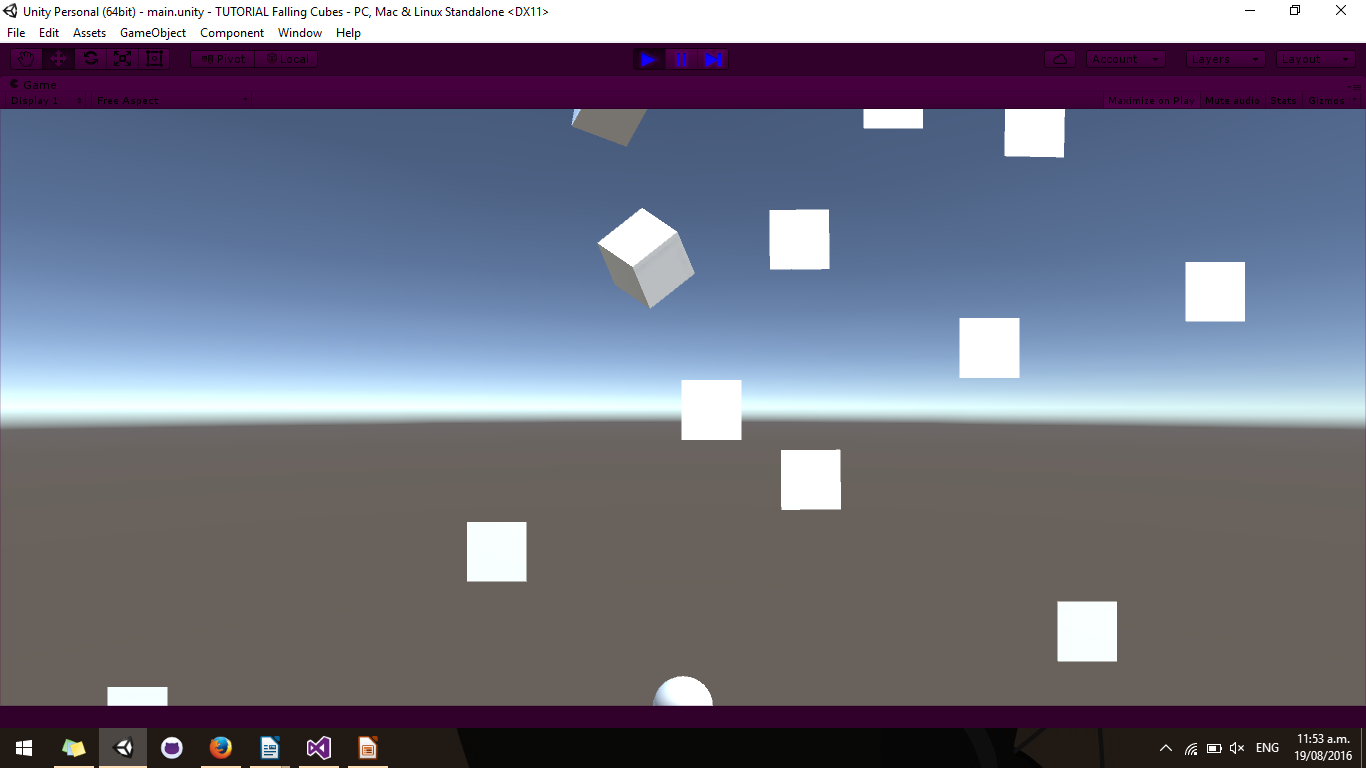
Falling cubes game

# Overview

  
Falling cubes game showing the player-controlled sphere, and the falling cubes that the player must dodge.

**Game contains two elements:**

* A sphere that the player can move left and right with arrow keys
* Cubes that fall from the top of the screen

**And their interaction:**

* Player is destroyed when contacted by a falling cube

**Perspective:**

* The game objects are 3D models (unity cube and sphere); however, the camera does not show depth – the camera’s projection property has been set to orthographic
* The camera does not move – *no code needed here.*

**Physics:**

* Collider and rigidbody components take care of the physics – *no code needed here.*
* Rigidbody causes cubes to be affected by gravity
* In order for collisions to occur (i.e. for objects to not pass through one another) both the player and the falling cubes require colliders and *at least one* of the objects requires a rigidbody (in this case, a rigidbody is attached to each of the cubes)

**Game Objects and attached Components:**

* Main Camera
  + Camera
* Directional Light
  + Light
* Player
  + Mesh renderer
  + Mesh filter (Sphere mesh)
  + Sphere collider
  + Player script
* Cube Spawner
  + Spawner script
* Cube (saved as prefab)
  + Mesh renderer
  + Mesh filter (Cube mesh)
  + Box collider
  + Rigidbody

**Two scripts to write:**

1. Player

* Moves player left or right on key press
* Has a ‘Speed’ property to describe the maximum distance moved per second
* Destroys the player when contacted by a cube

1. Spawner

* Continuously spawns cubes
* Has a property of type GameObject which stores the prefab to be spawned
* Has a ‘SpawnRate’ property to describe the rate at which prefabs are spawned
* Has ‘MinX and MaxX’ properties to describe the range of x-axis coordinates at which the cubes will be spawned

# Scene set-up

Save and name your scene. Remember to save changes to your scene as you progress through this tutorial.

### Camera

**The scene *should* begin with a camera already in it. If it doesn’t add one:**

***GameObject > Camera  
OR  
Right click in ‘Hierarchy’ window > Camera***

* The camera should start at position (x: 0, y: 1, z: -10); if it is not, set its position. To set the camera’s position to a specific value, it is easiest to edit the camera’s transform component in the ‘Inspector’ window
* The cameras rotation should be (x: 0, y: 0, z: 0); if it is not, set its rotation.
* Set the camera’s ‘Projection’ property to ‘Orthographic’

|  |
| --- |
| *NB: You can reset the values of a component’s properties by clicking on the cog in the top-right corner of the relevant component in the ‘Inspector’ window* |

### Light

**The scene *should* begin with a light already in it. If it doesn’t add one:**

***GameObject > Light > Directional Light  
OR  
Right click in ‘Hierarchy’ window > Light > Directional Light***

* Set the light’s rotation to (x: 50, y: -30, z: 0)

|  |
| --- |
| *NB: The position of a directional light does not affect it, only its rotation.* |

### Player

**Create a new sphere object.**

***GameObject > 3D Object > Sphere  
OR  
Right click in ‘Hierarchy’ window > 3D Object > Sphere***

**Position the sphere in the bottom-middle of the ‘Main Camera’s field of view.**

***Edit the position property of the cameras transform component  
OR  
Click and drag the handles attached to the game object in the scene view***

|  |
| --- |
| *NB: Selecting a camera in the ‘Hierarchy’ window will show a picture-in-picture preview in the ‘Scene’ window of what the camera can see. To show the position handles on a game object in the ‘Scene’ view click:* |

**Rename the sphere object ‘Player’:**

***Edit the name field in the ‘Inspector’ window  
OR  
Press F2 with the object selected in the ‘Hierarchy’***

**Create a new C# script called ‘Player’ and attach it to the Player game object.**

***In the ‘Inspector’ window: Add Component > New Script > Create and Add  
OR  
Right click in ‘Project’ window > Create > C# Script***

### Cube Spawner

**Create a new empty game object**

***GameObject > Create Empty  
OR  
Right click in ‘Hierarchy’ window > Create Empty***

* Set the objects position to (x: 0, y: 12, z: 0)
* Rename the object to ‘Cube Spawner’
* Create a new C# script called ‘Spawner’ and attach it to the Cube Spawner game object

|  |
| --- |
| *NB: Notice that the ‘CubeSpawner’ game object has no physical or visual presence in the scene. This is because there are no collider, rigidbody, or renderer components attached to it. It is not unusual to have game objects of this manner in Unity, due to the fact that every script MUST be attached to a game object in order to be executed* |

### Cube

**Create a new Cube object**

***GameObject > 3D Object > Cube  
OR  
Right click in the ‘Hierarchy’ window > 3D Object > Cube***

**Attach a ‘Rigidbody’ component to it**

***In the ‘Inspector’ window: Add Component > Physics > Rigidbody  
OR  
Component > Physics > Rigidbody***

**Convert the cube object into a prefab**

***Drag the object from the ‘Hierarchy’ window to the ‘Project’ window***

|  |
| --- |
| *NB: The objects name will turn blue in the ‘Hierarchy’ window when it is a prefab. Once the object is a prefab it can be safely deleted from the ‘Hierarchy’. DO NOT delete the object from the ‘Project’ window* |

**Delete the cube object from the ‘Hierarchy’ window**

***Select the object in the ‘Hierarchy’ window: Press Delete key OR Right click > Delete***

# Scripts

To open a script for editing double click on it in the ‘Project’ window. This will open the script in the IDE you have associated with Unity (MonoDevelop or Visual Studio)

Notice that by default Unity sets up a script in the following manner:

* Scripts use UnityEngine namespace, this gives us access to unity classes and methods
* Scripts inherit from MonoBehaviour – this is what makes it a component that can be attached to game objects
* Two methods are provided for us: Start() and Update(). These are two of the methods that make up a script’s life cycle.

|  |
| --- |
| *NB: Start(), Update() and other script life cycle events do not override any method from the MonoBehavour inheritance hierarchy. Unity implements its own messaging system for handling these events; such that, these methods are treated as special methods that will be called if they exist in a script. For this reason, it doesn’t matter if a script life cycle event method is declared as private, protected, or public, it will still be called if it is present in a script. In visual studio press Ctrl + Shift + M to show a list of all unity script events.* |

|  |
| --- |
| *NB: MonoBehaviours should not implement a constructor. The Unity game engine is in charge of creating instances of the scripts we write.* |

**Script Compilation**

Unity automatically compiles scripts and updates their associated components when you save or build your code, no action is required; however, you will likely notice a brief 1-2 second period of unresponsiveness when switching immediately back to Unity.

**Script Errors**

Any errors in a script that prevent compilation, or any exceptions that are thrown during game play will be output to the Unity console. To open the console: Window > Console, or click on the error in the status bar at the bottom of the editor window.

### Player script

* In the player class, declare a public float variable called ‘Speed’

public float Speed;

|  |
| --- |
| *NB: By declaring the variable as public it will be exposed in Unity in the ‘Inspector’ window listed under the associated component. Its value can be changed in Unity without the need to recompile the associated script.* |

* In the ‘Update()’ method get the value of the horizontal input axis, assign this value to a variable called ‘horizontalInput’

float horizontalInput = Input.GetAxis(“Horizontal”);

* Calculate the horizontal movement to apply to the player by multiplying Vector3.right with the player’s ‘Speed’ property, the ‘horizontalInput’ variable, and Time.deltaTime

Vector3 movement = Vector3.right \* Speed \* horizontalInput \* Time.deltaTime;

|  |
| --- |
| *NB: Time.deltaTime is the amount of time it took, in seconds, for the last frame to complete. By factoring this value into our calculation we make the horizontal movement frame rate independent. The calculation now equates to ‘the amount of movement per second’.* |

* Apply the movement to the player by manipulating its transform component. This can be done by adding it to the transform’s position property, or by calling transform.Translate()

transform.position += movement;  
**OR**transform.Translate(movement);

* Add the ‘OnCollisionEnter()’ event to the player script. Call the Destroy method on the player’s GameObject within the ‘OnCollisionEnter()’ method.

void OnCollisionEnter()  
{  
 Destroy(gameObject);  
}

|  |
| --- |
| *NB: Destroying a game object removes the game object and all attached components from the scene.* |

### Spawner script

* In the Spawner class, declare the following public variables:

public GameObject Prefab;  
public float SpawnRate;  
public float MinX;  
public float MaxX;

|  |
| --- |
| *NB: Setting an initial value for a public field will carry over to Unity. When a component gets reset (by clicking the cog icon) its values will reset to the initial value set in the script.* |

* Declare a method called Spawn()

void Spawn()

* Get the CubeSpawner game object’s position from its transform component. Save this to a variable called ‘spawnPosition’

Vector3 spawnPosition = transform.position;

* Generate a random number between ‘MinX’ and ‘MaxX’. Save this to a variable called ‘randomX’

float randomX = Random.Range(MinX, MaxX);

* Set the ‘spawnPosition’ vector’s x-coordinate to be ‘randomX’

spawnPosition.x = randomX;

* Create a cube prefab at spawnPosition

Instantiate(Prefab, spawnPosition, Quaternion.identity);

|  |
| --- |
| *NB: The instantiate method handles the creation of a prefab. It will create a new game object, attach the same components that are present in the prefab we passed the method, and set the values of the components properties to be the same as the prefab.* |

|  |
| --- |
| *NB: Quaternions describe rotations in three dimensional space. The value ‘Quaternion.identity’ means “no rotation”.* |

* In the ‘Start()’ method use the InvokeRepeating() method to repeatedly call the ‘Spawn()’ method at regular intervals.

InvokeRepeating(“Spawn”, 0, SpawnRate);

|  |
| --- |
| *NB: The InvokeRepeating() method repeatedly executes a method at the given interval. To stop the method from being continuously called, use the CancelInvoke() method.* |

### Component property values

* In the Unity editor, set the ‘Spawner’ component ‘Prefab’ property to be the cube prefab we made previously.

***Click the small circle next to the ‘Prefab’ input box in the ‘Inspector’ window > Select the cube prefab  
OR  
Drag the cube prefab from the ‘Project’ window to the ‘Prefab’ input box in the ‘Inspector’ window***

* Set appropriate values for the ‘Player’ and ‘Spawner’ component properties, try different values until you find some you like.

|  |
| --- |
| *NB: Some values may give interesting or unintended results. For example, a ‘SpawnRate’ of 0 will cause only one cube to be spawned. A negative value for the player’s ‘Speed’ will cause the player to move in the opposite direction to the axis key that is pressed* |

|  |
| --- |
| *NB: Changes to component property values during play mode ARE LOST when exiting play mode.* |

### Achievement Unlocked!

Basic game functionality has been achieved! Read on for additional explanations and functionality...

**Falling cubes?**

Notice that, in some cases, not all of the cubes move exclusively vertically. Some have a considerable amount of horizontal movement as well as rotation. This is a result of the unity physics system handling collisions. When cubes are being spawned in the game quickly and in a confined space, some of the spawned cubes overlap one another. Unity registers this as a collision and responds accordingly: both objects will have a repulsive force and torque (rotational force) applied to them so as to resolve the collision. This causes some objects to shoot off in random directions. You can observe this effect by watching the cubes as they spawn.

**Communication between components**

Components are the functional pieces of every game object; they encapsulate the objects behaviours. In order to change the behaviour of an object it is necessary to talk to its components. Often, we want this to happen as the result of an interaction between two objects. In this manner, it becomes necessary for communication between components to occur (one component calling another components methods).

To talk to another component from within a script we must first get a reference to the desired component. If the desired component is attached to the SAME game object as the script we are writing code in, then we can use the ‘GetComponent’ method. If the desired component is attached to a DIFFERENT game object, we must first get reference to the game object the desired script is attached to.

### Additional functionality

This section challenges you with additional functionalities that you can add to the game. Give each of them a go. Or don’t. I’m a handout, not the police.

**Stop cubes spawning**

Stop cubes spawning when the player dies. To tell the ‘Spawner’ component to stop spawning cubes we must get a reference to it. To do this via scripting, we must first find and talk to the game object that owns the ‘Spawner’ script. Unity has various methods that will search the scene for a particular object. One of these is GameObject.Find(“name”) which finds the first instance of a game object with the given name in the current scene.

GameObject cubeSpawnerObject = GameObject.Find(“CubeSpawner”);

Once we have a reference to the ‘CubeSpawner’ game object we can get a reference to its contained ‘Spawner’ component with the GetComponent method.

Spawner cubeSpawner = cubeSpawnerObject.GetComponent<Spawner>();

Now we can call the ‘Spawner’ components methods. In order to stop the cubes from spawning we must ‘turn off’ the repeated calling of the ‘Spawn’ method that we began by calling the ‘InvokeRepeating’ method. To do so, call the ‘Spawner’ components ‘CancelInvoke’ method.

cubeSpawner.CancelInvoke();

|  |
| --- |
| *NB: The InvokeRepeating and CancelInvoke methods are inherited from MonoBehaviour, and as such all scripts have access to them. Calling CancelInvoke for a script cancels all invoke calls that that script is currently making.* |

* Add the above lines of code to the ‘Player’s ‘OnCollisionEnter’ method. The method should look as follows:

void OnCollisionEnter()  
{  
 Destroy(gameObject);  
  
 GameObject cubeSpawnerObject = GameObject.Find(“CubeSpawner”);  
 Spawner cubeSpawner = cubeSpawnerObject.GetComponent<Spawner>();  
 cubeSpawner.CancelInvoke();  
}

**Player lives**

Add lives (or HP) to the player object: the player only dies once they’ve been hit by a certain number of cubes. Allow the number of lives a player has to be changed from within Unity.

**Coloured sphere**

Change the colour of the player sphere when it loses a life. From within the ‘Player’s OnCollisionEnter method get the player object’s ‘Renderer’ component (see above). The renderer component has a ‘material’ property, which itself has a ‘color’ property.

|  |
| --- |
| *NB: A material is a collection of properties that determine how a 3D mesh is drawn* |
| *NB: Color.Lerp() can be used to ‘mix’ colours.* |

**Destroy off screen cubes**

The script life cycle event ‘OnBecameInvisible’ is called whenever an object is no longer visible to ANY camera. Importantly, this event is only called once the object has been visible at least once. Create a new script, add the ‘OnBecameInvisible’ method, and add the script to the cube prefab we made previously.

What code goes inside the ‘OnBecameInvisible’ method?

### Solutions

Check out my organization GitHub page (<https://github.com/DoesntSuck/>) to get the code associated with this lecture. The project files contain two scenes, one with the basic functionality, and another with the additional functionalities. Plenty of comments throughout.

### Additional resources

**Learning**

* Cooking with Unity: <https://www.youtube.com/user/PushyPixels>
  + 3 Minute game (the basis of this lecture): https://www.youtube.com/watch?v=i30menw6gvU
* Unity tutorials: <https://unity3d.com/learn/tutorials>
  + Basic tutorials
  + Advanced tutorials
  + Tutorials focusing on a specific game – downloadable project resources
* Making stuff look good in unity
  + A steep introduction to shaders, but damn... does that stuff look good
* Explanation of GameObject and Component architecture in games
  + <http://gameprogrammingpatterns.com/component.html>

**Other stuff**

* Unity subreddits: /r/Unity3D, /r/Unity2D
  + Repository of gifs / videos etc of people showing off their games
* Free 3D models and animations: <https://www.mixamo.com/>
* Unity asset store: <https://www.assetstore.unity3d.com/>
  + Free and paid assets (models, sprites, scripts, editor extensions)

**Groups**

* Game Developers in Dunedin (monthly meet ups):
  + <https://www.facebook.com/gamedevdunedin/>