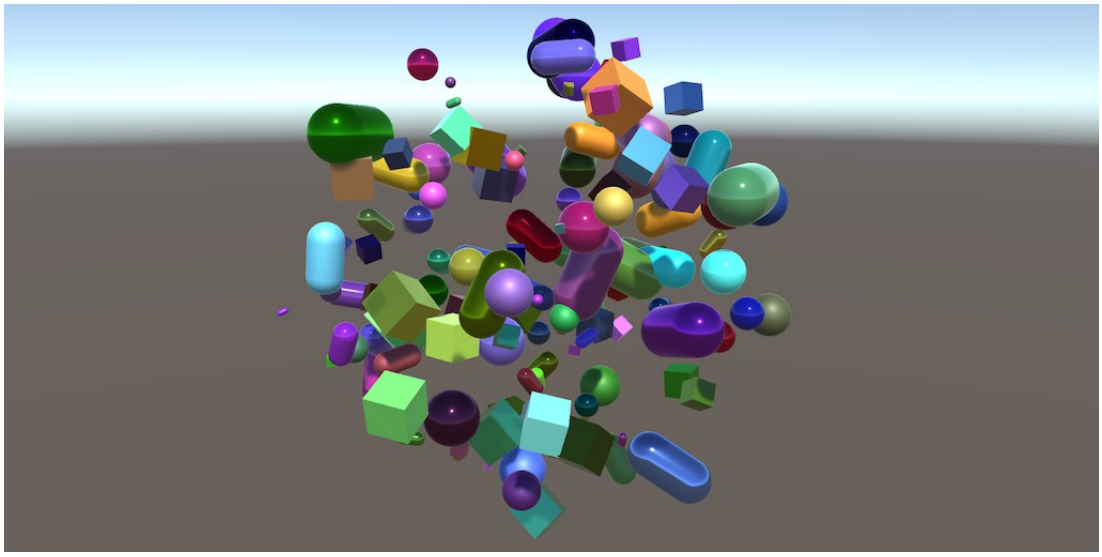
Object Variety Fabricating Shapes

Create a factory for shapes.
Save and load shape identifiers.
Support multiple materials and random colors.
Enable GPU instancing.

This is the second tutorial in a series about Object Management. In this part we'll add support for multiple shapes with varying materials and colors, while remaining backwards compatible with the previous version of our game.

This tutorial is made with Unity 2017.4.1f1.



These cubes survived the termination of their game.

1 Shape Factory

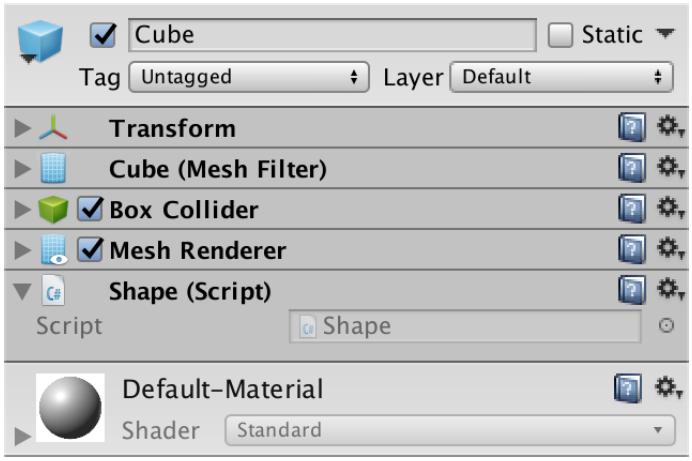
The goal of this tutorial is to make our game more interesting, by allowing the creation of other shapes than just white cubes. Just like the position, rotation, and scale, we'll randomize what shape is created each time the player spawns a new one.

1.1 Shape Class

We're going to be specific about what kind of things our game spawns. It spawns shapes, not generic persistable objects. So create a new `Shape` class, which represents geometric 3D shapes. It just extends `PersistableObject`, without adding anything new, at least for now.

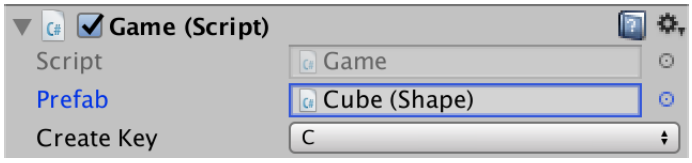
```
using UnityEngine;

public class Shape : PersistableObject {}
```

Remove the `PersistableObject` component from the *Cube* prefab and give it a `Shape` component instead. It cannot have both, because we gave `PersistableObject` the `DisallowMultipleComponent` attribute, which also applies to `Shape`.



Cube with a Shape component.

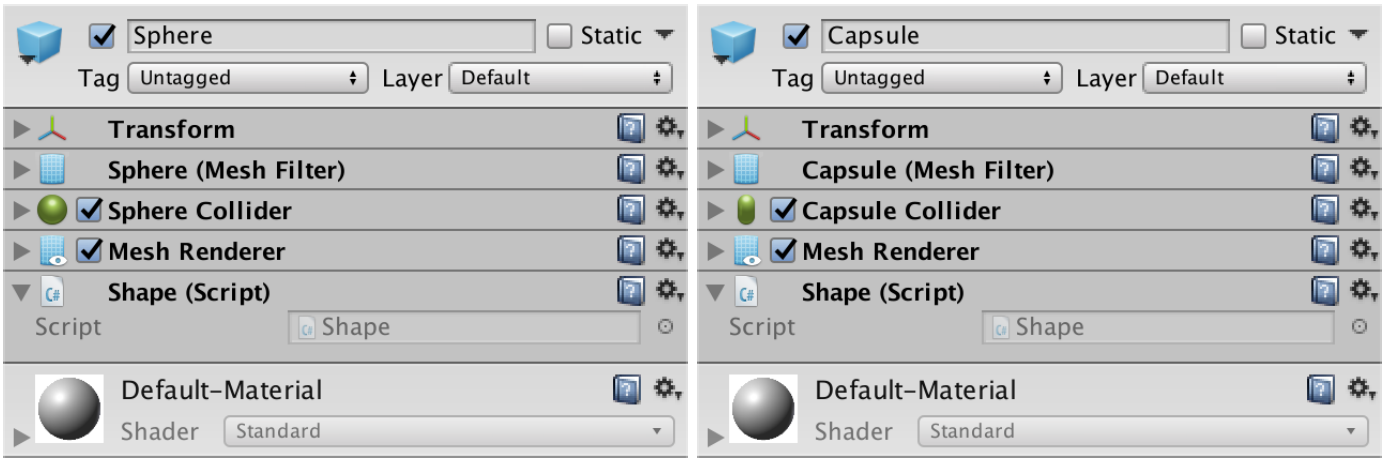
This breaks the reference that the *Game* object has to the prefab. But because `Shape` is also a `PersistableObject` we can assign it again.



Game with prefab assigned again.

1.2 Multiple Different Shapes

Create a default sphere and capsule object, give each a `Shape` component, and turn them into prefabs too. These are the other shapes that our game will support.



Sphere and capsule shape prefabs.

> **What about cylinders?**
>
> You could also add a cylinder object, but I omitted it because cylinders don't have their own collider type. Instead, they use a capsule collider, which doesn't really fit. That isn't an issue right now, but might be later.

1.3 Factory Asset

Currently, `Game` can only spawn a single thing, because it only has one reference to a prefab. To support all three shapes, it would need three prefab references. This would require three fields, but that wouldn't be flexible. A better approach would be to use an array. But maybe we'll come up with a different way to create shapes later. That could make `Game` rather complex, as it is also responsible for user input, keeping track of objects, and triggering the saving and loading.

To keep `Game` simple, we're going to put the responsibility for what shapes are supported in its own class. This class will act like a factory, creating shapes on demand, without its client having to know how those shapes are made or even how many different options there are. We will name this class `ShapeFactory`.

```
using UnityEngine;

public class ShapeFactory {}
```

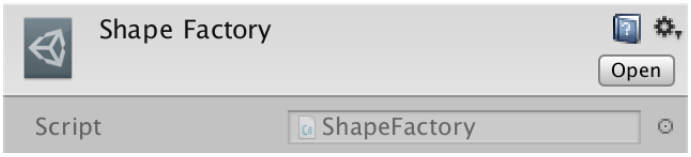
The factory's only responsibility is to deliver shape instances. It doesn't need a position, rotation, or scale, and neither does it need an `Update` method to change it state. So it doesn't need to be a component, which would have to be attached to a game object. Instead, it can exist on it own, not part of a specific scene, but part of the project. In other words, it is an asset. This is possible, by having it extend `ScriptableObject` instead of `MonoBehaviour`.

```
public class ShapeFactory : ScriptableObject {}
```

We now have a custom asset type. To add such an asset to our project, we'll have to add an entry for it to Unity's menu. The simplest way to do this is by adding the `CreateAssetMenu` attribute to our class.

```
[CreateAssetMenu]
public class ShapeFactory : ScriptableObject {}
```

You can now create our factory via *Assets › Create › Shape Factory*. We only need one.

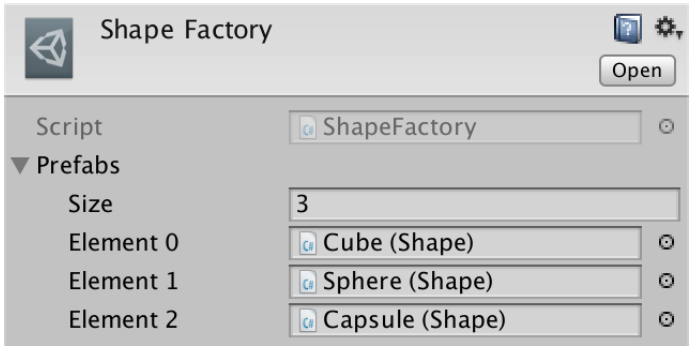


Shape factory asset.

To let our factory know about the shape prefabs, give it a `Shape[] prefabs` array field. We don't want this field to be public, as its inner workings should not be exposed to other classes. So keep it private. To have the array show up in the inspector and be saved by Unity, add the `SerializeField` attribute to it.

```
public class ShapeFactory : ScriptableObject {

	[SerializeField]
	Shape[] prefabs;
}
```

After the field appears in the inspector, drag all three shape prefabs onto it, so references to them get added to the array. Make sure that the cube is the first element. Use the sphere for the second element and the capsule for the third.



Factory with references to the prefabs.

1.4 Getting Shapes

For the factory to be of any use, there must be a way to get shape instances out of it. So give it a public `Get` method. The client can indicate what kind of shape it wants via a shape identifier parameter. We'll simply use an integer for this purpose.

```
	public Shape Get (int shapeId) {}
```

> **Why not use an enumeration?**
>
> That is certainly possible, so you could do that instead. But we don't really care about identifying exact shape types in code, so an integer works fine. That makes it possible to control what shapes are supported purely by changing the factory's array contents, without having to change any code.

We can directly use the identifier as the index to find the appropriate shape prefab, instantiate it, and return it. This means that 0 identifies a cube, 1 a sphere, and 2 a capsule. Even if we change how the factory works later, we have to make sure that this identification remains the same, to remain backwards compatible.

```
	public Shape Get (int shapeId) {
		return Instantiate(prefabs[shapeId]);
	}
```

Besides requesting a specific shape, let's also make it possible to get a random shape instance out of the factory, via a `GetRandom` method. We can use the `Random.Range` method to to select an index at random.

```
	public Shape GetRandom () {
		return Get(Random.Range(0, prefabs.Length));
	}
```

> **Shouldn't it be `Random.Range(0, prefab.Length - 1)` instead?**
>
> Unity's `Random.Range` method with integer parameters uses an exclusive maximum. The output range is from the minimum to the maximum minus 1. This was done because the typical use case was expected to be getting a random array index, which is exactly what we're doing here.
>
> Note that `Random.Range` with float parameters uses an inclusive maximum instead.

1.5 Getting Shapes

Because we're now creating shapes in `Game`, let's be explicit and rename its list to `shapes`. So everywhere that `objects` is written, replace it with `shapes`. It is easiest to use your code editor's refactor functionality to change the field's name, and it will take care of renaming it everywhere that it is used.

```
	List<PersistableObject> shapes;
```

Also change the list's item type to `Shape`, which is more specific.

```
	List<Shape> shapes;

	void Awake () {
		shapes = new List<Shape>();
	}
```

Next, remove the prefab field and add a `shapeFactory` field to hold a reference to the shape factory instead.

```
//	public PersistableObject prefab;
	public ShapeFactory shapeFactory;
```

In `CreateObject`, we'll now create an arbitrary shape by invoking `shapeFactory.GetRandom` instead of instantiating an explicit prefab.

```
	void CreateObject () {
//		PersistableObject o = Instantiate(prefab);
		Shape o = shapeFactory.GetRandom();
		…
	}
```

Let's also rename the instance's variable so it's very explicit that we're dealing with a shape instance, not a prefab reference that we still need to instantiate. Once again, you can use refactoring to quickly and consistently rename the variable.

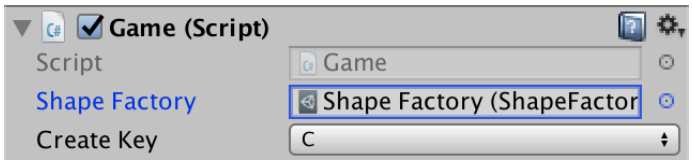
```
	void CreateShape () {
		Shape instance = shapeFactory.GetRandom();
		Transform t = instance.transform;
		t.localPosition = Random.insideUnitSphere * 5f;
		t.localRotation = Random.rotation;
		t.localScale = Vector3.one * Random.Range(0.1f, 1f);
		shapes.Add(instance);
	}
```

When loading, we now also have to use the shape factory. In this case, we do not want random shapes. We have only ever worked with cubes before, so we should get cubes, which is done by invoking `shapeFactory.Get(0)`.

```
	public override void Load (GameDataReader reader) {
		int count = reader.ReadInt();
		for (int i = 0; i < count; i++) {
//			PersistableObject o = Instantiate(prefab);
			Shape o = shapeFactory.Get(0);
			o.Load(reader);
			shapes.Add(o);
		}
	}
```

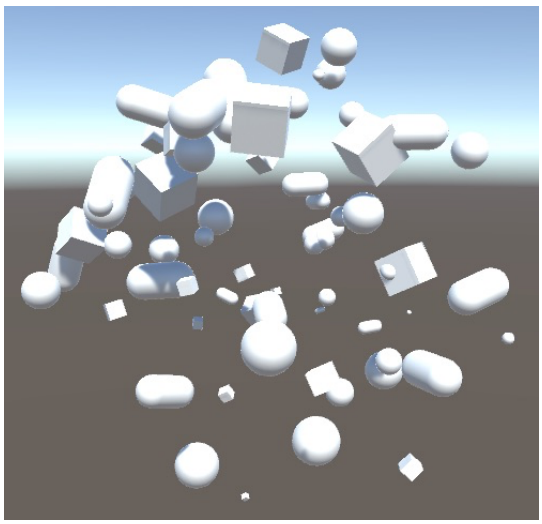
Let's also make it explicit here that we're dealing with an instance.

```
			Shape instance = shapeFactory.Get(0);
			instance.Load(reader);
			shapes.Add(instance);
```



Game with factory instead of prefab.

After giving *Game* a reference to our factory, it will now create random shapes each time the player spawns a new one, instead of always getting cubes.



Creating random shapes.

2 Remembering the Shapes

While it is now possible to create three different shapes, this information is not yet saved. So each time we load a saved game, we end up with nothing but cubes. This is correct for games that were saved earlier, but not for games that were saved after we added support for multiple shapes. We also have to add support for saving the different shapes, ideally while still being able to load the old save files as well.

2.1 Shape Identifier Property

To be able to save which shape an object has, the object has to remember this information. The most straightforward way to do this is by adding a shape identifier field to `Shape`.

```
public class Shape : PersistableObject {

	int shapeId;
}
```

Ideally, this field is read-only, because a shape instance is always of one type and doesn't change. But it has to be assigned a value somehow. We could mark the private field as serializable and assign it a value via the inspector of each prefab. However, this doesn't guarantee that the identifier matches the array index used by the factory. It might also be possible that we use a shape prefab somewhere else, which has nothing to do with the factory, or maybe even add it to another factory at some point. So the shape identifier depends on the factory, not the prefab. Thus, it's something to be tracked per instance, not per prefab.

Private fields do not get serialized by default, so the prefab has nothing to do with it. A new instance will simple get the field's default value, which is 0 in this case, because we didn't gave it another default. To make the identifier publicly accessible, we'll add a `ShapeId` property to `Shape`. We use the same name, except the first letter is a capital. Properties are methods that pretend to be fields, so they need a code block.

```
	public int ShapeId {}

	int shapeId;
```

Properties actually need two separate code blocks. One to get the value it represents, and one to set it. These are identified via the `get` and `set` keywords. It is possible to only use one of them, but we need both in this this case.

```
public int ShapeId {
    get {}
    set {}
}
```

The getter part simply returns the private field. The setter simply assigns to the private field. The setter has an implicit parameter of the appropriate type named `value` for this purpose.

```
public int ShapeId {
    get {
        return shapeId;
    }
    set {
        shapeId = value;
    }
}

int shapeId;
```

By using a property it becomes possible to add additional logic to what appears to be a simple retrieval or assignment. In our case, the shape identifier has to be set exactly once per instance, when it is instantiated by the factory. Setting it again after that would be a mistake.

We can check whether the assignment is correct by verifying that the identifier still has its default value at the time of assignment. If so, the assignment is valid. If not, we log an error instead.

```
public int ShapeId {
    get {
        return shapeId;
    }
    set {
        if (shapeId == 0) {
            shapeId = value;
        }
        else {
            Debug.LogError("Not allowed to change shapeId.");
        }
    }
}
```

However, 0 is a valid identifier. So we have to use something else as the default value. Let's use the minimum possible integer instead, `int.MinValue`, which is −2147483648. Also, we should ensure that the identifier cannot be reset to its default value. This approach will work as long as the minimum integer isn't a valid identifier.

```
	public int ShapeId {
		…
		set {
			if (shapeId == int.MinValue && value != int.MinValue) {
				shapeId = value;
			}
			…
		}
	}

	int shapeId = int.MinValue;
```

Why not just use a `readonly` property?

A `readonly` field or property can only be assigned a default value, or be assigned to in a constructor method. Unfortunately, we cannot use constructor methods when instantiating Unity objects. So we have to use an approach like this.

Adjust `ShapeFactory.Get` so it sets the identifier of the instance before returning it.

```
	public Shape Get (int shapeId) {
//		return Instantiate(prefabs[shapeId]);
		Shape instance = Instantiate(prefabs[shapeId]);
		instance.ShapeId = shapeId;
		return instance;
	}
```

2.2 Identifying the File Version

We didn't have shape identifiers before, so we didn't save them. If we save them from now on, we're using a different save file format. It's fine if the old version of our game—from the previous tutorial—cannot read this format, but we should ensure that the new game can still work with the old format.

We'll use a save version number to identify the format used by a save file. As we introduce this concept now, we start with version 1. Add this as a constant integer to `Game`.

```
	const int saveVersion = 1;
```

What does `const` mean?

It declares a simple value to be a constant, not a field. It cannot be changed and doesn't exist in memory. Instead, it's only part of the code and its explicit value gets used wherever it is referenced, substituted during compilation.

When saving the game, start with writing the save version number. When loading, begin by reading the stored version. This tells us what version we're dealing with.

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

	public override void Load (GameDataReader reader) {
		int version = reader.ReadInt();
		int count = reader.ReadInt();
		…
	}
```

However, this only works for files that contain the save version. The old save files from the previous tutorial don't have this information. Instead, the first thing written to those files is the object count. So we'd end up interpreting the count as the version.

The object count stored in old save files could be anything, but it will always be at least zero. We can use this to distinguish between the save version and the object count. This is done by not writing the save version verbatim. Instead, flip the sign of the version when writing it. As we start with 1, this means that the stored save version is always less than zero.

```
		writer.Write(-saveVersion);
```

When reading the version, flip its sign again to retrieve the original number. If we're reading and old save file, this ends up flipping the sign of the count, so it becomes either zero or negative. Thus, when we end up with a version less than or equal to zero, we know that we're dealing with an old file. In that case, we already have the count, just with a flipped sign. Otherwise, we still have to read the count.

```
		int version = -reader.ReadInt();
		int count = version <= 0 ? -version : reader.ReadInt();
```

What does the question mark mean?

It is the ternary operator, `condition ? trueResult : falseResult`, which is a shorthand alternative for an if-else expression. In this case, the code is equivalent to the following:

```
int version = -reader.ReadInt();
int count;
if (version <= 0) {
	count = -version;
}
else {
	count = reader.ReadInt();
}
```

This makes it possible for the new code to deal with the old save file format. But the old code cannot deal with the new format. We cannot do anything about that, because the old code has already been written. What we can do is make sure that from now on the game will refuse to load future save file formats that it doesn't know how to deal with. If the loaded version is higher than our current save version, log an error and return immediately.

```
int version = -reader.ReadInt();
if (version > saveVersion) {
	Debug.LogError("Unsupported future save version " + version);
	return;
}
```

2.3 Saving the Shape Identifier

A shape should not write its own identifier, because it has to be read to determine which shape to instantiate, and only after that the shape can load itself. So it's the responsibility of `Game` to write the identifiers. Because we're storing all shapes in a single list, we have to write each shape's identifier before the shape saves itself.

```
public override void Save (GameDataWriter writer) {
	writer.Write(-saveVersion);
	writer.Write(shapes.Count);
	for (int i = 0; i < shapes.Count; i++) {
		writer.Write(shapes[i].ShapeId);
		shapes[i].Save(writer);
	}
}
```

Note that this is not the only way to save the shape identifiers. For example, it is also possible to use a separate list for each shape type. In that case, it would only be necessary to write each shape identifier once per list.

2.4 Loading the Shape Identifier

For each shape in the list, begin by loading its shape identifier, then use that to get the correct shape from the factory.

```
	public override void Load (GameDataReader reader) {
		…
		for (int i = 0; i < count; i++) {
			int shapeId = reader.ReadInt();
			Shape instance = shapeFactory.Get(shapeId);
			instance.Load(reader);
			shapes.Add(instance);
		}
	}
```

But this is only valid for the new save version 1. If we're reading from an older save file, just get cubes instead.

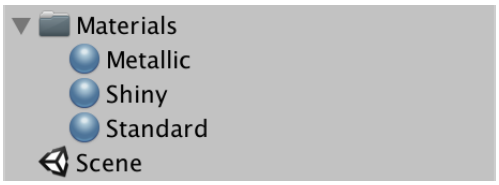
```
			int shapeId = version > 0 ? reader.ReadInt() : 0;
```

3 Material Variants

Besides varying the shape of the objects that are spawned, we could also vary what they're made of. At the moment, all shapes use the same material, which is Unity's default material. Let's change that into a random selection of materials.

3.1 Three Materials

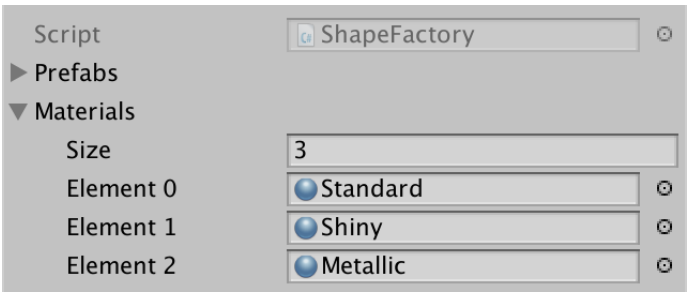
Create three new materials. Name the first *Standard*, leaving it unchanged so it matches Unity's default material. Name the second one *Shiny* and increase its *Smoothness* to 0.9. Name the third one *Metallic* and set both its *Metallic* and *Smoothness* to 0.9.



Standard, shiny, and metallic.

When getting a shape from the factory, it should now also be possible to specify what kind of material it has to be made of. This requires `ShapeFactory` to be aware of the allowed materials. So give it a material array—just like its prefab array—and assign the three materials to it. Make sure that the standard material is the first element. The second is the shiny material, and the third one is metallic.

```
	[SerializeField]
	Material[] materials;
```



Factory with materials.

3.2 Setting a Shape's Material

In order to save which material a shape has, we now also have to keep track of a material identifier. Add a property for this to `Shape`. However, instead of explicitly coding how the property works, omit the code blocks for the getter and setter. End each with a semicolon instead. This generates a default property, with an implicit hidden private field.

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

When setting a shape's material, we have to both give it the actual material as well as its identifier. This suggests that we have to use two parameters at once, but this is impossible for properties. So we're not going to rely on the property's setter. To disallow its use outside the `Shape` class itself, mark the setter as private.

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

Instead, we add a public `SetMaterial` method with the required parameters.

```
	public void SetMaterial (Material material, int materialId) {}
```

This method can get the shape's `MeshRenderer` component by invoking the `GetComponent<MeshRenderer>` method. Note that this is a generic method, like `List` is a generic class. Set the renderer's material and also the material identifier property. Make sure that you assign the parameter to the property, the difference being whether M is a capital letter.

```
	public void SetMaterial (Material material, int materialId) {
		GetComponent<MeshRenderer>().material = material;
		MaterialId = materialId;
	}
```

3.3 Getting Shapes with a Material

Now we can adjust `ShapeFactory.Get` to work with materials. Give it a second parameter to indicate which material should be used. Then use that to set the shape's material and its material identifier.

```
	public Shape Get (int shapeId, int materialId) {
		Shape instance = Instantiate(prefabs[shapeId]);
		instance.ShapeId = shapeId;
		instance.SetMaterial(materials[materialId], materialId);
		return instance;
	}
```

It might be possible that whoever invokes `Get` doesn't care about materials and is satisfied with the standard material. We can support a variant of `Get` with a single shape identifier parameter. We can do this by assigning a default value to its `materialId` parameter, using 0. This makes it possible to omit the `materialId` parameter when invoking `Get`. As a result, the existing code compiles at this point without errors.

```
	public Shape Get (int shapeId, int materialId = 0) {
		…
	}
```

We could do the same for the `shapeId` parameter, giving it a default of 0 too.

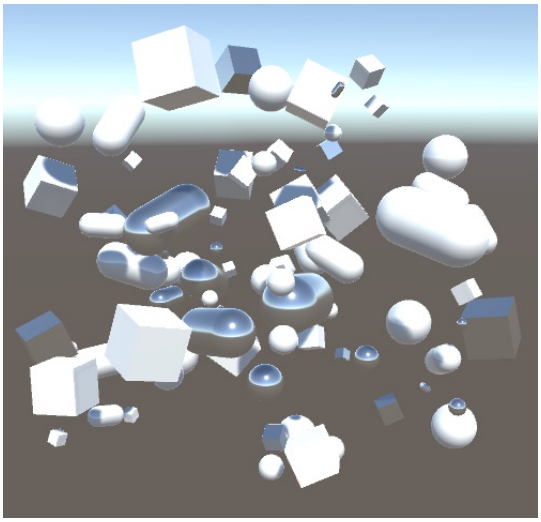
```
	public Shape Get (int shapeId = 0, int materialId = 0) {
		…
	}
```

How do you indicate which default values you want?

To omit `materialId`, simply omit it, so you invoke the method like `Get(0)`. You can also omit both arguments, by invoking `Get()`. However, if you want to omit `shapeId` but not `materialId`, then you have to be explicit about which arguments you are providing. You can do this by labeling your argument, by writing the parameter name in front of the argument value, followed by a colon. For example, `Get(materialId: 0)`.

The `GetRandom` method should now both select a random shape and a random material. So have it use `Random.Range` to pick a random material identifier as well.

```
	public Shape GetRandom () {
		return Get(
			Random.Range(0, prefabs.Length),
			Random.Range(0, materials.Length)
		);
	}
```



Shapes with random materials.

3.4 Saving and Loading the Material Identifier

Saving the material identifier works the same as saving the shape identifier. Write it after the shape identifier of each shape.

```
	public override void Save (GameDataWriter writer) {
		…
		for (int i = 0; i < shapes.Count; i++) {
			writer.Write(shapes[i].ShapeId);
			writer.Write(shapes[i].MaterialId);
			shapes[i].Save(writer);
		}
	}
```

Loading works the same too. We won't bother incrementing the save version for this change, because we're still in the same tutorial, which symbolizes a single public release. So loading will fail for a save file that stores shape identifiers but not material identifiers.

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

4 Randomized Colors

Besides whole materials, we can also vary the color of our shapes. We do this by adjusting the color property of each shape instance's material.

We could define a selection of valid colors and add them to the shape factory, but we'll use unrestricted colors in this case. That means that the factory doesn't have to be aware of shape colors. Instead, the color of a shape is set just like its position, rotation, and scale.

4.1 Shape Color

Add a `SetColor` method to `Shape` that makes it possible to adjust its color. It has to adjust the color property of whatever material it's using.

```
	public void SetColor (Color color) {
		GetComponent<MeshRenderer>().material.color = color;
	}
```

In order to save and load the shape's color, it has to keep track of it. We don't need to provide public access to the color, so a private field suffices, set via `SetColor`.

```
	Color color;

	public void SetColor (Color color) {
		this.color = color;
		GetComponent<MeshRenderer>().material.color = color;
	}
```

Saving and loading the color is done by overriding the `Save` and `Load` methods of `PersistableObject`. First take care of the base, then the color data after that.

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

	public override void Load (GameDataReader reader) {
		base.Load(reader);
		SetColor(reader.ReadColor());
	}
```

But this assumes that there are methods to write and read a color, which is currently not the case. So let's add them. First a new `Write` method for `GameDataWriter`.

```
	public void Write (Color value) {
		writer.Write(value.r);
		writer.Write(value.g);
		writer.Write(value.b);
		writer.Write(value.a);
	}
```

Then also a `ReadColor` method for `GameDataReader`.

```
	public Color ReadColor () {
		Color value;
		value.r = reader.ReadSingle();
		value.g = reader.ReadSingle();
		value.b = reader.ReadSingle();
		value.a = reader.ReadSingle();
		return value;
	}
```

Do we need to store the color channels as floats?

You could also decide to store them as bytes, but if you do so it's better to consistently use `Color32` everywhere. That ensures that the saved and loaded data is always the same. You don't need to bother with this just to save twelve less bytes per shape, unless you really need to minimize your save file size. Likewise, you could decide to skip the alpha channel as it's not needed for opaque materials, but it's also not something worth worrying about in general.

4.2 Remaining Backwards Compatible

While this approach makes it possible to store the shape color, it now assumes that the color is stored in the save file. This isn't the case for the old save format. To still support the old format, we have to skip loading the color. In `Game`, we use the read version to decide what to do. However, `Shape` doesn't know about the version. So we somehow have to communicate the version of the data we're reading to `Shape` when it is loading. It makes sense to define the version as a property of `GameDataReader`.

Because the version of a read file doesn't change while reading it, the property should be set only once. As `GameDataReader` isn't a Unity object class, we can use a read-only property, by only giving it a `get` part. Such properties can be initialized via a constructor method. To do this we have to add the version as a constructor parameter.

```
	public int Version { get; }

	BinaryReader reader;

	public GameDataReader (BinaryReader reader, int version) {
		this.reader = reader;
		this.Version = version;
	}
```

Now writing and reading the version number has become the responsibility of `PersistentStorage`. The version has to be added as a parameter to its `Save` method, which must write it before anything else. And the `Load` method reads it while constructing the `GameDataReader`. It is also here that we will perform the sign-change trick to support reading version zero files.

```
	public void Save (PersistableObject o, int version) {
		using (
			var writer = new BinaryWriter(File.Open(savePath, FileMode.Create))
		) {
			writer.Write(-version);
			o.Save(new GameDataWriter(writer));
		}
	}

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

This means that `Game` no longer needs to write the save version.

```
	public override void Save (GameDataWriter writer) {
//		writer.Write(-saveVersion);
		writer.Write(shapes.Count);
		…
	}
```

Instead, it must provide it as an argument when invoking `PersistentStorage.Save`.

```
	void Update () {
		…
		else if (Input.GetKeyDown(saveKey)) {
			storage.Save(this, saveVersion);
		}
		…
	}
```

In its `Load` method, it can now retrieve the version via `reader.Version`.

```
	public override void Load (GameDataReader reader) {
		int version = reader.Version;
		…
	}
```

And we can now also check the version in `Shape.Load`. If we have at least version 1, then read the color. Otherwise, use white.

```
	public override void Load (GameDataReader reader) {
		base.Load(reader);
		SetColor(reader.Version > 0 ? reader.ReadColor() : Color.white);
	}
```

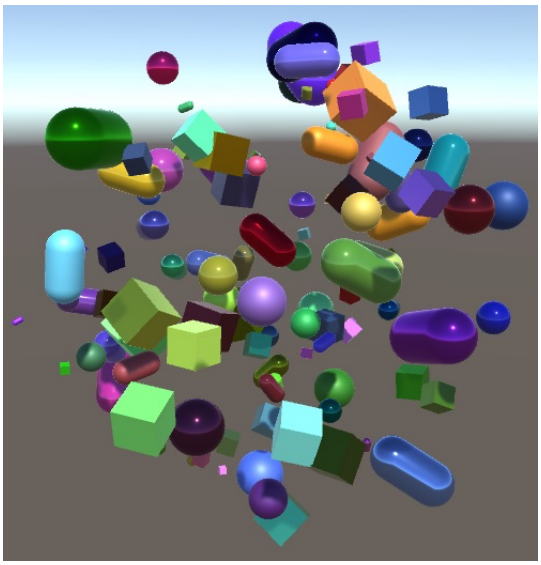
4.3 Choosing a Shape Color

To create shapes with arbitrary colors, simply invoke `SetColor` on the new instance in `Game.CreateShape`. We can use the `Random.ColorHVS` method to generate random colors. Without arguments, that method can create any valid color, which can get a bit messy. Let's limit ourselves to a colorful palette, by restricting the saturation range to 0.5–1 and the value range of 0.25–1. As we're not using alpha at this point, we'll always set it to 1.

```
	void CreateShape () {
		Shape instance = shapeFactory.GetRandom();
		Transform t = instance.transform;
		t.localPosition = Random.insideUnitSphere * 5f;
		t.localRotation = Random.rotation;
		t.localScale = Vector3.one * Random.Range(0.1f, 1f);
		instance.SetColor(Random.ColorHSV(0f, 1f, 0.5f, 1f, 0.25f, 1f, 1f, 1f));
		shapes.Add(instance);
	}
```

Using all eight parameters of `ColorHVS` makes it hard to understand, as it's not immediately clear which value controls what. You can make the code easier to read by explicitly naming the arguments.

```
		instance.SetColor(Random.ColorHSV(
			hueMin: 0f, hueMax: 1f,
			saturationMin: 0.5f, saturationMax: 1f,
			valueMin: 0.25f, valueMax: 1f,
			alphaMin: 1f, alphaMax: 1f
		));
```



Shapes with random colors.

4.4 Remembering the Renderer

We now need to access the `MeshRenderer` component of `Shape` both when settings its material and when setting its color. Using `GetComponent<MeshRenderer>` twice is not ideal, especially if we decide to change a shape's color multiple times in the future. So let's store the reference in a private field and initialize it in a new `Awake` method of `Shape`.

```
	MeshRenderer meshRenderer;

	void Awake () {
		meshRenderer = GetComponent<MeshRenderer>();
	}
```

Now we can use that field in `SetColor` and `SetMaterial`.

```
	public void SetColor (Color color) {
		this.color = color;
//		GetComponent<MeshRenderer>().material.color = color;
		meshRenderer.material.color = color;
	}

	public void SetMaterial (Material material, int materialId) {
//		GetComponent<MeshRenderer>().material = material;
		meshRenderer.material = material;
		MaterialId = materialId;
	}
```

4.5 Using a Property Block

A downside of setting a material's color is that this results in the creation of a new material, unique to the shape. This happens each time its color is set. We can avoid this by using a `MaterialPropertyBlock` instead. Create a new property block, set a color property named *_Color*, then use it as the renderer's property block, by invoking `MeshRenderer.SetPropertyBlock`.

```
	public void SetColor (Color color) {
		this.color = color;
//		meshRenderer.material.color = color;
		var propertyBlock = new MaterialPropertyBlock();
		propertyBlock.SetColor("_Color", color);
		meshRenderer.SetPropertyBlock(propertyBlock);
	}
```

Instead of using a string to name the color property, it is also possible to use an identifier. These identifiers are setup by Unity. They can change, but remain constant per session. So we can suffice with getting the identifier of the color property once, storing it in a static field. The identifier is found by invoking the `Shader.PropertyToID` method with a name.

```
	static int colorPropertyId = Shader.PropertyToID("_Color");

	…

	public void SetColor (Color color) {
		this.color = color;
		var propertyBlock = new MaterialPropertyBlock();
		propertyBlock.SetColor(colorPropertyId, color);
		meshRenderer.SetPropertyBlock(propertyBlock);
	}
```

it is also possible to reuse the whole property block. When setting a renderer's properties, the contents of the block are copied. So we do not have to create a new block per shape, we can keep changing the color of the same block for all shapes.

We can again use a static field to keep track of the block, but it is not possible to create a block instance via static initialization. Unity doesn't allow it. Instead, we can check whether the block exists before we use it. If not, we create it at that point.

```
	static MaterialPropertyBlock sharedPropertyBlock;

	…

	public void SetColor (Color color) {
		this.color = color;
//		var propertyBlock = new MaterialPropertyBlock();
		if (sharedPropertyBlock == null) {
			sharedPropertyBlock = new MaterialPropertyBlock();
		}
		sharedPropertyBlock.SetColor(colorPropertyId, color);
		meshRenderer.SetPropertyBlock(sharedPropertyBlock);
	}
```

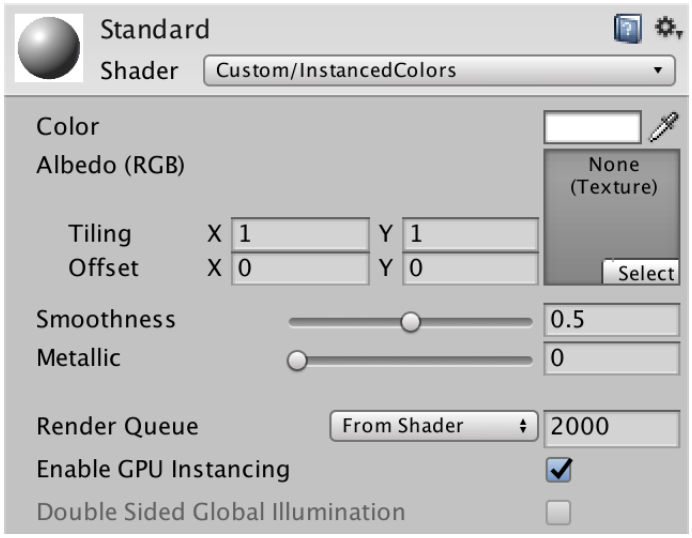
Now we no longer get duplicate materials, which you can verify by adjusting one of the materials while shapes are using it in play mode. The shapes will adjust their appearance based on the changes, which wouldn't happen if they used duplicate materials. Of course this doesn't work when you adjust the material's color, because each shape uses its own color property, which overrides the material's color.

4.6 GPU Instancing

As we're using property blocks, it is possible to use GPU instancing to combine shapes that use the same material in a single draw call, even though they have different colors. However, this requires a shader that supports instanced colors. Here is such a shader, which you can find on the Unity GPU Instancing manual page. The only differences are that I removed the comments and added the `#pragma instancing_options assumeuniformscaling` directive. Assuming uniform scaling makes instancing more efficient as it requires less data, and works because all our shapes use a uniform scale.

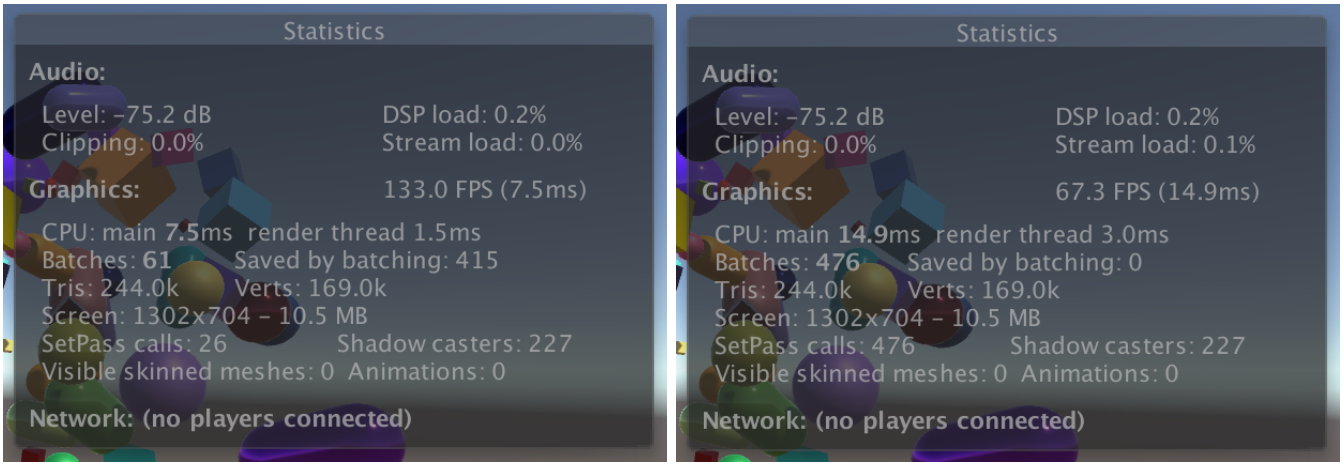
```
Shader "Custom/InstancedColors" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
		_Metallic ("Metallic", Range(0,1)) = 0.0
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		#pragma surface surf Standard fullforwardshadows
		#pragma instancing_options assumeuniformscaling

		#pragma target 3.0

		sampler2D _MainTex;

		struct Input {
			float2 uv_MainTex;
		};

		half _Glossiness;
		half _Metallic;

		UNITY_INSTANCING_BUFFER_START(Props)
			UNITY_DEFINE_INSTANCED_PROP(fixed4, _Color)
		UNITY_INSTANCING_BUFFER_END(Props)

		void surf (Input IN, inout SurfaceOutputStandard o) {
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex) *
				UNITY_ACCESS_INSTANCED_PROP(Props, _Color);
			o.Albedo = c.rgb;
			o.Metallic = _Metallic;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

Change our three materials so they use this new shader instead of the standard one. It supports less features and has a different inspector interface, but it is sufficient for our needs. Then make sure that *Enable GPU Instancing* is checked for all materials.



Standard material with instanced colors.

You can verify the difference via the *Stats* overlay of the *Game* window.



With vs. without GPU instancing.

The next tutorial is Reusing Objects.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick