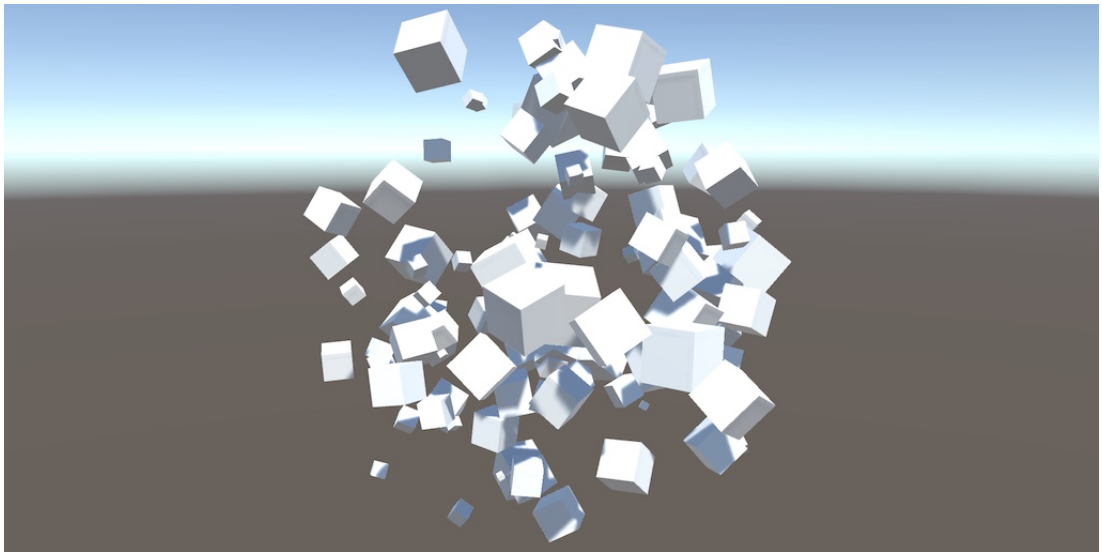
Persisting Objects Creating, Saving, and Loading

Spawn random cubes in response to a key press.
Use a generic type and virtual methods.
Write data to a file and read it back.
Save the game state so it can be loaded later.
Encapsulate the details of persisting data.

This is the first tutorial in a series about managing objects. It covers creating, tracking, saving, and loading simple prefab instances. It builds on the foundation laid by the tutorials in the Basics section.

This tutorial is made with Unity 2017.3.1p4.



These cubes survived the termination of their game.

1 Creating Objects On Demand

You can create scenes in the Unity editor and populate them with object instances. This allows you to design fixed levels for your game. The objects can have behavior attached to them, which can alter the state of the scene while in play mode. Often, new object instances are created during play. Bullets are fired, enemies spawn, random loot appears, and so on. It might even be possible for players to create custom levels inside the game.

Creating new stuff during play is one thing. Remembering it all, so the player can quit and later return to the game is something else. Unity doesn't automatically keep track of the potential changes for us. We have to do that ourselves.

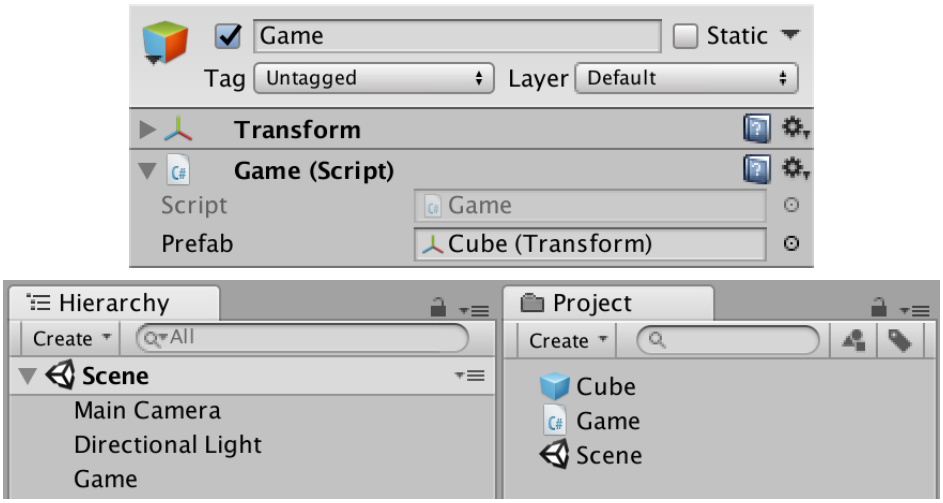
In this tutorial we'll create a very simple game. All it does is spawn random cubes in response to pressing a key. Once we're able to keep track of the cubes between play sessions, we can increase the game's complexity in a later tutorial.

1.1 Game Logic

Because our game is so simple, we'll control it with a single `Game` component script. It will spawn cubes, for which we'll use a prefab. So it should contain a public field to hook up a prefab instance.

```
using UnityEngine;

public class Game : MonoBehaviour {

	public Transform prefab;
}
```

Add a game object to the scene and attach this component to it. Then also create a default cube, turn it into a prefab, and give the game object a reference to it.

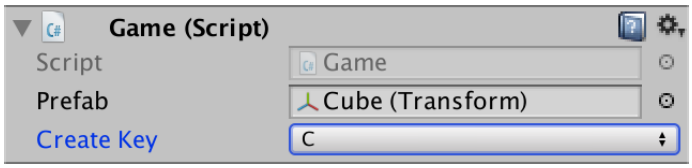


Game setup.

1.2 Player Input

We're going to spawn cubes in response to player input, so our game must be able to detect this. We'll use Unity's input system to detect key presses. Which key should be used to spawn a cube? The C key seems appropriate, but we can make this configurable via the inspector, by adding a public `KeyCode` enumeration field to `Game`. Use C as the default option when defining the field, via an assignment.

```
	public KeyCode createKey = KeyCode.C;
```



Create key set to C.

We can detect whether the key is pressed by querying the static `Input` class in an `Update` method. The `Input.GetKeyDown` method returns a boolean that tells us whether a specific key was pressed in the current frame. If so, we have to instantiate our prefab.

```
	void Update () {
		if (Input.GetKeyDown(createKey)) {
			Instantiate(prefab);
		}
	}
```

When exactly does `Input.GetKeyDown` return `true`?

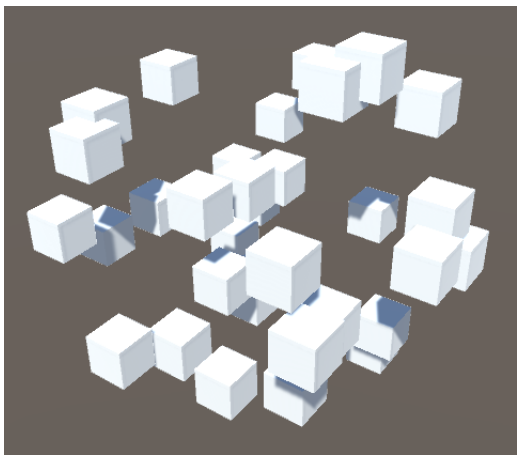
It does so only during the frame that the key's state has changed from not-pressed to pressed, because the player pressed on it. Typically, the key remains in the pressed state for a few frames until the player lets go of the button, but `Input.GetKeyDown` returns `true` only during the first frame. In contrast, `Input.GetKey` keeps returning `true` each frame that the key is held down. There is also `Input.GetKeyUp`, which returns `true` during the frame that the player let go of the key.

1.3 Randomized Cubes

While in play mode, our game now spawns a cube each time we press the C key, or whichever key you configured it to respond to. But it looks like we only get a single cube, because they all end up at the same position. So let's randomize the position of each cube that we create.

Keep track of the instantiated `Transform` component, so we can change its local position. Use the static `Random.insideUnitSphere` property to get a random point, scale it up to a radius of five units, and use that as the final position. Because that's more work than just a trivial instantiation, put the code for that in a separate `CreateObject` method and invoke it when the key is pressed.

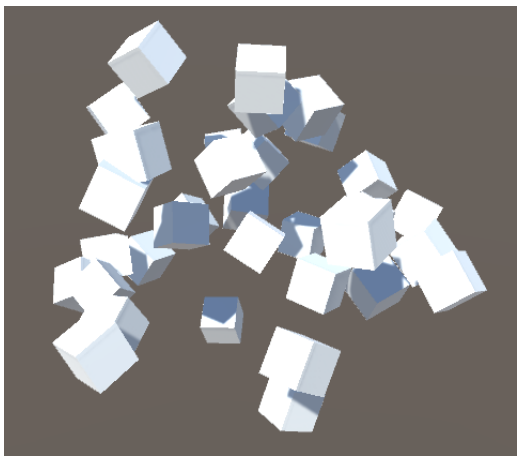
```
	void Update () {
		if (Input.GetKeyDown(createKey)) {
//			Instantiate(prefab);
			CreateObject();
		}
	}

	void CreateObject () {
		Transform t = Instantiate(prefab);
		t.localPosition = Random.insideUnitSphere * 5f;
	}
```



Randomly placed cubes.

The cubes now spawn inside a sphere instead of all at the exact same position. They can still overlap, but that's fine. However, they're all aligned and that doesn't look interesting. So let's give each cube a random rotation, for which we can use the static `Random.rotation` property.

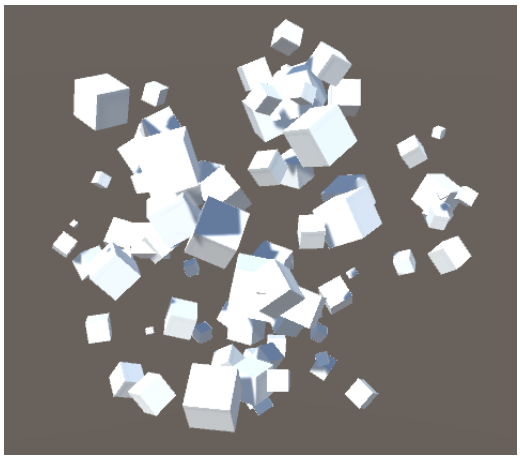
```
	void CreateObject () {
		Transform t = Instantiate(prefab);
		t.localPosition = Random.insideUnitSphere * 5f;
		t.localRotation = Random.rotation;
	}
```



Randomized rotations.

Finally, we can also vary the size of the cubes. We'll use uniformly-scaled cubes, so they're always perfect cubes, just with different sizes. The static `Random.Range` method can be used to get a random `float` inside a certain range. Let's go from small size 0.1 cubes up to regular size 1 cubes. To use this value for all three dimensions of the scale, simple multiply `Vector3.one` with it, then assign the result to the local scale.

```
	void CreateObject () {
		Transform t = Instantiate(prefab);
		t.localPosition = Random.insideUnitSphere * 5f;
		t.localRotation = Random.rotation;
		t.localScale = Vector3.one * Random.Range(0.1f, 1f);
	}
```



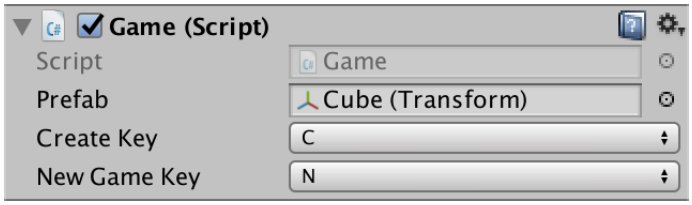
Randomized uniform scale.

1.4 Starting a New Game

If we want to begin a new game, we have to exit play mode and then enter it again. But that's only possible in the Unity Editor. The player would need to quit our app and start it again to be able to play a new game. It's much better if we could begin a new game while remaining in play mode.

We could start a new game by reloading the scene, but this isn't necessary. We can suffice with destroying all the cubes that were spawned. Let's use another configurable key for that, using N as the default.

```
	public KeyCode createKey = KeyCode.C;
	public KeyCode newGameKey = KeyCode.N;
```



New-game key set to N.

Check whether this key is pressed in `Update`, and if so invoke a new `BeginNewGame` method. We should only handle one key at a time, so only check for the N key if the C key isn't pressed.

```
	void Update () {
		if (Input.GetKeyDown(createKey)) {
			CreateObject();
		}
		else if (Input.GetKey(newGameKey)) {
			BeginNewGame();
		}
	}

	void BeginNewGame () {}
```

1.5 Keeping Track of Objects

Our game can spawn an arbitrary number of randomized cubes, which all get added to the scene. But `Game` has no memory of what it spawned. In order to destroy the cubes, we first need to find them. To make this possible, we'll have `Game` keep track of a list of references to the objects it instantiated.

Why not just use `GameObject.Find`?

This is possible for simple cases, where it's easy to distinguish between objects and there aren't many in the scene. For larger scenes, relying on `GameObject.Find` is a bad idea. `GameObject.FindWithTag` is better, but it's best to keep track of things yourself if you know you'll need them later.

We could add an array field to `Game` and fill it with references, but we don't know ahead of time how many cubes will be created. Fortunately, the `System.Collections.Generic` namespace contains a `List` class that we can use. It works like an array, except that its size isn't fixed.

How can the list's size be dynamic?

Internally, `List` uses an array to store its contents, which it initializes at some size. Items added to the list get put in this array, until it is full. If more items are added, the list will copy the contents of the full array to a new larger array and uses that one from now on. We could do this array management manually, but `List` takes care of it for us. Also, Unity supports `List` fields just like it supports array fields. They're editable via the inspector, their contents are saved by the editor, and they survive recompilation while in play mode.

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

public class Game : MonoBehaviour {

	…

	List objects;

	…
}
```

But we don't want a generic list. We specifically want a list of `Transform` references. In fact, `List` insists that we specify the type of its contents. `List` is a generic type, which means that it acts like a template for specific list classes, each for a concrete content type. The syntax is `List<T>`, where the template type `T` is appended to the generic type, between angle brackets. In our case the correct type is `List<Transform>`.

```
	List<Transform> objects;
```

Like an array, we have to ensure that we have a list object instance before we use it. We'll do that by creating the new instance in the `Awake` method. In the case of an array, we'd have to use `new Transform[]`. But because we're using a list, we have to use `new List<Transform>()` instead. This invokes the special constructor method of the list class, which can have parameters, which is why we have to append round brackets after the type name.

```
    void Awake () {
        objects = new List<Transform>();
    }
```

Next, add a `Transform` reference to our list each time we instantiate a new one, via the `Add` method of `List`.

```
    void CreateObject () {
        Transform t = Instantiate(prefab);
        t.localPosition = Random.insideUnitSphere * 5f;
        t.localRotation = Random.rotation;
        t.localScale = Vector3.one * Random.Range(0.1f, 1f);
        objects.Add(t);
    }
```

Do we have to wait until the end of `CreateObject` before adding the reference?

We could have added the reference to the list as soon as we got hold of it, so directly after the assignment of the result of `Instantiate` to the local variable. I just put it at the end to point out that we should only add fully-initialized things to the list.

1.6 Clearing the List

Now we can loop through the list in `BeginNewGame` and destroy all the game objects that were instantiated. This works the same as for array, except that the length of the list is found via its `Count` property.

```
	void BeginNewGame () {
		for (int i = 0; i < objects.Count; i++) {
			Destroy(objects[i].gameObject);
		}
	}
```

This leaves us with a list of references to destroyed objects. We must get rid of these as well, by emptying the list via invoking its `Clear` method.

```
	void BeginNewGame () {
		for (int i = 0; i < objects.Count; i++) {
			Destroy(objects[i].gameObject);
		}
		objects.Clear();
	}
```

2 Saving and Loading

To support saving a loading during a single play session, it would be sufficient to keep a list of transformation data in memory. Copy the position, rotation, and scale of all cubes on a save, and reset the game and spawn cubes using the remembered data on a load. However, a true save system is able to remember the game state even after the game is terminated. This requires the game state to be persisted somewhere outside the game. The most straightforward way is to store the data in a file.

> **What about using `PlayerPrefs`?**
>
> As its name suggests, `PlayerPrefs` is designed with game settings and preferences in mind, not game state. While it's possible to pack game state in strings, this is inefficient, hard to manage, and doesn't scale.

2.1 Save Path

Where game files should be stored depends on the file system. Unity takes care of the differences for us, making the path to the folder that we can use available via the `Application.persistentDataPath` property. We can grab the text string from this property and store it in a `savePath` field in `Awake`, so we need to retrieve it only once.

```
	string savePath;

	void Awake () {
		savePath = Application.persistentDataPath;
	}
```

This gives us the path to a folder, not a file. We have to append a file name to the path. Let's just use *saveFile*, not bothering with a file extension. Whether we should use a forward or backward slash to separate the file name from the rest of the path again depends on the operating system. We can use the `Path.Combine` method to take care of the specifics for us. `Path` is part of the `System.IO` namespace.

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

public class Game : MonoBehaviour {

	…

	void Awake () {
		savePath = Path.Combine(Application.persistentDataPath, "saveFile");
	}

	…
}
```

2.2 Opening a File for Writing

To be able to write data to our save file, we first have to open it. This is done via the `File.Open` method, providing it with a path argument. It also needs to know why we're opening the file. We want to write data to it, creating the file if it didn't already exist, or replacing an already existing file. We specify this by providing `FileMode.Create` as a second argument. Do this in a new `Save` method.

```
	void Save () {
		File.Open(savePath, FileMode.Create);
	}
```

`File.Open` returns a file stream, which isn't useful on its own. We need a data stream that we could write data into. This data has to be of a certain format. We'll use the most compact uncompressed format available, which is raw binary data. The `System.IO` namespace has the `BinaryWriter` class to make this possible. Create a new instance of this class, using its constructor method, providing the file stream as an argument. We don't need to keep a reference to the file stream, so we can directly use the `File.Open` invocation as the argument. We do need to keep a reference to the write, so assign it to a variable.

```
    void Save () {
        BinaryWriter writer =
            new BinaryWriter(File.Open(savePath, FileMode.Create));
    }
```

We now have a binary writer variable named writer that references a new binary writer. That's using the word "writer" three times in one expression, which is a bit much. As we're explicitly creating a new `BinaryWriter`, it is redundant to explicitly declare the variable's type as well. Instead, we can use the `var` keyword. This implicitly declares the variable's type to match whatever is immediately assigned to it, which is something that the compiler can figure out in this case.

```
    void Save () {
        var writer = new BinaryWriter(File.Open(savePath, FileMode.Create));
    }
```

We now have a writer variable that references a new binary writer. Its type is obvious.

When should `var` be used?

The `var` keyword is syntactic sugar that you don't need to use at all. While you could use it everywhere that the compiler can infer which type is meant, it's better to only do this when readability is improved and types are explicit. I only use `var` in these tutorials when a variable is declared and immediately assigned to, using the `new` keyword. So only in expressions of the form `var t = new Type`.

The `var` keyword is very useful when working with Language Integrated Query (LINQ) and anonymous types, but that's outside the scope of these tutorials.

2.3 Closing the File

If we open a file, we must make sure that we also close it. It's possible to do this via a `Close` method, but this isn't safe. If something goes wrong between opening and closing the file, an exception could be raised and execution of the method could be terminated before it got to closing the file. We have to carefully handle exceptions to ensure that the file is always closed. There is syntactic sugar to make this easy. Put the declaration and assignment of the `writer` variable inside round brackets, place the `using` keyword in front of it, and a code block after it. The variable is available inside that block, just like the iterator variable `i` of a standard `for` loop.

```
void Save () {
    using (
        var writer = new BinaryWriter(File.Open(savePath, FileMode.Create))
    ) {}
}
```

This will ensure that whatever `writer` references will be properly disposed of, after code execution exist the block, no matter how. This works for special disposable types, which the writer and stream are.

How does `using` work, without the sugar?

In our case, it would look like the following code.

```
var writer = new BinaryWriter(File.Open(savePath, FileMode.Create);
try { … }
finally {
    if (writer != null) {
        ((IDisposable)writer).Dispose();
    }
}
```

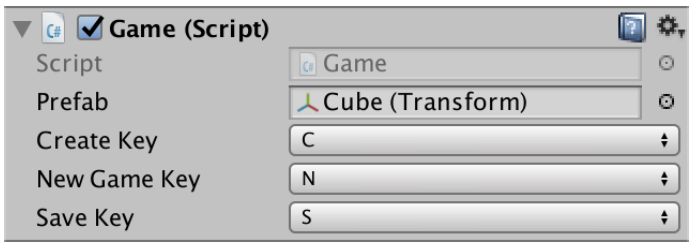
2.4 Writing Data

We can write data to our file by invoking our writer's `Write` method. It is possible to write simple values, like a boolean, integer, and so on, one at at time. Let's begin by writing only how many objects we have instantiated.

```
    void Save () {
        using (
            var writer = new BinaryWriter(File.Open(savePath, FileMode.Create))
        ) {
            writer.Write(objects.Count);
        }
    }
```

To actually save this data, we have to invoke the `Save` method. We'll again control this via a key, in this case using S as the default.

```
    public KeyCode createKey = KeyCode.C;
    public KeyCode saveKey = KeyCode.S;

    …

    void Update () {
        if (Input.GetKeyDown(createKey)) {
            CreateObject();
        }
        else if (Input.GetKey(newGameKey)) {
            BeginNewGame();
        }
        else if (Input.GetKeyDown(saveKey)) {
            Save();
        }
    }
```



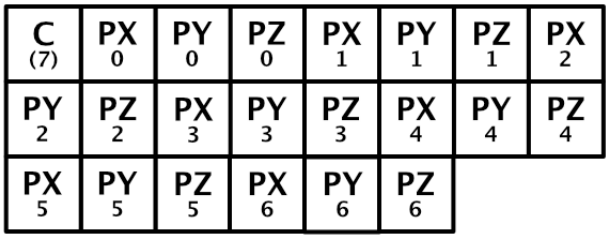
Save key set to S.

Enter play mode, create a few cubes, then save the game by pressing the key. This will have created a *saveFile* file on your file system. If you're not sure where it is located, you can use `Debug.Log` to write the file's path to the Unity console.

You'll find that the file contains four bytes of data. Opening the file in a text editor will show nothing useful, because the data is binary. It might show nothing at all, or might interpret the data as weird characters. There are four bytes because that's the size of an integer.

Besides writing how many cubes we have, we must also store the transformation data of each cube. We do this by looping through the objects and writing their data, one number at a time. For now, we'll limit ourselves to just their positions. So write the X, Y, and Z components of each cube's position, in that order.

```
		writer.Write(objects.Count);
		for (int i = 0; i < objects.Count; i++) {
			Transform t = objects[i];
			writer.Write(t.localPosition.x);
			writer.Write(t.localPosition.y);
			writer.Write(t.localPosition.z);
		}
```



File containing seven positions, in four-byte blocks.

> **Why not use `BinaryFormatter`?**
>
> While relying on `BinaryFormatter` can be convenient, it isn't possible to just serialize a game object hierarchy using a `BinaryFormatter` and deserialize it later. The game object hierarchy has to be recreated manually. Also, writing every bit of data ourselves gives us total control and understanding. Besides that, manually writing data requires less space and memory, is quicker, and makes it easier to support an evolving save file format. Sometimes, games that have already been released drastically change what's stored after an update or expansion. Some of those games can then no longer load a player's old save files. Ideally, a game remains backwards-compatible with all its save file versions.

2.5 Loading Data

To load the data that we just saved, we have to again open the file, this time with `FileMode.Open` as the second argument. Instead of a `BinaryWriter`, we have to use a `BinaryReader`. Do this in a new `Load` method, once again with a `using` statement.

```
	void Load () {
		using (
			var reader = new BinaryReader(File.Open(savePath, FileMode.Open))
		) {}
	}
```

The first thing we wrote to the file was the count property of our list, so that is also the first thing to read. We do this with the `ReadInt32` method of our reader. We have to be explicit what we read, because there is no parameter that makes this clear. The suffix 32 refers to the size of the integer, which is four bytes, thus 32 bits. There are also larger and smaller integer variants, but we don't use those.

```
		using (
			var reader = new BinaryReader(File.Open(savePath, FileMode.Open))
		) {
			int count = reader.ReadInt32();
		}
```

After reading the count, we know how many objects were saved. We have to read that many positions from the file. Do this with a loop, reading three floats per iteration, for the X, Y, and Z components of a position vector. A single-precision `float` is read with the `ReadSingle` method. A double-precision `double` would be read with the `ReadDouble` method.

```
			int count = reader.ReadInt32();
			for (int i = 0; i < count; i++) {
				Vector3 p;
				p.x = reader.ReadSingle();
				p.y = reader.ReadSingle();
				p.z = reader.ReadSingle();
			}
```

Use the vector to set the position of a newly instantiated cube, and add it to the list.

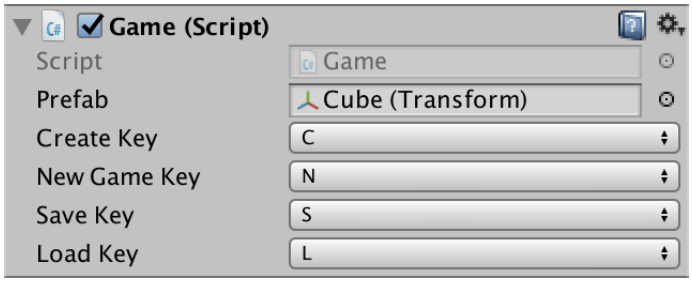
```
			for (int i = 0; i < count; i++) {
				Vector3 p;
				p.x = reader.ReadSingle();
				p.y = reader.ReadSingle();
				p.z = reader.ReadSingle();
				Transform t = Instantiate(prefab);
				t.localPosition = p;
				objects.Add(t);
			}
```

At this point we can recreate all cubes that we saved, but they get added to the cubes that were already in the scene. To properly load a previously saved game, we have to reset the game before recreating it. We can do this by invoking `BeginNewGame` before loading the data.

```
	void Load () {
		BeginNewGame();
		using (
			var reader = new BinaryReader(File.Open(savePath, FileMode.Open))
		) {
			…
		}
	}
```

Have `Game` invoke `Load` when a key is pressed, using L as the default.

```
	public KeyCode createKey = KeyCode.C;
	public KeyCode saveKey = KeyCode.S;
	public KeyCode loadKey = KeyCode.L;

	…

	void Update () {
		…
		else if (Input.GetKeyDown(saveKey)) {
			Save();
		}
		else if (Input.GetKeyDown(loadKey)) {
			Load();
		}
	}
```



Load key set to L.

Now the player can save their cubes and later load them, either during the same play session or another one. But because we're only storing the position data, the rotation and scale of cubes are not stored. As a result, loaded cubes all end up with the default rotation and scale of the prefab.

What would happen if we loaded before saving anything?

Then you would try to open a file that doesn't exist, which would result in an exception. This tutorial doesn't bother with checking whether the file exist or contains valid data, but we'll be more careful in a future tutorial.

3 Abstracting Storage

Although we need to know the specifics of reading and writing binary data, that's rather low-level. Writing a single 3D vector requires three invocations of `Write`. When saving and loading our objects, it's more convenient if we could work at a slightly higher level, reading or writing an entire 3D vector with a single method invocation. Also, it would be nice if we could just use `ReadInt` and `ReadFloat`, instead of having to worry about all the different variants that we don't use. Finally, it shouldn't matter whether the data is stored in binary, plain text, base-64, or another encoding method. `Game` doesn't need to know such details.

3.1 Game Data Writer and Reader

To hide the details of reading and writing data, we're going to create our own reader and writer classes. Let's begin with the writer, naming it `GameDataWriter`.

`GameDataWriter` does not extend `MonoBehaviour`, because we won't attach it to a game object. It will acts as a wrapper for `BinaryWriter`, so give it a single writer field.

```
using System.IO;
using UnityEngine;

public class GameDataWriter {

	BinaryWriter writer;
}
```

A new object instance of our custom writer type can be created via `new GameDataWriter()`. But this only makes sense if we have a writer to wrap. So create a custom constructor method with a `BinaryWriter` parameter. This is a method with the type name of its class as its own name, which also acts as its return type. It replaces the implicit default constructor method.

```
	public GameDataWriter (BinaryWriter writer) {
	}
```

Although invoking a constructor method results in a new object instance, such methods don't explicitly return anything. The object gets created before the constructor is invoked, which can then take care of any required initialization. In our case, that's simply assigning the writer parameter to the object's field. As I've used the same name for both, I have to use the `this` keyword to explicitly indicate that I'm referring to the object's field instead of the parameter.

```
	public GameDataWriter (BinaryWriter writer) {
		this.writer = writer;
	}
```

The most basic functionality is to write a single `float` or `int` value. Add public `Write` methods for this, simply forwarding the invocation to the actual writer.

```
	public void Write (float value) {
		writer.Write(value);
	}

	public void Write (int value) {
		writer.Write(value);
	}
```

Besides that, also add methods to write a `Quaternion`—for rotations—and a `Vector3`. These methods have to write all the components of their parameter. In the case of a quaternion, that's four components.

```
	public void Write (Quaternion value) {
		writer.Write(value.x);
		writer.Write(value.y);
		writer.Write(value.z);
		writer.Write(value.w);
	}

	public void Write (Vector3 value) {
		writer.Write(value.x);
		writer.Write(value.y);
		writer.Write(value.z);
	}
```

Next, create a new `GameDataReader` class, using the same approach as for the writer. In this case, we wrap a `BinaryReader`.

```
using System.IO;
using UnityEngine;

public class GameDataReader {

	BinaryReader reader;

	public GameDataReader (BinaryReader reader) {
		this.reader = reader;
	}
}
```

Give it methods that are simply named `ReadFloat` and `ReadInt`, that forward the invocations to `ReadSingle` and `ReadInt32`.

```
	public float ReadFloat () {
		return reader.ReadSingle();
	}

	public int ReadInt () {
		return reader.ReadInt32();
	}
```

Also create `ReadQuaternion` and `ReadVector3` methods. Read their components in the same order that we write them.

```
	public Quaternion ReadQuaternion () {
		Quaternion value;
		value.x = reader.ReadSingle();
		value.y = reader.ReadSingle();
		value.z = reader.ReadSingle();
		value.w = reader.ReadSingle();
		return value;
	}

	public Vector3 ReadVector3 () {
		Vector3 value;
		value.x = reader.ReadSingle();
		value.y = reader.ReadSingle();
		value.z = reader.ReadSingle();
		return value;
	}
```

3.2 Persistable Objects

Now it's a lot simpler to write the transform data of cubes in **`Game`**. But we can go one step further. What if **`Game`** could simply invoke `writer.Write(objects[i])`? That would be very convenient, but would require **`GameDataWriter`** to know the details of writing a game object. But it's better to keep the writer simple, limited to primitive values and simple structs.

We can turn this reasoning around. **`Game`** doesn't need to know how to save a game object, that's the responsibility of the object itself. All the object needs is a writer to save itself. Then **`Game`** could use `objects[i].Save(writer)`.

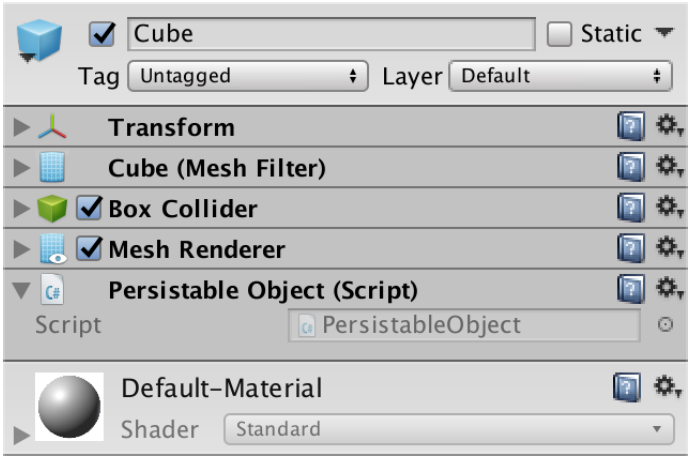
Our cubes are simple objects, without any custom components attached. So the only thing that's to save is the transform component. Let's create a **`PersistableObject`** component script that knows how to save and load that data. It simply extends **`MonoBehaviour`** and has a public `Save` and `Load` method with either a **`GameDataWriter`** or **`GameDataReader`** parameter. Have it save the transform position, rotation, and scale, and load them in the same order.

```
using UnityEngine;

public class PersistableObject : MonoBehaviour {

	public void Save (GameDataWriter writer) {
		writer.Write(transform.localPosition);
		writer.Write(transform.localRotation);
		writer.Write(transform.localScale);
	}

	public void Load (GameDataReader reader) {
		transform.localPosition = reader.ReadVector3();
		transform.localRotation = reader.ReadQuaternion();
		transform.localScale = reader.ReadVector3();
	}
}
```

The idea is that a game object that can be persisted only has one `PersistableObject` component attached to it. Having multiple such components makes no sense. We can enforce this by adding the `DisallowMultipleComponent` attribute to the class.

```
[DisallowMultipleComponent]
public class PersistableObject : MonoBehaviour {

	…
}
```

Add this component to our cube prefab.



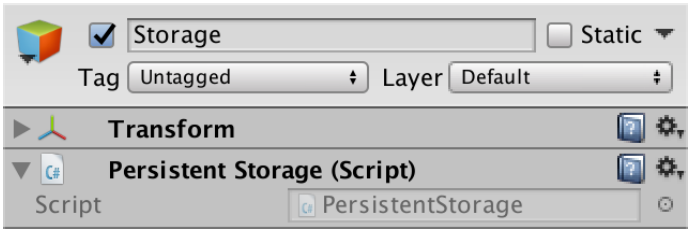
Persistable prefab.

3.3 Persistent Storage

Now that we have a persistent object type, let's also create a `PersistentStorage` class to save such an object. It contains the same saving and loading logic as `Game`, except that it only saves and loads a single `PersistableObject` instance, provided via a parameter to public `Save` and `Load` methods. Make it a `MonoBehaviour`, so we can attach it to a game object and it can initialize its save path.

```
using System.IO;
using UnityEngine;

public class PersistentStorage : MonoBehaviour {

	string savePath;

	void Awake () {
		savePath = Path.Combine(Application.persistentDataPath, "saveFile");
	}

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

Add a new game object to the scene with this component attached. It represents the persistent storage of our game. Theoretically, we could have multiple such storage objects, used to store different things, or to provide access to different storage types. But in this tutorial we use just this single file storage object.



Storage object.

3.4 Persistable Game

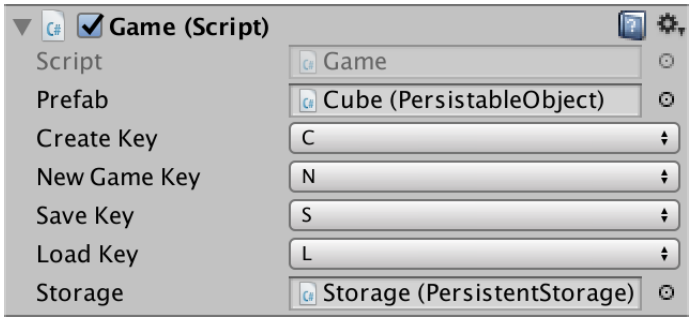
To make use of the new persistable object approach, we have to rewrite `Game`. Change the `prefab` and `objects` content type to `PersistableObject`. Adjust `CreateObject` so it can deal with this type change. Then remove all the code specific to reading from and writing to files.

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

public class Game : MonoBehaviour {

	public PersistableObject prefab;

	…

	List<PersistableObject> objects;

//	string savePath;

	void Awake () {
		objects = new List<PersistableObject>();
//		savePath = Path.Combine(Application.persistentDataPath, "saveFile");
	}

	void Update () {
		…
		else if (Input.GetKeyDown(saveKey)) {
//			Save();
		}
		else if (Input.GetKeyDown(loadKey)) {
//			Load();
		}
	}

	…

	void CreateObject () {
		PersistableObject o = Instantiate(prefab);
		Transform t = o.transform;
		…
		objects.Add(o);
	}

//	void Save () {
//		…
//	}

//	void Load () {
//		…
//	}
}
```

We'll have `Game` rely on a `PersistentStorage` instance to take care of the details of storing data. Add a public `storage` field of this type, so we can give `Game` a reference to our storage object. To again save and load the game state, we have `Game` itself extend `PersistableObject`. Then it can load and save itself, using the storage.

```
public class Game : PersistableObject {

	…

	public PersistentStorage storage;

	…

	void Update () {
		if (Input.GetKeyDown(createKey)) {
			CreateObject();
		}
		else if (Input.GetKeyDown(saveKey)) {
			storage.Save(this);
		}
		else if (Input.GetKeyDown(loadKey)) {
			BeginNewGame();
			storage.Load(this);
		}
	}

	…
}
```

Connect the storage via the inspector. Also reconnect the prefab, as its reference was lost due to the field's type change.



Game connected to prefab and storage.

3.5 Overriding Methods

When we save and load the game now, we end up writing and reading the transformation data of our main game object. This is useless. Instead, we have to save and load its list of objects.

I loaded before saving, and the game object got a strange position?

If you're loading an older save file at this point, you end up misinterpreting the data. The count integer will be mistaken for the X position, the first saved position's X and Y end up used for the Y and Z position, then the rotation will be filled with the next values, and so on. If you had saved less than four positions, the file would contain too little data to load a complete transform. Then you'd get an error complaining that you tried to read beyond the end of the file.

Instead of relying on the `Save` method defined in `PersistableObject`, we have to give `Game` its own public version of `Save` with a `GameDataWriter` parameter. In it, write the list as we did before, now using the convenient `Save` method of the objects.

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

This is not yet enough to make it work. The compiler complains that `Game.Save` hides the inherited member `PersistableObject.Save`. While `Game` can work with its own `Save` version, `PersistentStorage` only knows about `PersistableObject.Save`. So it would invoke this method, not the one from `Game`. To make sure that the correct `Save` method gets invoked, we have to explicitly declare that we override the method that `Game` inherited from `PersistableObject`. That's done by adding the `override` keyword to the method declaration.

```
	public override void Save (GameDataWriter writer) {
		…
	}
```

However, we cannot just override any method we like. By default, we're not allowed to do this. We have to explicitly enable it, by adding the `virtual` keyword to the `Save` and `Load` method declarations in `PersistableObject`.

```
	public virtual void Save (GameDataWriter writer) {
		writer.Write(transform.localPosition);
		writer.Write(transform.localRotation);
		writer.Write(transform.localScale);
	}

	public virtual void Load (GameDataReader reader) {
		transform.localPosition = reader.ReadVector3();
		transform.localRotation = reader.ReadQuaternion();
		transform.localScale = reader.ReadVector3();
	}
```

What does the `virtual` keyword mean?

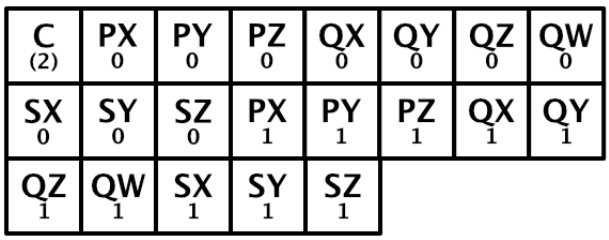
At a very low level, there aren't really objects or methods. There is just data, a portion of which is used as instructions to be executed by the CPU. Method invocations—unless optimized away—become instructions that tell the CPU to jump to another data point and continue execution from there. Besides that, it might also put some argument values in place. So when `PersistentStorage` invokes the `Save` method of the `PersistableObject` type, it becomes an instruction to jump to a fixed location. That we passed it and instance of `Game`—a subtype of `PersistableObject`—doesn't affect this at all. The object instance used to invoke the method is just another argument.

The `virtual` keyword changes this approach. Instead of using a hard-coded location, the compiler adds instructions to look up where to jump to, based on the type that was involved. Instead of going "It's this method, so always jump there." it becomes "Does this type contain the jump destination for this method? If yes, go there. If no, check its direct parent type. Repeat until the destination is found." This approach is know as a virtual method, function, or call table. Hence `virtual`. It allows subtypes to override the functionality of their parent types.

Note that the specifics of the low-level instructions that end up executed by the CPU can vary a lot, especially when using Unity's IL2CPP to create native executables. Wherever possible, IL2CCP eliminates the use of virtual method tables.

`PersistentStorage` will now end up invoking our `Game.Save` method, even though it's passed to it as a `PersistableObject` argument. Also have `Game` override the `Load` method.

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



File containing two transforms.

The next tutorial is Object Variety.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

BECOME A PATRON

Or make a direct donation!

made by Jasper Flick