getter property, so `Game` can access it.

```
	public int PopulationLimit {
		get {
			return populationLimit;
		}
	}
```

To enforce the limit, destroy shapes at the end of `Game.FixedUpdate` as long as there are too many of them. We'll only do that if the limit is positive, so zero or a negative value indicates that there is no limit.

```
	void FixedUpdate () {
		…

		int limit = GameLevel.Current.PopulationLimit;
		if (limit > 0) {
			while (shapes.Count > limit) {
				DestroyShape();
			}
		}
	}
```


Enforced population limit.

The next tutorial is Lifecycle.

repository