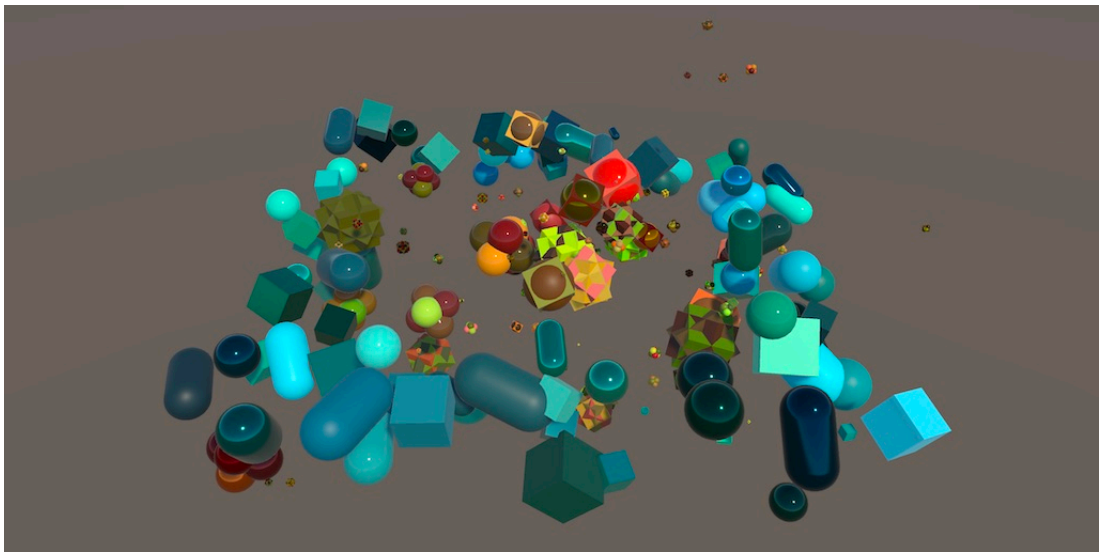
Lifecycle Growth and Death

Make shapes grow and shrink.
Allow behavior to kill shapes.
Delay killing until after the game update loop.
Replace immediate shape destruction with shrinking.

This is the eleventh tutorial in a series about Object Management. It introduces more fluid shape creation and destruction, by adding behavior for growing and dying.

This tutorial is made with Unity 2017.4.12f1.



Limited lifespans keep the population stable.

1 Growing Shapes

Whenever a shape is spawned it instantly appears, at full size. Shapes pop into existence without warning, which can be a jarring experience. One way to make the introduction of a new shape smoother and more gradual is to give them an initial scale of zero and slowly grow them to their full size. Another approach is to initially make them fully transparent and gradually make them more opaque. It's also possible do both at the same time, or do something else. We don't need to limit ourselves to a single approach, so the most flexible way to go about it is to create a behavior, rather than build it into the `Shape` class. In this tutorial we'll go for the first approach: growth.

1.1 Growing Behavior

To support growing shapes, add a `Growing` option to the **`ShapeBehaviorType`** enum.

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

Add a corresponding case to the `GetInstance` method that returns a **`GrowingShapeBehavior`**.

```
			case ShapeBehaviorType.Satellite:
				return ShapeBehaviorPool<SatelliteShapeBehavior>.Get();
			case ShapeBehaviorType.Growing:
				return ShapeBehaviorPool<GrowingShapeBehavior>.Get();
```

Create a bare-bones implementation for the new **`GrowingShapeBehavior`** class.

```
using UnityEngine;

public sealed class GrowingShapeBehavior : ShapeBehavior {

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

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

	public override void Save (GameDataWriter writer) {}

	public override void Load (GameDataReader reader) {}

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

1.2 Going from Zero to Full Scale

The purpose of `GrowingShapeBehavior` is to grow the shape from zero to the scale that we originally gave it. So we have to keep track of the original scale in a field. Also, it takes a while to grow, so we also need a duration field. Both values must also be saved and loaded.

```
	Vector3 originalScale;
	float duration;

	…

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

	public override void Load (GameDataReader reader) {
		originalScale = reader.ReadVector3();
		duration = reader.ReadFloat();
	}
```

The idea is that we add this behavior to a shape that already has its final scale. We'll configure the behavior with via an `Initialize` method, in which we can retrieve the original scale and provide the duration via a parameter. Then we set the shape's scale to zero.

```
	public void Initialize (Shape shape, float duration) {
		originalScale = shape.transform.localScale;
		this.duration = duration;
		shape.transform.localScale = Vector3.zero;
	}
```

In `GameUpdate`, we have to adjust the shape's scale as long as its age is less than the growth duration. The scale factor is simply the age divided by the duration. When the shape is old enough we revert to the original scale and the behavior is no longer needed.

```
	public override bool GameUpdate (Shape shape) {
		if (shape.Age < duration) {
			float s = shape.Age / duration;
			shape.transform.localScale = s * originalScale;
			return true;
		}
		shape.transform.localScale = originalScale;
		return false;
	}
```

1.3 Configuring Growth

The duration of the growing phase is something that we'll configure per spawn zone. Like for the satellite options, we'll define a nested `LifecycleConfiguration` struct in `SpawnZone.SpawnConfiguration` to group all options related to a shape's lifecycle. Right now that's just the growing duration, but we'll add more later. The growing duration can be randomized, but shouldn't be too long, like Somewhere between zero and two seconds.

```
	public struct SpawnConfiguration {
		…

		[System.Serializable]
		public struct LifecycleConfiguration {

			[FloatRangeSlider(0f, 2f)]
			public FloatRange growingDuration;
		}

		public LifecycleConfiguration lifecycle;
	}
```

Growing takes between one and two seconds.

Add a method to `SpawnZone` to take care of setting up the lifecycle of a shape. Besides the shape parameter, also add a parameter for the desired growing duration. If that duration is larger than zero, add a `GrowingShapeBehavior` to the shape. Otherwise we don't need to bother with the behavior.

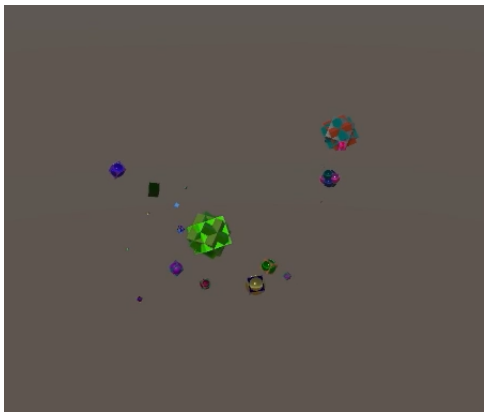
```
	void SetupLifecycle (Shape shape, float growingDuration) {
		if (growingDuration > 0f) {
			shape.AddBehavior<GrowingShapeBehavior>().Initialize(
				shape, growingDuration
			);
		}
	}
```

We made the growing duration a parameter so we can use the same duration for a shape and all its satellites. To make that work, add a duration parameter to `CreateSatelliteFor` and have it invoke `SetupLifecycle` for the satellite shape at the end.

```
	void CreateSatelliteFor (Shape focalShape, float growingDuration) {
		…
		SetupLifecycle(shape, growingDuration);
	}
```

At the end of `SpawnShapes`, randomly determine the growing duration and pass it to all satellites. After the satellites are created we can set up the lifecycle of the main shape. We cannot do that earlier because the scales of the satellites depend on the scale of the focal shape. Initializing the growing behavior sets the scale to zero, so it must be delayed.

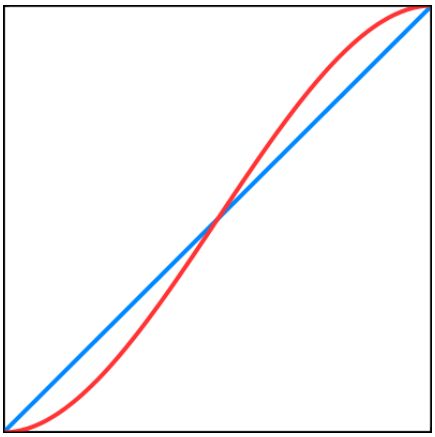
```
	public virtual void SpawnShapes () {
		…

		float growingDuration =
			spawnConfig.lifecycle.growingDuration.RandomValueInRange;

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

		SetupLifecycle(shape, growingDuration);
	}
```



Growing shapes.

1.4 Smooth Growth

When the growing behavior is used, shapes no longer pop into existence immediately. But the growth is linear, so the player has no clue when a shape is done growing. The growing phase just stops at an arbitrary moment. We can make that a bit more smooth and organic by using $3s^2 - 2s^3$ instead of the linear s scale factor. That's known as the smoothstep curve.



Linear and smoothstep.

```
		float s = shape.Age / duration;
		s = (3f - 2f * s) * s * s;
		shape.transform.localScale = s * originalScale;
```

The difference is subtle, but shapes now vary how fast they grow, starting slow, being fastest halfway, and slowing down again when they're almost done.

Smooth growing shapes.

2 Dying Shapes

If we support growing shapes, it's not a big jump to also support dying shapes. Rather than increasing their scale, dying shapes shrink until their scale has been reduced to zero.

2.1 Dying Behavior

Add a `Dying` option to **`ShapeBehaviorType`** and a corresponding case to the `GetInstance` method. Then create a **`DyingShapeBehavior`** class, by duplicating and renaming **`GrowingShapeBehavior`** and adjusting the `BehaviorType` property and `Recycle` method as needed. Such adjustments should be routine by now, so I'm not explicitly showing them.

The dying behavior needs the original scale and a duration, just like growing. But growing assumes that we start at age zero, while dying can start at any age. So we also need to keep track of the age at which we began dying, which is when `Initialize` is invoked. Also, because we're shrinking, the original scale should not be set to zero in `Initialize`.

```
	Vector3 originalScale;
	float duration, dyingAge;

	public void Initialize (Shape shape, float duration) {
		originalScale = shape.transform.localScale;
		this.duration = duration;
		dyingAge = shape.Age;
		//shape.transform.localScale = Vector3.zero;
	}

	…

	public override void Save (GameDataWriter writer) {
		writer.Write(originalScale);
		writer.Write(duration);
		writer.Write(dyingAge);
	}

	public override void Load (GameDataReader reader) {
		originalScale = reader.ReadVector3();
		duration = reader.ReadFloat();
		dyingAge = reader.ReadFloat();
	}
```

`GameUpdate` only needs slight modification. The dying duration is found by subtracting the dying age from the shape's current age. The final scale is zero. And the scalar is reversed, which is done by using 1 minus the duration division as the initial scalar, before smoothing.

```
	public override bool GameUpdate (Shape shape) {
		float dyingDuration = shape.Age - dyingAge;
		if (dyingDuration < duration) {
			float s = 1f - dyingDuration / duration;
			s = (3f - 2f * s) * s * s;
			shape.transform.localScale = s * originalScale;
			return true;
		}
		shape.transform.localScale = Vector3.zero;
		return false;
	}
```

2.2 Configuring Death

How long dying lasts is also something that we'll configure per spawn zone, so add a field for that to `LifecycleConfiguration`, using the same range as for the growing duration.

```
		[FloatRangeSlider(0f, 2f)]
		public FloatRange growingDuration;

		[FloatRangeSlider(0f, 2f)]
		public FloatRange dyingDuration;
```

Because we now have to determine two durations for the lifecycle, let's add a convenient property to `LifecycleConfiguration` that returns two random durations at once, as a `Vector2` with growing as its first and dying as its second component.

```
		public Vector2 RandomDurations {
			get {
				return new Vector2(
					growingDuration.RandomValueInRange,
					dyingDuration.RandomValueInRange
				);
			}
		}
```

Shouldn't we use a custom struct instead of relying on `Vector2`?

That would be a good idea if the functionality was publicly available and used in other parts of our project. But we're only using it once in `SpawnZone`, so a generic vector is fine.

Change `SetupLifecycle` so it uses such a vector as its parameter, instead of a single duration. To quickly test our dying behavior in isolation, we'll have it either add a growing or a dying behavior, but not both. Only if there's no growing duration while we do have a dying duration will we add a dying behavior.

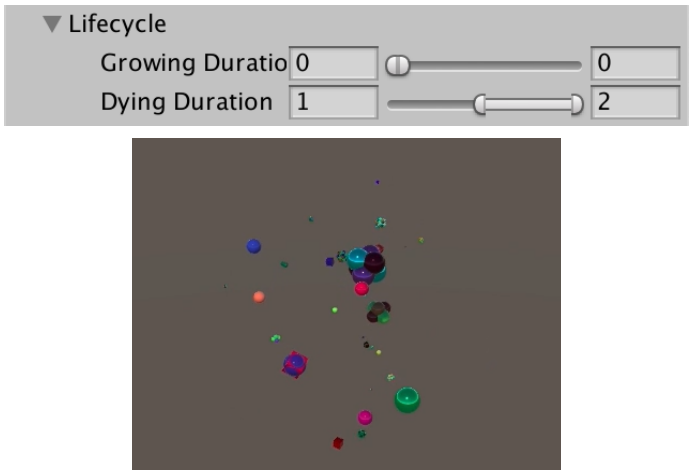
```
	void SetupLifecycle (Shape shape, Vector2 durations) {
		if (durations.x > 0f) {
			shape.AddBehavior<GrowingShapeBehavior>().Initialize(
				shape, durations.x
			);
		}
		else if (durations.y > 0f) {
			shape.AddBehavior<DyingShapeBehavior>().Initialize(
				shape, durations.y
			);
		}
	}
```

Adjust `CreateSatelliteFor` so it passes through the vector instead of a single duration.

```
	void CreateSatelliteFor (Shape focalShape, Vector2 lifecycleDurations) {
		…
		SetupLifecycle(shape, lifecycleDurations);
	}
```

And update `SpawnShapes` as well.

```
		//float growingDuration =
		//	spawnConfig.lifecycle.growingDuration.RandomValueInRange;
		Vector2 lifecycleDurations = spawnConfig.lifecycle.RandomDurations;

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

		SetupLifecycle(shape, lifecycleDurations);
```



Instant growth, slow dying.

2.3 Killing Shapes

When only the dying behavior is in use, we'll see shapes pop into existence that immediately start shrinking and disappear. But even though their scale is reduced to zero they're still alive. The amount of shapes will increase until the level maximum is reached—if set—at which point shapes will get destroyed at random.

The point of the dying behavior is that shapes should die when their scale reaches zero. To support this we have to make it possible for other classes besides `Game` to terminate shapes. So add a public `Kill` method with a shape parameter to `Game`. Just like when destroying a shape, grab its save index, recycle the shape, move the last shape into the hole in the list, then remove the last element of the list.

```
	public void Kill (Shape shape) {
		int index = shape.SaveIndex;
		shape.Recycle();
		int lastIndex = shapes.Count - 1;
		shapes[lastIndex].SaveIndex = index;
		shapes[index] = shapes[lastIndex];
		shapes.RemoveAt(lastIndex);
	}
```

Now we can kill shapes anywhere by invoking `Game.Instance.Kill(shape)`, but let's add a convenient `Die` method to `Shape` to make this easier.

```
	public void Die () {
		Game.Instance.Kill(this);
	}
```

This makes it possible to simply invoke `shape.Die()` in `DyingShapeBehavior.GameUpdate` when it is finished, instead of setting the scale to zero. But because the shape gets recycled—which recycles all its behavior—we must no longer indicate that the behavior should be removed. So return `true` instead of `false`.

```
	public override bool GameUpdate (Shape shape) {
		…
		//shape.transform.localScale = Vector3.zero;
		shape.Die();
		return true;
	}
```

2.4 Delayed Killing

While dead shapes indeed get removed at this point, we're killing them while `Game` is working through its shape list. This causes the order of the shape list to change, moving the last shape in the list to the index that is currently getting updated. A consequence of this is that the shuffled shape gets skipped this update. While shuffling the order in which shapes get updated doesn't matter much, we must ensure that they do always get updated.

To detect the problem when it is about to happen, we must first know whether `Game` is currently working through its shape list. We can do that by adding a boolean field to indicate whether we're currently in the game update loop. Set it to `true` immediately before the loop and to `false` immediately after it.

```
	bool inGameUpdateLoop;

	…

	void FixedUpdate () {
		inGameUpdateLoop = true;
		for (int i = 0; i < shapes.Count; i++) {
			shapes[i].GameUpdate();
		}
		inGameUpdateLoop = false;

		…
	}
```

If we are inside the loop, then we must not mess with the list. If a shape is killed, its removal from the list has to be postponed. We can do that by adding killed shapes to a separate kill list, which we have to keep track of besides the regular shape list.

```
	List<Shape> shapes, killList;

	…

	void Start () {
		mainRandomState = Random.state;
		shapes = new List<Shape>();
		killList = new List<Shape>();

		…
	}
```

Now `Kill` can check whether we're in the game update loop. If so, add the shape to the kill list. Otherwise, the shape can be killed immediately. Move the original kill code to a separate `KillImmediately` method, which should be private.

```
	public void Kill (Shape shape) {
		if (inGameUpdateLoop) {
			killList.Add(shape);
		}
		else {
			KillImmediately(shape);
		}
	}

	void KillImmediately (Shape shape) {
		int index = shape.SaveIndex;
		shape.Recycle();
		int lastIndex = shapes.Count - 1;
		shapes[lastIndex].SaveIndex = index;
		shapes[index] = shapes[lastIndex];
		shapes.RemoveAt(lastIndex);
	}
```

At the end of **`FixedUpdate`**, check whether there are any shapes in the kill list. If so, kill them all immediately and then clear the list.

```
	void FixedUpdate () {
		…

		if (killList.Count > 0) {
			for (int i = 0; i < killList.Count; i++) {
				KillImmediately(killList[i]);
			}
			killList.Clear();
		}
	}
```

We can also use `KillImmediately` in `DestroyShape`, getting rid of duplicate code.

```
	void DestroyShape () {
		if (shapes.Count > 0) {
			//int index = Random.Range(0, shapes.Count);
			//…
			//shapes.RemoveAt(lastIndex);
			Shape shape = shapes[Random.Range(0, shapes.Count)];
			KillImmediately(shape);
		}
	}
```

2.5 Preventing Redundant Kills

The delayed killing approach guarantees that all shapes get updated as they should, but it introduces another potential problem. It is now possible that the same shape gets killed more than once. For example, it's possible for a dying behavior to kill a shape, which then immediately gets destroyed due to the shape limit. And maybe there will be other behaviors that could kill any shape at any time.

We must avoid killing a shape a second time when it's already dead, because that would cause it to get recycled when it shouldn't. Maybe it's already recycled, which would cause it to get pooled twice, leading to trouble later. Maybe it already got reused and immediately gets recycled again when it shouldn't. We can guard agains all these problems by turning the kill list into a list of shape instances and checking whether they're still valid before the kill.

```
	List<Shape> shapes;

	List<ShapeInstance> killList;

	…

	void FixedUpdate () {
		…

		if (killList.Count > 0) {
			for (int i = 0; i < killList.Count; i++) {
				if (killList[i].IsValid) {
					KillImmediately(killList[i].Shape);
				}
			}
			killList.Clear();
		}
	}
```

3 Complete Lifecycle

We have a behavior for growing and a behavior for dying. If we put them together, with a span of adult life in between, we get an entire lifecycle. We could do that by creating a single behavior that contains all the code for growing and dying, but it's also possible to keep using the behavior that we already have, plus an additional lifecycle behavior that adds the others when needed. That might be overkill in this case, but it's an interesting approach to try out so we'll go for it.

3.1 Lifecycle Behavior

Create a new **`LifecycleShapeBehavior`**, linked to a `Lifecycle` enum option. Start with a duplicate of **`DyingShapeBehavior`** and make the require changes.

The lifecycle consists of three phases—growing, adult, and dying—each with its own duration. The growing phase begins immediately, so `Lifecyclebehavior` can immediately add the required behavior in `Initialize`, if needed. This means that it does not need to keep track of the growing duration in a field of its own, just pass the duration to the growing behavior. It also doesn't need to know the original scale. It does need to keep track of both the adult duration and the dying duration. Besides that, the dying age is equal to the growing duration plus the adult duration.

```
	//Vector3 originalScale;
	float adultDuration, dyingDuration, dyingAge;

	public void Initialize (
		Shape shape,
		float growingDuration, float adultDuration, float dyingDuration
	) {
		//originalScale = shape.transform.localScale;
		//this.duration = duration;
		this.adultDuration = adultDuration;
		this.dyingDuration = dyingDuration;
		dyingAge = growingDuration + adultDuration;

		if (growingDuration > 0f) {
			shape.AddBehavior<GrowingShapeBehavior>().Initialize(
				shape, growingDuration
			);
		}
	}

	…

	public override void Save (GameDataWriter writer) {
		writer.Write(adultDuration);
		writer.Write(dyingDuration);
		writer.Write(dyingAge);
	}

	public override void Load (GameDataReader reader) {
		adultDuration = reader.ReadFloat();
		dyingDuration = reader.ReadFloat();
		dyingAge = reader.ReadFloat();
	}
```

In `GameUpdate`, the lifecycle only needs to check whether the shape has reached its dying age. When that happens, it adds the dying behavior and removes itself. This will probably trigger a little too late, so reduce the final dying duration by the lost time, by adding the dying age and subtracting the current age.

```
	public override bool GameUpdate (Shape shape) {
		if (shape.Age >= dyingAge) {
			shape.AddBehavior<DyingShapeBehavior>().Initialize(
				shape, dyingDuration + dyingAge - shape.Age
			);
			return false;
		}
		return true;
	}
```

Actually, a dying behavior is only needed if there is a duration for it. If not, the shape can die immediately and the lifecycle behavior doesn't need to be removed explicitly.

```
		if (shape.Age >= dyingAge) {
			if (dyingDuration <= 0f) {
				shape.Die();
				return true;
			}
			shape.AddBehavior<DyingShapeBehavior>().Initialize(
				shape, dyingDuration + dyingAge - shape.Age
			);
			return false;
		}
```

3.2 Configuring the Lifecycle

To configure the full lifecycle, add an adult duration to `LifecycleConfiguration`. Its `Durations` property becomes a `Vector3`, with the adult duration in place of the dying duration, which moves to the third component.

```
		[FloatRangeSlider(0f, 2f)]
		public FloatRange growingDuration;

		[FloatRangeSlider(0f, 100f)]
		public FloatRange adultDuration;

		[FloatRangeSlider(0f, 2f)]
		public FloatRange dyingDuration;

		public Vector3 RandomDurations {
			get {
				return new Vector3(
					growingDuration.RandomValueInRange,
					adultDuration.RandomValueInRange,
					dyingDuration.RandomValueInRange
				);
			}
		}
```

We only have to change the vector type in `SpawnZone.SpawnShapes`.

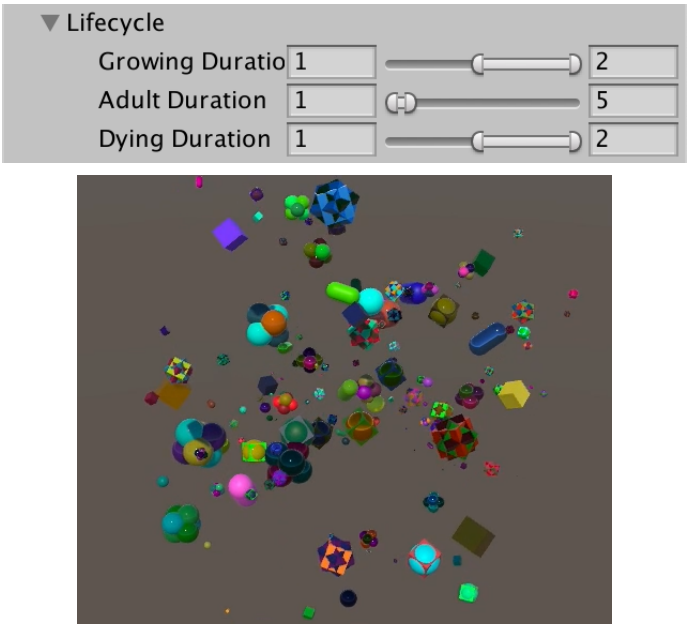
```
		Vector3 lifecycleDurations = spawnConfig.lifecycle.RandomDurations;
```

And in `CreateSatelliteFor`.

```
	void CreateSatelliteFor (Shape focalShape, Vector3 lifecycleDurations) {
		…
	}
```

`SetupLifecycle` becomes a bit more complicated. If there's a growing duration, then if we have at least one of the other durations, we need a full lifecycle. Otherwise only growth is needed. Otherwise, if we have an adult duration then we also need a lifecycle. Finally, we can make do with dying is there's only a dying duration. Make sure to change that code so it uses the third component of the vector.

```
	void SetupLifecycle (Shape shape, Vector3 durations) {
		if (durations.x > 0f) {
			if (durations.y > 0f || durations.z > 0f) {
				shape.AddBehavior<LifecycleShapeBehavior>().Initialize(
					shape, durations.x, durations.y, durations.z
				);
			}
			else {
				shape.AddBehavior<GrowingShapeBehavior>().Initialize(
					shape, durations.x
				);
			}
		}
		else if (durations.y > 0f) {
			shape.AddBehavior<LifecycleShapeBehavior>().Initialize(
				shape, durations.x, durations.y, durations.z
			);
		}
		else if (durations.z > 0f) {
			shape.AddBehavior<DyingShapeBehavior>().Initialize(
				shape, durations.z
			);
		}
	}
```



Adult between one and five seconds.

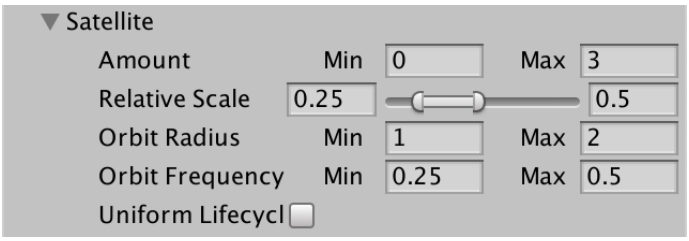
3.3 Different Lifecyle per Satellite

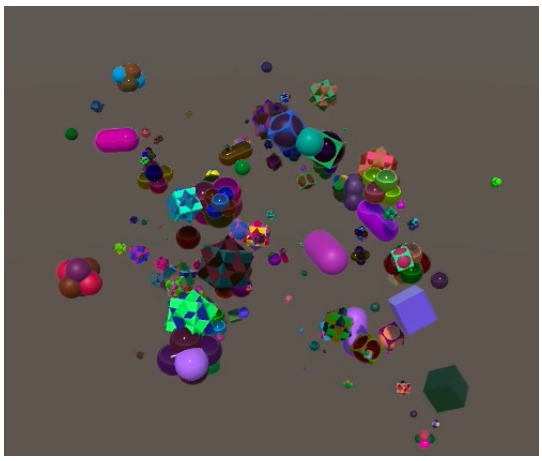
Currently, a shape and all its satellites have the same lifecycle, but that is not required. Let's add a toggle option to the satellite configuration to control whether the lifecycles are uniform.

```
		public struct SatelliteConfiguration {

			…

			public bool uniformLifecycles;
		}
```

In the case of uniform life cycles we keep using the same approach. Otherwise, we use a new set of random durations per satellite.

```
		for (int i = 0; i < satelliteCount; i++) {
			CreateSatelliteFor(
				shape,
				spawnConfig.satellite.uniformLifecycles ?
					lifecycleDurations : spawnConfig.lifecycle.RandomDurations
			);
		}
```





Non-uniform lifecycles.

Now deaths due to the lifecycle can result in escaping satellites, if the focal shape ended up dying first.

4 Destroying Slowly

Killing shapes causes them to shrink and then die, instead of immediately disappearing. But when a shapes gets destroyed—either by the player or because there were too many shapes—they still disappear immediately. We can change such destructions so they become slow shrinking deaths too, but this requires some care.

If shape destruction is just another way to kill them, then we shouldn't bother destroying shapes that are already dying. And as dying shapes are already on their way out, it makes sense to not consider them when checking the shape limit. To be able to do that it must be possible to distinguish between shapes that are dying and those that are not.

One way to make the distinction is to put all dying shapes in a separate shape list and removing them from the regular shape list. Then we automatically ignore dying shapes when picking a random one to destroy and when checking the limit. However, that affects the save index and all places where we manipulate the shape list, because we'd have two lists instead of one.

Another way to make the distinction is via the order of the shape list. We can split the list in two sections, effectively working with two lists while all the code that works with a single list remains valid. That minimizes the changes that we have to make, so we'll use that approach.

Dying and alive shapes in a single list.

4.1 Segregating Dying Shapes

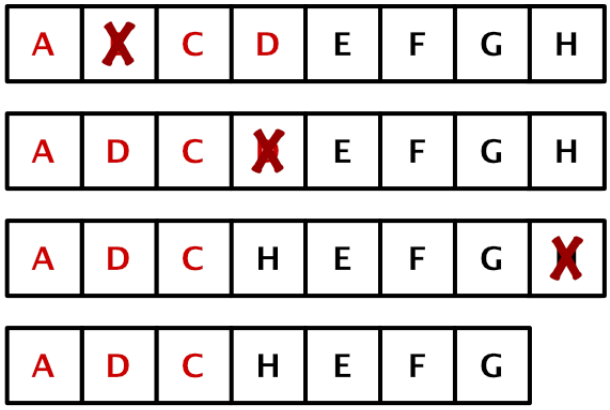
We'll split the shapes list into a dying and non-dying section by moving all dying shapes to the front of the list. As that is a list-order manipulation, it's something that we must be careful with. Add a private `MarkAsDyingImmediately` method to `Game` to put a shape in the dying section. Keep track of the dying shape count in a field and use that as the new index for a shape that's marked as dying, swapping the places with the shape at that index. Afterwards, increment the dying shape count.

```
int dyingShapeCount;

…

void MarkAsDyingImmediately (Shape shape) {
	int index = shape.SaveIndex;
	shapes[dyingShapeCount].SaveIndex = index;
	shapes[index] = shapes[dyingShapeCount];
	shape.SaveIndex = dyingShapeCount;
	shapes[dyingShapeCount++] = shape;
}
```

Marking a shape as dying more than once should have no effect, so abort if the shape is already in the dying section. That's the case when its index is lower than the dying shape count.

```
	int index = shape.SaveIndex;
	if (index < dyingShapeCount) {
		return;
	}
	shapes[dyingShapeCount].SaveIndex = index;
```

This change does affect `KillShapeImmediately`. First, we have to decrement the dying count if a dying shape gets removed. Second, we can no longer blindly move the last shape to the remove shape's index. Doing so could put a non-dying shape in the dying section. That's only the case when a dying shape is removed, if it is not the last in the dying section. In other words, when the shape's index is less than the dying count and also less than the dying count minus one. When that is the case we have to perform a double move: the last dying shape to the removed shape, and the last shape in the list to the hole that was created.



Double move when killing a dying shape.

```
		shape.Recycle();

		if (index < dyingShapeCount && index < --dyingShapeCount) {
			shapes[dyingShapeCount].SaveIndex = index;
			shapes[index] = shapes[dyingShapeCount];
			index = dyingShapeCount;
		}

		int lastIndex = shapes.Count - 1;
```

How does that conditional statement work?

The first condition evaluates as true if we're dealing with a dying shape. Only if that is the case will the second condition get evaluated, which first decrements the dying count and then performs the other comparison. You could also turn it into two nested `if` blocks:

```
		if (index < dyingShapeCount) {
			dyingShapeCount -= 1;
			if(index < dyingShapeCount) {
				shapes[dyingShapeCount].SaveIndex = index;
				shapes[index] = shapes[dyingShapeCount];
				index = dyingShapeCount;
			}
		}
```

But a double move only is only possible if there is at least a single non-dying shape. If there isn't, the hole that we created is at the end of the list, so we don't need to bother moving the last shape at all. As it's never needed to fill a hole at the end of the list, we can just skip that step in general.

```
		int lastIndex = shapes.Count - 1;
		if (index < lastIndex) {
			shapes[lastIndex].SaveIndex = index;
			shapes[index] = shapes[lastIndex];
		}
		shapes.RemoveAt(lastIndex);
```

Now that we know the dying shape count, subtract it from the shape count when checking whether we've exceeded the limit in `FixedUpdate`. That makes it only apply to non-dying shapes. So the total amount of shapes could exceed the limit, until all dying shapes are dead.

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

Likewise, in `DestroyShape` we only go ahead is there are non-dying shapes and then only pick a random shape from the second segment of the list.

```
	void DestroyShape () {
		if (shapes.Count - dyingShapeCount > 0) {
			Shape shape = shapes[Random.Range(dyingShapeCount, shapes.Count)];
			KillImmediately(shape);
		}
	}
```

Finally, make sure to set the dying shape count back to zero each time we begin a new game.

```
	void BeginNewGame () {
		…
		dyingShapeCount = 0;
	}
```

4.2 Delayed Marking

Because marking a shape as dying alters the order of the shape list, we have to make sure that this doesn't happen when we're in the game update loop. We can use the same approach that we use for the kill list, so add a second list for shapes that need to be marked.

```
	List<ShapeInstance> killList, markAsDyingList;

	…

	void Start () {
		mainRandomState = Random.state;
		shapes = new List<Shape>();
		killList = new List<ShapeInstance>();
		markAsDyingList = new List<ShapeInstance>();

		…
	}
```

Loop through this list at the end of **`FixedUpdate`**, immediately marking those elements that are still valid.

```
	void FixedUpdate () {
		…

		if (markAsDyingList.Count > 0) {
			for (int i = 0; i < markAsDyingList.Count; i++) {
				if (markAsDyingList[i].IsValid) {
					MarkAsDyingImmediately(markAsDyingList[i].Shape);
				}
			}
			markAsDyingList.Clear();
		}
	}
```

Now we can add a public `MarkAsDying` method that either add a shape to the list or immediately marks it.

```
	public void MarkAsDying (Shape shape) {
		if (inGameUpdateLoop) {
			markAsDyingList.Add(shape);
		}
		else {
			MarkAsDyingImmediately(shape);
		}
	}
```

And we can also add another convenient method to `Shape`.

```
	public void MarkAsDying () {
		Game.Instance.MarkAsDying(this);
	}
```

The only place where we need to invoke this method is when a `DyingShapeBehavior` is initialized.

```
	public void Initialize (Shape shape, float duration) {
		…
		shape.MarkAsDying();
	}
```

If there were other behaviors that represented different ways to die, then those should also mark their shape as dying during initialization.

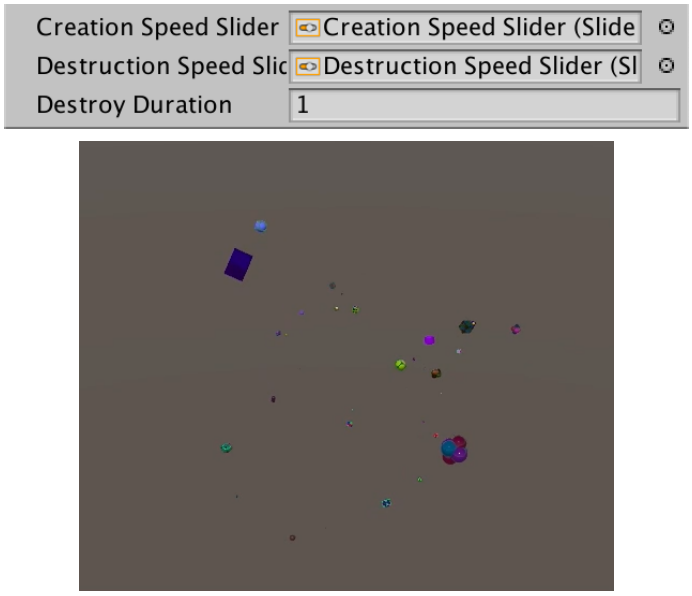
4.3 Slow Destruction

To finally support slow destruction we need to decide on a destruction duration. Make that configurable by adding a serializable field to `Game`.

```
	[SerializeField] float destroyDuration;
```

When the duration is positive, have `DestroyShape` add a dying behavior to the shape with that duration, instead of killing it immediately.

```
	void DestroyShape () {
		if (shapes.Count - dyingShapeCount > 0) {
			Shape shape = shapes[Random.Range(dyingShapeCount, shapes.Count)];
			if (destroyDuration <= 0f) {
				KillImmediately(shape);
			}
			else {
				shape.AddBehavior<DyingShapeBehavior>().Initialize(
					shape, destroyDuration
				);
			}
		}
```



Destruction takes one second; population limit 20.

4.4 Preventing Double Dying

The destruction of shapes happens independent of their lifecycle. That means that a random destruction could add the dying behavior to a shape that is still growing. That isn't a problem, because the dying behavior was added later so overrides the scale change of the growing behavior. What's more troublesome is that the lifecycle can still add a dying behavior even though one has already been added. The second behavior initiates a new shrinking effect that overrides the first, but whichever completes first decides when the shape is killed.

To prevent adding a second dying behavior to a shape it must be possible to check whether the shape is already dying, no matter why. We can add an `IsMarkedAsDying` method to `Game` to check this. All it has to do is check whether the shape's index is less than the dying count.

```
	public bool IsMarkedAsDying (Shape shape) {
		return shape.SaveIndex < dyingShapeCount;
	}
```

Once again we make this conveniently available via `Shape`, though in this case a readonly property is appropriate.

```
	public bool IsMarkedAsDying {
		get {
			return Game.Instance.IsMarkedAsDying(this);
		}
	}
```

Finally, in `LifecycleShapeBehavior.GameUpdate` check whether the shape is already dying when it reached its dying age. If so, don't add the dying behavior.

```
			if (dyingDuration <= 0f) {
				shape.Die();
				return true;
			}
			if (!shape.IsMarkedAsDying) {
				shape.AddBehavior<DyingShapeBehavior>().Initialize(
					shape, dyingDuration + dyingAge - shape.Age
				);
			}
			return false;
```

The next tutorial is More Complex Levels.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick