CE318/CE818: High-level Games Development

Lecture 1: Introduction. C# & Unity3D Basics

Diego Perez

dperez@essex.ac.uk
Office 3A.527

2016/17

Outline

Outline

Course Overview

This course will teach you the fundamentals of games console programming. We will focus on 3D games, developed using **Unity3D 5.1.4**. Topics covered include:

- Creating 3D games in Unity3D.
- Use of the Unity3D editor.
- 2D-3D Math game concepts.
- Managing player input.
- 3D game physics.
- Cameras.
- 3D Animations.
- Navigation.
- Graphical User Interfaces.
- Lights and audio.
- Particle Systems.
- Terrains.
- Game design.
- Gameplay and Game AI

The course is very practical, using numerous code samples throughout the lectures and encourages creative game design in the labs.

Module details: `http://www.essex.ac.uk/modules/default.aspx?coursecode=CE318&level=6&period=AU`

Description and resources: `http://orb.essex.ac.uk/ce/ce318/`

Learning Outcomes & Assessment

Following the completion of this module, you will be able to:

- Demonstrate an understanding of the architecture of 2D / 3D games
- Design and implement simple 2D / 3D games using Unity3D
- Implement basic AI behaviours for non-player characters
- Design and implement graphical effects in 3D
- Design and implement game objects

The learning outcomes will be **assessed** by

- Progress tests (30%)
  - Progress Test 1 (15%) - Lectures 1 - 4
  - Progress Test 2 (15%) - Lectures 5 - 9
- Bi-weekly lab assignments (10%)
- Single assignment (60%)
  - Part I (20%; due **22nd November**, 11:59:59, 2016)
  - Part II (40%; due **9th January**, 11:59:59, 2017)

Sample test questions will be provided during the lectures.

Course schedule

There will be a 2-hour lecture every Tuesday at 1pm in room 3.320.

- 10 minute break every 50 minutes.
- Includes classroom discussions, software demos & test questions (Kahoot!).

2-hour labs will take place every Friday 1pm in CES Lab 1.

- Step-by-step instructions, tutorials & open-ended exercises
- The laboratory assistant is Emily Marriot (`ecmarr@essex.ac.uk`)

	Day	Week	Time	Room
Lectures	Tuesday	2-5, 7-10	1pm - 3pm	3.320
Lecture	Tuesday	6, 11	2pm - 3pm	3.320
Labs	Friday	2-11	1pm - 3pm	CES Lab 1
Test	Tuesday	6 (8th Nov)	1pm - 2pm	3.320
Test	Tuesday	11 (13th Dec)	1pm - 2pm	3.320

Finally, the demonstrations/presentation of the assignment (part II) in **N** lab sessions, on **potentially** Week 16.

Assignments

All assignments in CE318 are individual. All labs, presentations and demonstrations are **mandatory**.

- (0% in case of unjustified absence).
- Tell me **NOW**.

(Bi-)Weekly Assignments (10%):

- There will be 4 bi-weekly assignments
- These assignments will be assessed in the lab each second week

Main Assignment (60%):

- Development of a full game
- 2 deliverables:
    - Part I (20%, Wed 22nd November):
        - CE318: Game Prototype (10%) + Development plan (5%) + 5 minute presentation (5%) on Week 9
    - Part II (40%, Mon 9th January):
        - CE318: Full developed game (20%) + Final Report (10%) + 10 minute Presentation (10%, peer reviewed).
        - CE818: Full developed game (15%) + Final Report (10%) + Case Study (10%) + 10 minute Presentation (5%, peer reviewed).

Peer Review and Best Game Awards

You will evaluate each other:

1. Final presentation.
2. How “fun” the game is.
3. Aesthetics of the game.
4. Best game overall

- The identity of who marks this will be anonymous to the person being assessed...
- ... but not to the lecturer/GLA of the module.
- We reserve the right to not taking into account (non-objective) extremely high or extremely low marks.

On the Spring term, three awards will be given, based on your classmates' evaluations, to the last three categories. Most likely, these will be 3 vouchers (around 20 GBP) for Amazon or Waterstones.

Recommended Reading and Web Resources

C#

1. *C# 4.0 Pocket Reference* by Albaharai and Albahari
2. *Essential C# 4.0* by Mark Michaelis
3. MSDN C# Ref: `msdn.microsoft.com/en-us/library/618ayhy6.aspx`

Unity3D

1. `http://unity3d.com/`: Manual, Scripting API, Samples, Tutorials... (most of materials of this course are taken from there).
2. *Beginning 3D Game Development with Unity 4, 2nd Edition* by Sue Blackman (highly recommended).
3. *Unity 4.x Cookbook* by Matt Smith (complementary book).
4. *Unity 4.x Game AI Programming* by Aung Sithu Kyaw, Thet Naing Swe (for Game AI in Unity).
5. *Unity 4x Game Development by Example, Beginners Guide* by Ryan Henson Creighton (for less confident programmers).

Game AI

1. *Artificial Intelligence for Games* by Millington and Funge
2. *Game Artificial Intelligence* by Ahlquist and Novak

Outline

Introduction to C#

All Unity3D games usually include scripts written in C#, Javascript or Boo.

In this module, we will only use C#, an object-oriented programming language developed by Microsoft within its .NET framework.

Unity3D comes with a default IDE called MonoDevelop, that provides syntax colour highlighting and debug capabilities.

The Basics of C#: From a Java Perspective

```
using System;

namespace HelloWorld
{
  class Hello
  {
    static void Main(string[] args)
    {
      PrintHelloWorld();
      Debug.Log("Press any key to exit.");
      Console.ReadKey();
    }

    static void PrintHelloWorld()
    {
      Debug.Log("Hello World!");
    }
  }
}
```

- The word `using` is used to import packages each of which is known as a `namespace`.
- You can have many classes in each namespace and the filename does not need to correspond to either the namespace nor any of the enclosed classes.
- Method names (Main) start with a capital letter, whereas the built-in types `class` and `string` do not.

The Basics of C#: Variables, parameters & namespaces

Variables and constants:

```
private int var1 = 100;
protected double var2 = 20d;
public bool var3 = true;
string var4 = "A string";
string var5 = null;
public const int con1 = 3;
```

Static variables in a class: All objects from the class share this variable with the same value.

```
private static int var1 = 100;
```

Conversions (implicit and explicit):

```
static int i = 10;
static double d = i;
static short s = (short)d;
```

`params` and optional parameters:

```
void Pars(params int[] numbers){}
Pars(1,2,3,4);

void Opt(int a, int b = 10){}
Opt(1,2);
Opt(1);
```

Namespaces are like packages. You can create a hierarchical organisation of your code by nesting namespaces:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

You import namespaces using the `using` directive, to access methods and variables available in that namespace.

The Basics of C#: value and reference types

Value datatypes: contain *a value*.

- int, float, double, bool, char, Structs (in Unity3D: Vector3, Quaternion).
- Changing a value type only changes the value of that variable.

Reference datatypes: contain *a memory address* that contains a value.

- Classes (in Unity3D: Transform, GameObject)
- Changing a reference type changes it for everyone that has access to it.

The left sample **doesn't** change the transform's position. The right one **does**.

```
Vector3 pos = transform.position;
pos = new Vector3(0,2,0);
```

```
Transform tr = transform;
tr.position = new Vector3(0,2,0);
```

`ref` and `out`: arguments by reference, not by value:

`out` keyword:

```
int a;
Out(out a);

void Out(out int a)
{
  a = 10;
}
```

`ref` keyword: requires initialized parameter.

```
int a = 5; int b = 10;
Swap(ref a, ref b);

void Swap(ref int a, ref int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
```

The Basics of C#: Classes, Interfaces and Inheritance

Classes and interfaces are similar to Java. In C#, a class can only ever subclass one class at a time but may implement numerous interfaces.

A simple class:

```
class MyClass1
{
  public int id;

  public MyClass(int id)
  {
    this.id = id;
  }

  public int getID()
  {
    return id;
  }
}
```

A simple interface (notice 'I'):

```
interface IMyInterface
{
  void AMethod();
}
```

Class implements interface:

```
class MyClass2 : IMyInterface
{
  public void AMethod()
  {
    // Do something
  }
}
```

Use `override` keyword to override methods from base:

```
class BaseClass{
  public void AMethod(){}
}

class SubClass : BaseClass{
  public override void AMethod(){}
}
```

Use `base` to access super class.

The Basics of C#: Generics

Allows the programmer to defer the specification of one or more types until the class or method is declared and instantiated. For example:

```
public class GenericList<T>
{
    void Add(T input) { }
}
```

```
class TestGenericList
{
    private class AClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass.
        GenericList<AClass> list3 = new GenericList<AClass>();
    }
}
```

The Basics of C#: Properties

Properties are shortcuts for the getters and setters usually associated with class variables. It can be used to make a variable read-only.

```
class TimePeriod
{
    private double seconds;

    //Short version (good rule: make first letter upper case)
    public double Seconds {get; set; }

    public double Hours
    {
        //Can do operations, or call other functions.
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}
```

```
TimePeriod t = new TimePeriod();
t.Hours = 24;
System.Console.WriteLine("Time in hours: " + t.Hours);
```

Now the class has a property called `Seconds` which can be set by calling

```
TimePeriod.Seconds=60;
```

Use access modifiers (e.g., `private`) to prevent anyone from changing the value of the property.

The Basics of C#: Structs

Like a class but a value type. Also, they do not support inheritance.

```
struct Rectangle
{
  public int Width { get; set; }
  public int Height { get; set; }

  public Rectangle(int width, int height)
  {
    Width = width;
    Height = height;
  }

  public Rectangle Add(Rectangle rect)
  {
    Rectangle newRect = new Rectangle();

    newRect.Width = Width + rect.Width;
    newRect.Height = Height + rect.Height;

    return newRect;
  }
}
```

Default constructor initialises all variables to 0; you cannot assign a value to them (unless they are `const`). Structs do not require instantiation of an object on the heap. By default, a struct is passed by value, while a class is passed by reference.

The Basics of C#: Arrays, Lists & Dictionaries

C# has two types of array: **rectangular** and **jagged**. In almost all cases, you will use rectangular arrays (more efficient).

```
//One dimensional arrays:
int[] a = new int[3];
int[] b = new int[] { 4, 3, 8 };
int[] c = { 4, 3, 8 };

//Jagged array: each element is
//an array of different length
int[][] d = new int[3][];
d[0] = new int[3];
d[1] = new int[4];
d[2] = new int[1];

//Rectangular arrays:
double[,] e = new double[2, 2];

int[,] f = new int[2, 2]
{
  {0,1},
  {2,3}
};
```

For individual dimensions:

Jagged arrays, use `.Length`

Rectangular arrays, use `getLength(DIM)`

List and Dictionaries use generics (like Java).

```
List<string> l = new List<string>();

l.Add("one");
l.Add("two");

l.Insert(0, "three");
l.Sort();

foreach(string s in l)
  Console.WriteLine(s);
```

```
Dictionary<string, int> d;
d = new Dictionary<string, int>();

d.Add("one", 1);
d.Add("two", 2);

int val = d["two"];

bool exists;
exists = d.TryGetValue("one",out v);
if(exists)
  Debug.Log(v); //Success!
```

The Basics of C#: Enumerations

Enumerations are used frequently in games to indicate different states of the game (e.g., splash screen, paused, etc.):

```
public enum MyEnum { TYPE1 , TYPE2 , TYPE3 };
```

They are often used in conjunction with switch statements.

```
public void PrintEnum(MyEnum t)
{
  switch (t)
  {
    case MyEnum.TYPE1:
      Console.WriteLine("T1");
    break;
    case MyEnum.TYPE2:
      Console.WriteLine("T2");
    break;
    case MyEnum.TYPE3:
      Console.WriteLine("T3");
    break;
  }
}
```

Each `enum` member has an integral value, following declaration order, from 0 to N.

Explicit values can be assigned in declaration:

```
public enum MyEnum {
  TYPE1 = 10,
  TYPE2 = 20
};
```

And retrieved by explicit casting:

```
int t1Val = (int)MyEnum.TYPE1;
Console.WriteLine(t1Val); //=10
```

The Basics of C#: Loops

C# has numerous iteration statements, similar to Java.

```
string[] a = new string[2];
a[0] = "one";
a[1] = "two";

for (int i = 0; i < a.GetLength(0); i++)
  Console.WriteLine(a[i]);

foreach (string s in a)
  Console.WriteLine(s);

int count = 0;

while (count < a.GetLength(0))   //Same as a.Length
  Console.WriteLine(a[count++]);

count = 0;

do
{
  Console.WriteLine(a[count]);
}
while (++count < a.GetLength(0));
```

C# also has `break`, `continue` and `goto` statements.

The Basics of C#: Additional Concepts

Comments:

```
// Line comment

/* Multi-line
 * comment
 */

/// <sumary> XML </summary>
```

`is` keyword: Checks if an object is compatible with a given type.

```
if (obj is MyObject)
{
   /*obj is an instance of the
        MyObject type, or a type
        that derives from MyObject*/
}
```

Preprocessor directives: allows the compiler to ignore certain lines of code when building (and executing).

```
#if UNITY_WII
  Debug.Log("Compiling/Executing for Wii");
#elif UNITY_WEBPLAYER
  Debug.Log("Compiling/Executing for browser");
#endif
```

Unity 5.1 allows the creation of new defines for scripting.

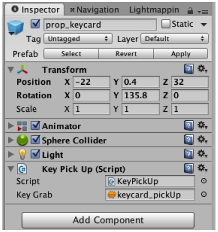
```
#if DEBUG
  Debug.Log("Debugging");
#else
  Debug.Log("No debug");
#endif
```

Outline

Scripting for Unity

Scripting is essential for games: determines how the player input is captured and used, how the interactions with the gameplay elements must function, creates graphical effects, enemy Artificial Intelligence (AI), physics, etc. In Unity3D 5.1, it is possible to write scripts in three languages: C#, UnityScript (Javascript) and Boo. We will be using C# in this module.

- A **Game Object** is each one of the entities present in a Scene of your game. Game objects created in Unity's Editor can be controlled by **scripts**.
- Each game object in Unity contains a collection of **Components** (lights, colliders, animations, etc.).
- **Scripts** can be seen as behaviour component of a game object.
- Scripts **need** to be attached to a game object to work.



☑ prop_keycard					☐ Static	
Tag	Untagged		Layer	Default		
Transform						
Position	X	-22	Y	0.4	Z	32
Rotation	X	0	Y	135.8	Z	0
Scale	X	1	Y	1	Z	1
☑ **Animator**						
☑ **Sphere Collider**						
☑ **Light**						
Key Pick Up (Script)						
Script	KeyPickUp					
Key Grab	keycard_pickUp					

Anatomy of a Script

A game object may contain multiple scripts. Ideally, each script should take care of a particular behaviour of the component (i.e. functionality).

When creating a new C# Script from the Unity3D editor, the initial code looks like this:

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

No constructor! Actually, you must not add any. Unity creates the object for you before the game starts.

Event Functions (1/2)

Unity passes the control to each script intermittently by calling a determined set of functions, called *Event Functions*. The list of available functions is very large, here are the most commonly used ones:

- Initialization:
  - `void Awake()`: First function to be called, when the scene is load. It is only called if its game object is **active**. If not, it will be the first function called when the game object becomes active.
  - `void Start()`: Start is called before the first frame update only if the script instance (the component) is **enabled**.
- Regular Update Events:
  - `void FixedUpdate()`: often called more frequently than Update. All physics calculations and updates occur immediately after FixedUpdate. When applying movement calculations inside FixedUpdate, you don't need to multiply your values by `Time.deltaTime`, as it's called indep. of the framerate.
  - `void Update()`: called at every frame, just before the frame is rendered. `Time .deltaTime`: time in seconds it took to complete the last frame (time since last call to Update()).
  - `void LateUpdate()`: called once per frame, after Update has finished. Any calculations that are performed in Update will have completed when Late-Update begins. Example of use: camera that follows the player.

Event Functions (2/2)

- GUI Events:
  - `void OnGUI()`: For functionality of the graphical user interface
  - `void OnMouseDown()`, `void OnMouseEnter()`, `void OnMouseOver()`: for mouse events related to GUI Components.
- Physics Events:
  - `void OnCollisionEnter()`, `void OnCollisionStay()`, `void OnCollisionExit()`: For collisions with colliders.
  - `void OnTriggerEnter()`, `void OnTriggerStay()`, `void OnTriggerExit()`: for collisions with triggers.

The complete list of functions and variables usable from the base class (MonoBehaviour) can be consulted in the manual:
`http://docs.unity3d.com/ScriptReference/MonoBehaviour.html`

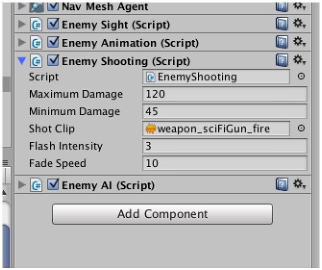
The order in which all the Event Functions are called is depicted here:
`http://docs.unity3d.com/Manual/ExecutionOrder.html`

Variables in Scripts

```
using UnityEngine;
using System.Collections;

public class EnemyShooting : MonoBehaviour
{
  public float maximumDamage = 120f;
  public float minimumDamage = 45f;
  public AudioClip shotClip;
  public float flashIntensity = 3f;
  public float fadeSpeed = 10f;


  private Animator anim;
  private Light laserShotLight;
  private SphereCollider col;
```



Script	EnemyShooting
Maximum Damage	120
Minimum Damage	45
Shot Clip	weapon_sciFiGun_fire
Flash Intensity	3
Fade Speed	10

- Values on the editor takes preference over the values declared as attributes.
- However, Awake() and Start() could override the editor values.

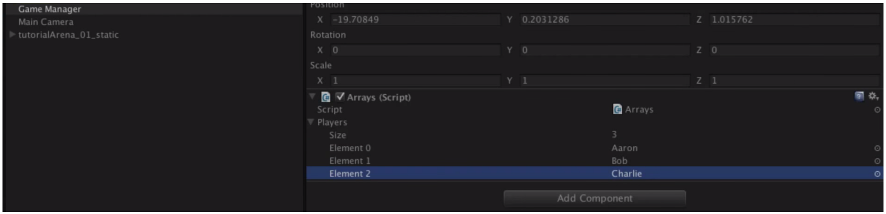
Arrays in Scripts

```
using UnityEngine;
using System.Collections;

public class Arrays : MonoBehaviour
{
   public GameObject[] players;

   void Start ()
   {
      players = GameObject.FindGameObjectsWithTag("Player");

      for(int i = 0; i < players.Length; i++)
      {
         Debug.Log("Player Number "+i+" is named "+players[i].name);
      }
   }
}
```



Components of a game object *

Each GameObject contains a collection of components. Here are some of the most commonly used ones:

- Transform: Position, rotation and scale of an object.
- Collider: Different types of colliders, with different shapes.
- Rigidbody: Control of an object's position through physics simulation.
- Scripts.
- Animator: Interface to control the Mecanim animation system.
- AudioSource: A representation of audio sources in 3D.
- Light: Script interface for light components.

A script can have a game object or a component as a variable:

- It is possible to assign game objects from the editor as variables.
- It is also possible to assign components of game objects from the editor: Unity will assign the component (if it exists) to the variable.

Scripts and Game Objects (1/4) *

All components can be accessed from the scripts one way or another.
A script assigned to a game object can access it's own components in several ways:

1. Some of them are so commonly used that they are accessible as variables: transform, ~~rigidbody, collider, audio, light~~ (**Not** in Unity3D 5+ anymore!).
2. `rigidbody` $\longrightarrow$ `GetComponent <Rigidbody >()`
3. In the general case, all components can be accessed by calling:

```
//This call picks the first component of the object with the
    given type (Rigidbody, Transform, etc).
Rigidbody rb = GetComponent<Rigidbody>();
Transform tr =  GetComponent<Transform>();
```

There is an alternative way of the calling `GetComponent()` method, using a `string`:

```

rigidBody = GetComponent("Rigidbody") as Rigidbody;
```

However, it's better to use the generic version, for performance reasons.

Scripts and Game Objects (2/4)

Other variants of `GetComponent`:

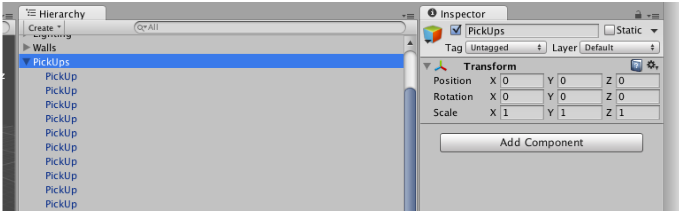
- `Component GetComponentInChildren<T>()`: Returns the component of type `T` in the GameObject **or** any of its children using depth first search.
- `Component GetComponentInParent<T>()`: Returns the component of type `T` in the GameObject **or** of its parent.

It is possible to get more than one component at once:

- `Component[] GetComponents<T>()`: Returns all components of type `T` in the GameObject.
- `Component[] GetComponentsInChildren<T>()`: Returns all components of type `T` in the GameObject **or** any of its children.
- `Component[] GetComponentsInParent<T>()`: Returns all components of type `T` in the GameObject **or** of its parent.
- Example: all enemies of a given type.

Scripts and Game Objects (3/4)

Often, it is quite usual that you have several game objects of the same *type* (examples: coins in Mario Bros, points in a patrol path, checkpoints in a racing game, ...). A way to keep these Game Objects organized is to make all of them children of a common empty game object:



It is possible to access all these game objects through the Transform of the parent game object:

```
void Start() {
    Transform[] pickups = new Transform[transform.childCount];
    int i = 0;

    for (Transform t in transform)
       pickups[i++] = t;
}
```

Scripts and Game Objects (4/4)

An example:

```
//Script component of the game object "Enemy"
public class EnemyAttack : MonoBehaviour
{
   // Reference to the player GameObject.
   GameObject player;

   // PlayerHealth is a Script component in another object ("Player").
   // The class name is PlayerHealth
   PlayerHealth playerHealth;

   // EnemyHealth is a Script component in the same object ("Enemy").
   // The class name is EnemyHealth
   EnemyHealth enemyHealth;

   void Awake ()
   {
      //Costly operations: better in Awake (once) than in Update (n).

      //First, obtain the Player game object with its tag.
      player = GameObject.FindGameObjectWithTag ("Player");

      //Obtain a (script) component from the Player's game object.
      playerHealth = player.GetComponent <PlayerHealth> ();

      //And get a (script) component from this game object.
      enemyHealth = GetComponent<EnemyHealth>();
   }
}
```

Operations with game objects and components

Destroying game objects or components at runtime:

- `void Destroy(Object obj, float t = 0.0F)`: destroys the object/component passed as first parameter. The second argument, optional, indicates a delay in seconds for the operation (if not provided, it's destroyed instantaneously).
- If *obj* is a Component it will remove the component from the GameObject and destroy it. If *obj* is a GameObject it will destroy the GameObject, all its components and all transform children of the GameObject.
- `Destroy` won't destroy the object immediately. It marks the object to be destroyed, and all marked objects will be destroyed at the end of the frame.

Activating / **Deactivating** game objects:

- `void SetActive(bool value)`: Activates/Deactivates the GameObject.
- Making a GameObject inactive will deactivate every component, turning off any attached renderers, colliders, rigidbodies, scripts, etc... Any scripts that you have attached to the GameObject will no longer have Update() called. Check `GameObject.activeSelf` and `GameObject.activeInHierarchy`.

Enabling and **disabling** components:

- `bool Behaviour.enabled`: `true` to *enable* a component, `false` to *disable* it.
- Enabled Behaviours (components) are Updated, disabled Behaviours aren't.

Names and Tags *

It is also possible to find game objects by its name, given in the editor:

- `GameObject Find(string name)`:
  - Finds a game object by *name* and returns it.
  - If no game object with name can be found, *null* is returned.
  - For performance reasons it is recommended to not use this function every frame. Instead cache the result in a member variable at startup or use `GameObject.FindWithTag`.

Tagging: It is possible to define tags for all game objects in a scene. Tags must be defined in the tag manager, and each object can be assigned to one of this tags.

Given a tag, it is possible to retrieve one or several game objects.

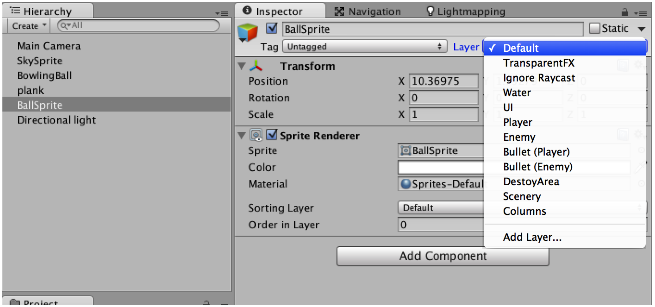
- `GameObject FindWithTag(string tag)`: Returns one **active** GameObject tagged *tag*. Returns *null* if no GameObject was found.
- `GameObject[] FindGameObjectsWithTag(string tag)`: Returns a list of **active** GameObjects tagged *tag*. Returns *empty array* if no GameObject was found.
- The tag must be defined in the editor, or a *UnityException* will be thrown.
- Note that these functions are *static*, belonging to the GameObject class.

Layers (1/4) *

It is possible to organize your game objects in **layers**. Layers are used:

- by cameras to render only a part of the screen.
- by raycasting to selectively ignore colliders.
- by Lights to illuminate the scene.
- to determine collisions (Layer-based Collision).
- to determine the order in which sprites are rendered.

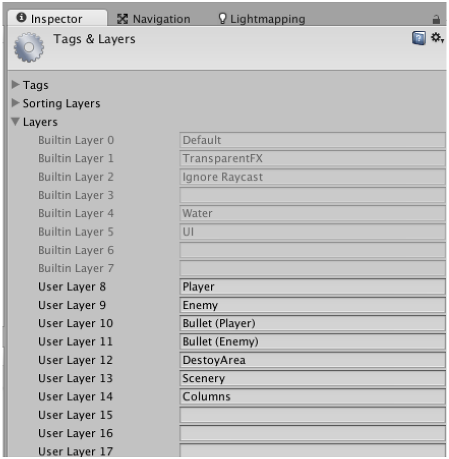
To assign a game object to a layer, simply select the layer at the top of the *Inspector View*, with the game object selected:



BallSprite
Static
Tag Untagged
Layer
Transform
Position X 10.36975
Rotation X 0
Scale X 1
Sprite Renderer
Sprite BallSprite
Color
Material Sprites-Defaul
Sorting Layer Default
Order in Layer 0

Layers (2/4)

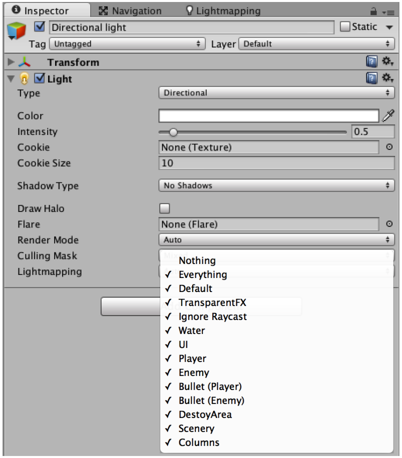
Unity provides some layers by default, but you can create new layers in *Edit* → *Project Settings* → *Tags and Layers*.



Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
Builtin Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
Builtin Layer 6	
Builtin Layer 7	
User Layer 8	Player
User Layer 9	Enemy
User Layer 10	Bullet (Player)
User Layer 11	Bullet (Enemy)
User Layer 12	DestoyArea
User Layer 13	Scenery
User Layer 14	Columns
User Layer 15	
User Layer 16	
User Layer 17	

Layers (3/4)

It is possible to specify, for both camera and light components, which layers will be affected by these (*Culling Mask* selector). By default, all layers are rendered by the camera and illuminated by the lights.

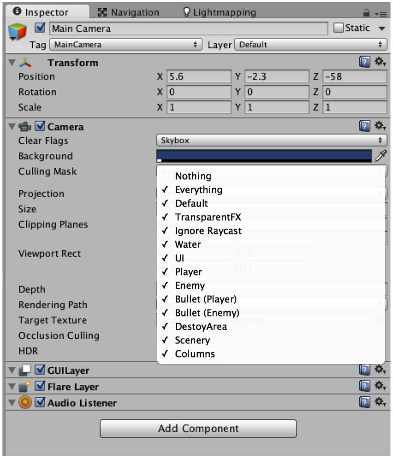


Directional light — Static
Tag: Untagged — Layer: Default

Transform

Light
Type: Directional
Color:
Intensity: 0.5
Cookie: None (Texture)
Cookie Size: 10
Shadow Type: No Shadows
Draw Halo: ☐
Flare: None (Flare)
Render Mode: Auto
Culling Mask
Lightmapping

Nothing
✓ Everything
✓ Default
✓ TransparentFX
✓ Ignore Raycast
✓ Water
✓ UI
✓ Player
✓ Enemy
✓ Bullet (Player)
✓ Bullet (Enemy)
✓ DestoyArea
✓ Scenery
✓ Columns



Main Camera — Static
Tag: MainCamera — Layer: Default

Transform
Position: X 5.6 Y -2.3 Z -58
Rotation: X 0 Y 0 Z 0
Scale: X 1 Y 1 Z 1

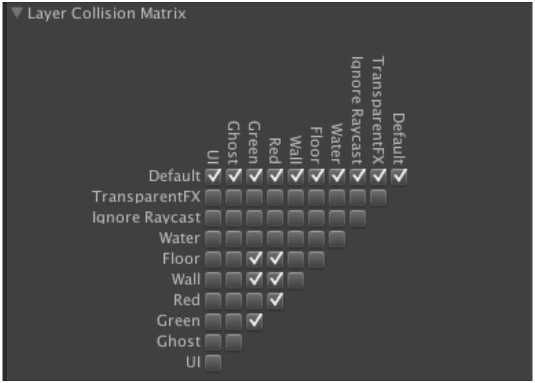
Camera
Clear Flags: Skybox
Background
Culling Mask
Projection
Size
Clipping Planes
Viewport Rect
Depth
Rendering Path
Target Texture
Occlusion Culling
HDR

Nothing
✓ Everything
✓ Default
✓ TransparentFX
✓ Ignore Raycast
✓ Water
✓ UI
✓ Player
✓ Enemy
✓ Bullet (Player)
✓ Bullet (Enemy)
✓ DestoyArea
✓ Scenery
✓ Columns

GUILayer
Flare Layer
Audio Listener

Layers (4/4)

Layer-based collision detection: It is also possible to specify which game objects can collide with which by enabling/disabling collisions between their layers.



Layer Collision Matrix

	UI	Ghost	Green	Red	Wall	Floor	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	☐	☐	☐	☐	☐	☐	☐	☐	☐	
Ignore Raycast	☐	☐	☐	☐	☐	☐	☐	☐		
Water	☐	☐	☐	☐	☐	☐	☐			
Floor	☐	☐	✓	✓	☐	☐				
Wall	☐	☐	✓	✓	☐					
Red	☐	☐	☐	✓						
Green	☐	☐	✓							
Ghost	☐	☐								
UI	☐									

In the Physics 2D Manager, *Edit* → *Project Settings* → *Physics2D*.

Prefab assets *

A **prefab** is an asset that stores a game object with all its components and properties.

- It acts as a template for that game object.
- Any edits made to a prefab asset are immediately reflected in all instances produced from it.
- It is possible to override components and settings for each instance individually.

A prefab can be instantiated dynamically on runtime:

- `Object Instantiate(Object original)`
- `Object Instantiate(Object original, Vector3 position, Quaternion rotation)`
- Creates an object, clone of the prefab.

Invoke: scheduling calls

The family of *Invoke* methods allow to schedule calls to a function with a delay. Only functions with return type `void` and with no arguments can be called using the *Invoke* family.

- `void Invoke(string methodName, float time)`: Invokes the method *methodName* in *time* seconds.
- `void InvokeRepeating(string methodName, float time, float repeatRate)`: Invokes the method *methodName* in *time* seconds, then repeatedly every *repeatRate* seconds.
- `void CancelInvoke()`: Cancels all Invoke calls on this MonoBehaviour.
- `void CancelInvoke(string methodName)`: Cancels all Invoke calls on *methodName*.

```
Invoke ("SpawnObject", 2);
InvokeRepeating("SpawnObjectRandomPosition", 2, 1);

void SpawnObject()
{
   Instantiate(target, new Vector3(0, 2, 0), Quaternion.identity);
}

void SpawnObjectRandomPosition()
{
   float x = Random.Range(-2.0f, 2.0f);
   float z = Random.Range(-2.0f, 2.0f);
   Instantiate(target, new Vector3(x, 2, z), Quaternion.identity);
}
```

Coroutines (1/2)

A coroutine is a function that is executed in intervals:

- It is started calling the StartCoroutine method: `StartCoroutine (method());` or `StartCoroutine (string methodName);`
- The method called must be of type `IEnumerator`.
- It can return using the keywords `yield return`.
- In the next update call, this function resumes its execution from the `yield return` point (note: this is decoupled from the `Update()` event function).
- It finishes when *method* ends, or `StopCoroutine (string methodName);` is called.

```
void Start () {
   StartCoroutine (TestCoroutine ());
}

IEnumerator TestCoroutine(){
   while(transform.position.y < 10) {
      transform.position = new Vector3(0, transform.position.y+1, 0);

      //yield returns and stops / resumes execution at this point.
      yield return null;
   }

   //Stops here again, and resume in 3 seconds
   yield return new WaitForSeconds(3f);

   //Coroutine ends.
}
```

Coroutines (2/2)

```
using UnityEngine;
using System.Collections;

public class PropertiesAndCoroutines : MonoBehaviour
{
   public float smoothing = 7f;
   public Vector3 Target
   {
     get{ return target; }
     set
     {
         target = value;

         //Stops a coroutine with the given method name.
         StopCoroutine("Movement");
         StartCoroutine("Movement", target);
      }
   }

   private Vector3 target;

   IEnumerator Movement (Vector3 target)
   {
      while(Vector3.Distance(transform.position, target) > 0.05f)
      {
         transform.position = Vector3.Lerp(transform.position, target,
               smoothing * Time.deltaTime);
         yield return null;
      }
   }
}
```

Good Coding Practices

Guidelines to keep the code maintainable, extensible, and readable.

- **Single responsibility:** A class should be responsible for a single task. Example:
  - *Player.cs*: Control of input, physics, weapons, health, inventory.

  vs.
  - *PlayerInput.cs*: Manages input for the player.
  - *PlayerWeapons.cs*: Weapon system for the player.
  - *PlayerInventory.cs*: Inventory for the player.
  - ...
- **Dependency Inversion:** If a class A depends on B, B should be an interface or an abstract class.
- **Modularization:** behaviours that are independent should be placed in separated function, or even classes.

More details here:

`http://unity3d.com/learn/tutorials/modules/intermediate/scripting/coding-practices`

Outline

Lab Preview

The first time you open Unity, you'll be prompted to choose a license. For CE318, we'll be using the *Free License* option. Additionally, you need to have a Unity3D account to develop games in Unity (this is also free). You can create the account when you open Unity for the first time, or on the website:

`https://accounts.unity3d.com/sign-up`

It'd be great if you could **create your account before the lab** this week, so you can directly get your hands dirty with the assignment.

During this and next week's lab you will create a 3D game:

- The game will consist of rolling ball that collects items and collides with some boundaries in the level.
- This is the first part of the two-weeks first assignment. You will be requested to complete certain points detailed in the instructions.

Next lecture: Math in 3D Games and Input.