Runner, a minimal side-scroller

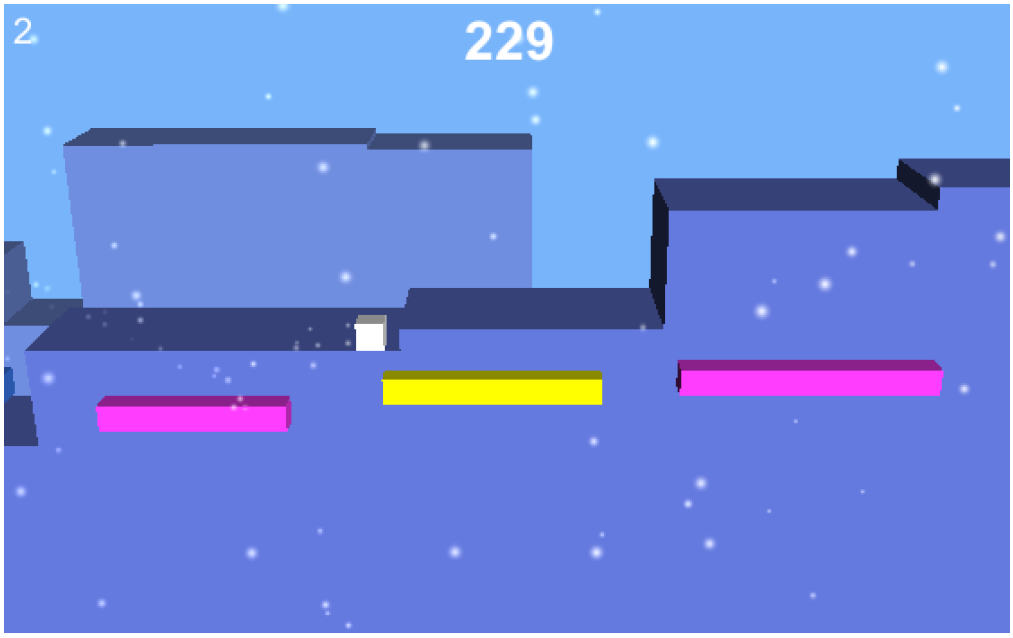
In this tutorial we'll create a very simple endless running game. You'll learn to

- generate a layered background;
- reuse objects;
- use simple physics;
- detect input to make the player jump;
- implement a power-up;
- write a small event manager;
- switch stuff on and off on demand;
- make a minimal GUI.

You're assumed to know your way around Unity's editor and know the basics of creating C# scripts. If you've completed the Clock tutorial you're good to go. The Graphs tutorial is useful too, but not necessary.

Note that I will often omit chunks of code that have remained the same, only new code is shown. The context of the new code should be clear.

This tutorial is quite old. I created it for Unity 3 and later updated it to Unity 4, but I won't update it to take advantage of the new featuers of Unity 5. I recommend you go through the Swirly Pipe tutorial instead, which is the spiritual successor of this one. Having said that, this tutorial still contains useful things that aren't mentioned in the new one.



You'll make a run for it.

Game Design

Before we get started, we should make some decisions about what we put in the game. We're going to make a very simple 2D side-scroller, but that's still very vague. Let's narrow it down a bit.

For gameplay, we'll have a runner who dashes towards the right of the screen. The player needs to jump from platform to platform for as long as possible. These platforms can come in different flavors, slowing down or speeding up the runner. We'll also include a single power-up, which is a booster that allows mid-air jumps.

For graphics, we'll simply use cubes and standard particle systems. The cubes will be used for the runner, power-up, platforms, and a skyline background. We'll use particle systems to add a trail effect and lots of floating stuff to give a better sense of speed and depth.

There won't be any sound or music.

Play the Game

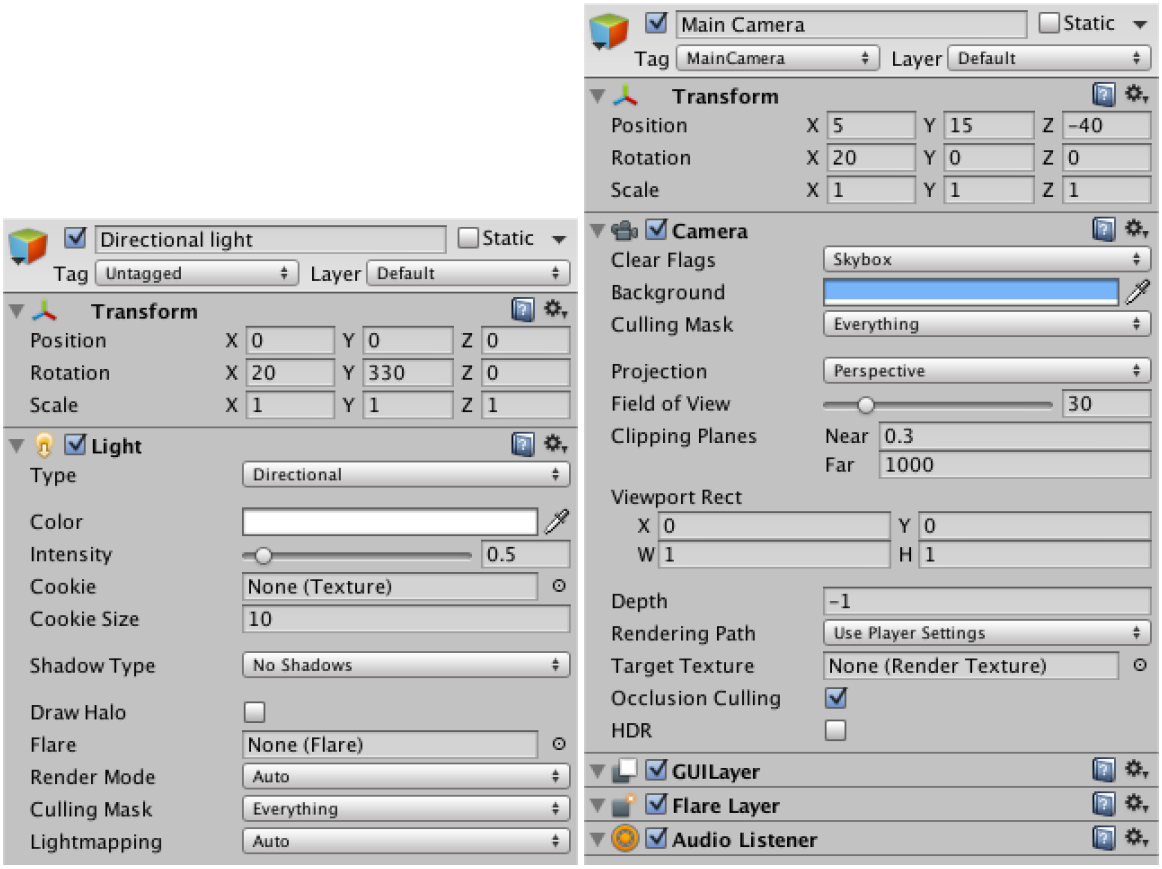
Setting the Scene

Open a new project without any packages. The default *2 by 3* editor layout is a good one for this project, but you can use whatever layout you prefer. Let's make this game with a 16:10 display ratio in mind, so select this option in the *Game* view.

Our game is basically 2D, but we want to keep a little feeling of 3D. An orthographic camera doesn't allow for 3D, so we stick to a perspective camera. This way we can also get a multilayered scrolling background by simply placing stuff at various distances. Let's say the foreground is at depth 0 and we have a background layer at depth 50 and another one at depth 100. Let's place three cubes at these depths and use them as guides to construct the scene. I went ahead and picked a view angle and color setup, but you're free to experiment and choose whatever you like.

Add a directional light (*GameObject / Create Other / Directional Light*) with a rotation of (20, 330, 0). This gives us a light source that's shining over our right shoulder. Because it's a directional light its position doesn't matter.

Reduce the *Field of View* of the *Main Camera* to 30, position it at (5, 15, -40), and rotate it by (20, 0, 0). Also change its *Background* color to (120, 180, 250).



Light and camera.

Create three cubes (*GameObject / Create Other / Cube*) with Z positions of 0, 50, and 100. Call them *Runner*, *Skyline Close*, and *Skyline Far Away* respectively. Remove the collider from both skyline cubes, because we won't be needing those.

Create a material for each in the *Project* view via *Create / Material*, naming them *Runner Mat* and so on, then assign them to the cubes by dragging. I used default diffuse shaders with the colors white, (100, 120, 220), and (110, 140, 220).

What's a collider?

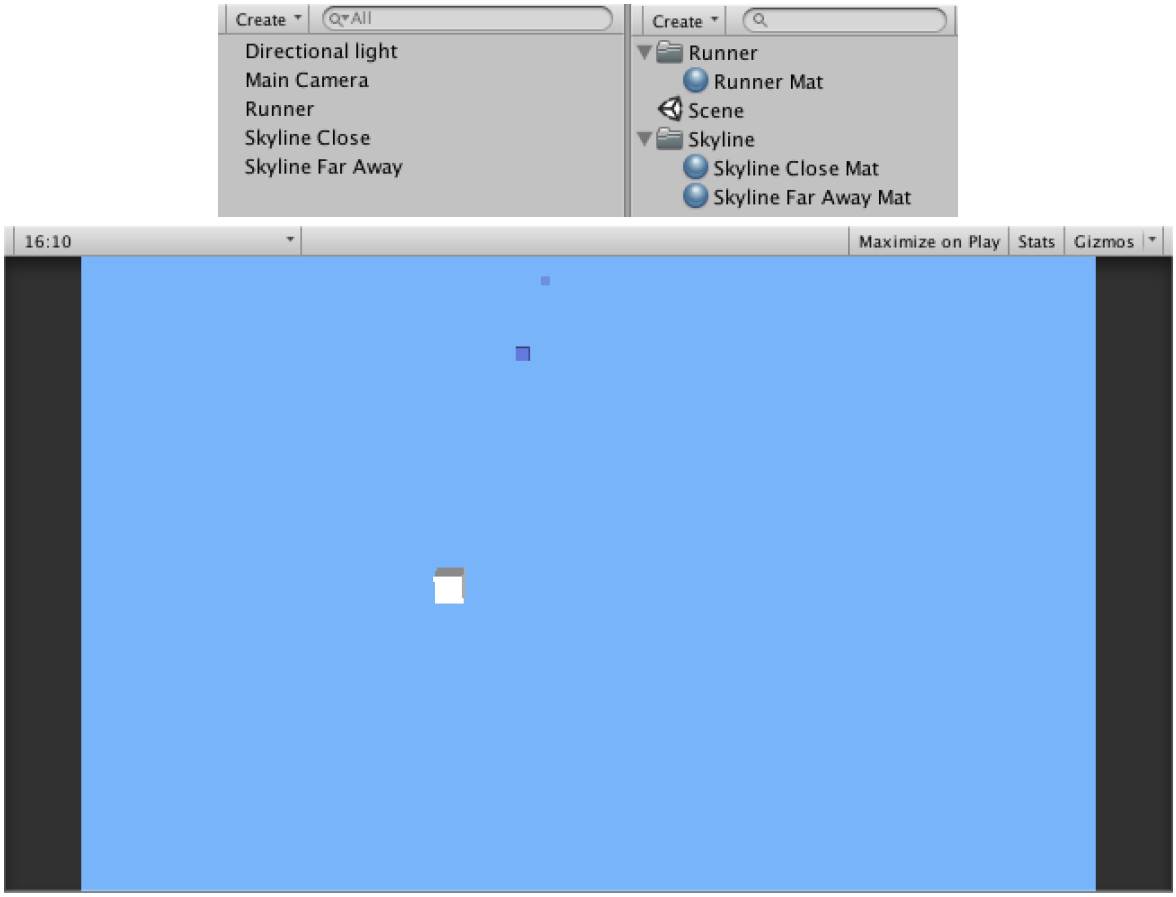
Why remove the colliders?

What's a material?

What about the fourth color component?

The three cube configurations.

To keep things organized, create a *Runner* and a *Skyline* folder in the *Project* view via *Create / Folder* and put the materials in there.



Hierarchy, project, and game views.

Running

So far it doesn't look like much and nothing's happening yet, but that will change. Let's start by creating a mock-up of the game in action by instructing the *Runner* to move to the right.

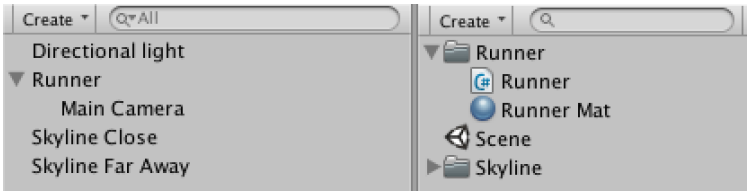
Create a new C# script called *Runner* inside the *Runner* folder and attach it to our *Runner* cube. Write the following code to make it move.

```
using UnityEngine;

public class Runner : MonoBehaviour {

	void Update () {
		transform.Translate(5f * Time.deltaTime, 0f, 0f);
	}
}
```

While it's not much, it already shows us a problem when entering play mode. The camera does not follow the cube. To fix this, drag *Main Camera* onto *Runner* so is becomes a child of it.

Now *Runner* remains at a fixed position in our view and we can see that the close skyline cube appears to move faster than the one further away.

What does it mean to be a child?



Setup for moving camera.

Generating a Skyline

Now that we have rudimentary movement, let's generate a row of cubes to construct an endless skyline. The first thing we should realize is that only the visible part of the skyline needs to exist. As soon as a cube falls off the left side of the screen, it can be destroyed. Or better yet, it can be reused to build the next part of the skyline that's about to enter view. We can program this behaviour by generating a queue of cubes and constantly moving the front cube to the back as soon as it's no longer visible.

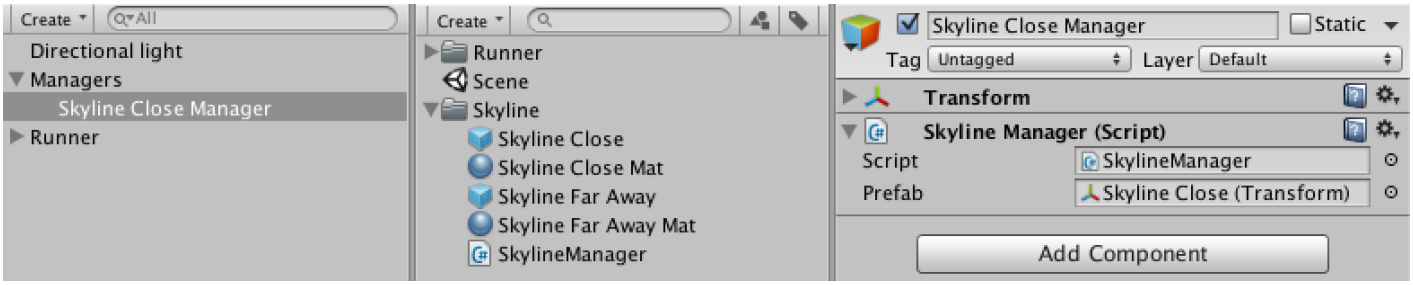
Create a new C# script in the *Skyline* folder and name it *SkylineManager*. We will use it to create two managers, one for each of the skyline layers. At minimum, it needs to know which prefab to use to generate the skyline, so let's start by adding a public variable for that.

```
using UnityEngine;

public class SkylineManager : MonoBehaviour {

	public Transform prefab;
}
```

To keep things organized, create a new empty object (*GameObject / Create Empty*) named *Managers* which we'll use as a container for all of our manager objects. Create another empty object named *Skyline Close Manager*, make it a child of *Managers*, and create a *SkylineManager* component for it by dragging the script on it.

Now turn both skyline cubes into prefabs by dragging them into the *Skyline* project folder or via *Create / Prefab* and then dragging onto that. Afterwards, delete both cubes from the *Hierarchy*. Now drag the *Skyline Close* prefab onto the *Prefab* field of our *Skyline Close Manager*.

What's a prefab?



Manager and prefabs.

We need a starting point from where we'll begin spawning cubes, so let's add a *startPosition* variable. We also need to determine how many cubes we need to spawn to fill the screen. Let's simply use a variable named *numberOfCubes* for that. To keep track of where the next cube needs to spawn we'll use a private variable named *nextPosition*.

```
	public Transform prefab;
	public int numberOfObjects;
	public Vector3 startPosition;

	private Vector3 nextPosition;

	void Start () {
		nextPosition = startPosition;
	}
```

The next step is spawning the initial row of cubes. We'll use a simple loop for that, instantiating new objects, setting their position, and advancing *nextPosition* by the width of the object so they

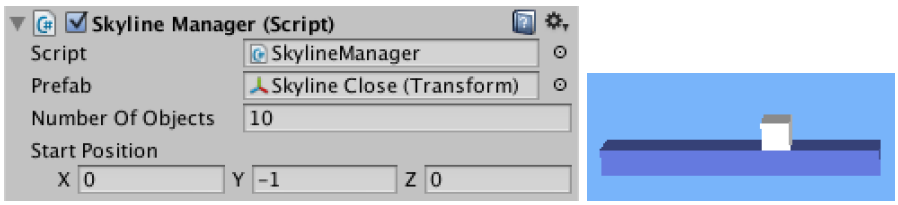
form an unbroken line. Because we're using unit cubes, their with is equal to their `localScale.x`.

What does `Instantiate` do?

What does `+=` do?

```
	void Start () {
		nextPosition = startPosition;
		for (int i = 0; i < numberOfObjects; i++) {
			Transform o = (Transform)Instantiate(prefab);
			o.localPosition = nextPosition;
			nextPosition.x += o.localScale.x;
		}
	}
```

Now set *Start Position* to (0, -1, 0) and set *Number of Objects* to 10. When entering play mode, we'll see a short row of cubes appear below *Runner*. However, once the cubes move out of view we'll never see them again.



Instantiating a skyline.

The idea is that we'll recyle objects once *Runner* has moved past them by some distance. For this to work, the manager must know how far *Runner* has traveled. We can provide for this by adding a static variable named `distanceTraveled` to `Runner` and making sure that it's always up to date.

Why is `distanceTraveled` static?

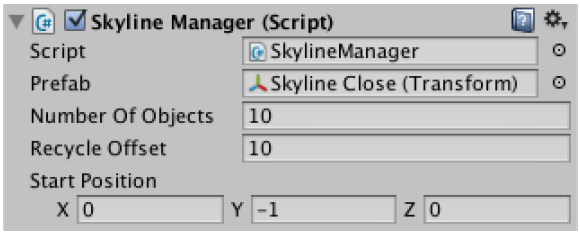
```
using UnityEngine;

public class Runner : MonoBehaviour {

	public static float distanceTraveled;

	void Update () {
		transform.Translate(5f * Time.deltaTime, 0f, 0f);
		distanceTraveled = transform.localPosition.x;
	}
}
```

Now we'll store our skyline objects in a queue and keep checking whether the first object in it should be recycled. If so, we'll reposition it and move it to the back of the queue. Let's use a *recycleOffset* variable to configure how far behind *Runner* this reuse should occur. Set it to 10 for now.

What's a `Queue`?

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

public class SkylineManager : MonoBehaviour {

	public Transform prefab;
	public int numberOfObjects;
	public float recycleOffset;
	public Vector3 startPosition;
```

```
	private Vector3 nextPosition;
	private Queue<Transform> objectQueue;

	void Start () {
		objectQueue = new Queue<Transform>(numberOfObjects);
		nextPosition = startPosition;
		for (int i = 0; i < numberOfObjects; i++) {
			Transform o = (Transform)Instantiate(prefab);
			o.localPosition = nextPosition;
			nextPosition.x += o.localScale.x;
			objectQueue.Enqueue(o);
		}
	}

	void Update () {
		if (objectQueue.Peek().localPosition.x + recycleOffset < Runner.distanceTraveled) {
			Transform o = objectQueue.Dequeue();
			o.localPosition = nextPosition;
			nextPosition.x += o.localScale.x;
			objectQueue.Enqueue(o);
		}
	}
}
```



Recycling skyline configuration.

This works! After entering play mode, you'll see the cubes reposition themselves. However, it doesn't look much like a skyline yet. To make it more like a real irregular skyline, let's randomly scale the cubes whenever they're placed or recycled.

First, consider that both initially placing and later recycling a cube is basically doing the same thing. Let's put this code in its own `Recycle` method and rewrite our `Start` and `Update` methods to both use it.

```
	void Start () {
		objectQueue = new Queue<Transform>(numberOfObjects);
		for (int i = 0; i < numberOfObjects; i++) {
			objectQueue.Enqueue((Transform)Instantiate(prefab));
		}
		nextPosition = startPosition;
		for (int i = 0; i < numberOfObjects; i++) {
			Recycle();
		}
	}

	void Update () {
		if (objectQueue.Peek().localPosition.x + recycleOffset < Runner.distanceTraveled) {
			Recycle();
		}
	}

	private void Recycle () {
		Transform o = objectQueue.Dequeue();
		o.localPosition = nextPosition;
		nextPosition.x += o.localScale.x;
		objectQueue.Enqueue(o);
	}
```

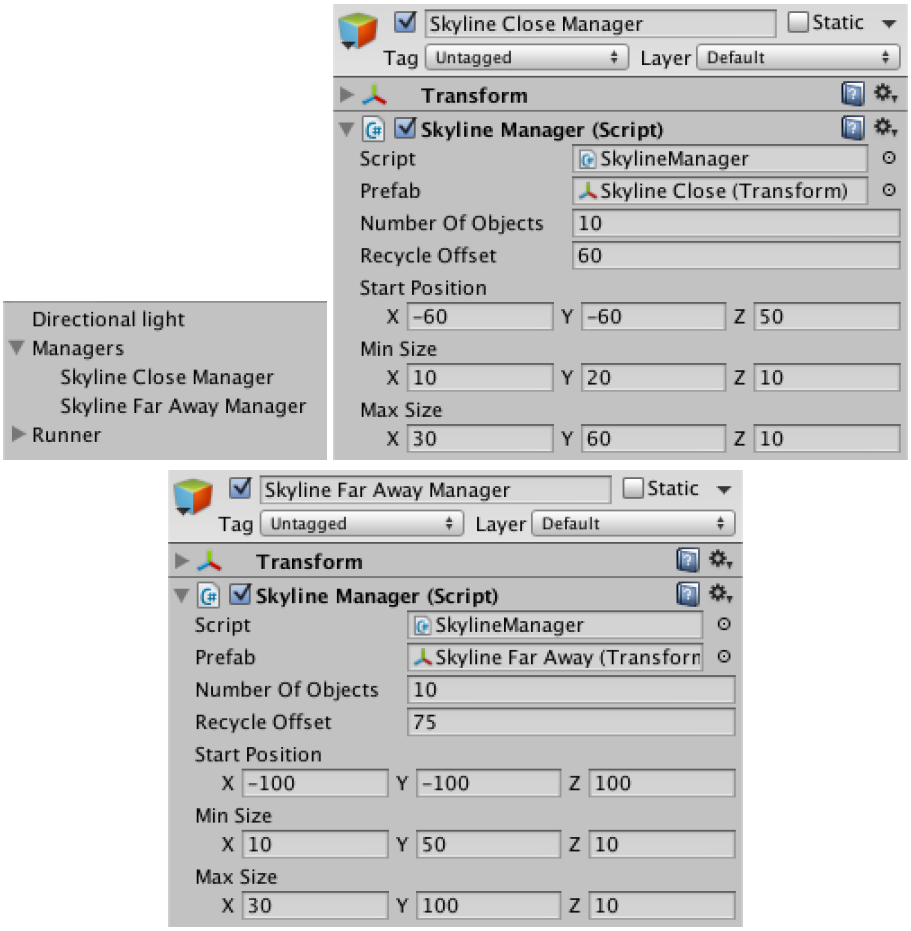
Next, we'll introduce two variables to configure the maximum and minimum allowed size and use them to randomly scale our objects. After picking a scale, we'll make sure to position the object so they're all aligned at the bottom. For that we'll need to offset by half their size, because the objects are centered around their position.

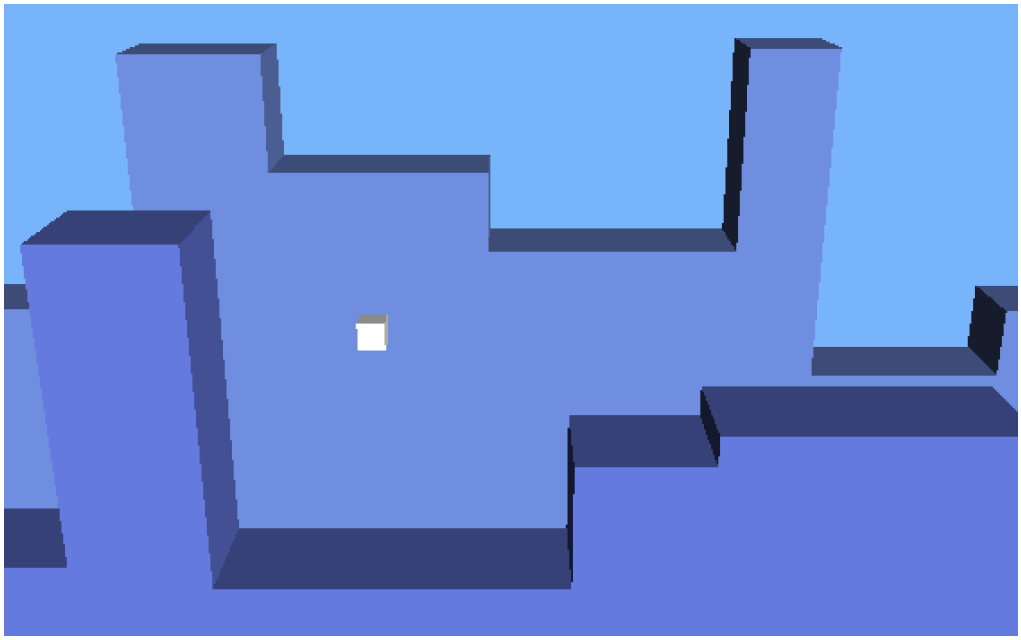
What does `Random.Range` do?

```
public Vector3 minSize, maxSize;

private void Recycle () {
	Vector3 scale = new Vector3(
		Random.Range(minSize.x, maxSize.x),
		Random.Range(minSize.y, maxSize.y),
		Random.Range(minSize.z, maxSize.z));

	Vector3 position = nextPosition;
	position.x += scale.x * 0.5f;
	position.y += scale.y * 0.5f;

	Transform o = objectQueue.Dequeue();
	o.localScale = scale;
	o.localPosition = position;
	nextPosition.x += scale.x;
	objectQueue.Enqueue(o);
}
```

To get a nice skyline effect, set *Start Position* to (-60, -60, 50), set *Min Size* to (10, 20, 10), set *Max Size* to (30, 60, 10), and set *Recycle Offset* to 60.

Let's go ahead and add the second skyline layer as well. Duplicate *Skyline Close Manager* and change its name to *Skyline Far Away Manager*. Change its *Prefab* to the *Skyline Far Away* prefab. Set its *Start Position* to (-100, -100, 100), its *Recycle Offset* to 75, its *Min Size* to (10, 50, 10), and its *Max Size* to (30, 100, 10). Of course you can use any values you like instead.





The complete skyline.

Generating Platforms

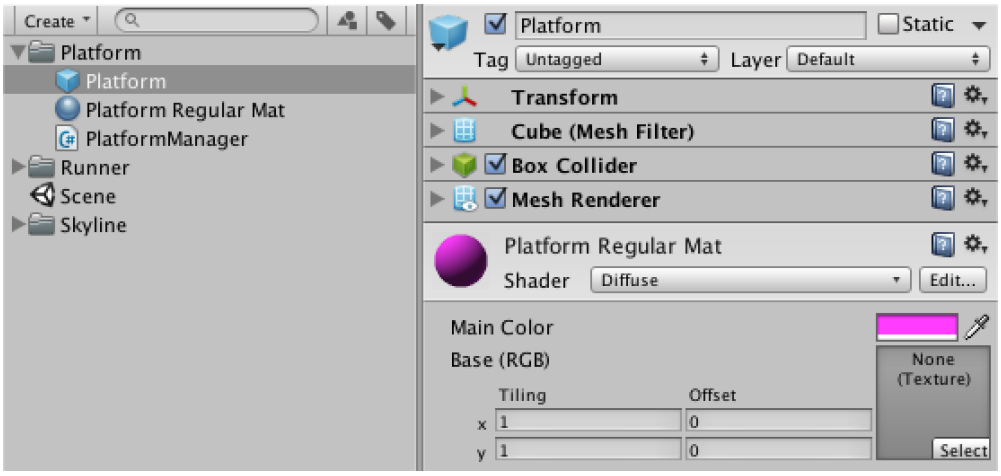
Adding platforms to the game is basically doing the same thing as generating a skyline, with only a few differences. The elevation of the platforms needs to change at random and there need to be gaps between them. Also, we want to constrain the elevation of the platforms to make sure our skyline remains properly in view. If a platform is placed outside this range, we should bounce it back.

Create a new folder in the *Project* view named *Platform*. Create a new C# script in there called *PlatformManager* and copy the code from *SkylineManager* into it. Then change the code as shown below to make if conform to our needs.

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

public class PlatformManager : MonoBehaviour {

	public Transform prefab;
	public int numberOfObjects;
	public float recycleOffset;
	public Vector3 startPosition;
	public Vector3 minSize, maxSize, minGap, maxGap;
	public float minY, maxY;

	private Vector3 nextPosition;
	private Queue<Transform> objectQueue;

	void Start () {
		objectQueue = new Queue<Transform>(numberOfObjects);
		for(int i = 0; i < numberOfObjects; i++){
			objectQueue.Enqueue((Transform)Instantiate(prefab));
		}
		nextPosition = startPosition;
		for(int i = 0; i < numberOfObjects; i++){
			Recycle();
		}
	}

	void Update () {
		if(objectQueue.Peek().localPosition.x + recycleOffset < Runner.distanceTraveled){
			Recycle();
		}
	}

	private void Recycle () {
```

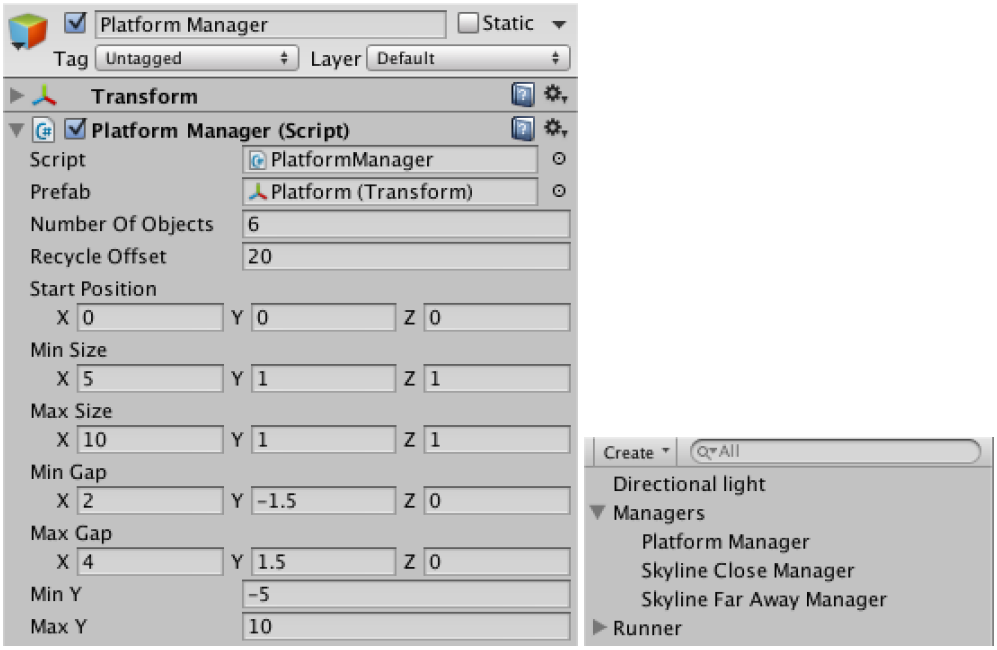
```
			Vector3 scale = new Vector3(
				Random.Range(minSize.x, maxSize.x),
				Random.Range(minSize.y, maxSize.y),
				Random.Range(minSize.z, maxSize.z));

			Vector3 position = nextPosition;
			position.x += scale.x * 0.5f;
			position.y += scale.y * 0.5f;

			Transform o = objectQueue.Dequeue();
			o.localScale = scale;
			o.localPosition = position;
			objectQueue.Enqueue(o);

			nextPosition += new Vector3(
				Random.Range(minGap.x, maxGap.x) + scale.x,
				Random.Range(minGap.y, maxGap.y),
				Random.Range(minGap.z, maxGap.z));

			if(nextPosition.y < minY){
				nextPosition.y = minY + maxGap.y;
			}
			else if(nextPosition.y > maxY){
				nextPosition.y = maxY - maxGap.y;
			}
		}
	}
```

Now let's make a prefab for the platforms, along with a material for it. You can do this by duplicating one of the skyline materials and changing its color to (255, 60, 255). Call it *Platform Regular Mat*. Then create a new cube, assign the material to it, and turn it into a prefab called *Platform*. Put them both in the *Platform* folder.



Platform prefab.

Also create a new empty object named *Platform Manager* and make it a child of *Managers*. Give it a *Platform Manager* component by draggin the script onto it. Assign our new *Platform* prefab to its *Prefab* field. Set its *Number Of Objects* to 6, its *Recycle Offset* to 20, its *Start Position* to (0, 0, 0), its *Min Size* to (5, 1, 1), and its *Max Size* to (10, 1, 1). Then set the new fields *Min Gap*, *Max Gap*, *Min Y*, and *Max Y* to (2, -1.5, 0), (4, 1.5, 0), -5, and 10 respectively. Once again, you can experiment with different settings.



Platform manager.

Jumping and Falling

Now that we have platforms, it's time to upgrade our runner. We'll use Unity's physics engine to make it jump, fall, and collide with the platforms, so add a *Rigidbody* component to *Runner* via *Component / Physics / Rigidbody*. We don't want it to rotate or disappear into the distance, so constrain it by freezing its Z position and all rotation axes.

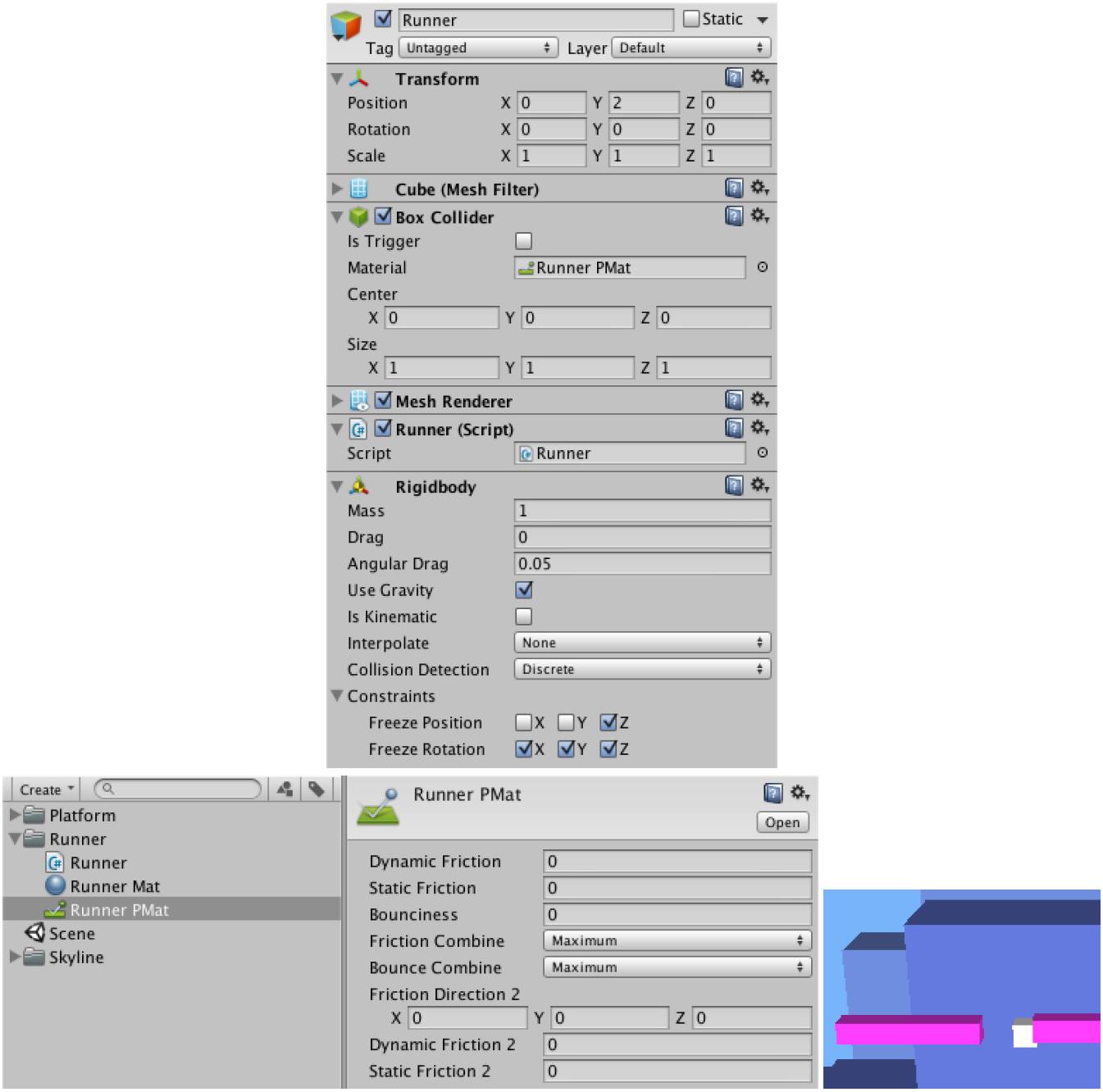
As movement will be accomplished by gliding across the platforms, let's create a physic material (*Create / Physic Material*) with no friction whatsoever. Set all its fields to zero and both combine options to maximum. This way friction will be determined by whatever it's gliding across.

Name the new physic material *Runner PMat*, put it in the *Runner* folder, and assign it to the *Material* field of the *Box Collider* of *Runner*.

Reposition *Runner* to (0, 2, 0) so that it will begin by falling down on the first platform. Then try out play mode to see what happens!

What's a rigidbody?

What's a physic material?



Runner with physics.

So *Runner* falls on the platform and then moves to the right. It even falls again after it moves past the platform. But when it happens to collide with the side of the next platform it behaves a bit weird. This is because we're still changing its position in an `Update` method. We should leave its movement to the physics engine and instead apply forces to it.

Remove the call to `Translate` from the `Update` method of `Runner`. Instead, we'll use two of Unity's collision event methods – `OnCollisionEnter` and `OnCollisionExit` – to detect when we touch or leave a platform. As long as we're touching a platform, we apply an acceleration to make us run faster.

Let's make the acceleration configurable and set it to 5 in the editor.

When is `FixedUpdate` called?

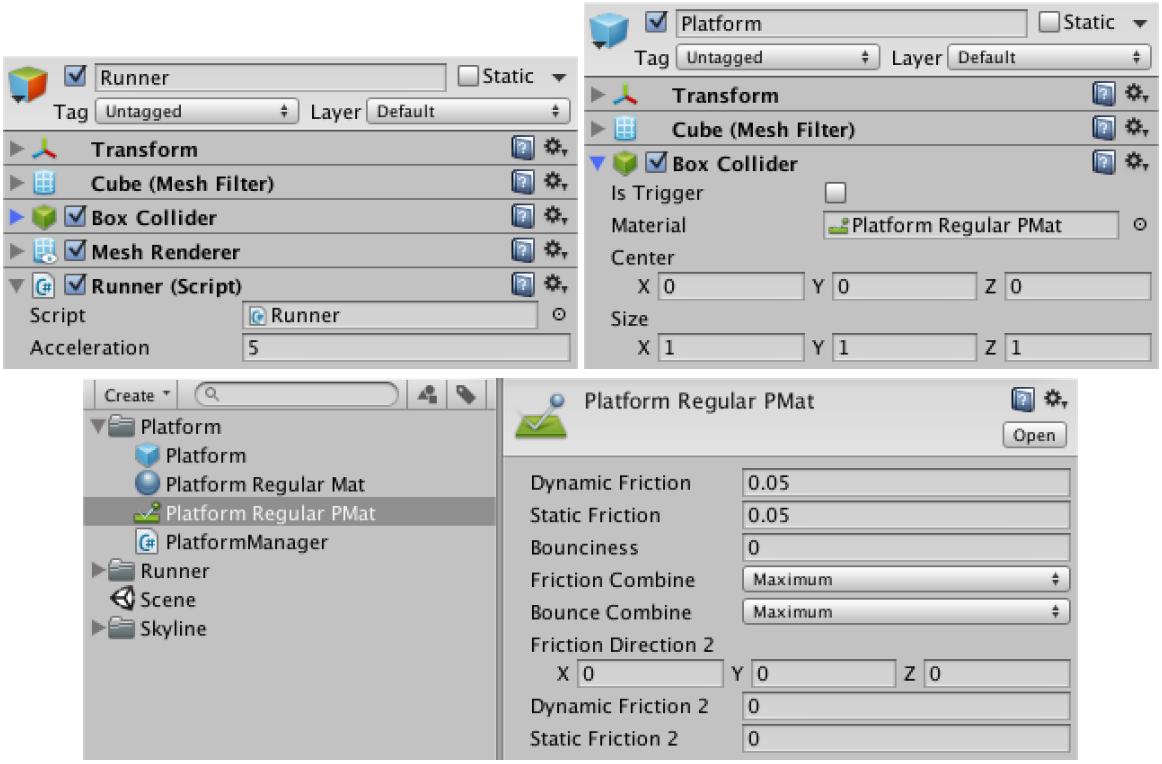
What does `AddForce` do?

```
using UnityEngine;

public class Runner : MonoBehaviour {
```

```
	public static float distanceTraveled;

	public float acceleration;

	private bool touchingPlatform;

	void Update () {
		distanceTraveled = transform.localPosition.x;
	}

	void FixedUpdate () {
		if(touchingPlatform){
			rigidbody.AddForce(acceleration, 0f, 0f, ForceMode.Acceleration);
		}
	}

	void OnCollisionEnter () {
		touchingPlatform = true;
	}

	void OnCollisionExit () {
		touchingPlatform = false;
	}
}
```

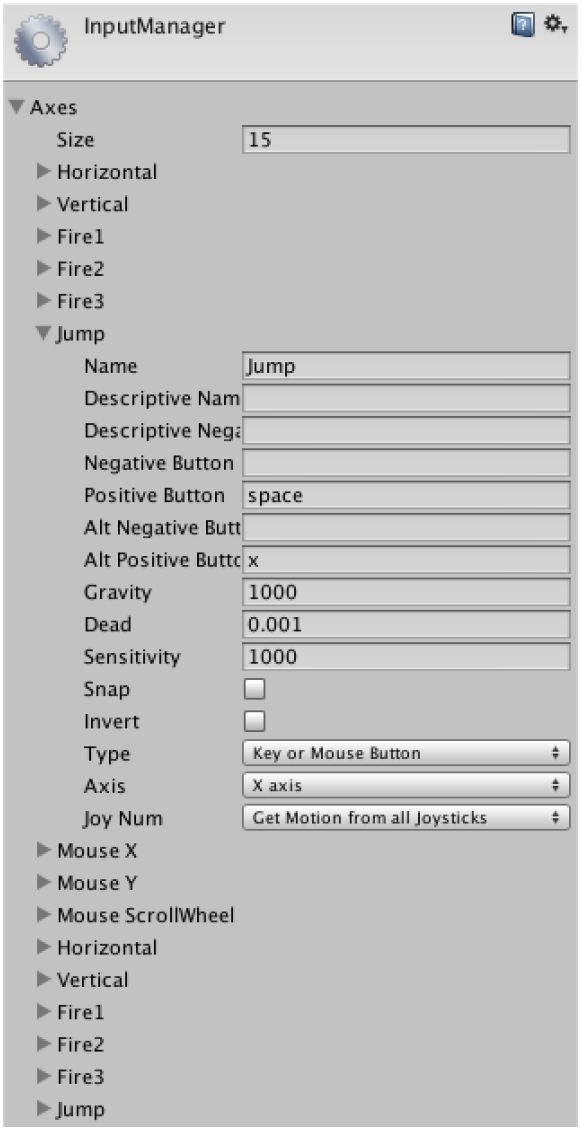
One more thing we need to do before this works is assign a physic material to out platforms. Duplicate *Runner PMat*, rename it to *Platform Regular PMat* and move it to the *Platform* folder. Set both friction fields to 0.05 and drag the physic material onto the *Platform* prefab.

Now our platforms provide a little friction, but *Runner* has a large enough acceleration pick up speed while moving across them.



Acceleration and regular platform.

To make *Runner* jump, we need to detect the player's input. Unity's default settings for the jump action, found under *Edit / Project Settings / Input*, is to be triggered by pressing the space bar. We'll use that, and also configure 'x' as an alternative by putting it in the *Alt Positive Button* fied.



Jump input configuration.

We'll add a variable to `Runner` so we can configure its jump velocity. We'll use a vector instead of just a float so we can profide both a vertical and horizontal component. Set the corresponding field in the editor to (1, 7, 0).

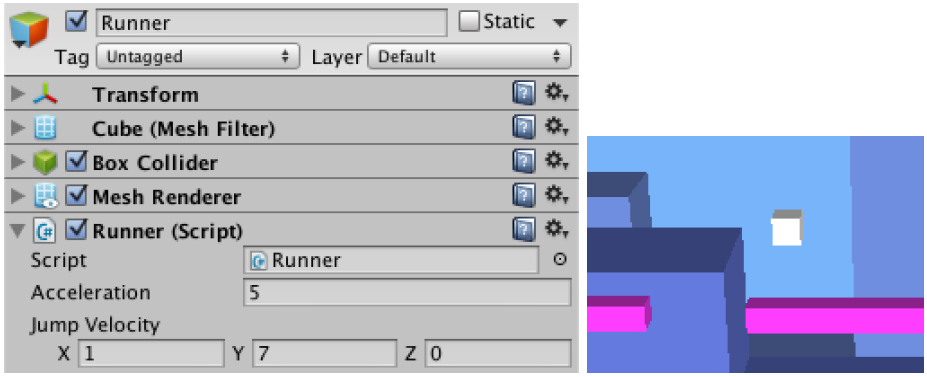
We want *Runner* to jump only when it's touching a platform while the jump button is pressed. Let's add code for this to the `Update` method.

What does `&&` do?

What does `Input.GetButtonDown` do?

Shouldn't the jump be in `FixedUpdate`?

```
public Vector3 jumpVelocity;

void Update () {
	if(touchingPlatform && Input.GetButtonDown("Jump")){
		rigidbody.AddForce(jumpVelocity, ForceMode.VelocityChange);
	}
	distanceTraveled = transform.localPosition.x;
}
```



Runner jumping.

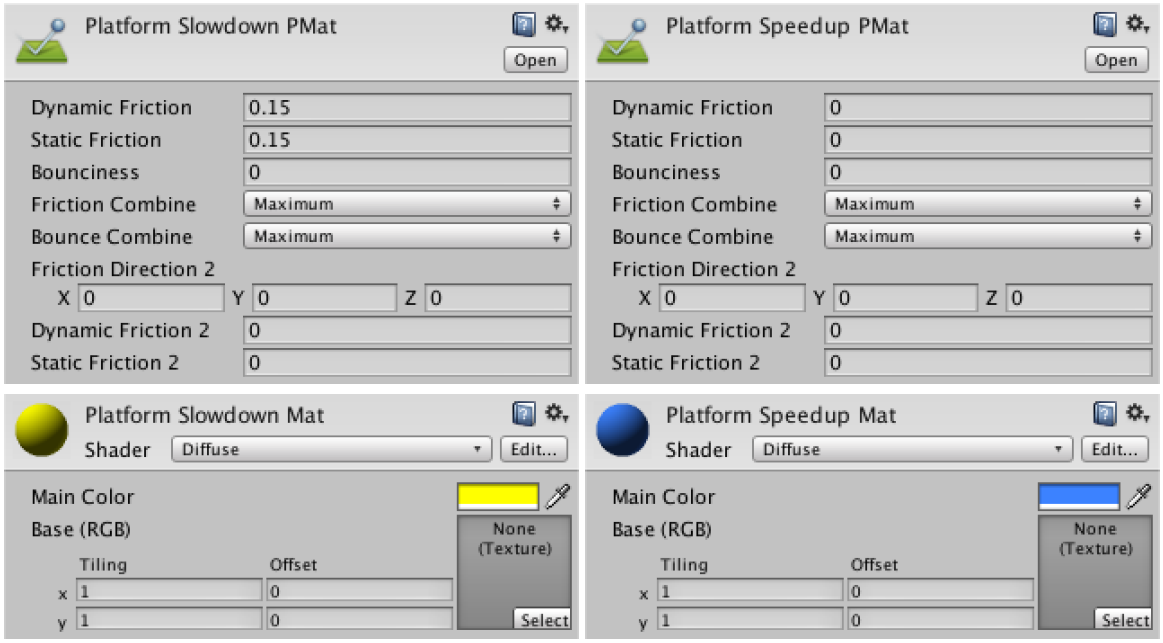
Now we can jump! However, if *Runner* hits a platform from the side, we can perform multiple jumps while it's still touching the platform, launching ourselves out of the gap. To prevent this, we'll decree that once jumped we are no longer touching the platform, even if we really are. This allows for one jump after colliding, which usually isn't enough to escape from a gap.

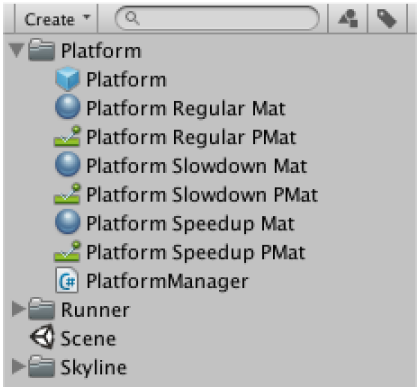
```
	void Update () {
		if(touchingPlatform && Input.GetButtonDown("Jump")){
			rigidbody.AddForce(jumpVelocity, ForceMode.VelocityChange);
			touchingPlatform = false;
		}
		distanceTraveled = transform.localPosition.x;
	}
```

Platform Variety

To spice things up, let's add two new platform types. One slows *Runner* down a bit, while the other speeds it up. We'll accomplish this by adding two physic materials, accompanied by two new colors, and pick which to use per platform at random.

Duplicate *Platform Regular PMat* twice and name them *Platform Slowdown PMat* and *Platform Speedup PMat*. Also duplicate *Platform Regular Mat* twice and name them in a similar fashion. Set the friction values to 0.15 and 0, and their colors to (255, 255, 0) and (60, 130, 255), respectively.



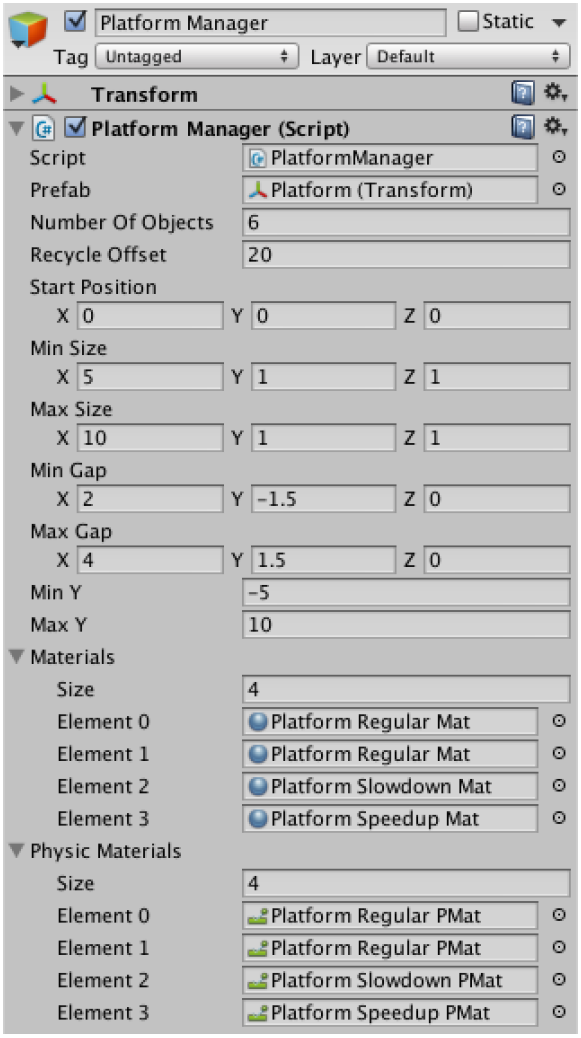


Slowdown and speedup platforms.

We now have to modify `PlatformManager` so it will assign these materials. We'll add two arrays for the materials and pick from them at random when recycling a platform.

```
	public Material[] materials;
	public PhysicMaterial[] physicMaterials;

	private void Recycle () {
		Vector3 scale = new Vector3(
			Random.Range(minSize.x, maxSize.x),
			Random.Range(minSize.y, maxSize.y),
			Random.Range(minSize.z, maxSize.z));

		Vector3 position = nextPosition;
		position.x += scale.x * 0.5f;
		position.y += scale.y * 0.5f;

		Transform o = objectQueue.Dequeue();
		o.localScale = scale;
		o.localPosition = position;
		int materialIndex = Random.Range(0, materials.Length);
		o.renderer.material = materials[materialIndex];
		o.collider.material = physicMaterials[materialIndex];
		objectQueue.Enqueue(o);

		nextPosition += new Vector3(
			Random.Range(minGap.x, maxGap.x) + scale.x,
			Random.Range(minGap.y, maxGap.y),
			Random.Range(minGap.z, maxGap.z));

		if(nextPosition.y < minY){
			nextPosition.y = minY + maxGap.y;
		}
		else if(nextPosition.y > maxY){
			nextPosition.y = maxY - maxGap.y;
		}
	}
}
```

Now it's a matter of assigning stuff to the arrays, either by dragging or by setting the array's size and using the dots. Make sure that both arrays are ordered the same way. Furthermore, if we'd like some platform types to be more common than others, simply include them multiple times. By including the regular materials twice and the others just once, the regular option has a 50% chance of occurring, while the others have a 25% chance each.



Platform variety.

Game Events

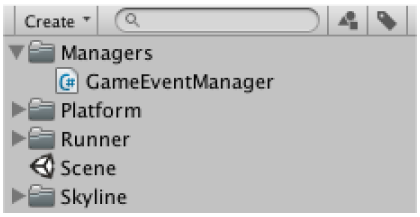
Right now, the game starts as soon as we enter play mode and it doesn't end at all. What we want instead is that the game begins with a title screen and ends with a game over notice, where pressing one of the jump buttons starts a new game.

For this approach we can identify three events that might require objects to take action. The first, game launch, is effectively handled by the `Start` methods. The other two, game start and game over, require a custom approach. We will create a very simple event manager class to handle them.

Create a new folder named *Managers* and put a new C# script named *GameEventManager* in it. We make `GameEventManager` a static class that defines a `GameEvent` delegate type inside it. Note that the manager isn't a `MonoBehaviour` and won't be attached to any Unity object.

Why is the class static?

What's a **delegate**?



Game event manager.

```
public static class GameEventManager {

	public delegate void GameEvent();
}
```

Next, we use the new `gameEvent` type to add two events to our manager, `GameStart` and `GameEnd`. Now other scripts can subscribe to these events by assigning methods to them, which will be called when the events are triggered.

What's an **event**?

```
public static class GameEventManager {

	public delegate void GameEvent();

	public static event GameEvent GameStart, GameOver;
}
```

Finally, we need to include a way to trigger these events. We'll add two methods for this. Care should be taken to only call an event if anyone is subscribed to it, otherwise it will be **null** and the call will result in an error.

What's **null**?

What does != do?

```
public static class GameEventManager {

	public delegate void GameEvent();

	public static event GameEvent GameStart, GameOver;

	public static void TriggerGameStart(){
		if(GameStart != null){
			GameStart();
		}
	}

	public static void TriggerGameOver(){
		if(GameOver != null){
			GameOver();
		}
	}
}
```

GUI and Game Start

Now that we have a game start event, let's create a GUI and a manager that uses it.

Let's add some text labels to our scene. To keep things organized, we'll use a container object to group them, so create a new empty game object with position (0, 0, 0) and name it *GUI*. Create three empty child objects for it and give each a *GUIText* component via

Component / Rendering / GUIText. Set their *Anchor* fields to *middle center* so their text gets centered on their position.

Name the first object *Game Over Text*, set its *Text* field to "GAME OVER", set its *Font Size* to 40, and set its *Font Style* to bold. Change its position to (0.5, 0.2, 0) so it ends up near the bottom center of the screen.

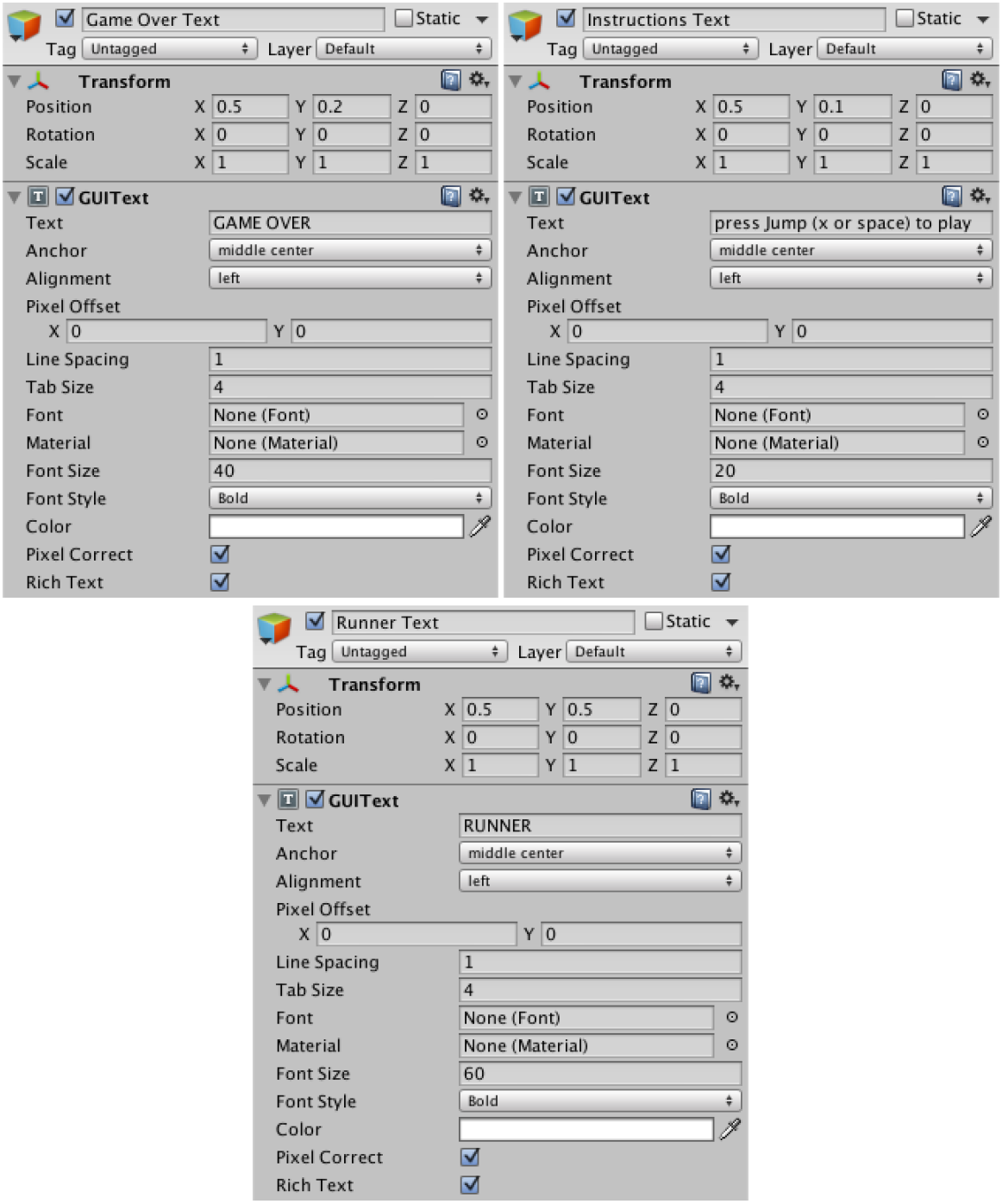
Name the second object *Instructions Text*, also bold but with a font size of 20, and set its text to "press Jump (x or space) to play". Change its position to (0.5, 0.1, 0), just below the game over text.

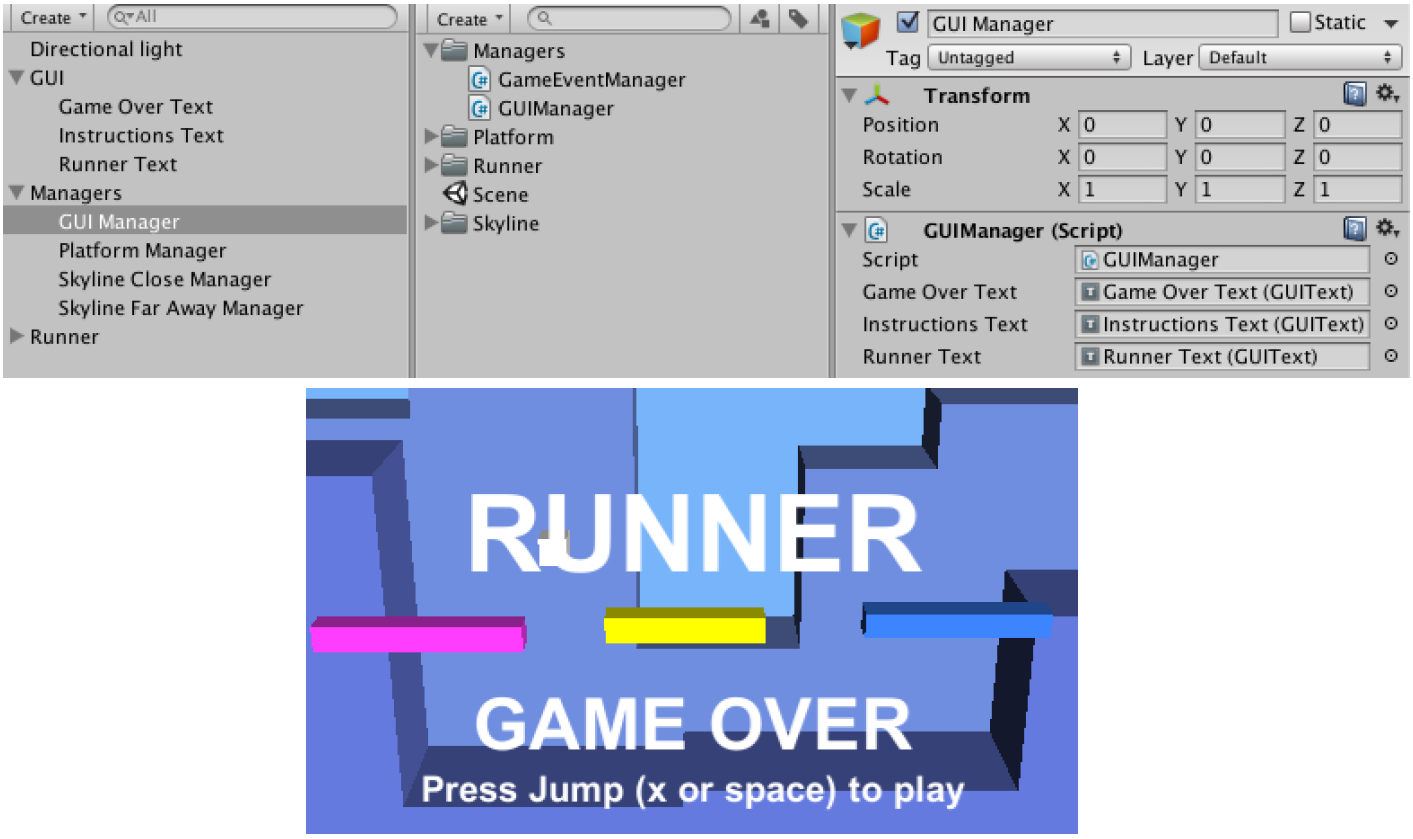
Name the third object *Runner Text*, with text "RUNNER", bold, and a font size of 60. It's position should be (0.5, 0.5, 0), right in the middle of the screen.

Now create a C# script named *GUIManager* in the *Managers* folder and give it a `GUIText` variable for each text object we just made. Create a new object named *GUI Manager* and assign the script as a component. Make it a child of *Managers*. Then assign the text objects to the manager's corresponding fields.

What's with the text positions?

```
using UnityEngine;

public class GUIManager : MonoBehaviour {

	public GUIText gameOverText, instructionsText, runnerText;
}
```


Game Over Text
Static
Tag Untagged
Layer Default
Transform
Position X 0.5 Y 0.2 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
GUIText
Text GAME OVER
Anchor middle center
Alignment left
Pixel Offset X 0 Y 0
Line Spacing 1
Tab Size 4
Font None (Font)
Material None (Material)
Font Size 40
Font Style Bold
Color
Pixel Correct
Rich Text
Instructions Text
Static
Tag Untagged
Layer Default
Transform
Position X 0.5 Y 0.1 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
GUIText
Text press Jump (x or space) to play
Anchor middle center
Alignment left
Pixel Offset X 0 Y 0
Line Spacing 1
Tab Size 4
Font None (Font)
Material None (Material)
Font Size 20
Font Style Bold
Color
Pixel Correct
Rich Text
Runner Text
Static
Tag Untagged
Layer Default
Transform
Position X 0.5 Y 0.5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
GUIText
Text RUNNER
Anchor middle center
Alignment left
Pixel Offset X 0 Y 0
Line Spacing 1
Tab Size 4
Font None (Font)
Material None (Material)
Font Size 60
Font Style Bold
Color
Pixel Correct
Rich Text



GUI Text.

Now add a `Start` method to our new manager and use it to disable `gameOverText` so it won't be shown anymore. Also add an `Update` method that checks whether a jump button was pressed, and if so triggers the game-start event.

```
	void Start () {
		gameOverText.enabled = false;
	}

	void Update () {
		if(Input.GetButtonDown("Jump")){
			GameEventManager.TriggerGameStart();
		}
	}
```

Now it's time to include a method to handle our game-start event, let's appropriately name it `GameStart`. We use this method to disable all text. We also disable the manager itself, so its `Update` method will no longer be called. If we didn't, each time we jump there'd be a new game-start event.

Why trigger and handle the same event?

```
	private void GameStart () {
		gameOverText.enabled = false;
		instructionsText.enabled = false;
		runnerText.enabled = false;
		enabled = false;
	}
```

The last step is informing our event manager that it should call the `GameStart` method of our manager object, whenever the game-start event is triggered. We do this by adding our method to the event in the `Start` method.

```
	void Start () {
		GameEventManager.GameStart += GameStart;
		gameOverText.enabled = false;
	}
```

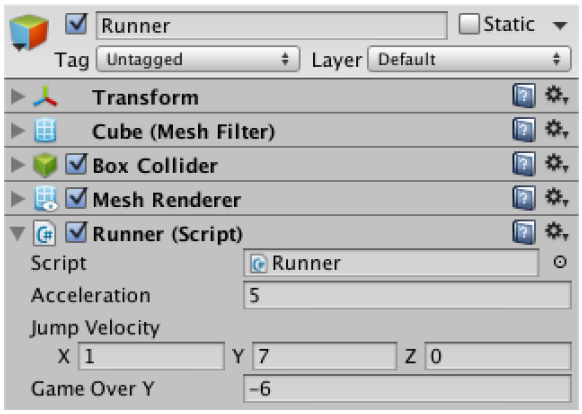
Game Over

Let's also add a handler for the game-over event to our gui manager. We go about the same way as for the game start event, but in this case we need to enable the manager again, along with the instructions and game over text.

```
	void Start () {
		GameEventManager.GameStart += GameStart;
		GameEventManager.GameOver += GameOver;
		gameOverText.enabled = false;
	}

	private void GameOver () {
		gameOverText.enabled = true;
		instructionsText.enabled = true;
		enabled = true;
	}
```

The game over event should be triggered whenever *Runner* falls below the platforms. We'll simply add a *Game Over Y* field to `Runner` with a value of -6, then check each update whether we dropped below it. If so, we trigger the game over event.

```
	public float gameOverY;

	void Update () {
		if(touchingPlatform && Input.GetButtonDown("Jump")){
			rigidbody.AddForce(jumpVelocity, ForceMode.VelocityChange);
			touchingPlatform = false;
		}
		distanceTraveled = transform.localPosition.x;

		if(transform.localPosition.y < gameOverY){
			GameEventManager.TriggerGameOver();
		}
	}
```



Game over threshold.

Using the Events

Now that our game events are triggered correctly, it's time for *Runner* to take them into account.

We want `Runner` to be disabled before the first game is started, though we want the camera inside of it to stay active. Disabling the runner means we have to deactivate its renderer and the runner component itself. We also switch its rigidbody to kinematic mode to freeze it in place. We

can do this in its `Start` method, then undo this change when the game-start event is triggered, and then redo it when the game-over event is triggered. We'll also remember its starting position so we can reset it each game start. Let's reset `distanceTraveled` too, so it's immediately up to date.

Why immediately reset the distance?

What does it mean to be kinematic?

```
	private Vector3 startPosition;

	void Start () {
		GameEventManager.GameStart += GameStart;
		GameEventManager.GameOver += GameOver;
		startPosition = transform.localPosition;
		renderer.enabled = false;
		rigidbody.isKinematic = true;
		enabled = false;
	}

	private void GameStart () {
		distanceTraveled = 0f;
		transform.localPosition = startPosition;
		renderer.enabled = true;
		rigidbody.isKinematic = false;
		enabled = true;
	}

	private void GameOver () {
		renderer.enabled = false;
		rigidbody.isKinematic = true;
		enabled = false;
	}
```

Now *Runner* reacts properly to the game events, but once the first platform has been recycled all new games will be over rather quickly, as *Runner* immediately plummets. Let's modify our platform manager so it reacts to the events as well. It should only be enabled when a game is in progress and the platforms should only become visible after the first game start event has been triggered.

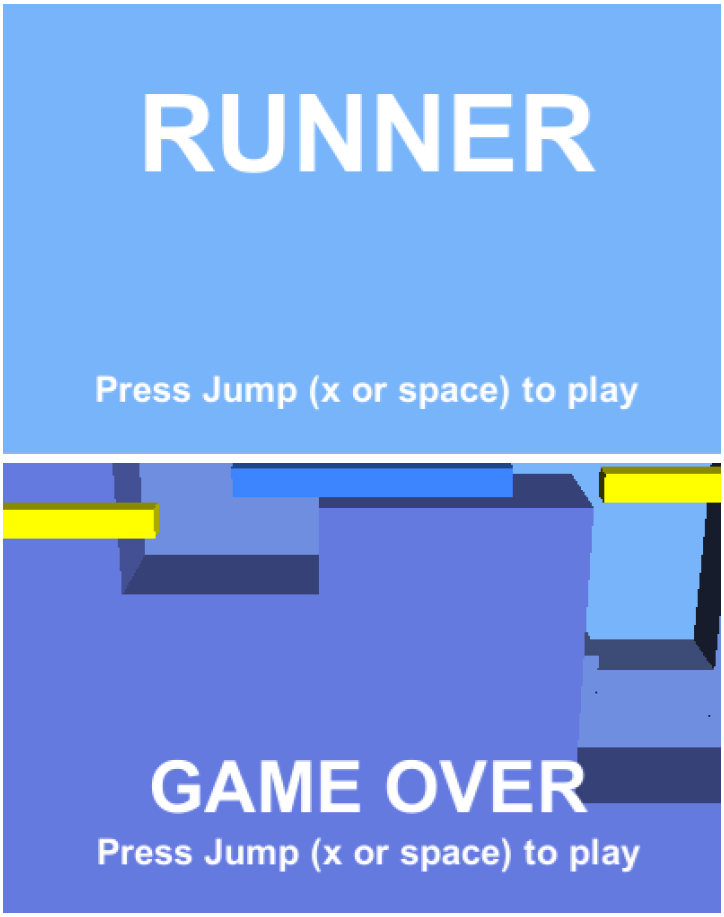
We can achieve this by having `PlatformManager` initially place the platforms somewhere far behind the camera and relocating its recycle loop to a new `GameStart` method.

What's `Quaternion.identity`?

```
	void Start () {
		GameEventManager.GameStart += GameStart;
		GameEventManager.GameOver += GameOver;
		objectQueue = new Queue<Transform>(numberOfObjects);
		for (int i = 0; i < numberOfObjects; i++) {
			objectQueue.Enqueue((Transform)Instantiate(
				prefab, new Vector3(0f, 0f, -100f), Quaternion.identity));
		}
		enabled = false;
	}

	private void GameStart () {
		nextPosition = startPosition;
		for(int i = 0; i < numberOfObjects; i++){
			Recycle();
		}
		enabled = true;
	}

	private void GameOver () {
		enabled = false;
	}
```

Now give `SkylineManager` the exact same treatment, so all parts of the game respond nicely to our events.

```
void Start () {
	GameEventManager.GameStart += GameStart;
	GameEventManager.GameOver += GameOver;
	objectQueue = new Queue<Transform>(numberOfObjects);
	for(int i = 0; i < numberOfObjects; i++){
		objectQueue.Enqueue((Transform)Instantiate(
			prefab, new Vector3(0f, 0f, -100f), Quaternion.identity));
	}
	enabled = false;
}

private void GameStart () {
	nextPosition = startPosition;
	for(int i = 0; i < numberOfObjects; i++){
		Recycle();
	}
	enabled = true;
}

private void GameOver () {
	enabled = false;
}
```



Game start and game over.

Power-Up

Let's include a power-up that allows for mid-air boosts. We'll make it a spinning cube that appears above platforms at random. We decide to have at most one such booster cube active in the scene at any moment, so we can suffice with one instance and reuse it.

Create a new folder named *Booster*. In it, create a new material named *Booster Mat*. Because it's spinning, we'll use the *Specular* shader for the material, giving it a green (0, 255, 0) color and a

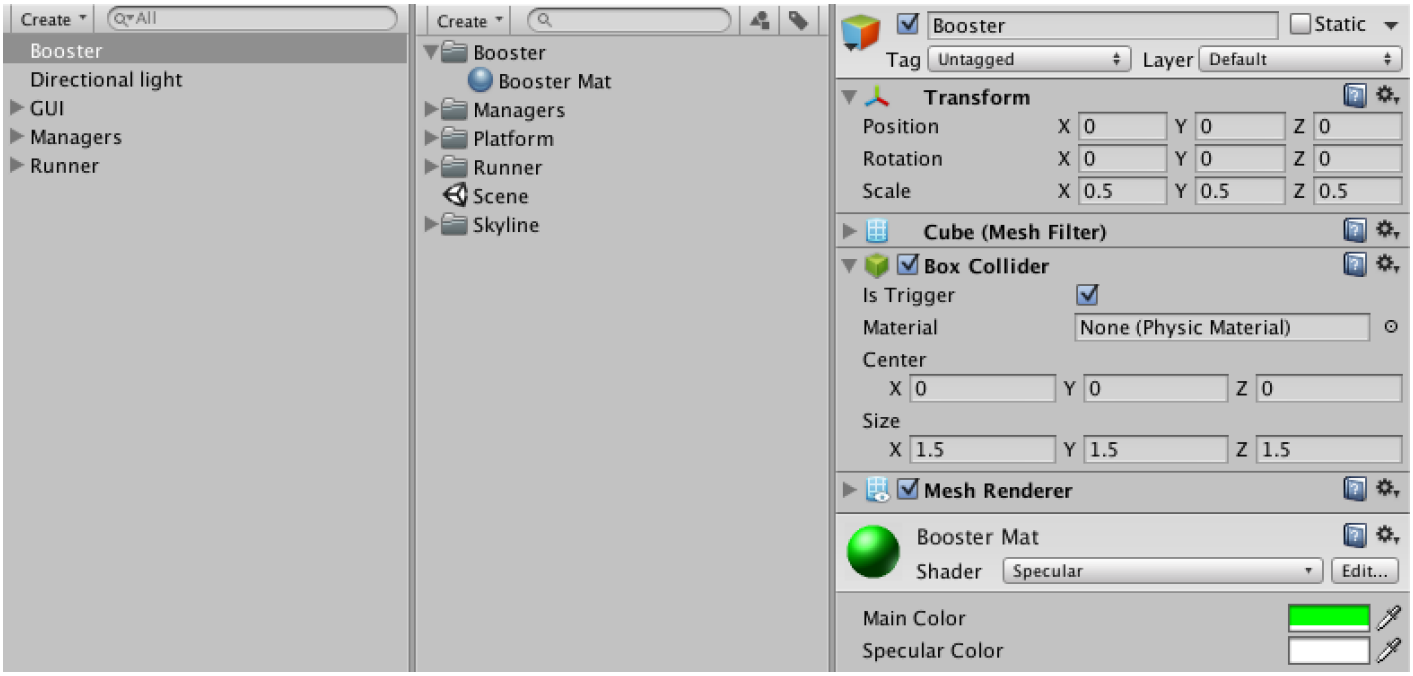
white specular color.

Now create a new cube, name is *Booster*, and set its scale to 0.5 to make it small. To make it a bit easier to hit, increase its collider's size to 1.5, which ends up being 0.75 due to the scale. Then assign its material to it.

Mark the collider as a trigger, by checking its *Is Trigger* field. We do this because we want *Runner* to pass right through it, instead of colliding.

Why a specular shader?

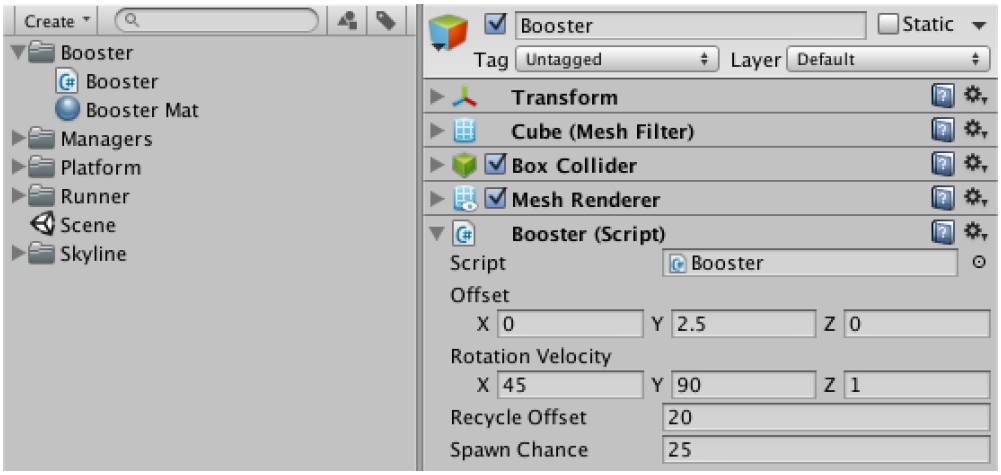
What does it mean to be a trigger?



Booster cube.

Now create a new C# script named *Booster* in the corresponding folder and assign it to the *Booster* object. We start by giving it four public variables used to configure it. First, we need an offset from the platform's center to place the booster. Let's set it to (0, 2.5, 0). Second, we need a rotation velocity to make it spin. Let's use (45, 90, 1) to make it a bit lively. Third, we need a recycle offset, just as for platforms, in case *Runner* misses the power-up. Let's use a distance of 20. Fourth, we include a spawn chance to make the appearance of the booster somewhat unpredictable. A 25% chance per platform is fine.

```
using UnityEngine;

public class Booster : MonoBehaviour {

	public Vector3 offset, rotationVelocity;
	public float recycleOffset, spawnChance;
}
```



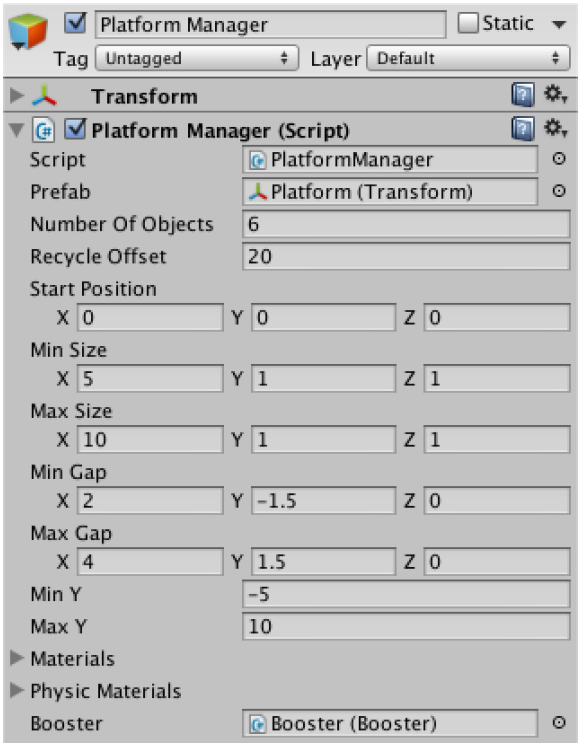
Booster configuration.

One way to make the spawning work is by requesting a booster placement each time a platform is recycled. Then it's up to the booster itself whether it'll be placed. We'll add a method named `SpawnIfAvailable` to `Booster` for this. It requires a platform position so we know where to place the booster. We leave it empty for now.

```
	public void SpawnIfAvailable(Vector3 position){
	}
```

We then add a variable to `PlatformManager` to which we assign *Booster*. Inside the `Recycle` method, we'll call its `PlaceIfAvailable` method after we've determined the new platform's position.

```
	public Booster booster;

	private void Recycle () {
		Vector3 scale = new Vector3(
			Random.Range(minSize.x, maxSize.x),
			Random.Range(minSize.y, maxSize.y),
			Random.Range(minSize.z, maxSize.z));

		Vector3 position = nextPosition;
		position.x += scale.x * 0.5f;
		position.y += scale.y * 0.5f;
		booster.SpawnIfAvailable(position);

		Transform o = objectQueue.Dequeue();
		o.localScale = scale;
		o.localPosition = position;
		int materialIndex = Random.Range(0, materials.Length);
		o.renderer.material = materials[materialIndex];
		o.collider.material = physicMaterials[materialIndex];
		objectQueue.Enqueue(o);

		nextPosition += new Vector3(
			Random.Range(minGap.x, maxGap.x) + scale.x,
			Random.Range(minGap.y, maxGap.y),
			Random.Range(minGap.z, maxGap.z));

		if(nextPosition.y < minY){
			nextPosition.y = minY + maxGap.y;
		}
		else if(nextPosition.y > maxY){
			nextPosition.y = maxY - maxGap.y;
		}
	}
```



Platform manager knows about booster.

Now that everything is connected, we need to update the `SpawnIfAvailable` method so it activates and positions the booster, but only if it's not already active, and also taking the spawn chance into account. Also, to make this work `Booster` must begin deactivated and must also deactivate when the game ends.

What does `setActive()` do?

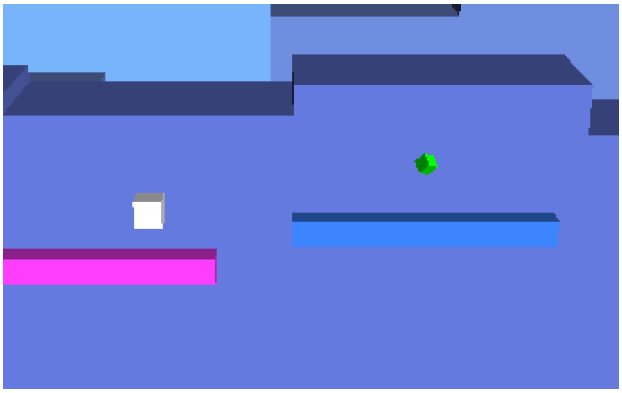
What does `return` do?

```
void Start () {
	GameEventManager.GameOver += GameOver;
	gameObject.SetActive(false);
}

public void SpawnIfAvailable (Vector3 position) {
	if(gameObject.activeSelf || spawnChance <= Random.Range(0f, 100f)) {
		return;
	}
	transform.localPosition = position + offset;
	gameObject.SetActive(true);
}

private void GameOver () {
	gameObject.SetActive(false);
}
```

To make the booster spin and recycle, we have to add an `Update` method to it. Recycling is achieved by simple deactivation, as that makes it eligible for a respawn via `SpawnIfAvailable`. Rotation is achieved by rotating based on the elapsed time since the last frame.

What's `Time.deltaTime`?

```
void Update () {
	if(transform.localPosition.x + recycleOffset < Runner.distanceTraveled){
		gameObject.SetActive(false);
		return;
	}
	transform.Rotate(rotationVelocity * Time.deltaTime);
}
```



Rotating booster.

At the moment *Runner* passed right through *Booster*, nothing happened. To change this, we add the Unity event method `OnTriggerEnter` to `Booster`, which is called whenever something hits its trigger collider. Because we know that the only thing that could possibly hit the booster is our runner, we can go ahead and give it a new booster power-up whenever there's a trigger. Let's assume `Runner` has a static method named `AddBoost` for this purpose, and use that. We also deactivate the booster, because it's been consumed.

```
	void OnTriggerEnter () {
		Runner.AddBoost();
		gameObject.SetActive(false);
	}
```

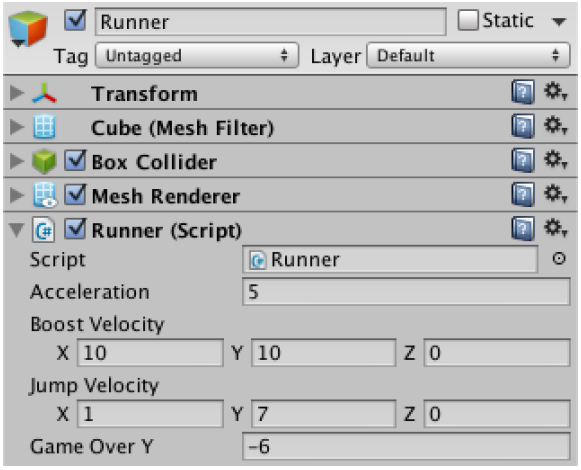
To make this work, we have to add an `AddBoost` method to `Runner`. To keep things simple, let's just add a private static variable to remember how many boosts we have accumulated.

```
	private static int boosts;

	private void GameStart () {
		boosts = 0;
		distanceTraveled = 0f;
		transform.localPosition = startPosition;
		renderer.enabled = true;
		rigidbody.isKinematic = false;
		enabled = true;
	}

	public static void AddBoost () {
		boosts += 1;
	}
```

To actually allow mid-air jumps by consuming boosts, we need to modify the code that checks whether a jump is possible. Let's define a seperate boost velocity as well and set it to (10, 10, 0) for a nice boost.

```
	public Vector3 boostVelocity, jumpVelocity;

	void Update () {
		if(Input.GetButtonDown("Jump")){
			if(touchingPlatform){
				rigidbody.AddForce(jumpVelocity, ForceMode.VelocityChange);
				touchingPlatform = false;
			}
			else if(boosts > 0){
				rigidbody.AddForce(boostVelocity, ForceMode.VelocityChange);
				boosts -= 1;
			}
		}
		distanceTraveled = transform.localPosition.x;

		if(transform.localPosition.y < gameOverY){
			GameEventManager.TriggerGameOver();
```

```
		}
	}
```



Boost velocity.

Informative GUI

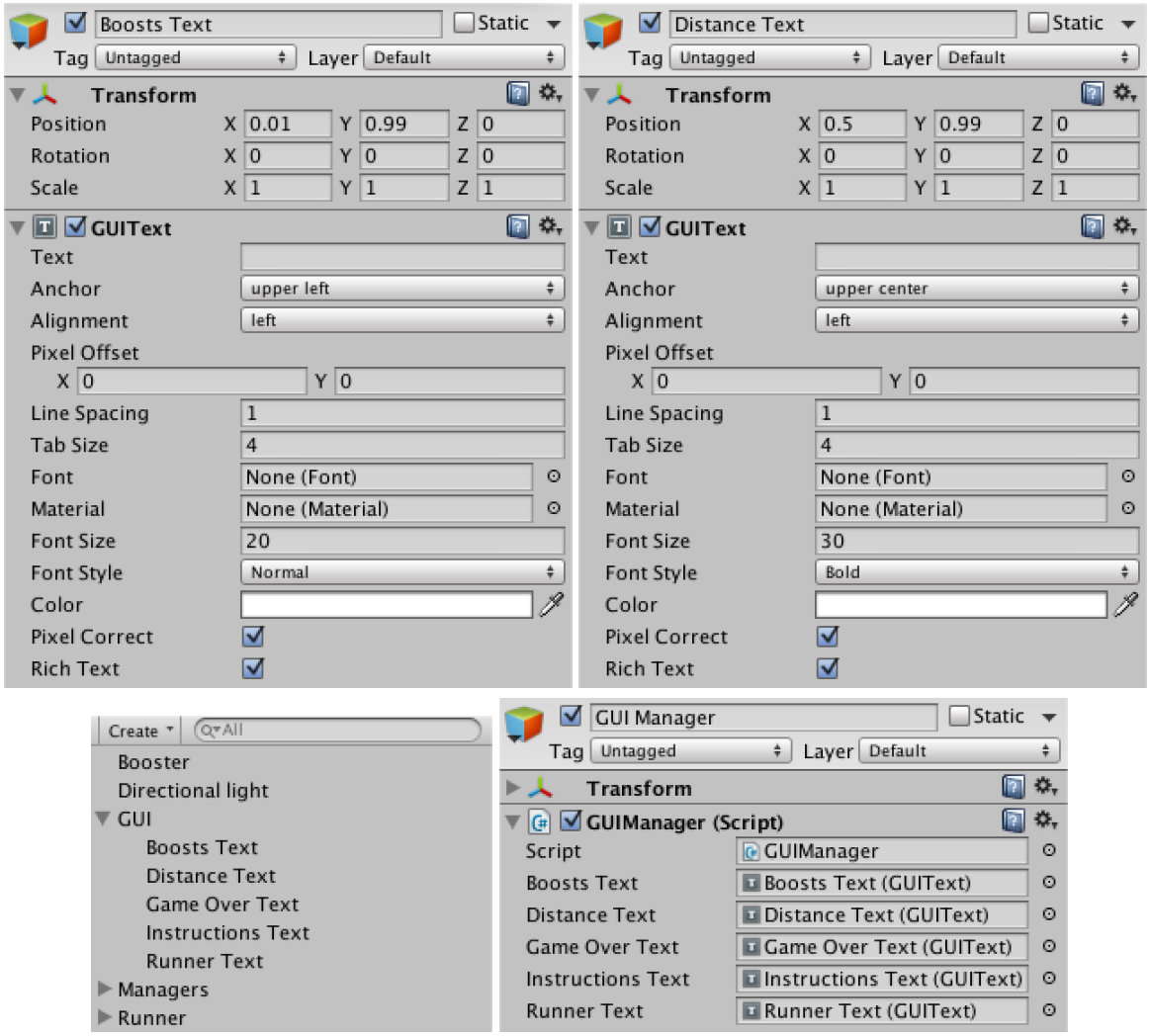
It works! As long as we have boosts remaining, *Runner* can boost itself while in flight. It would be useful to actually see how many boost are available, so let's add a display for it to the GUI. While we're at it, let's show the distance traveled so far as well.

Create a new object with a *GUIText* component as a child of *GUI*. Position it at (0.01, 0.99, 0), set its *Anchor* to upper left, give it font size 20 and a normal style. Name it *Boosts Text*.

Create another such object, naming it *Distance Text*. Set its position to (0.5, 0.99, 0), with font size 30 and bold style. Its *Anchor* should be set to upper center.

Add two variables to `GUIManager` for these new objects and assign them.

```
	public GUIText boostsText, distanceText, gameOverText, instructionsText, runnerText;
```



Boosts and distance.

Let's add two static methods to `GUIManager` which `Runner` can use to notify the GUI of changes to its distance traveled and boost count. Because the manager needs to use nonstatic variables in those methods, we add a static variable that references itself. That way the static code can get to the component instance which actually has the gui text elements.

What's `this`?

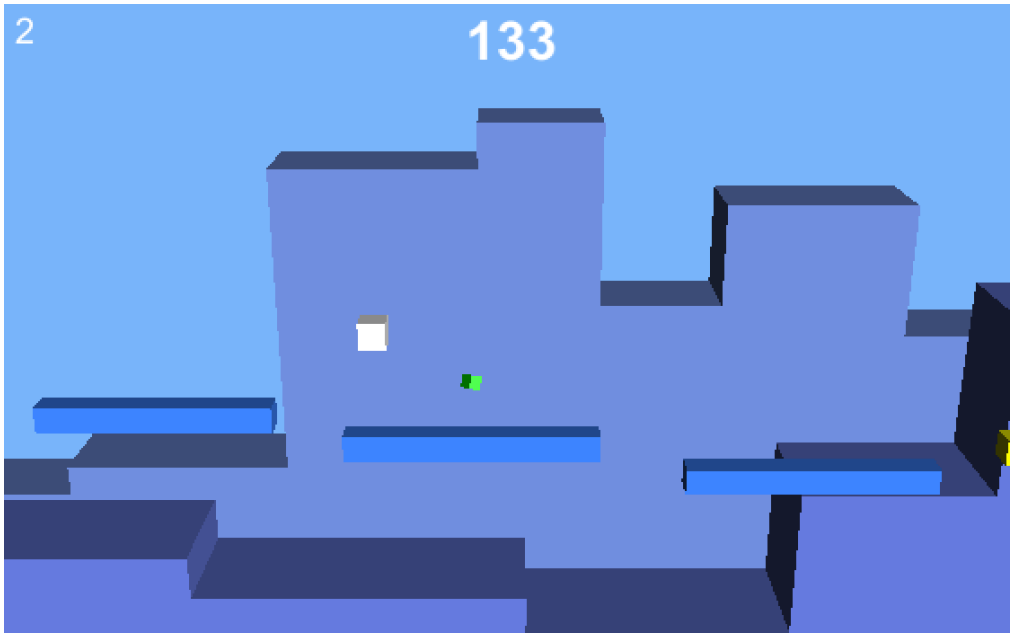
What does `ToString()` do?

```
private static GUIManager instance;

void Start () {
	instance = this;
	GameEventManager.GameStart += GameStart;
	GameEventManager.GameOver += GameOver;
	gameOverText.enabled = false;
}

public static void SetBoosts(int boosts){
	instance.boostsText.text = boosts.ToString();
}

public static void SetDistance(float distance){
	instance.distanceText.text = distance.ToString("f0");
}
```

Now all we need to do is let `Runner` call those methods whenever its distance or amount of boosts changes.

Why not set the labels from `Runner`?

```
	void Update () {
		if(Input.GetButtonDown("Jump")){
			if(touchingPlatform){
				rigidbody.AddForce(jumpVelocity, ForceMode.VelocityChange);
				touchingPlatform = false;
			}
			else if(boosts > 0){
				rigidbody.AddForce(boostVelocity, ForceMode.VelocityChange);
				boosts -= 1;
				GUIManager.SetBoosts(boosts);
			}
		}
		distanceTraveled = transform.localPosition.x;
		GUIManager.SetDistance(distanceTraveled);

		if(transform.localPosition.y < gameOverY){
			GameEventManager.TriggerGameOver();
		}
	}

	private void GameStart () {
		boosts = 0;
		GUIManager.SetBoosts(boosts);
		distanceTraveled = 0f;
		GUIManager.SetDistance(distanceTraveled);
		transform.localPosition = startPosition;
		renderer.enabled = true;
		rigidbody.isKinematic = false;
		enabled = true;
	}

	public static void AddBoost(){
		boosts += 1;
		GUIManager.SetBoosts(boosts);
	}
```



Complete GUI.

Particle Effects

By now we have a functional game, but it feels a bit empty. Let's add some dust particles to fill the empty space and enhance the sense of depth and speed.

Create a new a new particle system (*GameObject / Create Other / Particle System*) named *Dust Emitter*. Make it a child of *Runner* with a position of (25, 0, 0) and reset its rotation, so it'll always stay to the right of the camera view.

Set *Start Lifetime* to *Random Between Two Constants* with values 6 and 10, and set *Start Speed* to 0 so we get stationary particles with varied lifetimes to start with. Also set *Simulation Space* to *World* so the particles don't move with *Runner*. To increase variety, set *Start Size* to *Random Between Two Constants* with values 0.2 and 0.8.

Change the shape to a box with dimensions (1, 30, 10) so we get a large spawning area, and increase the *Rate* of *Emission* to 20.

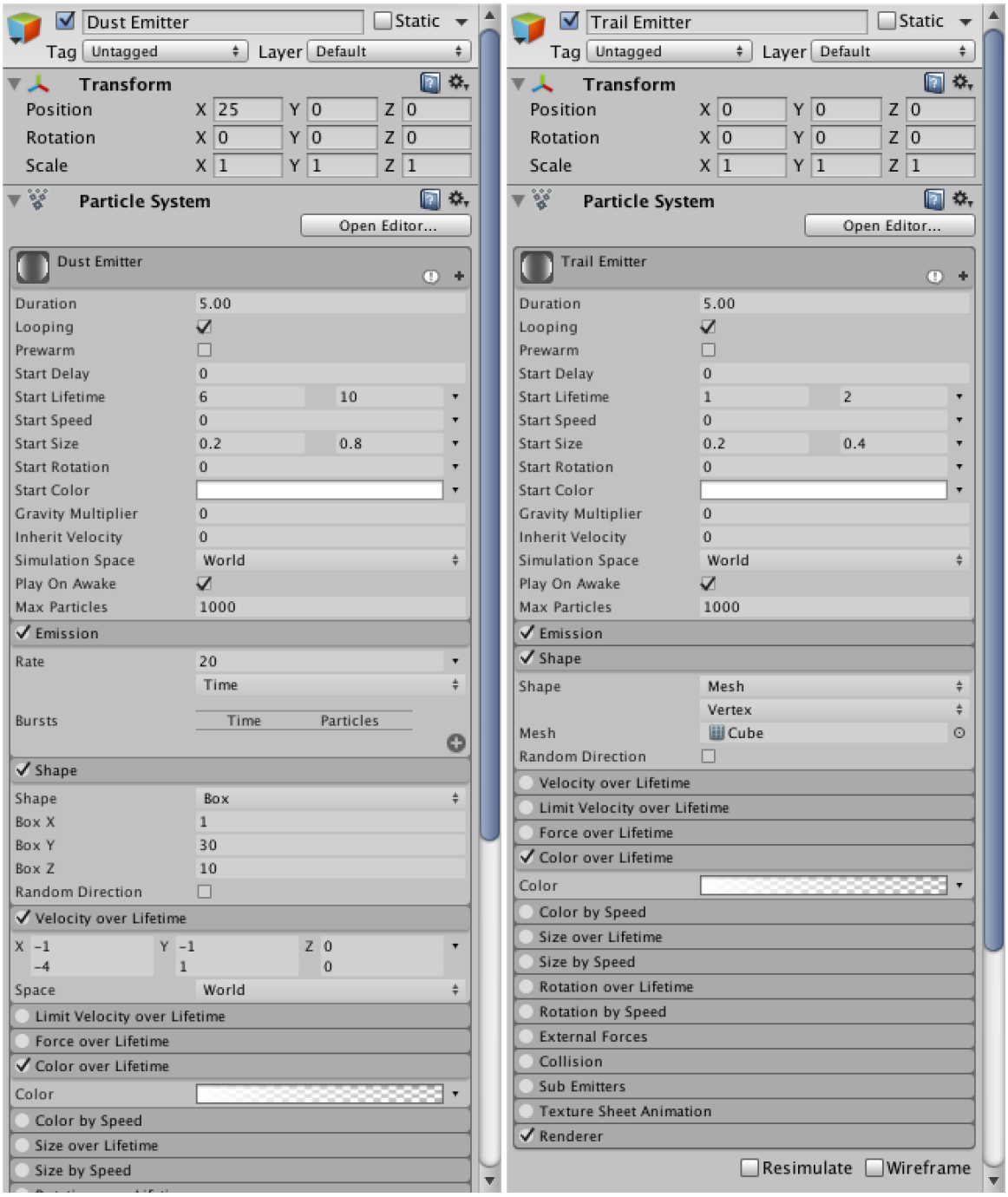
Activate *Velocity Over Lifetime*, set it to use world space and a random range between two constants, using the vectors (-1, -1, 0) and (-4, 1, 0). This way the particles have some individual movement.

Finally activate *Color over Lifetime* and change to gradient so it has an alpha value of 0 at 100%. This adds some fading to the particles.

Next, duplicate this particle system, keep it a child of *Runner*, reset its position, and name it *Trail Emitter*. We'll use this one for a condensation trail effect left behind by *Runner*.

Change its *Shape* to *Mesh* and set it to a cube (by clicking on the dot), and deactivate *Velocity over Lifetime*.

Decrease *Start Lifetime* to between 1 and 2 and *Start Size* to between 0.2 and 0.4, to keep the trail subtle and short.



Particle systems.

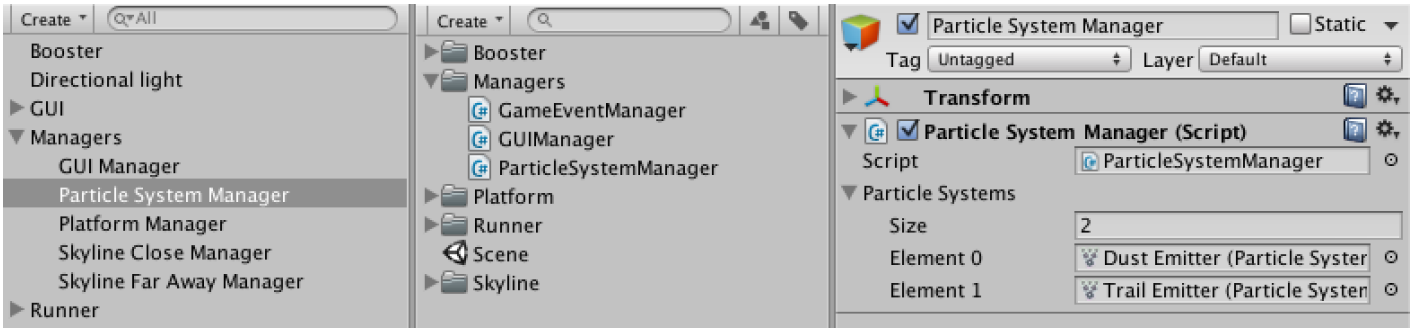
As we only want to spawn particles when a game is in progress, we'll create a manager for them. Add a new C# script in the *Managers* folder and name it *ParticleSystemManager*. Also create an appropriately named child object for *Managers* and assign the manager script to it.

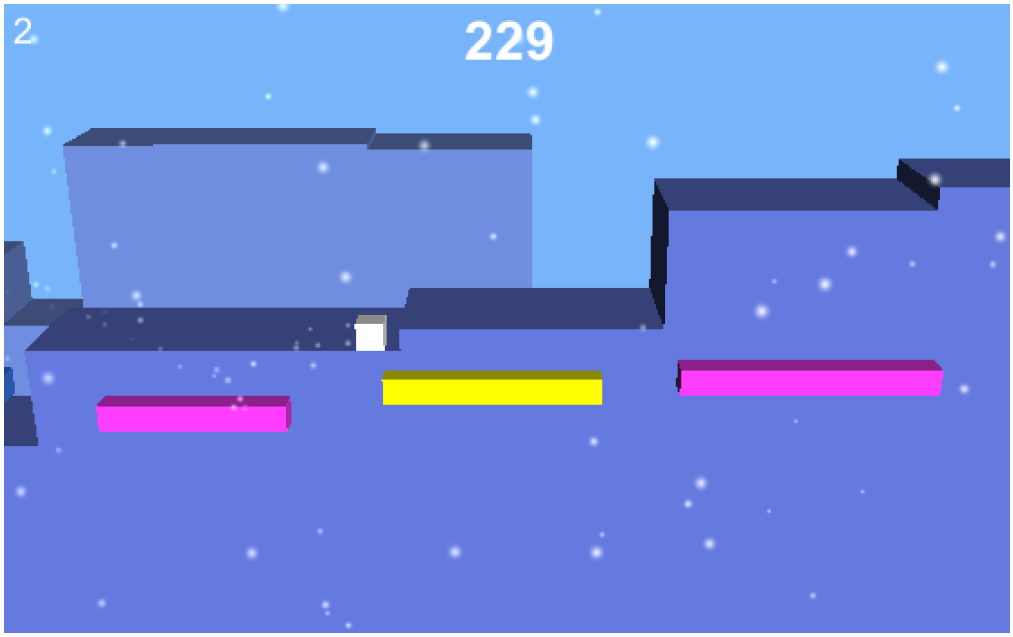
The only thing that `ParticleSystemManager` has to do is switch the particle systems on and off at the appropriate time. We'll use an array variable named `particleSystems` to hold references to all emitters that need to be managed. In this case, that's the two emitters we just created, but the manager can deal with any additional emitters you'd like to create.

Assign our two particle emitters by dragging them to the *Particle Systems* field.

Why call `GameOver` in `Start`?

```
using UnityEngine;

public class ParticleSystemManager : MonoBehaviour {

	public ParticleSystem[] particleSystems;

	void Start () {
		GameEventManager.GameStart += GameStart;
		GameEventManager.GameOver += GameOver;
		GameOver();
	}

	private void GameStart () {
		for(int i = 0; i < particleSystems.Length; i++){
			particleSystems[i].Clear();
			particleSystems[i].enableEmission = true;
		}
	}

	private void GameOver () {
		for(int i = 0; i < particleSystems.Length; i++){
			particleSystems[i].enableEmission = false;
		}
	}
}
```





Particles in action.

That's it, we've finished the game! We can run and jump, leave a trail, collect power-ups, see our score, and have a scrolling background. It's a nice prototype that, with a lot of polish, you can transform into a finished game.

Enjoyed the tutorial? Help me make more by becoming a patron!

Downloads

runner_01.unitypackage
The project at the start of Generating a Skyline.

runner_02.unitypackage
The project at the start of Generating Platforms.

runner_03.unitypackage
The project at the start of Jumping and Falling.

runner_04.unitypackage
The project at the start of Platform Variety.

runner_05.unitypackage
The project at the start of Game Events.

runner_06.unitypackage
The project at the start of Power-Up.

runner_07.unitypackage
The project at the start of Informative GUI.

runner_08.unitypackage
The project at the start of Particle Effects.

runner.unitypackage
The finished project.

Questions & Answers

What's a collider?

A collider is a physics concept used by Unity's physics engine. They can function as either obstacles, causing collisions, or detectors, triggering events.

Why remove the colliders?

Because the cubes will be used for the noninteractive background only, nothing will ever collide with then. There's no point in the physics engine keeping track of their colliders, so we simply remove them.

What's a material?

Materials are used to define the visual properties of objects. They can range from very simple, like a constant color, to very complex.

Materials consist of a shader and whatever data the shader needs. Shaders are basically scripts that tell the graphics card how an object's polygons should be drawn.

The standard diffuse shader uses a single color and optionally a texture, along with the light sources in the scene, to determine the appearance of polygons.

What about the fourth color component?

Although colors have four components, I'm mentioning only three. The fourth component is the alpha value, which represents the opacity of the color. I assume this value is 255 by default, though it doesn't really matter as we won't create any materials that take alpha into account.

What does it mean to be a child?

Beyond the obvious effect on the object hierarchy, being a child means that you are subject to the *Transform* component of your parent. Your own transformation is relative to your parent's. When it moves, so do you. When it rotates, you orbit around its pivot. When it scales, both your size and your relative position scale as well.

From a low-level graphics point of view, the hierarchy corresponds to how the transformation matrix stack is created. The parent's matrix is pushed first, then the child's matrix.

What's a prefab?

A prefab is a Unity object – or hierarchy of objects – that doesn't exist in the scene and hasn't been activated. You use it as a template, creating clones of it and adding those to the scene.

What does `Instantiate` do?

The **`Object`** class, which every **`MonoBehaviour`** inherits from, contains the static `Instantiate` method. This method creates a clone of whatever **`Object`** instance you pass to it. Optionally, you can supply a new position and rotation for the clone, otherwise it keeps the values of the original.

Note that `Instantiate` returns an **`Object`** reference. If you want to do something with the new clone, you have to cast it to the correct type.

Typically, this method is used with prefabs, but you can also clone objects that already exist in the scene.

What does += do?

The code `x += y;` adds x and y together and assigns the result back to x. You can consider it a short alternative for the code `x = x + y;`

There are other operators that behave in a similar fashion, like `-=`, `*=`, and `/=`.

Why is `distanceTraveled` static?

Because static variables exist independent of object instances, we can access it everywhere via **`Runner`**`.distanceTraveled`. If it were nonstatic, we first need to get a reference to our runner instance before we could get to `distanceTraveled`.

Of course, we could just add a **`Runner`** variable to **`SkylineManager`** and assign *Runner* to it. However, this approach gets unwieldy when we'll need the value in multiple scripts later.

What's a **`Queue`**?

The `System.Collections.Generic` namespace contains the **`Queue`** class, which can be used to represent a first-in, first-out queue. By constantly moving the first entry in the queue to the end of it, we effectively get a rotating ring.

`Queue` is a generic class that can deal with any one type of content. In this case, we use **`Queue`**<**`Transform`**> to declare a queue of **`Transform`** references.

You can add to the end of the queue by using the `Enqueue` method. Taking out the first item is done with the `Dequeue` method. Additionaly, you can get to the first item without removing it via the `Peek` method.

What does **`Random.Range`** do?

`Random` is a utility class that contains some stuff to create random values. Its **`Range`** method can be used to generate a random value within some range.

There are two versions of the **`Range`** method. You can call it with two floats, in which case it returns a float between the minimum and maximum value, both inclusive.

Alternatively, you can call **`Range`** with two integers, in which case it returns an integer between the minimum, inclusive, and maximum, exclusive. A typical use for this version is selecting an index at random, like `someArray[`**`Random.Range`**`(0, someArray.Length)]`.

What's a rigidbody?

A rigidbody is a physics concept, literally a rigid body that doesn't deform. Unity's physics engine will simulate real-world physics behavior for all objects with `RigidBody` components, causing them to fall, move, and collide with other stuff.

It is also possible to have soft bodies, which do deform, like cloth.

What's a physic material?

Physic materials are like regular materials, except they deal with collision instead of visual properties. When objects collide, what happens depends on whether they're made of stone, wood, ice, rubber, or some other substance. You use physic materials to simulate this behavior by configuring friction and bounciness.

When is `FixedUpdate` called?

The physics engine works by dividing time into little discrete steps – by default 0.02 seconds – during which it moves objects and then checks for collisions and triggers. It keeps doing that in a loop until it has caught up with real time.

The `FixedUpdate` method works like `Update`, except that it's called once per physics step instead of once per frame. In other words, `FixedUpdate` is independent of the frame rate.

What does `AddForce` do?

The `AddForce` method applies a force to a rigidbody, which might result in an acceleration, which builds up velocity, which results in movement.

There are actually various ways to use this method, which you control with the second parameter. For example, if you want to apply a specific acceleration, regardless of an object's mass, you can use the `ForceMode.Acceleration` option. If you want to directly adjust the velocity, you can use `ForceMode.VelocityChange`.

What does `&&` do?

The `&&` operator is used for boolean logic and stands for 'and also'. In other words, `x && y` is only true if both `x` and `y` are true.

Note that if `x` is found to be false, there's no point in checking `y` anymore. If `y` were a method call, it won't be invoked. Because of this, when *Runner* isn't touching the platform, the input won't be checked at all.

The companion of `&&` is the `||` operator, which stands for 'or else'. So `x || y` is true if at least one of them is. Also, if `x` is found to be true, then `y` will not be considered.

What does `Input.GetButtonDown` do?

`Input` is a utility class that contains stuff to detect the player's input. This can be anything from button presses to mouse movement to joystick motion.

The `GetButtonDown` method can be used to check whether the user just pressed down a key associated with some button or action. Correspondingly, the `GetButtonUp` method can be used the check whether the user just released it. Also, the `GetButton` method tells you whether the button is current held down.

Shouldn't the jump be in `FixedUpdate`?

When the player presses a jump button, we want the velocity change to happed exactly once. For single instantaneous events, putting the code in `Update` is equivalent to writing code that would activate once in the next `FixedUpdate`.

Why is the class static?

By marking a class as static you require that its contents are static as well. There can't be any nonstatic variables or methods and it cannot be used to create object instances. In other words, a static class is not a blueprint for objects.

What's a `delegate`?

Besides simple values and object references, you also store method references in a variable. Such a variable is known as a delegate.

You define a delegate type as if you're creating a method, except there's no code body. After that, you can use this type to create a delegate variable, to which you can assign any method that matches the type. You can then treat this variable like a method. In fact, you can treat a delegate like a list and add multiple methods to it. All of them will be called when you invoke the variable.

The Graphs tutorial uses delegates to dynamically select what kind of graph to generate.

What's an `event`?

For our purposes, an event is a restricted form of a delegate, forced to behalve like a list. We could use a regular delegate variable instead and it would work just fine.

Both events and delegates allow methods to be added and removed from them, via `myEvent += myMethod` and `myEvent -= myMethod`. A delegate also allows a direct assignment, via `myDelegate = myMethod`. Doing so replaces whatever other methods had been added to it before. We only want the former functionality and not the latter. By disallowing it altogether, we protect ourselves from a potentially hard to find bug caused by forgetting to write a single `+` somewhere.

Also, events can only be invoked by the class that defines them. Outsiders can only register and unregister methods to them.

What's `null`?

The default value of a variable that's not a simple value is **`null`**. This means that the variable doesn't reference anything yet. Trying to invoke or access anything from a variable that's **`null`** results in an error. You can test for this value to make sure that doesn't happen. You can also set such a variable to **`null`** yourself, in case you no longer need whatever it was referencing.

What does `!=` do?

The `!=` operator checks whether two things are different. For example, `1 != 2` is true, while `2 != 2` is false. In our case, we're checking whether our event isn't **`null`**, which it would be if no methods had been added to it.

In contrast, the `==` operator checks whether two things are equal.

Note that for object references, equality is usually a matter identity. Two different objects with the exact same contents are not considered equal.

What's with the text positions?

The GUI text is not drawn in 3D but in 2D, relative to the screen. A position of (0, 0) corresponds to the lower left corner, while (1, 1) corresponds to the top right corner.

Why trigger and handle the same event?

If we trigger the game start event, why not simply put the disabling code right after the call to `TriggerGameStart`?

Any code that deals with the game start has nothing to do with the `Update` method. Regardless how a game start is triggered, it should simply work. That's why we put the code in the appropriate event handler method. If we ever add another way to start a new game, **`GUIManager`** will respond to the event just fine.

Why immediately reset the distance?

Leaving it to the `Update` method to override `distanceTraveled` could lead to bugs. For example, if *Platform Manager* happens to be updated before *Runner*, it would recycle based on the old distance. If this distance is far ahead, the first platform will be recycled immediately, causing *Runner* to plummet to its doom.

There are ways to enforce the order in which components are updated, but it is better to guarantee correct results regardless of update order. If you provide public data, make sure it's always up to date.

What does it mean to be kinematic?

A kinematic rigidbody will not be moved by the physics engine. However, other things will still react to it appropriately. In a way, it's a physics object that defies the laws of physics.

What's `Quaternion.identity`?

`Quaternion.identity` is a static property that corresponds to the identity quaternion, which results in no rotation.

Why a specular shader?

The default specular shader works like the diffuse shader, except that it also has a specular color and a shininess value. The shader uses these to add a highlight to the visuals.

We use this shader for *Booster* because it results in more vivid color changes while it rotates.

What does it mean to be a trigger?

By default, a collider acts like a solid object. You can use the `OnCollisionEnter` method to detect when something hits it.

If a collider is a trigger, it's like a ghost and does not influence the movement of other physics object. Instead, it acts like a radar or alarm. You can use the `OnTriggerEnter` method to detect when something enters the collider's volume.

What does `setActive()` do?

You use this method to either activate or deactivate an entire game object, not just a single component. Also, when deactivating a game object all its child game objects will be deactivated as well. So a game object is only really active when both itself and all of its parents are active. You can check whether this is the case via the property `activeInHierarchy`. You can also check `activeSelf` which disregards the hierarchy, which is what we do because our *Booster* has no parents.

What does `return` do?

You use the `return` keyword to incidate that a method is finished. Implicitly, it's at the end of every method. You can use it to add multiple exit paths to a method.

In our case, we check whether *Booster* shouldn't be spawned, either because it's already active or because of the spawn chance. If we shouldn't spawn, we simply return back to where the method was called.

In case a method produces some result – like a number, shown in the Graphs tutorial – you need to explicitly declare what result it returns.

What's `Time.deltaTime`?

`Time` is a utility class for time-related stuff. Its `deltaTime` property contains the amount of seconds passed since the last frame, or since the last fixed time step if called inside `FixedUpdate`.

What's `this`?

The `this` keyword is a reference to an object itself. As a consequence, it can only be used inside nonstatic methods.

Whenever you're accessing a variable of an object inside one of its methods, you're implicitly using `this` to access it. For example, inside the `Update` method, `transform` is the same as `this.transform`.

What does `ToString()` do?

The `ToString` method can be used to create string representations of data. We use it for an `int` and a `float`. The first conversion is simply the decimal representation of the number (its real form is binary). We do the same for the `float`, except we also add a format description which tells the method to no display the fractional part.

Why not set the labels from `Runner`?

By putting a manager in between *Runner* and the GUI, we make both independent of each other. The `Runner` class doesn't deal with GUI details, only with runner details.

If we were to change the GUI – like using icons to display boosts instead of a label – we only need to modify `GUIManager`, the rest of the game doesn't care about the change.

We could go one step further and not make `Runner` call the GUI manager at all. Then it would be up to `GUIManager` to get the boost count from `Runner` instead. However, then the manager must know details about the runner, which it really shouldn't. Complete decoupling might be achieved by using an event for this, but that's a rather heavy-handed approach for a straightforward case like this. A simple call to a manager is fine.

Why call `GameOver` in `Start`?

Initially, we want our particle systems to not emit, until the first game-start event is triggered. So we need to loop over all emitters and shut them off. Because that's the exact same thing that our `GameOver` method does, we call it instead of writing the same code twice.
