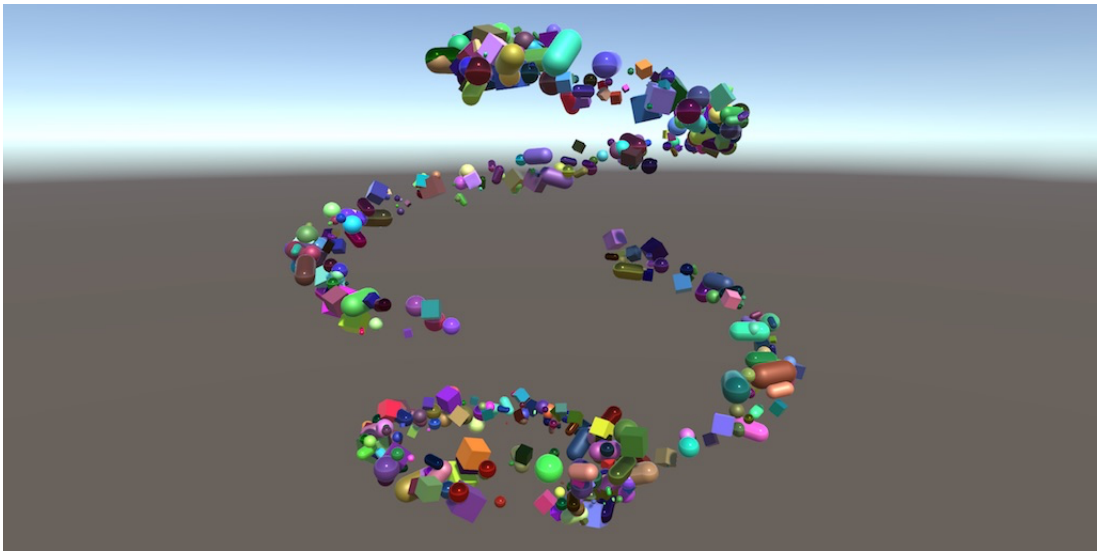
More Game State Saving All That Matters

Keep track of randomness.
Save level data.
Loop through spawn zones.
Create a rotating level object.

This is the sixth tutorial in a series about Object Management. It covers saving more game state, in addition to the spawned shapes and level index.

This tutorial is made with Unity 2017.4.4f1.



A reproducible trail of random shapes.

1 Saving Randomness

The point of using randomness when spawning shapes is to get unpredictable results. But this isn't always desirable. Suppose you save the game, then spawn a few more shapes. Then you load, and spawn the same amount of shapes again. Should you end up with the exact same shapes, or different ones? Currently, you'll get different ones. But the other option is equally valid, so let's support it.

The numbers generated by Unity's `Random` methods aren't truly random. They're pseudorandom. It's a sequence of numbers generated by a mathematical formula. At the start of the game, this sequence gets initialized with an arbitrary seed value, based on the current time. If you start a new sequence with the same seed, you'll get the same numbers.

1.1 Writing Random State

Storing the initial seed value is not enough, because that would bring us back to the start of the sequence, instead of the point in the sequence when the game got saved. But `Random` must keep track of where in the sequence it is. If we can get to this state, then we can later restore it, continuing the old sequence.

The random state is defined as a `State` struct, nested inside the `Random` class. So we can declare a field or parameter of this type, `Random.State`. To save it, we have to add a method to `GameDataWriter` that can write such a value. Let's add this method now, but leave its implementation for later.

```
	public void Write (Random.State value) {}
```

With this method, we can save the random state of the game. We'll do this at the start of `Game.Save`, right after writing the shape count. Again, increment the save version to signal a new format.

```
	const int saveVersion = 3;

	…

	public override void Save (GameDataWriter writer) {
		writer.Write(shapes.Count);
		writer.Write(Random.state);
		writer.Write(loadedLevelBuildIndex);
		…
	}
```

1.2 Reading Random State

To read the random state, add a `ReadRandomState` method to `GameDataReader`. As we're not writing anything yet, skip reading anything for now. Instead, return the current random state, so nothing changes. The current state can be found via the static `Random.state` property.

```
	public Random.State ReadRandomState () {
		return Random.state;
	}
```

Setting the random state is done via the same property, which we'll do `Game.Load`, but only for save file versions 3 and higher.

```
	public override void Load (GameDataReader reader) {
		…
		int count = version <= 0 ? -version : reader.ReadInt();

		if (version >= 3) {
			Random.state = reader.ReadRandomState();
		}

		StartCoroutine(LoadLevel(version < 2 ? 1 : reader.ReadInt()));
		…
	}
```

1.3 JSON Serialization

`Random.State` contains four floating-point numbers. However, they aren't publicly accessible, so it is not possible for us to simply write them. We have to use an indirect approach instead.

Fortunately, `Random.State` is a serializable type, so it is possible to convert it to a string representation of the same data, using the `ToJson` method of Unity's `JsonUtility` class. This gives us a JSON string. To see what that looks like, log it to the console.

```
	public void Write (Random.State value) {
		Debug.Log(JsonUtility.ToJson(value));
	}
```

What does Json mean?

The proper spelling is JSON, all capital letters. It stands for JavaScript Object Notation. It defines a simple human-readable data format.

After saving a game, the console will now log a string of four numbers named *s0* through *s3*, between curly brackets. Something like *{"s0":-1409360059,"s1":1814992068,"s2":-772955632,"s3":1503742856}*.

We'll write this string to our file. If you were to open the safe file with a text editor, you'll be able to see this string near the beginning of the file.

```
	public void Write (Random.State value) {
		writer.Write(JsonUtility.ToJson(value));
	}
```

In `ReadRandomState`, read this string by invoking `ReadString` and then use `JsonUtility.FromJson` to convert it back to a proper random state.

```
	public Random.State ReadRandomState () {
		return JsonUtility.FromJson(reader.ReadString());
	}
```

Besides the data, `FromJson` also needs to know the type of whatever it's supposed to create from the JSON data. We can use the generic version of the method, specifying that it should create a `Random.State` value.

```
	public Random.State ReadRandomState () {
		return JsonUtility.FromJson<Random.State>(reader.ReadString());
	}
```

1.4 Decoupling Levels

Our game now saves and restores the random state. You can verify this by beginning a game, saving, creating a few shapes, then loading, and creating the exact same shapes again. But you can go a step further. You could even begin a new game after loading, and still create the same shapes after that. So we can influence the randomness of a new game by loading a game right before it. This is not desirable. Ideally, the randomness of distinct games is separate, as if we restarted the whole game. We can achieve this by seeding a new random sequence each time we begin a new game.

To pick a new seed value, we have to use randomness. We can use `Random.value` for that, but have to make sure that these values come from their own random sequence. To do this, add a main random state field to `Game`. At the start of the game, set it to the random state that was initialized by Unity.

```
Random.State mainRandomState;

…

void Start () {
    mainRandomState = Random.state;
    …
}
```

When the player begins a new game, the first step is now to restore the main random state. Then grab a random value and use that as the seed to initialize a new pseudorandom sequence, via the `Random.InitState` method.

```
void BeginNewGame () {
    Random.state = mainRandomState;
    int seed = Random.Range(0, int.MaxValue);
    Random.InitState(seed);

    …
}
```

To make the seeds a little more unpredictable, we'll mix them with the current play time, accessible via `Time.unscaledTime`. The bitwise exclusive-OR operator `^` is good for this.

```
    int seed = Random.Range(0, int.MaxValue) ^ (int)Time.unscaledTime;
```

What does exclusive-OR do?

For each bit, the result is 1 if exactly one of the two inputs is 1 and the other is 0. Otherwise, the result is 0. In other words, whether the inputs are different. Because it is a bit manipulation, the result isn't mathematically obvious, like addition would be.

To keep track of the progression of the main random sequence, store its state after grabbing the next value, before initializing the state for the new game.

```
    Random.state = mainRandomState;
    int seed = Random.Range(0, int.MaxValue);
    mainRandomState = Random.state;
    Random.InitState(seed);
```

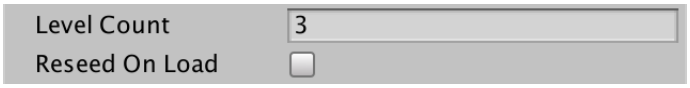
Now loading games and what you do in each game no longer affects the randomness of other games played during the same session. But to make sure this works correctly we also have to invoke `BeginNewGame` for the first game of each session.

```
	void Start () {
		…

		BeginNewGame();
		StartCoroutine(LoadLevel(1));
	}
```

1.5 Supporting Both Approaches

You might not want reproducible randomness, instead preferring to get new results after loading. So let's support both approaches, by adding a `reseedOnLoad` toggle to `Game`.

```
	[SerializeField] bool reseedOnLoad;
```



Option to reseed on load.

All we have to change is whether the random state is set when a game is loaded. We can keep saving—and loading—it, so the save files always support both options.

```
	public override void Load (GameDataReader reader) {
		…

		if (version >= 3) {
			//Random.state = reader.ReadRandomState();
			Random.State state = reader.ReadRandomState();
			if (!reseedOnLoad) {
				Random.state = state;
			}
		}

		…
	}
```

2 Persisting Level Data

We can save shapes that have been spawned during a game, we can save which level we're playing, and we can save the random state. We could use the same approach to save comparable data, like how many shapes were spawned and destroyed, or other things that could be created while playing. But what if we would like to save the state of something that is part of the level? Something we put in the level scene, but changes during play? To support that, we would have to save the state of the level too.

2.1 Current Level Instead of Game Singleton

To save the level, `Game` must include it when saving itself. This means that it must somehow get a reference to the current level. We could add a property to `Game` and have the loaded level assign itself to that, but then we're putting knowledge two level-related things directly inside `Game`: the level itself and its spawn zone. This can be a valid approach, but let's turn it around. Instead of relying on a `Game` singleton, make the current level globally accessible.

Add a static `Current` property to `GameLevel`. Everyone can get the current level, but only the level itself can set it, which it does when it becomes enabled.

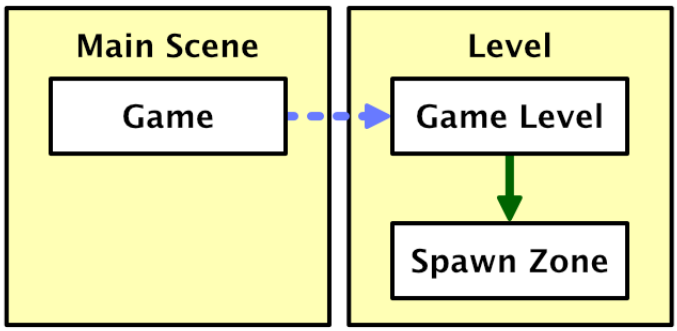
```
	public static GameLevel Current { get; private set; }

	…

	void OnEnable () {
		Current = this;
	}
```

Now instead of setting the game's spawn point, the level can expose its spawn point for the game to use. In fact, we can go a step further and have `GameLevel` directly offer a `SpawnPoint` property, forwarding the request to its spawn zone. Thus, the level acts as a facade for its spawn zone.

```
	public Vector3 SpawnPoint {
		get {
			return spawnZone.SpawnPoint;
		}
	}

	…

	//void Start () {
	//	Game.Instance.SpawnZoneOfLevel = spawnZone;
	//}
```

This means that `Game` no longer needs to know about spawn zones. It just asks the current level for a point.

```
	//public SpawnZone SpawnZoneOfLevel { get; set; }

	…

	void CreateShape () {
		…
		t.localPosition = GameLevel.Current.SpawnPoint;
		…
	}
```



Game only knows about the current level.

At this point, `GameLevel` doesn't need to reference `Game` anymore. As the static instance isn't used anywhere else, let's remove it.

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

	…

	//void OnEnable () {
	//	Instance = this;
	//}

	void Start () {
		mainRandomState = Random.state;
		//Instance = this;
		…
	}
```

Can't we keep `Game.Instance`, even if it isn't used?

You can, but leaving unused code—known as dead code—in a project makes it harder to maintain. It's such simple code that if we need it in the future, we'll just add it again.

2.2 Persistable Game Level

To make it possible to save the level, make it a `PersistableObject`. The transformation data of the level object itself is of no use, so overwrite the `Save` and `Load` methods so they do nothing, for now.

```
public class GameLevel : PersistableObject {

	…

	public override void Save (GameDataWriter writer) {}

	public override void Load (GameDataReader reader) {}
}
```

In `Game.Save`, it makes sense to write the level data before everything that was created while playing. Let's put it directly after the level build index.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(shapes.Count);
		writer.Write(Random.state);
		writer.Write(loadedLevelBuildIndex);
		GameLevel.Current.Save(writer);
		for (int i = 0; i < shapes.Count; i++) {
			…
		}
	}
```

2.3 Loading Level Data

When loading, we must now read the level data after reading the level build index. However, we can only do so after the level scene has been loaded, otherwise we would be applying it to the level scene that is about to be unloaded. Thus, we must postpone reading the rest of the save file until after the `LoadLevel` coroutine has finished. To make this possible, let's turn the entire loading process into a coroutine.

After verifying that the save version is supported, start a new `LoadGame` coroutine and then end `Game.Load`. The code that used to come after this point becomes the new `LoadGame` coroutine method, which requires the reader as a parameter.

```
	public override void Load (GameDataReader reader) {
		int version = reader.Version;
		if (version > saveVersion) {
			Debug.LogError("Unsupported future save version " + version);
			return;
		}
		StartCoroutine(LoadGame(reader));
	}

	IEnumerator LoadGame (GameDataReader reader) {
		int version = reader.Version;
		int count = version <= 0 ? -version : reader.ReadInt();

		if (version >= 3) {
			Random.State state = reader.ReadRandomState();
			if (!reseedOnLoad) {
				Random.state = state;
			}
		}

		StartCoroutine(LoadLevel(version < 2 ? 1 : reader.ReadInt()));

		for (int i = 0; i < count; i++) {
			int shapeId = version > 0 ? reader.ReadInt() : 0;
			int materialId = version > 0 ? reader.ReadInt() : 0;
			Shape instance = shapeFactory.Get(shapeId, materialId);
			instance.Load(reader);
			shapes.Add(instance);
		}
	}
```

In `LoadGame`, yield on `LoadLevel` instead of invoking `StartCoroutine`. After that we can invoke `GameLevel.Current.Load`, assuming we have a file version 3 or higher.

```
		//StartCoroutine(LoadLevel(version < 2 ? 1 : reader.ReadInt()));
		yield return LoadLevel(version < 2 ? 1 : reader.ReadInt());
		if (version >= 3) {
			GameLevel.Current.Load(reader);
		}
```

Unfortunately, we get an error at this point when trying to load a game.

2.4 Buffering the Data

The error that we get tells us that we're trying to read from a closed `BinaryReader` instance. It got closed because of the `using` block inside `PersistentStorage.Load`. It guarantees that the hold that we got on the file gets released as soon as that method invocation finishes. We're now trying to read the level data later—via the coroutine—so it fails.

There are two ways to solve this problem. The first is to do away with the `using` block, instead releasing the hold on the save file manually later, by closing the reader explicitly. That would require us to careful track whether we're holding on to a reader and make sure to close it, even if we encounter an error along the way. The second approach is to read the entire file in one go, buffering it, and then reading from the buffer later. That means that we don't have to worry about releasing the file, but have to store its entire contents in memory for a while. As our save files are small, we'll use the buffer.

Reading the entire file can be done by invoking `File.ReadAllBytes`, which gives us a byte array. This will be our new approach in `PersistentStorage.Load`.

```
	public void Load (PersistableObject o) {
		//using (
		//	var reader = new BinaryReader(File.Open(savePath, FileMode.Open))
		//) {
		//	o.Load(new GameDataReader(reader, -reader.ReadInt32()));
		//}

		byte[] data = File.ReadAllBytes(savePath);
	}
```

We still have to use a `BinaryReader`, which requires a stream, not an array. We can create a `MemoryStream` instance that wraps the array, and give that to the reader. Then load with a `GameDataReader`, as before.

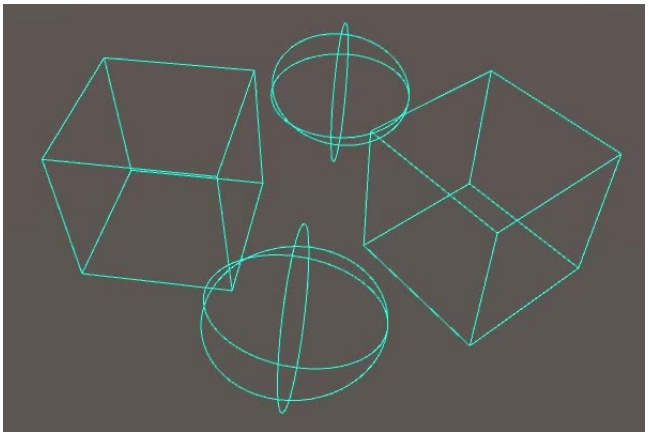
```
	public void Load (PersistableObject o) {
		byte[] data = File.ReadAllBytes(savePath);
		var reader = new BinaryReader(new MemoryStream(data));
		o.Load(new GameDataReader(reader, -reader.ReadInt32()));
	}
```

3 Level State

We've made it possible to save level data, but at this point we don't have anything to store yet. So let's come up with something to save.

3.1 Sequential Composite Spawn Zone

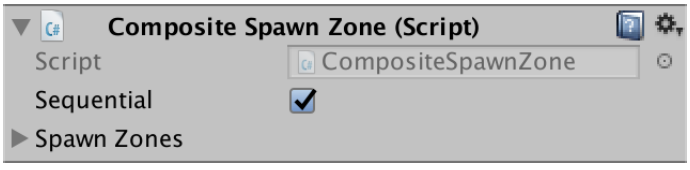
The most complex level structure that we have so far is the composite spawn zone. It has an array of spawn zones, of which one element is used each time a new spawn point is needed. When looking at this in action, you cannot predict which zone is used next. The placement of shapes is arbitrary and need not be uniform, although it will average out across all zones in the long run.



Random spawning.

We can change this by looping through the spawn zones in sequence. Both approaches are valid, so we'll support both. Add a toggle option to `CompositeSpawnZone` to make it sequential.

```
	[SerializeField]
	bool sequential;
```



Sequential composite spawn zone.

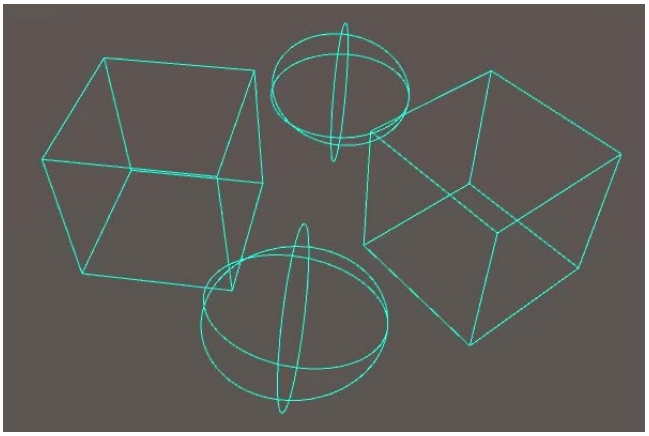
Sequential spawning requires us to keep track of which zone index has to be used next. So add a `nextSequentialIndex` field and use that for the index in `SpawnPoint`, if we're in sequential mode. Increment the field afterwards.

```
	int nextSequentialIndex;

	public override Vector3 SpawnPoint {
		get {
			int index;
			if (sequential) {
				index = nextSequentialIndex++;
			}
			else {
				index = Random.Range(0, spawnZones.Length);
			}
			return spawnZones[index].SpawnPoint;
		}
	}
```

To make it loop, jump back to the first index when we move past the end of the array.

```
			if (sequential) {
				index = nextSequentialIndex++;
				if (nextSequentialIndex >= spawnZones.Length) {
					nextSequentialIndex = 0;
				}
			}
```

A sequential spawn zone behaves noticeably different than a random one. Its spawn pattern is clear and the shapes are evenly distributed among the zones, although their placement inside each zone is still random.



Sequential spawning.

3.2 Remembering the Next Index

When saving a game, the state of the sequential composite spawn zone must now be saved, otherwise the sequence will reset after a load. So it must become a persistable object. It already extends `SpawnZone`, so we must make `SpawnZone` extend `PersistableObject`. That makes it possible for all spawn zone types to persist their state.

```
public abstract class SpawnZone : PersistableObject {

	public abstract Vector3 SpawnPoint { get; }
}
```

Overwrite the `Save` and `Load` methods in `CompositeSpawnZone`, simply writing and reading `nextSequentialIndex`. We'll do this regardless whether the zone is sequential or not. We could also invoke the base methods, to also save the zone's transform data, but let's focus on the sequence only. The zone doesn't move on its own.

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

	public override void Load (GameDataReader reader) {
		nextSequentialIndex = reader.ReadInt();
	}
```

3.3 Tracking Persistable Objects

The spawn zones are now persistable, but they aren't getting saved yet. `GameLevel` has to invoke their `Save` and `Load` methods. We could simply use the `spawnZone` field, but that only allows a single spawn zone to be saved. What if we want to put multiple sequential spawn zones in a level, all part of a hierarchy of composite zones?

We could make the composite zone responsible for also saving and loading all zones it contains, but what if we add other things to the level that should also be saved? To make it as flexible as possible, let's add a way to configure which objects are supposed to be persisted when a level is saved. The simplest approach is to add an array of persistent objects to `GameLevel` that we can fill when designing the level scene.

```
	[SerializeField]
	PersistableObject[] persistentObjects;
```

Now `GameLevel` can save how may such objects there are, then save each of them, just like `Game` does for its shape list.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(persistentObjects.Length);
		for (int i = 0; i < persistentObjects.Length; i++) {
			persistentObjects[i].Save(writer);
		}
	}
```

And the same goes for loading, but because the level object are part of the scene, nothing needs to be instantiated.

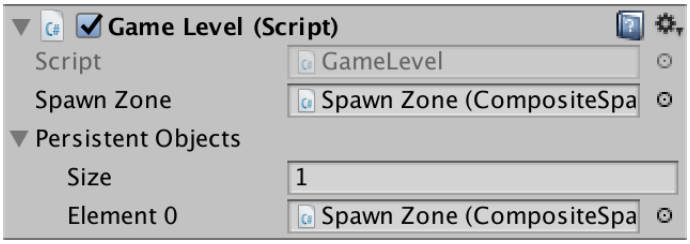
```
	public override void Load (GameDataReader reader) {
		int savedCount = reader.ReadInt();
		for (int i = 0; i < savedCount; i++) {
			persistentObjects[i].Load(reader);
		}
	}
```

Note that from now on you must ensure that what you put into this array stays in it, at the same index, otherwise you break backwards compatibility with older save files. However, you can add more to it in the future. Those new objects will be skipped when loading old files, remaining how they were saved in the scene.

Another important point is that the `GameLevel` instances in all our scenes haven't automatically gained the new array. You have to open and save all level scenes, otherwise you can end up with a null-reference exception when loading a level. Alternatively, we can check whether an array exists when a level object is enabled in play. If not, create one. That's a more convenient approach if you have many levels and the only option if third parties have created levels for your game that you still wish to support.

```
	void OnEnable () {
		Current = this;
		if (persistentObjects == null) {
			persistentObjects = new PersistableObject[0];
		}
	}
```

Now we can finally save the sequential composite spawn zone, by explicitly adding it to the persistent objects of the level.



Level 3 with persistent spawn zone.

3.4 Reloading for a New Game

The sequence index now gets restored when loading a level, but it currently doesn't get reset when the player begins a new game in the same level. The solution is to also load the level in this case, resetting the entire level state.

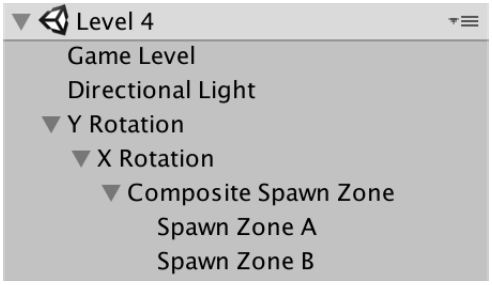
```
		else if (Input.GetKeyDown(newGameKey)) {
			BeginNewGame();
			StartCoroutine(LoadLevel(loadedLevelBuildIndex));
		}
```

3.5 Rotating Objects

Let's add another kind of level object that also has to store state. A simple rotating object. It's a persistable object that has a configurable angular velocity. Use a 3D vector, so the velocity can be in any direction. To make it rotate, give it an `Update` method that invokes the `Rotate` method of its transformation, with the velocity scaled by the time delta as its argument.

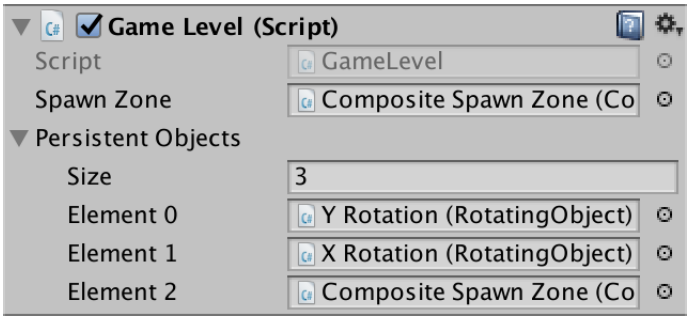
```
using UnityEngine;

public class RotatingObject : PersistableObject {

	[SerializeField]
	Vector3 angularVelocity;

	void Update () {
		transform.Rotate(angularVelocity * Time.deltaTime);
	}
}
```

To demonstrate the rotating object, I have created a fourth scene. In it, there is a root object that rotates around the Y axis with a velocity of 90. Its only child is another rotating object with a velocity of 15 around the X axis. One level deeper sits a sequential composite spawn zone, with two sphere spawn zone children. Both spheres have a radius of 1 and are positioned ten units from the origin in both directions along the Z axis.



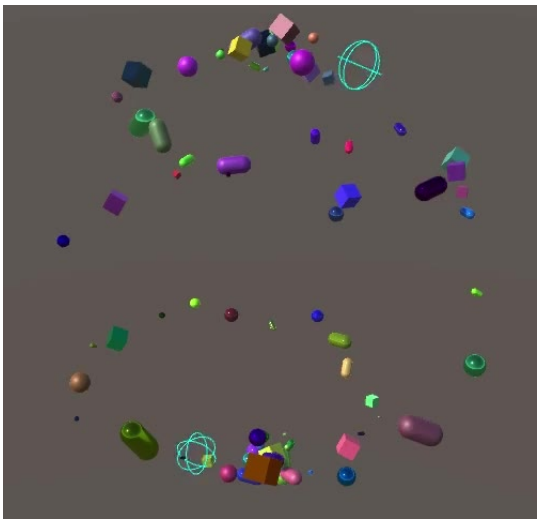
Rotating spawn zone hierarchy.

To persist the level state, both rotating objects and the composite spawn zone must by put in the persistent object array. Their order doesn't matter, but should not be changed later.



Persistent objects of level 4.

This configuration creates two small spawn zones on opposite sides of a larger sphere, spinning around it, and going up and down.



Rotating Spawn Zones.

It is easiest to see it in action by using an automatic creation speed, instead of spawning shapes manually. You can then also test saving and loading, to verify that the level state is indeed persisted and restored. However, sometimes we can end up with different spawn results. We'll deal with that in the next section.

4 Creation and Destruction

The automatic creation and destruction process is also part of the game state. We're currently not saving that, so the creation and destruction progress is unaffected by saving and loading. This means that when the creation speed is larger than zero, you likely won't get the exact same shape placement after loading a game. The same goes for the timing of shape destruction. We should make sure that the timing remains exactly the same.

4.1 Saving and Loading

Saving the progress is simply a matter of writing both values in `Game.Save`. Let's do so after writing the random state.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(shapes.Count);
		writer.Write(Random.state);
		writer.Write(creationProgress);
		writer.Write(destructionProgress);
		writer.Write(loadedLevelBuildIndex);
		GameLevel.Current.Save(writer);
		…
	}
```

When loading, read them back at the appropriate moment.

```
	IEnumerator LoadGame (GameDataReader reader) {
		int version = reader.Version;
		int count = version <= 0 ? -version : reader.ReadInt();

		if (version >= 3) {
			Random.State state = reader.ReadRandomState();
			if (!reseedOnLoad) {
				Random.state = state;
			}
			creationProgress = reader.ReadFloat();
			destructionProgress = reader.ReadFloat();
		}

		yield return LoadLevel(version < 2 ? 1 : reader.ReadInt());
		…
	}
```

4.2 Exact Timing

We still don't get the exact same timing. That's because our game's frame rate is not perfectly stable. The time delta of each frame is variable. If a frame takes longer than before, it might be enough for a shape to be spawned a frame earlier than the previous time. Or it might appear a frame later. Combined with a moving spawn zone based on the same time delta, the shape can end up somewhere else.

We can make the timing exact by use a fixed time delta to update the creation and destruction progress. This is done by moving the relevant code from the `Update` method to a new `FixedUpdate` method.

```
	void Update () {
		if (Input.GetKeyDown(createKey)) {
			CreateShape();
		}
		…
		else {
			…
		}
	}

	void FixedUpdate () {
		creationProgress += Time.deltaTime * CreationSpeed;
		while (creationProgress >= 1f) {
			creationProgress -= 1f;
			CreateShape();
		}

		destructionProgress += Time.deltaTime * DestructionSpeed;
		while (destructionProgress >= 1f) {
			destructionProgress -= 1f;
			DestroyShape();
		}
	}
```

Now the automatic creation and destruction of shapes is no longer affected by the variable frame rate. But the rotator still is. To make it perfect, we should use `FixedUpdate` for the rotation in `RotatingObject` as well.

```
	void FixedUpdate () {
		transform.Rotate(angularVelocity * Time.deltaTime);
	}
```

When does `FixedUpdate` get invoked?

It gets invoked each frame, after **`Update`**. How many times it gets invoked depends on the frame time and the fixed time step, which you can configure via *Edit / Project Settings / Time*.

The default fixed time step is 0.02, which is 50 times per second. So if your game runs at exactly 10 frames per second, **`FixedUpdate`** would get invoked five times each frame. And if your game runs at more than 50 frames per second, then sometimes **`FixedUpdate`** won't be invoked at all during a frame. You can use a different time step, if you need more or less time granularity.

You use **`FixedUpdate`** when working with the physics engine, or when you want reliable reproducible timing, which is the case in this tutorial.

4.3 Speed Settings

Besides the progress, we can also consider the speed settings part of the game state. All we have to do is write the speed properties too, when saving.

```
	public override void Save (GameDataWriter writer) {
		writer.Write(shapes.Count);
		writer.Write(Random.state);
		writer.Write(CreationSpeed);
		writer.Write(creationProgress);
		writer.Write(DestructionSpeed);
		writer.Write(destructionProgress);
		writer.Write(loadedLevelBuildIndex);
		GameLevel.Current.Save(writer);
		for (int i = 0; i < shapes.Count; i++) {
			writer.Write(shapes[i].ShapeId);
			writer.Write(shapes[i].MaterialId);
			shapes[i].Save(writer);
		}
	}
```

And read them when loading.

```
		if (version >= 3) {
			Random.State state = reader.ReadRandomState();
			if (!reseedOnLoad) {
				Random.state = state;
			}
			CreationSpeed = reader.ReadFloat();
			creationProgress = reader.ReadFloat();
			DestructionSpeed = reader.ReadFloat();
			destructionProgress = reader.ReadFloat();
		}
```

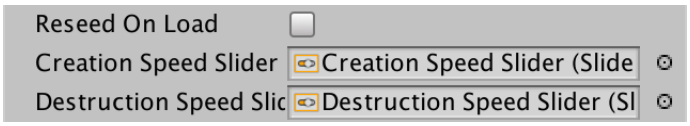
It also makes sense to reset the speeds when beginning a new game.

```
	void BeginNewGame () {
		…
		Random.InitState(seed);

		CreationSpeed = 0;
		DestructionSpeed = 0;

		…
	}
```

4.4 Updating the Labels

The speed settings are now saved, and get restored when we load a game. But the UI isn't aware of this, so won't change if we happen to load a different speed. We have to manually refresh the sliders after loading. To make this possible, `Game` needs to reference the sliders, so add two configuration fields for them.

```
…
using UnityEngine.UI;

[DisallowMultipleComponent]
public class Game : PersistableObject {

	…

	[SerializeField] Slider creationSpeedSlider;
	[SerializeField] Slider destructionSpeedSlider;

	…
}
```



References to the sliders.

Isn't there a way to bind the UI to the properties?

There is no built-in way to do this. We could come up with a custom solution, but that's out of the scope of this tutorial. For our simple case, slider references suffice.

When resetting the speeds, we can now also update the sliders, by assigning to their `value` properties as well.

```
		CreationSpeed = 0;
		creationSpeedSlider.value = 0;
		DestructionSpeed = 0;
		destructionSpeedSlider.value = 0;
```

This code can be made more concise by chaining the assignments.

```
		//CreationSpeed = 0;
		creationSpeedSlider.value = CreationSpeed = 0;
		//DestructionSpeed = 0;
		destructionSpeedSlider.value = DestructionSpeed = 0;
```

Do the same in the `Load` method.

```
		creationSpeedSlider.value = CreationSpeed = reader.ReadFloat();
		creationProgress = reader.ReadFloat();
		destructionSpeedSlider.value = DestructionSpeed = reader.ReadFloat();
		destructionProgress = reader.ReadFloat();
```

The UI now also updates after loading or beginning a new game.

Want to know when the next tutorial is released? Keep tabs on my Patreon page!

made by Jasper Flick