|  |
| --- |
| Continuing Education Program |
| Introduction to Unity |

1.What is Unity?

|  |
| --- |
| **Copyright Notice**  Copyright © 2013 DigiPen (USA) Corp. and its owners. All rights reserved.  No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052  **Trademarks**  DigiPen® is a registered trademark of DigiPen (USA) Corp.  All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged. |
|  |

Unity is a combined game-engine/IDE (integrated development environment) that allows to code and release in most contemporary game platforms.

Unity provides:

-A game engine (manages game loops, memory allocation, rendering and other low level behavior).

-A powerful API accessible through three different programming languages: C#, unityScript and Boo.

-Access to a subset of the .NET framework (most importantly file IO and Collections, but many more).

-PhysX integration.

-Shuriken (particle system) integration.

-Mechanim (animation system) integration.

-Plugin and third party support for extensions.

-A game scene and asset editor.

Unity main development goal is 3D games, although many 2D games use the engine nonetheless.

But most importantly, Unity provides a really straightforward art pipeline as well as rapid iteration capabilities, significantly reducing not only the technical team requirements for game projects, but also the managing overhead.

Is this ease of use that has made Unity by far the most used engine in mobile and small studio development.

In order to get a first look at the engine, it´s useful to divide it in three parts: the pure engine, the scripting system and API, and the Editor.

2. Unity Game engine:

First let´s take a look at how the engine looks from inside.

Unity is an object oriented, managed engine. That means it provides automatic garbage collection and memory allocation.

It is single threaded, although extra threads can be created through scripting. You can’t access engine functions, though (no access to the unity API when you’re not in the same thread).

It has a component paradigm. Every game object (entity with a position in the world) has a transform and any number of components. It is those components what regulate the behavior of objects. Everything, from physics to rendering, is done through components. Through scripting you can create new components to control game logic or to modify the behavior of in-built components.

Each frame, Unity goes through a series of update callbacks for every active and registered game object. Game objects can ONLY be instantiated as part of the current scene hierarchy, so to unreferenced a game object, you need to destroy it. Regular objects (created through scripting) do not participate in the hierarchy, and they will be destroyed as soon as no other object in active context references them.

It’s worth looking closely at Unity´s update loop calls, since there are a few of these, and not all of them is called every frame. These are called PER COMPONENT. By overriding these functions, you code your main logic.

The order of updates (generally –awake functions can change wildly-) is:

INITIALIZATION FUNCTIONS (note that these do override the concept of a constructor. You can use constructors in Unity, but not for components, which are added in editor or through API functions that don´t allow for parameters).

**Awake()**

Called before any object is made active OR when a function from a component in the object is called.

**OnEnable()**

Similar to Awake but called only if the object is enabled (active). That means function calls to components do not trigger the function. This is called EVERY TIME the object is enabled, not just the first time.

**OnLevelWasLoaded()**

Called if the level was just loaded this frame.

**Start()**

Called before the first Update function is ever called on the object. Only once.

UPDATE FUNCTIONS:

Every loop Unity looks at its execution time and decides whether a frame is to be drawn or whether it also requires a physics update.

**FixedUpdate()**

FixedUpdate is in charge of managing game logic. It is called BEFORE a physics update to the system, which can coincide with a drawn frame or not (either because there are many physic updates per drawn frame –slow draw- or many drawn frames per physics update –slow logic/physics-).This is called at a reliable time and if you avoid deltaTime calculations you can still get coherent behavior. It is good to use deltatime for consistency with physics, though.

**Update()**

Update is called before a draw call. Therefore time steps are not consistent between update calls. Everything related to rendering and not gameplay critical code goes here (physics for particle effects, for example).

**LateUpdate()**

Provided for ease of use, it is guaranteed to be called after all the Update() are called. This allows consistency, for example, for camera control (so you are sure nothing moves the target after the camera has set itself). The functionality provided by LateUpdate can also be obtained through changing the script execution order in the editor except for the fact LateUpdate executes AFTER animations are advanced to their current frame.

DESTRUCTION FUNCTIONS

**OnDisable()**

Called every time an object is disabled AFTER all update functions are called for that frame for the object.

**OnDestroy()**

Called before an object is destroyed but after all current frame update functions for the object.

**OnApplicationQuit()**

Called on every active object when you quit the program. Useful, for example, for autosave functions.

On top of this functions there are some rendering dependent callbacks that can be used to control rendering of objects.

Corroutines started through the unity API (since the engine, remember, is not multi threaded) execute their next update cycle after all other updates.

However the functions shown above are by far the most commonly used in every Unity project.

3.Scripting languages:

Scripting in Unity is a misnomer. In reality, what unity provides are three different memory managed, object oriented, fully developed programming languages. All three languages are hooked to Unity’s API and a subset of the .NET framework Unity supports.

Originally Unity used UnityScript (a language with java-like syntax). As the engine developed, they embraced C#, which has become the preferred language to work in Unity. Unity´s implementation of C# is at least 2.0 compatible, but does not include the full .NET framework. This is because Unity does NOT compile to CLR or Mono (multiplatform port of .NET). Unity scripts, although Mono based, compile to machine code. This actually makes Unity´s C# code incredibly fast and very similar in performance to native C++ code except when dealing with memory (since memory management and garbage collection has an overhead).

The use of a powerful language like C#, access to the .NET framework, and close to native performance makes Unity very reliable. You can replicate any part of the engine (except for very specific, third party code) by using C# code. Sometimes, programmers coming from more traditional development environments go out of their way to provide functionality in code similar to that found in the Unity editor.

However, even if doable, this is WRONG. Refusing to use the power of the environment is like cutting a tree with a chainsaw by dully chopping it instead of turning it on.

Try to keep code generation of objects and initialization of components to a minimum.

Since components can be scripts, there has to be a class every script inherits from. This is the MonoBehavior class, and it´s the class that has all the update callbacks discussed above.

In C# we need to explicitly extend from MonoBehavior. However, every time we create a script, Unity does generate the relevant code to get the class started.

This is how a newly created C# file looks:

using UnityEngine;

using System.Collections;

public class InvaderGroupController : MonoBehaviour {

// Use this for initialization

void Start () {

}

// Update is called once per frame

void Update () {

}

}

At first look this looks like standard C# code (even explicitly importing Unity´s API). However, on a closer look we see that Update and Start are neither public nor overridden. Unity uses reflection to see whether a MonoBehavior implements each update callback and doesn’t even add those components to the callbacks list if they don´t. It even detects if the callback has any meaningful code inside and ignores it if it doesn’t. Unity components have VERY little overhead.

Most of the time you don´t need the full power of C# to work in Unity.

C# PRIMER:

C# is not very different in syntax to c++, except that it does not use pointers.

BASIC SYNTAX:

Variables are declared like this:

public float \_variablename = 1.0f;

And functions like:

public abstract float \_FunctionName (float \_parameterName, int \_otherparameterName);

Built in types (the most important of which are int, bool, float and string) are value types (except for string, but comparers are overloaded to work like if it was a value type), while any other type, unless inheriting from System.ValueType is a reference type. Reference types work as pointers. Equality only returns true if the same object (same memory) is pointed and changes to the object from inside a function do change the original object.

Note that we will use float and not double. Although most processors now are 64 bits, Unity´s API is based on floats.

Basic control loops are:

while (expression) statement

switch (expression)

{

case constant-expression:

statement

jump-statement

[default:

Statement

jump-statement]

}

Jump statements are

goto

break

if (expression)

statement1

[else

statement2]

for ([initializers]; [expression]; [iterators]) statement

foreach (type identifier in expression) statement

this last one is an special case. It allows you to iterate through an enumerator of any object that implements a GetEnumerator() function. It is so useful it gets it´s own keyword (although it´s more part of the framework than of the language.

To code, Unity comes with it´s own version of MonoDevelop. It is possible to integrate Unity with Visual Studio or any other IDE.

4. EDITOR:

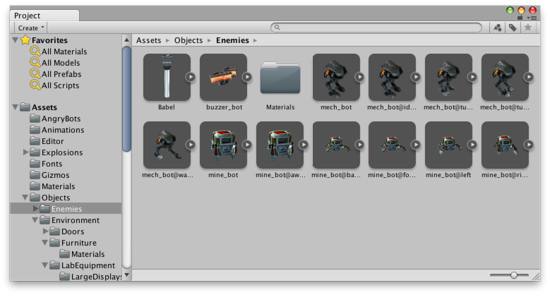
Unity’s most powerful feature, and where the engine and scripting come together, is in the editor.

The main areas of the editor are:

PROJECT BROWNSER:

This is were the project files are organized and WERE YOU CREATE AND MOVE ELEMENTS WITHIN THE PROJECT FOLDER.

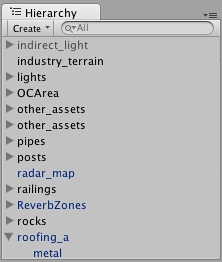
Moving elements in the finder can lead to loss of metadata in version controlled projects.



To create a script, material or prefab, right click on the project brownser.

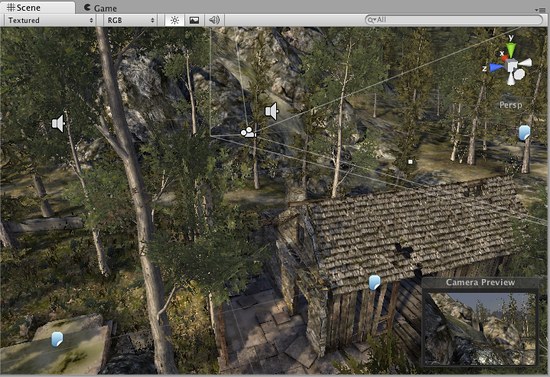
Also notice the little search box on top. This will become very useful to locate files once your project becomes bigger.

HIERARCHY



This is the object hierarchy of the current scene. GameObject can have children GameObjects that will be affected by the transform modifications of the father (and other things).

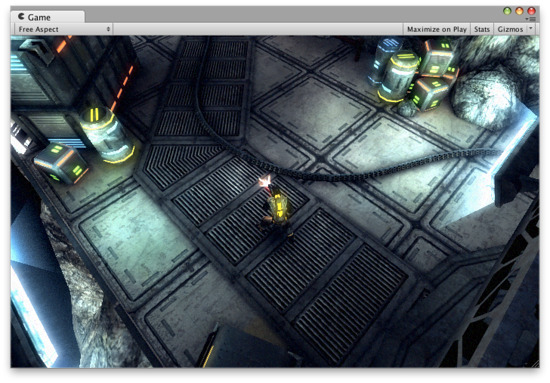
SCENE VIEW



A three-dimensional view of the current scene. This is where level design takes place.

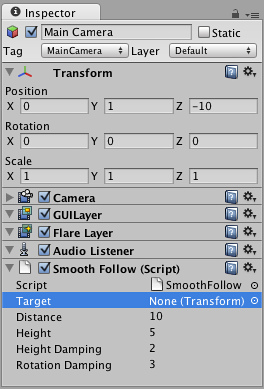
C:\Program Files (x86)\Unity\Editor\Data\Documentation\Documentation\Images\manual\SceneView40-1.jpg And these are the controls to move, rotate and scale the objects in the scene.

GAME VIEW



It shows the output of the main camera when the game is not playing or the game screen when the game is playing.

INPECTOR:



The inspector shows the currently selected object in the hierarchy (more specifically, its components, including the transform). This is where you assign variables (assigns are saved when you save the scene).

EVERY VARIABLE DECLARED AS PUBLIC APPEARS ON THE INSPECTOR.

Non-public variables appear as greyed out in debug view.

Only basic types and Unity’s API classes appear in the inspector (collections, for example, don’t, although arrays do).

It is possible to code your own views and editor for custom types.

5. DOCUMENTATION:

Unity has very decent documentation, including a basic manual. Although we´ll go over specific systems, it´s important you get used to the scripting reference to be able to slowly learn the API.

6. USING UNITY WITH EXTERNAL SOURCE CONTROL

First, go to project settings -> Editor and set version control to Meta files.

Save.

Create the project repository. In your version control exclude:

Folders:

Library

Temp

Files:

\*.csproj

\*.pidb

\*.unityproj

\*.sln

\*.userprefs