CE318/CE818 High-Level Games Development

Introduction – Lecture 1

Dr Aikaterini (Katerina) Bourazeri

16th October 2023

University of Essex

Introduction

Office Hours (1NW.3.19):

- Tuesday 11am - 12pm
- Friday 11am - 12pm

Email: `a.bourazeri@essex.ac.uk`

Introduction

- A **key** module for BSC G610 Computer Games, BSC G612 Computer Games (Including Year Abroad) and BSC I610 Computer Games (Including Placement Year).
- A **8**–week module (**16** hours of lectures) that counts as **15** credits (7.5 ECTS).
- **16** hours of labs (8 labs).

Module Description

The CE318/CE818 Module:

- teaches the main programming and modelling techniques required to implement a non-trivial 3D game,
- has a significant laboratory content and the practical aspects will be taught using a game development platform.

No previous game development experience is needed, although having object-orientated programming knowledge is strongly advised (all programming will be done in C#.

Module Overview

This module will teach you the fundamentals of games console programming. We will focus on 3D games, developed using **Unity3D 2022.3.8f1**. Topics covered include:

- Creating 3D Games in Unity3D
- Use of the Unity3D editor
- 2D-3D Math Game Concepts
- Managing Player Input
- 3D Animations
- Cameras and Navigation

Module Overview

- Graphical User Interfaces
- Lights and Audio
- Particle Systems
- Terrains
- Game Design
- 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.

Learning Outcomes

- Demonstrate an understanding of the software architecture of 3D game.
- Design and implement a 3D game.
- Implement AI behaviours for bots or non-player characters.
- Design and implement graphic effects.
- Design and implement game objects (e.g. weapon systems).
- Demonstrate an understanding of advanced techniques in game development.

Module Information

- Recap of essential mathematics for 3D games, how to implement the associated routines, and to use them in existing libraries.
- Software architecture for games.
- Game content and the content pipeline.
- 3D modelling and simulation, physics modelling, detecting and reacting to collisions, lighting, cameras and scene graphics.
- Game mechanics, efficiency tuning and the game loop.
- Case study: from design to implementation of a complete 3D game.
- Tips and tricks for ensuring your game meets the required frame rate.
- Analysis of inefficient program code and how to fix it.

Assessment

1. Bi-weekly Assignments (10%)
    - 4 lab exercises
2. Progress Tests (2 × 15%)
3. Assignment Part I (20%) – *(10 November 2023)*
    - Game Prototype
    - Game Design Document
4. Assignment Part II (40%) – *(15 December 2023)*
    - Fully Developed Game
    - Final Report
    - Case Study (CE818 students)

All assignments in CE318/818 are ***individual****.*

Recommended Books

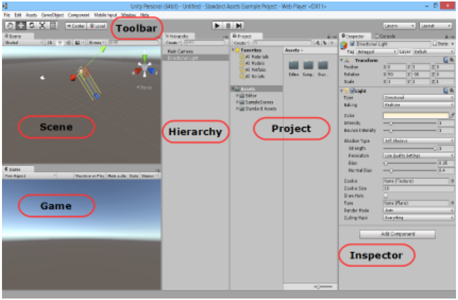
- Harrison Ferrone: Learning C# by Developing Games with Unity 2021 (2021)
- Franz Lanzinger: 3D Game Development with Unity (2022)
- Ian Millington: AI for Games (2019)

What is Unity?

- Game engine – A *tool* for building games.
- Allows exporting games for multiple platforms.
  - ∗ Not so much for the editor...
  - ∗ (Sorry Linux users)
- Currently very popular for game development.
  - ∗ Don't assume that it will always be the case...
- Supports 2D and 3D games.
- A large amount of tutorials, guides and videos.
- Fairly fast release cycle.
  - ∗ Take note – We will not accept 'works on my machine'.

The Unity Interface

- **Hierarchy:** lists all the objects present in the scene.
- **Scene:** displays the content of a scene.
- **Game:** visualises the scene as it will appear in the game.
- **Inspector:** displays information on the object currently selected.
- **Project:** includes all the available assets.



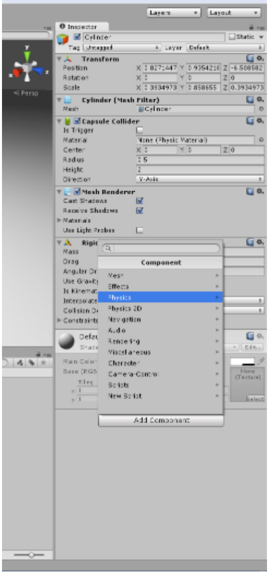
Key Terms

Game objects are the **stuff** that make up games in Unity. There are some key concepts attached to them:

- **Components** –Things that make up game objects.
- **Prefabs** – Preconfigured game objects which you can use later.
- **Tags** – Used to identify game objects.
- **Layers** – Group game objects together and apply rules to them.

Components

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



☑ Cylinder □ Static
Tag Untagged Layer Default

Transform
Position X 0.8271447 Y 0.9354218 Z -6.508582
Rotation X 0 Y 0 Z 0
Scale X 0.3934973 Y 0.858655 Z 0.3934973

Cylinder (Mesh Filter)
Mesh Cylinder

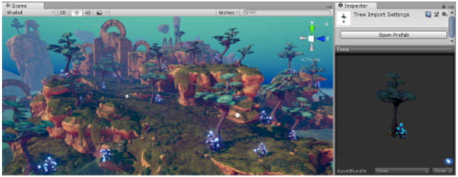
☑ Capsule Collider
Is Trigger □
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0.5
Height 2
Direction Y-Axis

☑ Mesh Renderer
Cast Shadows ☑
Receive Shadows ☑
Materials
Use Light Probes □

Prefab Assets

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

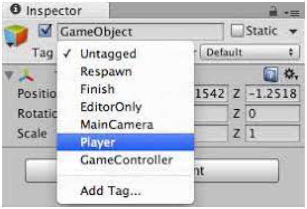
- It acts as a template for that game object.
- Any edits made to a prefab asset are immediately reflected in all instances produced from it.
- It is possible to override components and settings for each instance individually.



Names and Tags

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

- **Tag** – a general descriptor of an object, e.g., "level" or "enemy" (`GameObject.FindWithTag`).
- **Name** – a specific object name, e.g., "robot" (`GameObject Find(string name)`).

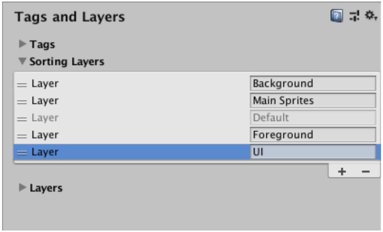


For performance reasons it is recommended to use `GameObject.FindWithTag`.

Layers

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

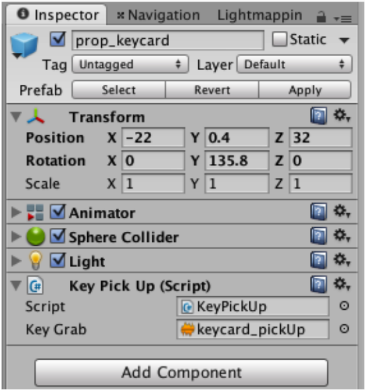
- by cameras to render only a part of the screen.
- by raycasting to selectively ignore colliders.
- by lights to illuminate the scene.
- to determine collisions (Layer-based Collision).
- to determine the order in which sprites are rendered.



Layer	Background
Layer	Main Sprites
Layer	Default
Layer	Foreground
Layer	UI

Attaching Scripts

- Scripts allow you to add behaviours to game objects.
- They are **components**.
- Therefore, we need to **attach** them to a **Game Object** to be used.



☑ 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

Script Notes

- You can have **multiple** scripts per game object.
- You *should* separate out code by function (i.e. 1 script = 1 function)
- When you create a new script, you get a **template**.
- Scripts make heavy use of **event functions**.

Event Functions

- When something happens in our game, Unity will let our script know by invoking functions on it.
- The **[Unity Documentation]** outlines these.
- The **[Order in which they are invoked]** is also covered.

Initialisation

- Awake() – initialises any variables or game state before the game starts. Awake() is called only once during the lifetime of the script instance.
- Start() – Called at the start of the scene. Like the Awake function, it is called exactly once in the lifetime of the script, but the script should be enabled.

Regular Updates

- Update() – Called every time the screen is refreshed (once per frame).
- FixedUpdate() – Called when dealing with Rigidbody; when adding a force to a rigidbody, you have to apply the force every fixed frame inside FixedUpdate() instead of every frame inside Update().
- LateUpdate() – LateUpdate() is identical to Update() in terms of run frequency (both run once per frame), but LateUpdate() runs after all Update functions; used to modify animated model bones or to implement a smooth camera follow.

GUI Events

Events correspond to user input (key presses, mouse actions).

- OnGUI() – Called for rendering and handling GUI events.
- OnMouseDown(), onMouseEnter(), onMouseOver() – For mouse events related to GUI components.

Physics Events

- OnCollisionEnter(), OnCollisonStay(), OnCollisionExit() – For collider collisions.
- OnTriggerEnter(), onTriggerStay(), onTriggerExit() – For collisions with triggers.

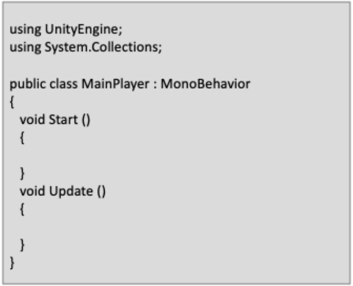
An easy way to differentiate between the two is to think of them **visually**. *OnCollisionEnter()* can be visualised as *colliding against a wall*, and *OnTriggerEnter()* can be visualised as *triggering an alarm*.

Language Choices

- Unity lets you use a few different languages for scripting:
  - * C#
  - * UnityScript (Javascript)
  - * Boo
- We'll be using C# for this module...

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 : MonoBehavior
{
  void Start ()
  {

  }
  void Update ()
  {

  }
}
```

Getting Object Objects

If we need to access other game objects, the *static* methods can help us:

- `GameObject Find(string name);`
  - Uses the name defined in the editor.
  - It is slow, so don't use it in every frame.
  - Returns *null* if no object exists.
- `GameObject FindWithTag(string name);`
  - Returns an *active* game object tagged with *tag*.
  - Returns *null* if no object exists.
  - Throws *UnityException* if *name* is not defined as a tag.
- `GameObject[] FindGameObjectsWithTag(string tag);`
  - Returns *active* game objects tagged with *tag*.
  - Returns *empty array* if no object exists.
  - Throws *UnityException* if *name* is not defined as a tag.

Examples

```
GameObject.Find("SomeGuy");

GameObject.FindWithTag("Player");

GameObject.FindGameObjectsWithTag("Enemy");
```

Destroying Objects

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 is 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 and Deactivating (Why?)

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:

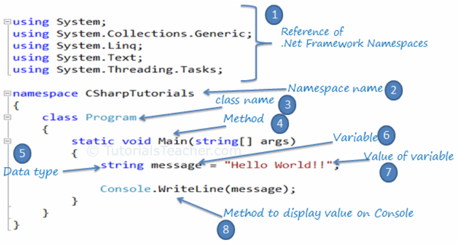
- **`bool`** `Behaviour.enabled:` **`true`** to *enable* a component, **false** to disable it.
- Enabled Behaviours (components) are Updated, disabled Behaviours aren't.

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 recommends Visual Studio for editing scripts.

The Basics of C#: From a Java Perspective

- 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.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpTutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = "Hello World!!";

            Console.WriteLine(message);
        }
    }
}
```

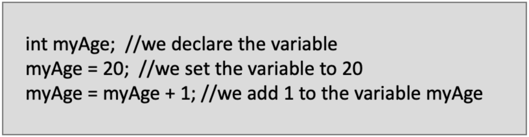
Figure 1: The Basics of C#

The Basics of C#: Variables

A variable is a container; it includes a value that might change overtime.

When using variables, we need to:

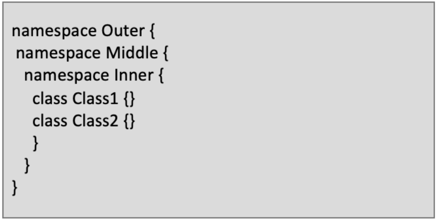
1. declare the variable by specifying its type,
2. assign a value to this variable,
3. possibly combine the variable with other variables using operators.



```
int myAge;  //we declare the variable
myAge = 20;  //we set the variable to 20
myAge = myAge + 1; //we add 1 to the variable myAge
```

The Basics of C#: Namespaces

Namespaces are like packages; they are used to organise code into logical groups. You can create a hierarchical organisation of your code by nesting namespaces:

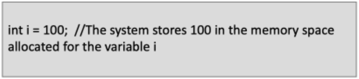


```
namespace Outer {
 namespace Middle {
  namespace Inner {
    class Class1 {}
    class Class2 {}
    }
  }
}
```

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

The Basics of C#: Value and Reference Types

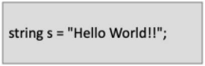
Value type: holds a data value within its own memory space (variables of these data types directly contain values).



```
int i = 100;  //The system stores 100 in the memory space
allocated for the variable i
```

These data types are all of value type: bool, byte, char, double, float, int, etc.

Reference type: doesn't store its value directly. Instead, it stores the address where the value is being stored.



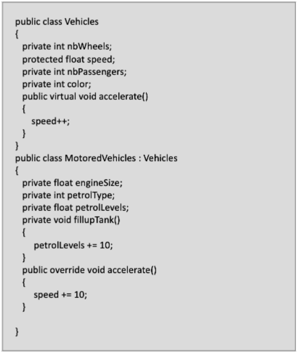
```
string s = "Hello World!!";
```

These are reference type data types: string, arrays, class, etc.

The Basics of C#: Inheritance

The main idea behind inheritance is that objects can inherit their properties from other objects (their parent).

- As they inherit these properties, they can remain *identical* or *evolve* and *overwrite* some of these properties.
- This makes possible to *minimise* our code by creating a parent class with general properties, and customise some of these properties for the children.

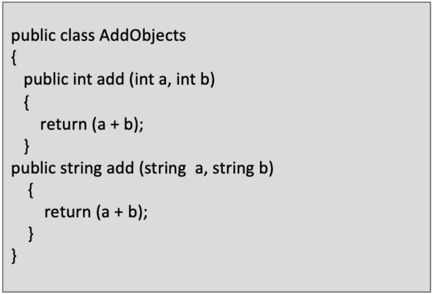


```
public class Vehicles
{
  private int nbWheels;
  protected float speed;
  private int nbPassengers;
  private int color;
  public virtual void accelerate()
  {
     speed++;
  }
}
public class MotoredVehicles : Vehicles
{
  private float engineSize;
  private int petrolType;
  private float petrolLevels;
  private void fillupTank()
  {
     petrolLevels += 10;
  }
  public override void accelerate()
  {
     speed += 10;
  }

}
```

The Basics of C#: Polymorphism

Polymorphism: **poly** (several) and **morph** (shape); the ability to process objects differently depending on their data type or class

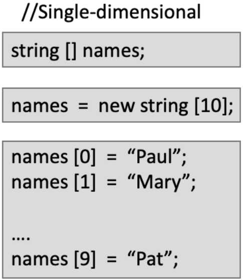


```
public class AddObjects
{
  public int add (int a, int b)
  {
     return (a + b);
  }
public string add (string  a, string b)
   {
      return (a + b);
   }
}
```

It is possible to add two different types of data: integers and strings. *Depending on whether two integers or two strings are passed as parameters, we will be calling either the first* **add** *method or the second* **add** *method.*

The Basics of C#: Arrays

When creating arrays, we can create single-dimensional arrays and multidimensional arrays.



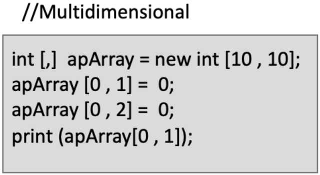
```
//Single-dimensional

string [] names;

names  =  new string [10];

names [0]  =  “Paul”;
names [1]  =  “Mary”;

....
names [9]  =  “Pat”;
```



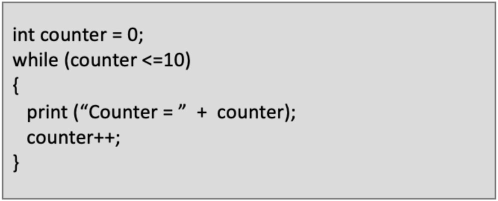
```
//Multidimensional

int [,]  apArray = new int [10 , 10];
apArray [0 , 1] =  0;
apArray [0 , 2] =  0;
print (apArray[0 , 1]);
```

The Basics of C#: Loops

There are times when you have to perform repetitive tasks as a programmer: loops are structures that will perform the same actions repetitively based on a condition. The process is:

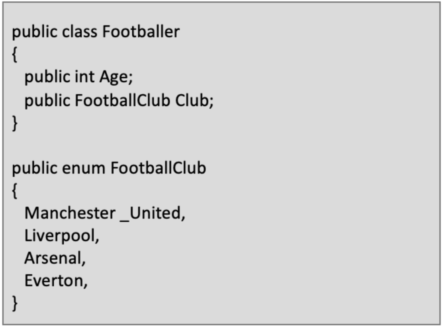
- Start the loop,
- Perform actions,
- Check for a condition,
- Exit the loop if the condition is fulfilled or keep looping.



```
int counter = 0;
while (counter <=10)
{
  print (“Counter = ”  +  counter);
  counter++;
}
```

The Basics of C#: Enumerations

Enumerations are user-defined value types used to represent a list of named integer constants. It is created using the enum keyword inside a class, structure, or namespace.



```
public class Footballer
{
  public int Age;
  public FootballClub Club;
}

public enum FootballClub
{
  Manchester _United,
  Liverpool,
  Arsenal,
  Everton,
}
```

It improves a program's readability, maintainability and reduces complexity.

Quiz!

https://forms.office.com/r/wEyQ9EGAKV