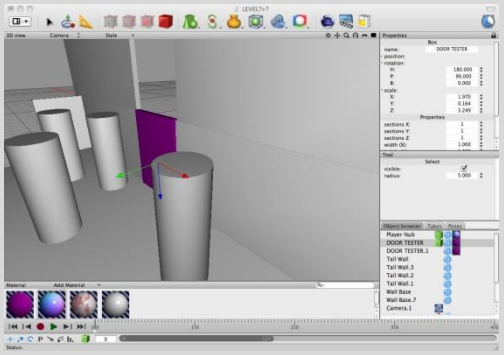




Tutorial Básico - UNITY

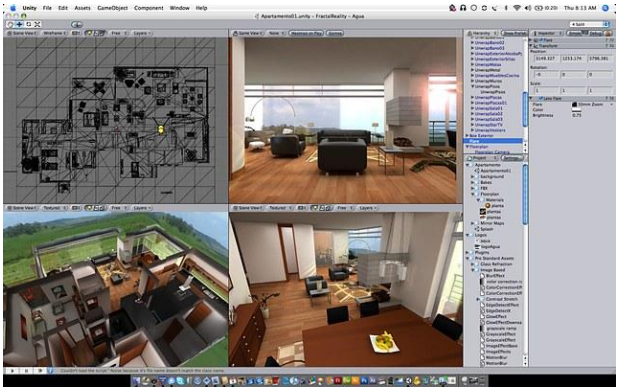
Autor:

- Huanay Martínez, Franz calhoun_123@hotmail.com

Contenido

I. Introducción

Unity es una herramienta de programación integrada para la creación de video juegos 3D u otros contenidos interactivos como visualizaciones arquitectónicas o animaciones 3D en tiempo real.



Visualización usada por arquitectos.

Simulación en tiempo real.

Video juegos en Unity pueden ser creados en 2D y 3D.

Características del Software

Desarrollador: Unity Technologies.

Última edición: 3.0.0 f5 / 27 de setiembre, 2010.

Sistemas Operativos: Windows, Mac OS X, Wii, iPhone/iPad, Xbox 360, Android, PS3.

Tipo: Game Engine.

Website: www.unity3d.com

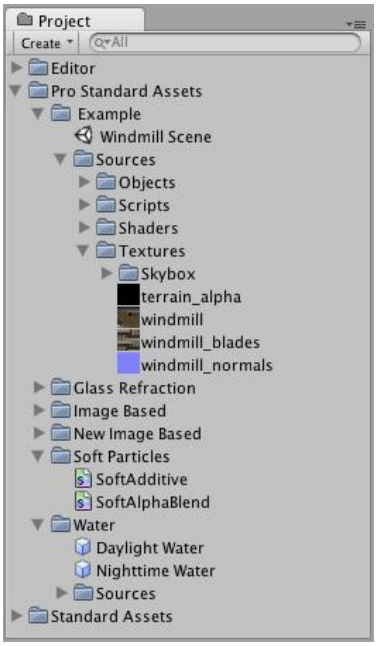
II. Interfaz gráfica - Basics

Ahora se explicarán detalles de la interfaz gráfica para realizar operaciones básicas y no tener problemas al navegar a través de los ambientes que ofrece el Unity.

Main Editor Window

La ventana del editor principal (***Main Editor Window***) está compuesta de varias sub-ventanas, llamadas vistas (***views***). Hay varios tipos de vistas en Unity, cada una con un propósito específico.

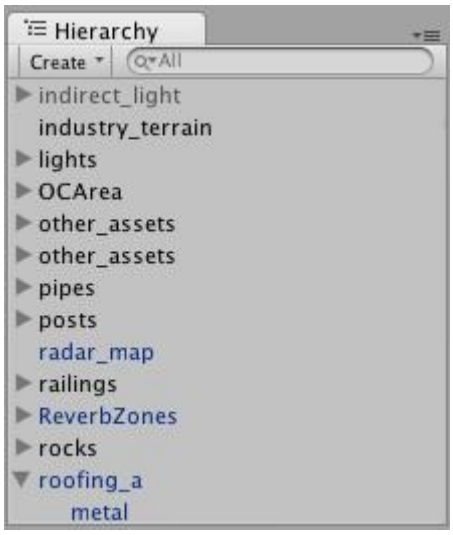
Project View



Todo proyecto en Unity tiene una carpeta de Propiedades (***Assets***). El contenido de esta carpeta es presentado en la vista de proyecto (***Project View***). Aquí es donde se puede guardar todas las propiedades necesarias para crear tu juego, como ***scenes***, ***scripts***, ***3D models***, ***textures***, ***audio files*** y ***Prefabs***.

Nota importante: Es recomendable nunca mover ***project assets*** usando el sistema operativo, ya que se podrían romper vínculos entre la data usada por el Unity. Se debe usar siempre el ***Project View*** para organizar los ***assets***

Hierarchy



Hierarchy contiene todo ***GameObject*** en la actual escena. Algunos de estos son aplicaciones directas de los ***assets files*** como ***3D models***, y otras son aplicaciones de los ***Prefabs***. Se pueden seleccionar y vincular objetos aquí. Mientras los objetos son añadidos y removidos de la escena, ellos aparecerán y desaparecerán del ***Hierarchy***.

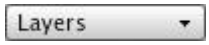
Toolbar

El ***Toolbar*** consiste en cinco controles básicos; cada uno relacionado a diferentes partes del Editor.

Transform Tools – usado con el Scene View.

Transform Gizmo Toggles – Afecta la forma del Scene View.

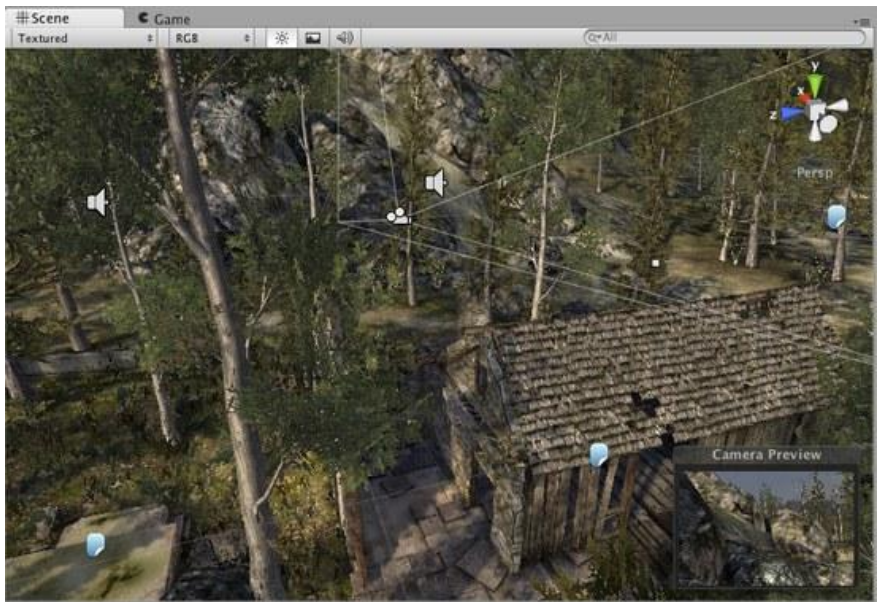
Play/Pause/Step Buttons – usado con el Game View.



Layers Drop-down – controla objetos que son mostrados en el Scene View.

Layout Drop-down – controla la disposición de las vistas.

Scene View



El ***Scene View*** es una ventana interactiva. Éste se puede usar para seleccionar y posicionar los ambientes, el jugador, la cámara, los enemigos, y otros ***GameObjects***. Maniobrar y manipular estos objetos dentro del ***Scene View*** es una de las funciones más importantes en el Unity.

Navegación a través del Scene View

En ésta vista podemos usar el ***Hand Tool***, especialmente aprovechar el mouse lo más posible.

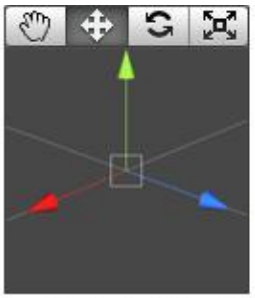

Hacer click y arrastrar la cámara sobre cualquier parte del espacio.

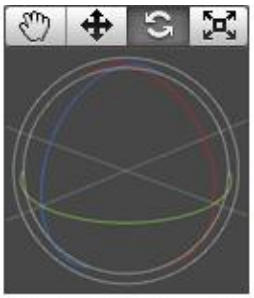

Mantener **Alt** y hacer click y arrastrar el orbita de la cámara por el punto pivote actual.

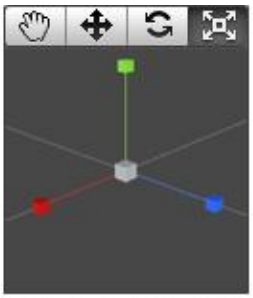

Mantener **Control** (sólo en Mac) and hacer click y arrastrar para ampliar la cámara.

También es posible cambiar el ***Scene Camera*** a modo isométrico o tan sólo haciendo click en cualquiera de los brazos se puede girar la cámara a las direcciones X, Y o Z.

Posicionamiento de GameObjects


Translate Tool - Hotkey "W"


Rotate Tool - Hotkey "E"

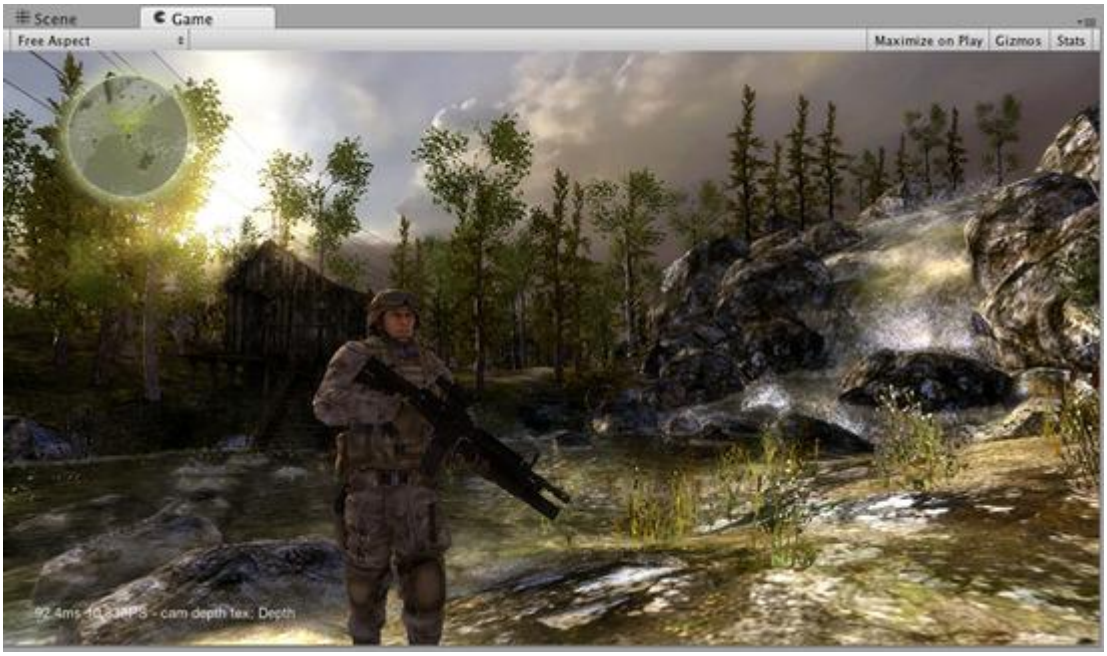

Scale Tool - Hotkey "R"

En el proceso de construir los juegos, se tendrá que ubicar varios objetos en el mundo del videojuego. Para hacer esto, debemos usar las ***Transform Tools*** en el ***Toolbar*** para trasladar, rotar y dar escalas a ***GameObjects***. Cada uno tiene un correspondiente ***Gizmo*** que aparece alrededor del ***GameObject*** seleccionado en el ***Scene View***. Se puede usar el mouse y manipular cualquier ***Gizmo axis*** para modificar el ***Transform component*** del ***GameObject***, o también se pueden introducir valores directamente al ***Transform component*** que se encuentra en el **Inspector**.

Barra de control del Scene Bar

La barra del control del ***Scene View*** nos permite ver la escena en varios modos de vista como ***textured***, ***wireframe***, ***RGB***, ***overdraw***, y otros. Éste permitirá ver y oír iluminación in-game, elementos de juego, y sonidos.

Game View



El ***Game View*** es generado desde la(s) cámara(s) en el juego. Éste es representativo del producto final. Se necesitará usar una o más cámaras para controlar lo que el jugador observa realmente cuando están jugando el juego.

Modo Play

Los botones en el ***Toolbar*** se usan para controlar el editor ***Play Mode***, así se observará como funcionará el producto final. Durante el ***Play Mode*** se esté ejecutando, todos los cambios que se realicen serán temporales, y serán inicializados de nuevo al salir del ***Play Mode***.

Barra de control del Game View

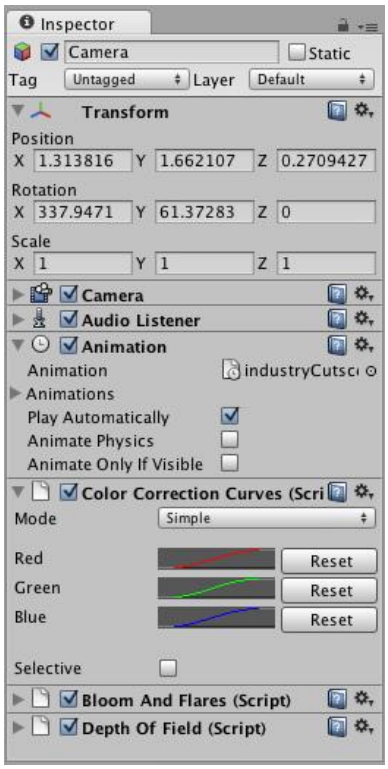
Free Aspect: Permite cambiar las dimensiones del ***Game View***.

Maximize on Play: El ***Game View*** se maximizará al 100% (vista en toda la pantalla) una vez en el ***Play Mode***.

Gizmos: Permite observar todos los ***Gizmos*** del ***Scene View***.

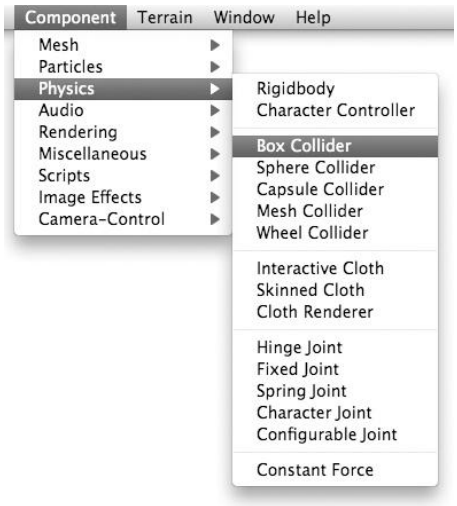
Stats: Muestra las estadísticas del renderizado, es útil para optimizar la performance de los gráficos.

Inspector



Los juegos en Unity están compuestos de multiples ***GameObjects*** que contienen ***meshes***, ***scripts***, ***sounds*** y otros elementos. El ***Inspector*** muestra información detallada sobre el ***GameObject*** seleccionado, incluyendo sus elementos seleccionados y sus propiedades.

Cualquier propiedad puede ser modificada directamente. Incluso las variables de los scripts pueden ser modificadas sin cambiar el script. Se pueden cambiar las variables mientras el juego está corriendo para realizar pruebas. En un script, si se definió una variable pública de un tipo objeto (como ***GameObject*** o ***Transform***), se puede arrastrar y soltar el ***GameObject*** o ***Prefab*** dentro del ***Inspector*** para asignarlo(s) como variable(s) del script.



Se pueden añadir componentes a cualquier ***GameObject*** o ***Prefab*** de acuerdo a las necesidades del usuario.

Otras Vistas

Las vistas previamente explicadas nos mostraron la interface básica en Unity. Hay otras vistas que se pueden utilizar para tareas más específicas, tenemos las siguientes vistas:

Console: Muestra mensajes, advertencias y errores.
Animation View: Usado para animar objetos en la escena.
Profiler: Usado para investigar y encontrar el performance de los cuellos de botella en el juego.
Asset Server View: Usado para administrar versión del proyecto usando el ***Asset Server*** del Unity.
Ligthmapping View: Usado para administrar iluminación de mapas.
Occlusion Culling View: Usado para administrar el ***Occlusion Culling*** y mejorar su performance.

III. Scripting en Unity – Basics

El Scripting es una parte esencial de Unity ya que define el comportamiento del juego.
En ésta parte se introducirá los fundamentos del scripting usando JavaScript.

Convenciones del Unity

Variables (variables)**:** Empiezan con una letra minúscula. Las variables se usan para almacenar información sobre cualquier aspecto de un estado de juego.
Functions (funciones)**:** Empiezan con una letra mayúscula. Las funciones son bloques de códigos que han sido escritos una vez y que se pueden rehusar tantas veces como sea necesario.
Classes (clases)**:** Empiezan con una letra mayúscula. Éstos pueden tomarse como colecciones de funciones.

Consejo: cuando se lea un código en Unity, se debe prestar mucha atención a la primera letra de las palabras. Esto ayudará a comprender mejor la relación entre los objetos.

Scripts comunes

Cuando se crea un nuevo JavaScript, por defecto éste contiene una función Update(). En esta sección trataremos otras opciones comunes disponibles, simplemente reemplazando el nombre de la función Update() con uno de la lista inferior.

FixedUpdate()
El código situado dentro de esta función se ejecuta en intervalos regulares (a un ritmo de presentación de frames fijado). Es muy común usar este tipo de función cuando aplicamos fuerzas a un "Rigidbody" (cuerpo rígido).

```
// Aplica una fuerza hacia el cuerpo rígido cada frame.
function FixedUpdate () {
rigidbody.AddForce (Vector3.up);
}
```

Awake()
El código dentro de esta función es llamado cuando se inicia el script.

Start()
Se llama antes de cualquier función Update, pero después de la función Awake(). La diferencia entre las funciones Start() y Awake() es que la función Start() es llamada únicamente si el script es habilitado (si su 'checkbox' está habilitada en la Vista de Inspector).

OnCollisionEnter()
El código dentro de esta función se ejecuta cuando el objeto de juego al que pertenece el script colisiona con otro objeto de juego.

OnMouseDown()
El código se ejecuta cuando el ratón hace clic sobre un objeto de juego que contiene un 'GUIElement' (elemento GUI) o un 'Collider'.

```
// Carga el nivel llamado "SomeLevel" como respuesta
// al hacer click en el objeto.
function OnMouseDown () {
Application.LoadLevel ("SomeLevel");
}
```

OnMouseOver()

El código aquí dentro se ejecuta cuando el ratón se coloca sobre un objeto de juego que contiene un 'GUIElement' (elemento GUI) o un 'Collider'.

```
// Desvanece a cero el componente rojo del material
// mientras el mouse está sobre la malla.
function OnMouseOver () {
renderer.material.color.r -= 0.1 * Time.deltaTime;
}
```

Operaciones comunes en el scripting

Posición y Rotación

La mayoría de las manipulaciones de ***GameObjects*** son hechas a través del componente ***Transform*** y/o ***Rigidbody*** de estos. Estos son accesibles dentro de los ***behaviour scripts*** a través de las variables miembro ***transform*** y ***rigidbody*** respectivamente.
Si se quiere rotar un objeto 5 grados alrededor del eje Y cada ***frame*** se podría hacer escribiendo el siguiente código:

```
function Update() {
transform.Rotate(0, 5, 0);
}
```

Si se quiere mover un objeto hacia otro punto se podría escribir el siguiente código:

```
function Update() {
transform.Translate(0, 0, 2);
}
```

Tiempo

La clase tiempo contiene una importante variable de clase llamada ***deltaTime***. Ésta variable contiene la cantidad de tiempo desde la última llamada hasta un ***Update*** o ***FixedUpdate*** (si es que hay alguna de éstas funciones dentro).

Rotando un objeto:

```
function Update() {
transform.Rotate(0, 5 * Time.deltaTime, 0);
}
```

Moviendo un objeto:

```
function Update() {
transform.Translate(0, 0, 2 * Time.deltaTime);
}
```

Cuando se multiplica por ***Time.deltaTime*** se está expresando que por ejemplo se quiere mover un objeto 10m/s en lugar de 10m/frame. Esto es no es sólo fácil porque el juego correrá independientemente del ***frame rate*** sino también porque más unidades de movimiento son fáciles de entender (10 metros por segundo).

Otro ejemplo, si se quiere incrementar el rango de luminosidad en el tiempo. El siguiente expresa el cambio de radio en 2 unidades por segundo.

```
function Update() {
light.range += 2.0 * Time.deltaTime;
}
```

Acceso a otros componentes

Componentes son vinculados a ***GameObjects***. Todos los scripts son componentes, así éstos pueden ser vinculados a ***GameObjects***. Los componentes más comunes son accesibles como simples variables miembro.

Componente	Accesible como
Transform	transform
Rigidbody	rigidbody
Renderer	renderer
Camera	camera (sólo camera objects)
Light	light (sólo light objects)
Animation	animation
Collider	collider
…etc.	

Cualquier componente o script vinculado a un ***GameObject*** puede ser accedido a través de ***GetComponent***.

```
transform.Translate(0, 1, 0);
// es equivalente a
GetComponent(Transform).Translate(0, 1, 0);
```

Note la diferencia entre ***transform*** y ***Transform***. La primera es una variable (minúscula), la segunda es una ***clase*** o ***script name*** (mayúscula).
Aplicando esto, se puede encontrar cualquier script o componente incorporado el cual es vinculado al mismo objeto usando ***GetComponent***. Note que para hacer el siguiente ejemplo se necesita tener un script llamado ***OtherScript*** conteniendo una función ***DoSomething***. El script ***OtherScript*** debe ser vinculado al mismo ***GameObject*** como en el siguiente script.

```
// Éste encuentra el script OtherScript en el mismo GameObject
// y llama a DoSomething en éste.
function Update ()
{
var otherScript = GetComponent(OtherScript);
otherScript.DoSomething();
}
```

Acceso a otros GameObjects

La mayoría de avanzados códigos de juego no sólo manipulan a simple objeto. Hay varias formas de encontrar y acceder a otros ***GameObjects*** y componentes dentro. A continuación asumiremos que hay un script nombrado ***OtherScript.js*** vinculado a ***GameObjects*** en la escena.

```
function Update () {
otherScript = GetComponent(OtherScript);
otherScript.DoSomething();
}
```

A. A través de las referencias asignables del Inspector
Podemos asignar variables a cualquier tipo de objeto a través del ***Inspector***.

```
// Trasladar el objeto arrastrado hacia el target slot.
var target : Transform;
function Update () {
target.Translate(0, 1, 0);
}
```

También se puede exponer referencias a otros objetos al Inspector. Debajo se puede arrastar un ***GameObject*** que contiene el ***OtherScript*** en target slot en el Inspector.

```
// Setea el foo DoSomething en la variable target asignada en el Inspector.
var target : OtherScript;

function Update () {
// Setea la variable foo del target object.
target.foo = 2;
// Llama a DoSomething en el target.
target.DoSomething("Hello");
}
```

B. Localizado a través del Object Hierarchy.
Se pueden encontrar objetos hijo y padre a un objeto existente a través del componente ***Transform*** de un ***GameObject***.

```
// Encuentra el hijo "Hand" del GameObject .
// Vinculamos el script a
transform.Find("Hand").Translate(0, 1, 0);
```

Una vez que se ha encontrado el ***transform*** en ***hierarchy***, se puede usar ***GetComponent*** para obtener otros scripts.

```
// Encuentra el hijo llamado "Hand".
// En OtherScript vinculado a éste, setear foo a 2.
transform.Find("Hand").GetComponent(OtherScript).foo = 2;

// Encuentra el hijo llamado "Hand".
// Llama a DoSomething en el OtherScript vinculado a éste.
transform.Find("Hand").GetComponent(OtherScript).DoSomething("Hello");

// Encuentra al hijo llamado "Hand".
// Entonces se aplica una fuerza al rigidbody vinculado a Hand.
transform.Find("Hand").rigidbody.AddForce(0, 10, 0);
```

Se puede enlazar a todos los hijos.

```
// Mueve todos los hijos transform 10 unidades hacia arriba.
for (var child : Transform in transform) {
child.Translate(0, 10, 0);
}
```

C. Localizado por nombre o Tag.
Se puede buscar ***GameObjects*** con tags precisos usando ***GameObject.FindWithTag*** y ***GameObject.FindGameObjectsWithTag***. Use ***GameObject.Find*** para encontrar un ***GameObject*** por nombre.

```
function Start () {
// Por nombre.
var go = GameObject.Find("SomeGuy");
go.transform.Translate(0, 1, 0);

// Por tag.
var player = GameObject.FindWithTag("Player");
player.transform.Translate(0, 1, 0);
}
```

Se puede usar usar ***GetComponent*** en el resultado para obtener cualquier script o componente en el ***GameObject*** encontrado.

```
function Start () {
// Por nombre.
var go = GameObject.Find("SomeGuy");
go.GetComponent(OtherScript).DoSomething();

// Por tag.
var player = GameObject.FindWithTag("Player");
player.GetComponent(OtherScript).DoSomething();
}
```

Vectores

Unity usa la clase ***Vector3*** en todo para representar vectores en 3D. Los componentes individuales de un vector 3D pueden ser accedidos a través de sus variables miembro x, y, z.

```
var aPosition : Vector3;
aPosition.x = 1;
aPosition.y = 1;
aPosition.z = 1;
```

También se puede usar el constructor ***Vector3*** para inicializar todos los componentes a la vez.

```
var aPosition = Vector3(1, 1, 1);
```

Vector3 también define algunos valores comunes como constantes.

```
var direction = Vector3.up; // lo mismo que Vector3(0, 1, 0);
```

Operaciones en un solo vector son accedidas de la siguiente manera:

```
SomeVector.Normalize();
```

Y operaciones usando múltiples vectores son hechas usando funciones de la clase ***Vector3***.

```
var oneVector : Vector3 = Vector3(0,0,0);
var otherVector : Vector3 = Vector3(1,1,1);

var theDistance = Vector3.Distance(oneVector, otherVector);
```

También se pueden usar operadores matemáticos comunes para manipular vectores:

```
combined = vector1 + vector2;
```

Variables

Member variables

Cualquier variable definida fuera de cualquier función define una variable miembro. Las variables son accesibles a través del Inspector dentro del Unity. Cualquier valor guardado en una variable miembro es también automáticamente guardado con el proyecto.

```
var memberVariable = 0.0;
```

La variable anterior se mostrará como propiedad numérica llamada "***Member Variable***" en el Inspector.
Si se setea el tipo de una variable a un tipo de componente (como ***Transform***, ***Rigidbody***, ***Collider***, cualquier ***script name***, etc.) y así se podrán setearlos arrastrando ***GameObjects*** dentro del valor en el ***Inspector***.

```
var enemy : Transform;

function Update()
{
if ( Vector3.Distance( enemy.position, transform.position ) < 10 )
print("Creo que el enemigo está cerca!");
}
```

También se pueden crear ***private member variables***. Estas son útiles para guardar estatus que no deberían ser visibles fuera del script. Variables miembro privadas no son guardadas en el disco y no son editables en el ***Inspector***. Estos son visibles en el Inspector cuando estén en el ***debug mode.*** Esto permite usar variables privadas como ***updating debuggers*** en tiempo real.

```
private var lastCollider : Collider;

function
OnCollisionEnter(collisionInfo : Collision ) {
lastCollider = collisionInfo.collider;
}
```

Global variables
También se puede crear variables globales usando el ***static keyword***.
Este crea una variable global llamada ***someGlobal***.

```
// Variable estática en un script llamado 'TheScriptName.js'.
static var someGlobal = 5;
// Se puede acceder a éste desde dentro del script como variables normales:
print(someGlobal);
someGlobal = 1;
```

Para acceder a éste desde otro script se debe usar el nombre del script seguido por un punto y la variable global.

```
name.print(TheScriptName.someGlobal);
TheScriptName.someGlobal = 10;
```

Instantiate

Instantiate duplica un objeto. Incluyendo todo los scripts vinculados y todo el hierarchy. Este mantiene referencias en la manera que se espera: Referencias a objetos que están fuera de la jerarquía clonada serán dejadas sin contacto, Referencias a objetos en la jerarquía clonada serán mapeadas en el objeto clonado.
Instantiate es increíblemente rápido y versátil. Por ejemplo aquí está un pequeño script que cuando se vincula a un rigidbody con collider se destruirá así mismo y pondrá en su lugar un ***explosion object***.

```
var explosion : Transform;

// Cuando ocurre una collision se autodestruye
// y aparece una explosión "prefab" en lugar del anterior
function OnCollisionEnter () {
Destroy (gameObject);

var theClonedExplosion : Transform;
theClonedExplosion = Instantiate(explosion,transform.position, transform.rotation);
}
```

IV. Clases importantes

Clase Transform

Variables

position	The position of the transform in world space.
localPosition	Position of the transform relative to the parent transform.
eulerAngles	The rotation as Euler angles in degrees.
localEulerAngles	The rotation as Euler angles in degrees relative to the parent transform's rotation.
right	The red axis of the transform in world space.
up	The green axis of the transform in world space.
forward	The blue axis of the transform in world space.
rotation	The rotation of the transform in world space stored as a Quaternion.
localRotation	The rotation of the transform relative to the parent transform's rotation.
localScale	The scale of the transform relative to the parent.
parent	The parent of the transform.
worldToLocalMatrix	Matrix that transforms a point from world space into local space (Read Only).
localToWorldMatrix	Matrix that transforms a point from local space into world space (Read Only).
root	Returns the topmost transform in the hierarchy.
childCount	The number of children the Transform has.
lossyScale	The global scale of the object (Read Only).

Functions

Translate	Moves the transform in the direction and distance of translation.
Rotate	Applies a rotation of eulerAngles.z degrees around the z axis, eulerAngles.x degrees around the x axis, and eulerAngles.y degrees around the y axis (in that order).
RotateAround	Rotates the transform about axis passing through point in world coordinates by angle degrees.
LookAt	Rotates the transform so the forward vector points at /target/'s current position.
TransformDirection	Transforms direction from local space to world space.
InverseTransformDirection	Transforms a direction from world space to local space. The opposite of Transform.TransformDirection.
TransformPoint	Transforms position from local space to world space.
InverseTransformPoint	Transforms position from world space to local space. The opposite of Transform.TransformPoint.
DetachChildren	Unparents all children.
Find	Finds a child by name and returns it.
IsChildOf	Is this transform a child of parent?

Inherited Variables

transform	The Transform attached to this GameObject (null if there is none attached).
rigidbody	The Rigidbody attached to this GameObject (null if there is none attached).
camera	The Camera attached to this GameObject (null if there is none attached).
light	The Light attached to this GameObject (null if there is none attached).
animation	The Animation attached to this GameObject (null if there is none attached).
constantForce	The ConstantForce attached to this GameObject (null if there is none attached).
renderer	The Renderer attached to this GameObject (null if there is none attached).
audio	The AudioSource attached to this GameObject (null if there is none attached).
guiText	The GUIText attached to this GameObject (null if there is none attached).
networkView	The NetworkView attached to this GameObject (Read Only). (null if there is none attached)
guiTexture	The GUITexture attached to this GameObject (Read Only). (null if there is none attached)
collider	The Collider attached to this GameObject (null if there is none attached).
hingeJoint	The HingeJoint attached to this GameObject (null if there is none attached).
particleEmitter	The ParticleEmitter attached to this GameObject (null if there is none attached).
gameObject	The game object this component is attached to. A component is always attached to a game object.
tag	The tag of this game object.
name	The name of the object.
hideFlags	Should the object be hidden, saved with the scene or modifiable by the user?

Inherited Functions

GetComponent	Returns the component of Type type if the game object has one attached, null if it doesn't.
GetComponentInChildren	Returns the component of Type type in the GameObject or any of its children using depth first search.
GetComponentsInChildren	Returns all components of Type type in the GameObject or any of its children.
GetComponents	Returns all components of Type type in the GameObject.
CompareTag	Is this game object tagged tag?
SendMessageUpwards	Calls the method named methodName on every MonoBehaviour in this game object and on every ancestor of the behaviour
SendMessage	Calls the method named methodName on every MonoBehaviour in this game object.
BroadcastMessage	Calls the method named methodName on every MonoBehaviour in this game object or any of its children.
GetInstanceID	Returns the instance id of the object.

Inhereted Class Functions

operator bool	Does the object exist?
Instantiate	Clones the object original and returns the clone.
Destroy	Removes a gameobject, component or asset.
DestroyImmediate	Destroys the object obj immediately. It is strongly recommended to use Destroy instead.
FindObjectsOfType	Returns a list of all active loaded objects of Type type.
FindObjectOfType	Returns the first active loaded object of Type type.
operator ==	Compares if two objects refer to the same
operator !=	Compares if two objects refer to a different object
DontDestroyOnLoad	Makes the object target not be destroyed automatically when loading a new scene.

Clase Rigidbody

Variables

velocity	The velocity vector of the rigidbody.
angularVelocity	The angular velocity vector of the rigidbody.
drag	The drag of the object.
angularDrag	The angular drag of the object.
mass	The mass of the rigidbody.
useGravity	Controls whether gravity affects this rigidbody.
isKinematic	Controls whether physics affects the rigidbody.
freezeRotation	Controls whether physics will change the rotation of the object.
collisionDetectionMode	The Rigidbody's collision detection mode.
centerOfMass	The center of mass relative to the transform's origin.
worldCenterOfMass	The center of mass of the rigidbody in world space (Read Only).
inertiaTensorRotation	The rotation of the inertia tensor.
inertiaTensor	The diagonal inertia tensor of mass relative to the center of mass.
detectCollisions	Should collision detection be enabled? (By default always enabled)
useConeFriction	Force cone friction to be used for this rigidbody.
position	The position of the rigidbody.
rotation	The rotation of the rigdibody.
interpolation	Interpolation allows you to smooth out the effect of running physics at a fixed frame rate.
solverIterationCount	Allows you to override the solver iteration count per rigidbody.
sleepVelocity	The linear velocity, below which objects start going to sleep. (Default 0.14) range { 0, infinity }
sleepAngularVelocity	The angular velocity, below which objects start going to sleep. (Default 0.14) range { 0, infinity }
maxAngularVelocity	The maximimum angular velocity of the rigidbody. (Default 7) range { 0, infinity }

Functions

SetDensity	Sets the mass based on the attached colliders assuming a constant density.
AddForce	Adds a force to the rigidbody. As a result the rigidbody will start moving.
AddRelativeForce	Adds a force to the rigidbody relative to its coordinate system.
AddTorque	Adds a torque to the rigidbody.
AddRelativeTorque	Adds a torque to the rigidbody relative to the rigidbodie's own coordinate system.
AddForceAtPosition	Applies force at position. As a result this will apply a torque and force on the object.
AddExplosionForce	Applies a force to the rigidbody that simulates explosion effects. The explosion force will fall off linearly with distance to the rigidbody.
ClosestPointOnBounds	The closest point to the bounding box of the attached colliders.
GetRelativePointVelocity	The velocity relative to the rigidbody at the point relativePoint.
GetPointVelocity	The velocity of the rigidbody at the point worldPoint in global space.
MovePosition	Moves the rigidbody to position.
MoveRotation	Rotates the rigidbody to rotation.
Sleep	Forces a rigidbody to sleep at least one frame.
IsSleeping	Is the rigidbody sleeping?
WakeUp	Forces a rigidbody to wake up.
SweepTest	Tests if a rigidbody would collide with anything, if it was moved through the scene.
SweepTestAll	Like Rigidbody.SweepTest, but returns all hits.

Messages Sent

OnCollisionEnter	OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.
OnCollisionExit	OnCollisionEnter is called when this collider/rigidbody has stopped touching another rigidbody/collider.
OnCollisionStay	OnCollisionStay is called once per frame for every collider/rigidbody that is touching rigidbody/collider.

Inherited Variables

transform	The Transform attached to this GameObject (null if there is none attached).
rigidbody	The Rigidbody attached to this GameObject (null if there is none attached).
camera	The Camera attached to this GameObject (null if there is none attached).
light	The Light attached to this GameObject (null if there is none attached).
animation	The Animation attached to this GameObject (null if there is none attached).
constantForce	The ConstantForce attached to this GameObject (null if there is none attached).
renderer	The Renderer attached to this GameObject (null if there is none attached).
audio	The AudioSource attached to this GameObject (null if there is none attached).
guiText	The GUIText attached to this GameObject (null if there is none attached).
networkView	The NetworkView attached to this GameObject (Read Only). (null if there is none attached)
guiTexture	The GUITexture attached to this GameObject (Read Only). (null if there is none attached)
collider	The Collider attached to this GameObject (null if there is none attached).
hingeJoint	The HingeJoint attached to this GameObject (null if there is none attached).
particleEmitter	The ParticleEmitter attached to this GameObject (null if there is none attached).
gameObject	The game object this component is attached to. A component is always attached to a game object.
tag	The tag of this game object.
name	The name of the object.
hideFlags	Should the object be hidden, saved with the scene or modifiable by the user?

Inherited Functions

GetComponent	Returns the component of Type type if the game object has one attached, null if it doesn't.
GetComponentInChildren	Returns the component of Type type in the GameObject or any of its children using depth first search.
GetComponentsInChildren	Returns all components of Type type in the GameObject or any of its children.
GetComponents	Returns all components of Type type in the GameObject.
CompareTag	Is this game object tagged tag?
SendMessageUpwards	Calls the method named methodName on every MonoBehaviour in this game object and on every ancestor of the behaviour
SendMessage	Calls the method named methodName on every MonoBehaviour in this game object.
BroadcastMessage	Calls the method named methodName on every MonoBehaviour in this game object or any of its children.
GetInstanceID	Returns the instance id of the object.

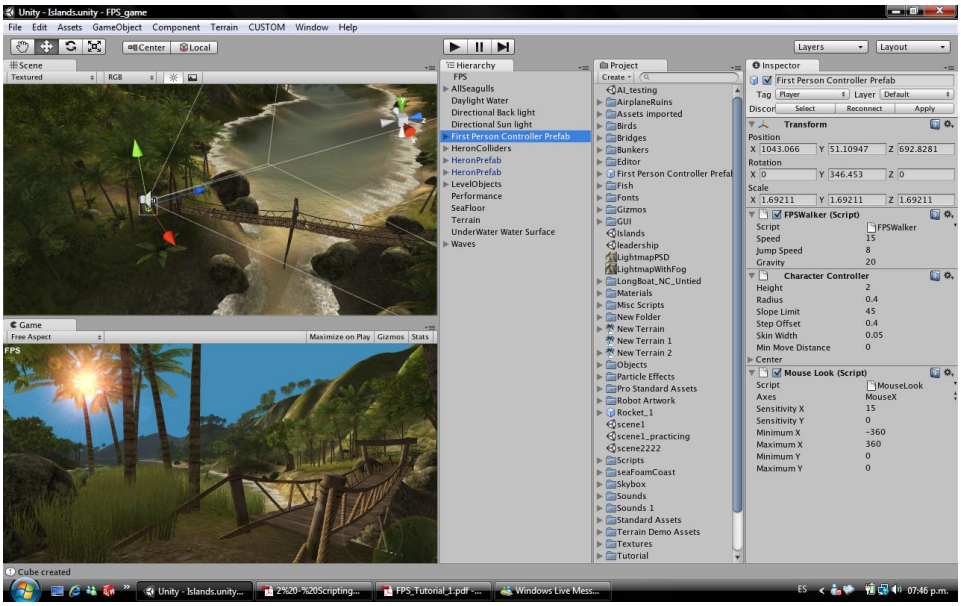
Inherited Class Functions

operator bool	Does the object exist?
Instantiate	Clones the object original and returns the clone.
Destroy	Removes a gameobject, component or asset.
DestroyImmediate	Destroys the object obj immediately. It is strongly recommended to use Destroy instead.
FindObjectsOfType	Returns a list of all active loaded objects of Type type.
FindObjectOfType	Returns the first active loaded object of Type type.

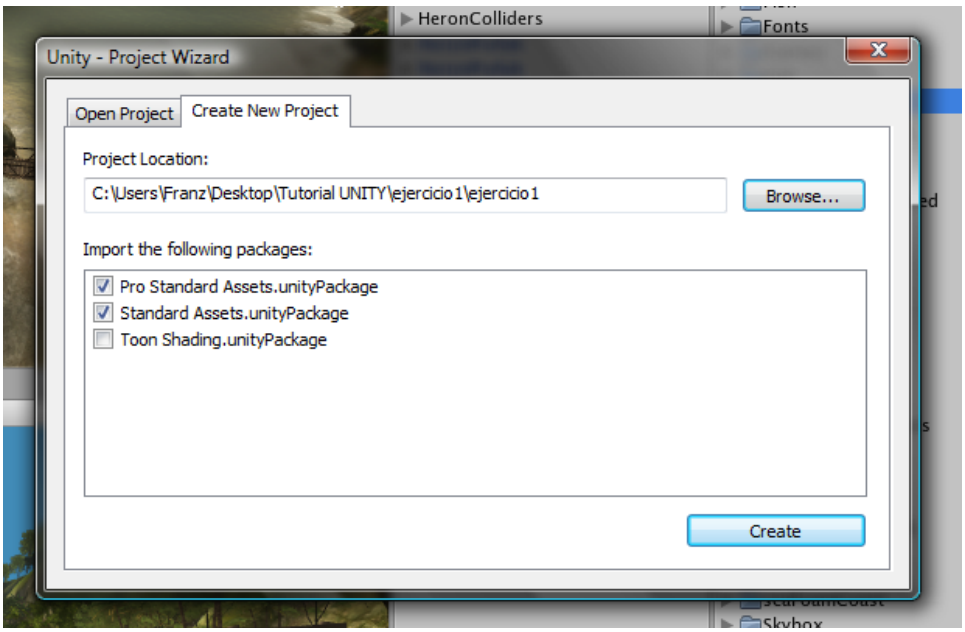
operator ==	Compares if two objects refer to the same
operator !=	Compares if two objects refer to a different object
DontDestroyOnLoad	Makes the object target not be destroyed automatically when loading a new scene.

V. Ejercicio 1: Aplicación de lo aprendido

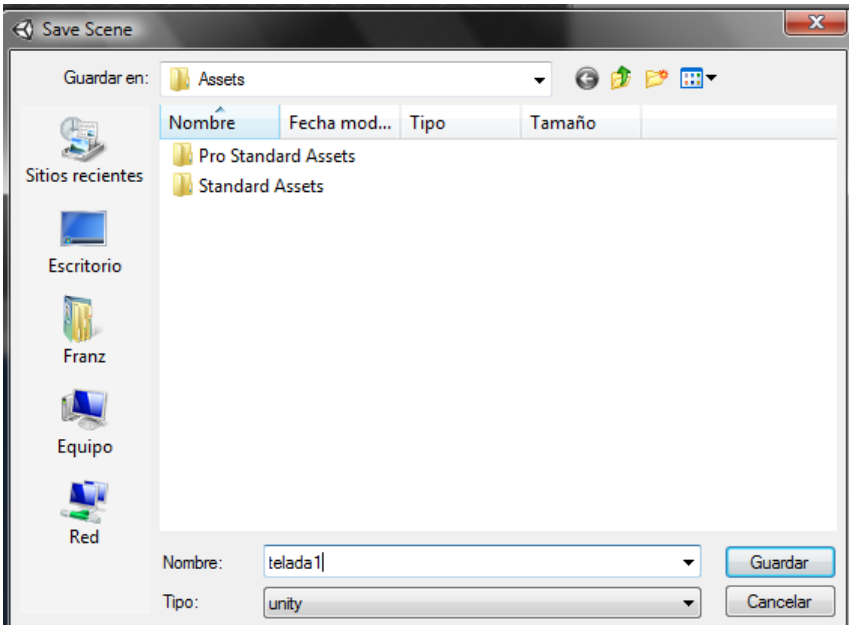
Ingresamos al software Unity. Al inicio se observará en pantalla el proyecto que se estuvo realizando anteriormente.



Creamos un nuevo proyecto. File->New Project. Seleccionar la locación del proyecto seleccionando las opciones "Pro Standard Assets.unityPackage" y "Standard Assets.unityPackage". La nombramos "ejercicio1".



Ahora creamos una escena. File->New Scene. La nombramos "telada1".



Para nuestro primer programa vamos a permitir que el jugador se mueva en un mundo de juego sencillo.

Entorno del jugador

Situar la escena

Primero vamos a crear una superficie en la que camine el usuario. La superficie que vamos a usar es la forma de un cubo aplastado.
- Crea un cubo y escala las dimensiones x,y,z a 5, 0.1, y 5 respectivamente, ahora debería parecerse a un amplio y llano plano. Renombra este objeto como 'Plane' en la Vista de Jerarquía (***Hierarchy View***).
- Crea un segundo cubo y colócalo en el centro del plano. Si no puedes ver los objetos en la Vista de Juego (***Game View***), altera la cámara principal para que estén visibles. Renombra el objeto como Cubo1 (Cube1).
-También deberías crear un punto de luz y colocarlo encima de los cubos de forma que sean visibles más fácilmente.

Nuestro primer Script

Ahora estamos listos para empezar a programar juegos. Vamos a permitir que el jugador camine por el mundo de juego controlando la posición de la cámara principal. Para esto vamos a escribir un script que leerá la entrada desde el teclado, luego adjuntamos (asociamos) el script con la cámara principal (más en la próxima sección).
- Comenzar creando un script vacío. Selecciona ***Assets***->***Create***->***JavaScript*** y renombra este script como Movimiento1 (Move1) en el Panel del Proyecto (***Project Panel***).
- Haz doble click en el script Move1 y se ya abrirá con la función **Update()** insertada (es un comportamiento por defecto), vamos a insertar nuestro código dentro de esta función. Cualquiera código que insertes dentro de la función Update() ejecutará cada frame.
Para mover un objeto de juego en Unity necesitamos cambiar la propiedad de posición de su transform, la función **Translate** perteneciente al transform nos permitirá hacerlo. La función Translate toma 3 parámetros: los movimientos x, y, z. Si queremos controlar el objeto de juego de la cámara principal con las teclas del cursor, simplemente asociamos código para determinar si las teclas del cursor están siendo presionadas para los respectivos parámetros:

```
function Update () {
transform.Translate(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
}
```

La función ***Input.GetAxis()*** devuelve un valor entre -1 y 1, por ejemplo en el eje horizontal, la tecla izquierda del cursor devuelve -1, la tecla derecha del cursor devuelve 1.
Fijate en que el parámetro 0 en el eje 'y' ya que no estamos interesados en mover la cámara hacia arriba. Los ejes Horizontal y Vertical están predefinidos en el ***Input Settings***, los nombres y claves trazados a ellos pueden ser fácilmente cambiados en ***Edit***->***Project Settings***->***Input***.
-Abre el JavaScript 'Move1' y escribe el código superior, presta atención a las mayúsculas.

Adjuntar el script

Ahora que nuestro primer script está escrito, ¿cómo le decimos a Unity qué objeto de juego debería tener ese comportamiento? Todo lo que tenemos que hacer es adjuntar el script al objeto de juego que queremos que muestre ese comportamiento.
- Para hacer esto, primero haz click en el objeto de juego que quieras que tenga el comportamiento definido en el script. En nuestro caso, esta es la Cámara Principal, y tú puedes seleccionarla desde la Vista de Jerarquía (***Hierarchy View***) o desde la Vista de Escena (***Scene View***).
- Después selecciona ***Components***->***Scripts***->Move1 desde el menú principal.
Así se adjunta el script a la cámara, deberías notar que el componente Move1 ahora aparece en la Vista de Inspector (***Inspector View***) para la cámara principal.

Consejo: puedes también asignar un script a un objeto de juego arrastrando el script desde la Vista de Proyecto (***Project View***) hasta el objeto en la Vista de Escena (***View Scene***).
-Pon en marcha el juego (presiona el icono 'play' en la parte superior), deberías ser capaz de mover la cámara principal con las teclas del cursor o W, S, A, D.
Probablemente notaste que la cámara se movió un poco demasiado rápido, vamos a buscar una mejor forma de controlar la velocidad de la cámara.

Tiempo Delta
Como el anterior código estaba dentro de la función Update(), la cámara se movía a la velocidad medida en metros por frame. De todas formas es mejor asegurarse de que tus objetos de juego se mueven al ritmo predecible de metros por segundo. Para conseguir esto tenemos que multiplicar el valor dado desde la función ***Input.GetAxis()*** por el tiempo Delta y también por la velocidad a la que queremos movernos por segundo:

```
var speed = 5.0;
function Update () {
var x = Input.GetAxis("Horizontal") * Time.deltaTime * speed;
var z = Input.GetAxis("Vertical") * Time.deltaTime * speed;
transform.Translate(x, 0, z);
}
```

-Actualiza el script Move1 con el código superior.

Date cuenta aquí que la velocidad variable se declara fuera de la función Update(), esto es llamado una variable expuesta (exponed variable), y aparecerá en la Vista de Inspector (***Inspector View***) para cualquier objeto de juego al que esté adjunto el script (la variable queda explicada en la Unity GUI).
Exponer variables es útil cuando el valor necesita ser modificado para conseguir el efecto deseado, esto es mucho más fácil que cambiar códigos.

Conectar variables

Conectar variables a través de GUI es una característica muy poderosa de Unity. Permite que las variables que normalmente estarían asignadas en código puedan ser hechas mediante el drag and drop (arrastrar y tirar) en la GUI de Unity.
Esto permite hacer rápidos y fáciles prototipos de ideas. Como las variables se conectan a través de la GUI de Unity, sabemos que siempre necesitaremos exponer una variable en nuestro código de script para poder asignar el parámetro en la Vista de Inspector (***Inspector View***).
Vamos a demostrar el concepto de conectar variables creando un foco de luz que seguirá al jugador (Cámara Principal) mientras se mueve.
- Añade un foco de luz (***spotlight***) a la Vista de Escena (***Scene View***). Muévelo si es necesario para que esté cerca de los otros objetos de juego.
- Crea un nuevo JavaScript y nómbralo 'Follow' (Seguir).
Pensemos en lo que queremos hacer. Queremos que nuestro nuevo foco de luz apunte a cualquiera que sea la cámara principal. Para que esto ocurra, hay una función en Unity que lo hace, transform.LookAt(). Si estabas empezando a pensar '¿cómo lo hago?' y te imaginabas un montón de códigos, entonces merece la pena recordar siempre que puedes revisar la API de Unity para usar una función que ya existe. También podríamos acertar mirando en la sección "***transform***" de la API, ya que estamos interesados en alterar la posición o rotación de un objeto de juego.
Ahora vamos a la sección de conectar variables; ¿qué usamos como un parámetro para ***LookAt()***? Podemos insertar un objeto de juego, no obstante sabemos que queremos asignar la variable mediante la GUI, así que sólo tendremos que usar una variable expuesta (del tipo ***Transform***). Nuestro ***Follow.js*** script debería parecerse a esto:

```
var target : Transform;
function Update () {
transform.LookAt(target);
}
```

- Adjunta el script al foco de luz y fijate que cuando el componente se añade, la variable "***target***" (objetivo) está expuesta.
- Con el foco de luz aun seleccionado, arrastra la Cámara Principal desde la Vista de Jerarquía (***Hierarchy View***) hasta la variable "***target***" en la Vista de Inspector (***Inspector View***). Así se asigna la variable objetivo, esto es, el foco de luz no seguirá a la Cámara Principal. Si queríamos el foco de luz para seguir a un objeto de juego diferente, podríamos simplemente arrastrar hasta él un objeto diferente (mientras sea del tipo ***Transform,*** claro).
- Pon en marcha el juego. Si miras la Vista de Escena (***Scene View***) deberías ver el foco de luz siguiendo a la Cámara Principal. Puede que quieras cambiar la posición del foco para mejorar el efecto.

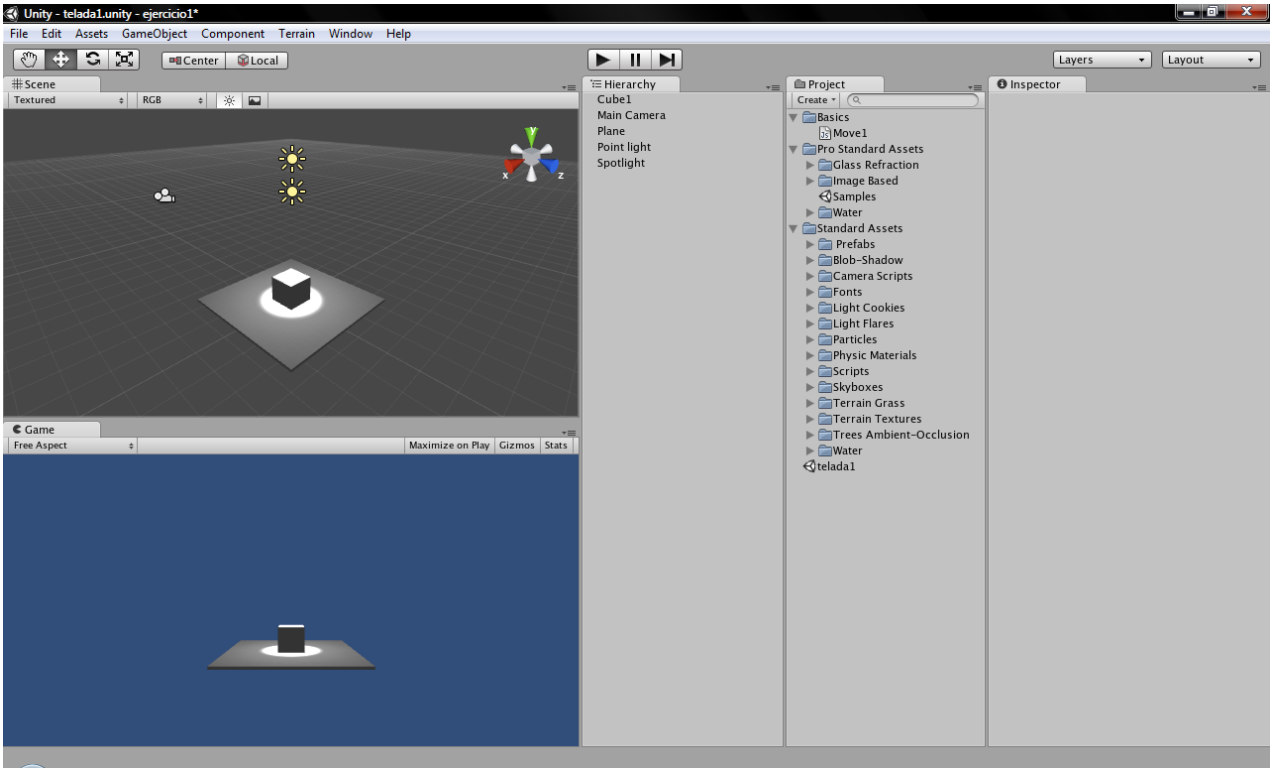
Acceder a los componentes

Como un objeto de juego puede tener adjuntos múltiples scripts (o otros componentes), a veces es necesario tener acceso a otras funciones o variables de los componentes. Unity permite de esta forma la función ***GetComponent()***.
Ahora vamos a añadir otro script a nuestro foco de luz, lo que hará que mire al Cube1 (Cubo1) cuando el botón de salto (barra espaciadora por omisión) se presione.
Pensemos en esto primero, lo que queremos hacer:
1. Detectar cuando el botón de salto ha sido presionado.
2. Cuando el botón haya sido presionado haz que el foco mire al Cube1.
¿Cómo lo hacemos? Bueno, el script Follow contiene una variable "target" (objetivo) cuyo valor determina a qué objeto de juego debería mirar el foco de luz. Debemos fijar un nuevo valor para este parámetro. Podríamos insertar el valor para el cubo (ver la sección "Hacerlo

con código" más tarde), sin embargo sabemos que exponer la variable y asignarla mediante la GUI es una mejor forma de hacerlo.
- Crea un nuevo JavaScript y llámalo "Switch" (Cambiar). Añade el siguiente código a ***"Switch.js"***:

```
var switchToTarget : Transform;
function Update () {
        if (Input.GetButtonDown("Jump"))
        GetComponent(Follow).target = switchToTarget;
}
```

Fijate en particular como "Follow" es el parámetro de ***GetComponent()***, éste devuelve una referencia al script "Follow" que nosotros podemos usar entonces para acceder a su "target" (objetivo) variable.
- Añade el script 'Switch' al foco de luz y asigna el 'Cube1' al parámetro "switchToTarget" (cambiar a objetivo) en la Vista de Inspector (***Inspector View***).
- Pon en marcha el juego. Muevete por ahí y verifica que el foco de luz te sigue como siempre, luego pulsa la barra espaciadora y el foco de luz debería enfocar el 'Cube1'.



Haciéndolo con código
Anteriormente en el tutorial ya mencionamos que sería posible asignar las variables por medio de código (a diferencia de la GUI de Unity), vamos a ver cómo hacerlo.
Recuerda que esto es sólo para comparar, asignar variables mediante la GUI es la forma recomendada.
El problema en el que estábamos interesados anteriormente era cómo podíamos hacer que el foco de luz mirara al 'Cube1' cuando la tecla espacio estuviese pulsada. Nuestra solución era exponer una variable en el script 'Switch' que pudiésemos asignar después llevando el 'Cube1' hasta ella desde la GUI de Unity. Hay dos maneras principales para hacerlo en código:

1. Usa el nombre del objeto de juego.
2. Usa la etiqueta (tag) del objeto de juego.

1. **Nombre del objeto de juego:**
Un nombre de objeto de juego puede verse en la Vista de Jerarquía (***Hierarchy View***). Para usar este nombre con código lo usamos como un parámetro en la función GameObject.Find(). Así que si queremos que el botón espacio cambie el foco de luz desde la Cámara Principal hasta 'Cube1', el código es el siguiente:

```
function Update () {
if (Input.GetButtonDown("Jump")){
        var newTarget = GameObject.Find("Cube").transform;
        GetComponent(Follow).target = newTarget;
        }
}
```

Nota que ninguna variable se expone si la nombramos directamente en código.
Revisa el API para más opciones usando ***Find().***

2. **Etiqueta del objeto de juego.**
Una etiqueta del objeto de juego es un string que puede ser usado para identificar un componente. Para ver las etiquetas incorporadas, haz click en el botón "Tag" (etiqueta) en al Vista de Inspector (***Inspector View***) y date cuenta de que puedes además crear las tuyas propias. La función para encontrar un componente con una etiqueta específica es ***GameObject.FindWithTag()*** y toma un string como parámetro. Nuestro código completo para hacerlo es:

```
function Update () {
if (Input.GetButtonDown("Jump")){
        var newTarget = GameObject.FindWithTag("Cube").transform;
        GetComponent(Follow).target = newTarget;
        }
}
```

Instantiate

A veces es conveniente crear objetos durante el tiempo de ejecución (mientras se está jugando al juego). Para hacer esto usamos la función "***Instantiate***".
Vamos a mostrar cómo funciona creando un nuevo objeto de juego cada vez que el usuario presione la tecla fire (el botón izquierdo del ratón o el Ctrl izquierdo del teclado).
Entonces ¿qué queremos hacer? Queremos que el usuario se mueva normalmente, y cuando presione la tecla fire, se cree un nuevo objeto. Varias cosas en las que pensar:
1. ¿Qué objeto creamos?
2. ¿Dónde lo creamos?
En cuanto al objeto a crear, la mejor forma de solventarlo es exponer una variable. Esto significa que podemos establecer qué objeto crear mediante el uso del drag and drop para asignar un objeto de juego a esta variable.
Para saber dónde crearlo, por ahora sólo crearemos el nuevo objeto de juego donde sea que el usuario (Cámara Principal) está actualmente situado cuando quiera que la tecla – sea presionada.
La función ***Instantiate*** toma 3 parámetros: (1) el objeto que queremos crear, (2) la posición 3D del objeto, y (3) la rotación del objeto.
El código completo para hacer esto es el siguiente (Create.js):

```
var newObject : Transform;
function Update () {
        if (Input.GetButtonDown("Fire1")) {
        Instantiate(newObject, transform.position, transform.rotation);
        }
}
```

No olvides que '***transform.position***' y '***transform.rotation***' son la posición y rotación de transformar al que está adjunto el script; en nuestro caso será la Cámara Principal.
Sin embargo, cuando un objeto se crea, es normal que ese objeto sea un 'prefabricado'. Vamos ahora a transformar el objeto de juego 'Cube1' en un prefabricado.
- Primero, vamos a crear un prefabricado (***prefab***) vacío. Selecciona 'Assets- Create-Prefab'. Llama al prefabricado 'Cube'.
- Arrastra el objeto de juego 'Cube1' desde la Vista de Jerarquía (***Hierarchy View***) hasta el prefabricado 'Cube' en la Vista de Proyecto (***Project View***). Nota que el icono prefabricado cambia.
Ahora podemos crear nuestro código JavaScript.
- Crea un nuevo JavaScript y llámalo 'Create' (crear). Inserta el código superior.
- Adjunta este script a la Cámara Principal y asigna el prefabricado 'Cube' a la variable "newObject" de la Cámara Principal.
- Juega al juego y muévete como de costumbre. Cada vez que el botón fire sea pulsado deberías notar que un nuevo cubo aparece.

Debugging

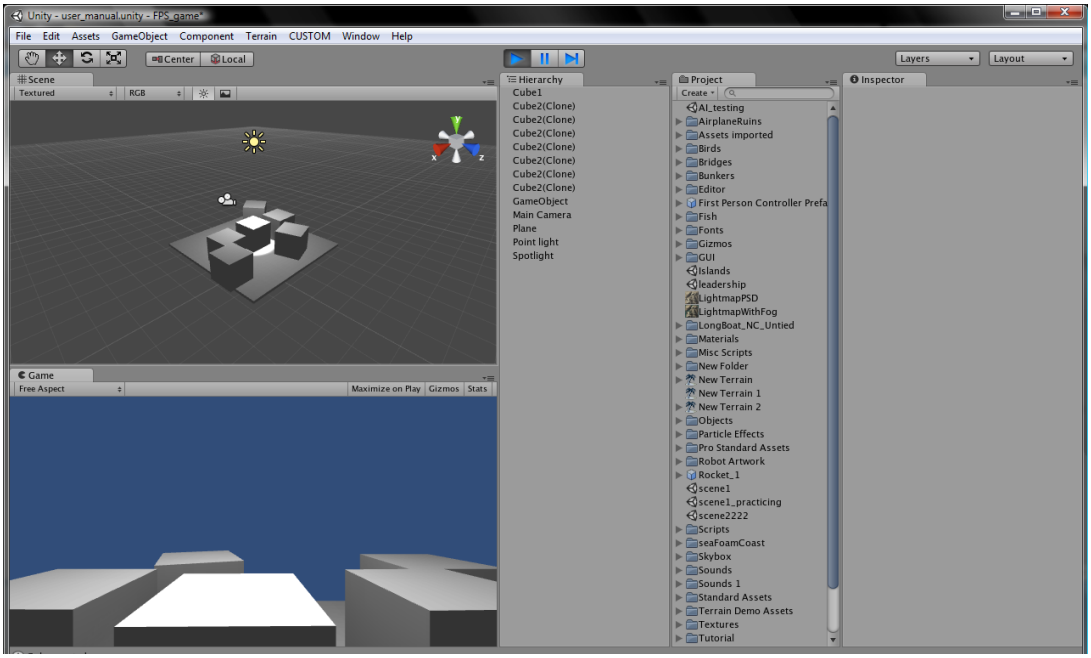
Depurar es la técnica para encontrar y solucionar errores humanos en tu código (vale, llamémosles equivocaciones). Unity proporciona ayuda mediante la clase "Debug" (depurar), vamos ahora a mirar la función Debug.Log().

Log
La función ***Log()*** permite al usuario mandar un mensaje a la Consola de Unity.
Algunas razones para hacer esto:
1. Para probar que una cierta parte del código está siendo alcanzada durante el tiempo de ejecución.
2. Para informar del estado de una variable.
Vamos a usar ahora la función Log() para mandar un mensaje a la Consola de Unity cuando el usuario pulse la tecla fire.
- Abre el script 'Create' y añade la siguiente línea tras el código 'Instantiate' (instanciar) dentro del bloque 'if': Debug.Log("Cube created");
- Pon en marcha el juego y pulsa la tecla ---. Deberías ver una línea que aparece al pie de la GUI de Unity diciendo "cube created" (cubo creado), puedes hacer click en ella para examinar la Consola de Unity.

Mirar
Otra característica útil para depurar es exponer una variable particular. Esto hace que la variable sea visible en la Vista de Inspector (***Inspector View***) cuando se selecciona el modo **Debug (depurar),** pero no puede ser editado.
Para demostrarlo, vamos a exponer una variable particular para contar el número de cubos que creamos.
- Abre el script 'Create' otra vez y añade 2 líneas:
(1) Añade una variable particular llamada "cubeCount".
(2) Incrementa esta variable cuando se cree un cubo.
El código completo es el siguiente (Create.js):

```
var newObject : Transform;
private var cubeCount = 0;
function Update () {
        if (Input.GetButtonDown("Fire1")) {
        Instantiate(newObject, transform.position, transform.rotation);
        Debug.Log("Cube created");
        cubeCount++;
        }
}
```

- Arranca el juego y pulsa la tecla fire (disparar) para crear algunos cubos. Date cuenta de cómo se incrementa la variable 'cubeCount' en la Vista de Inspector cada vez que se crea un cubo. Nota además cómo el número aparece sombreado en gris, lo que indica que es una variable únicamente de lectura (no se puede editar).



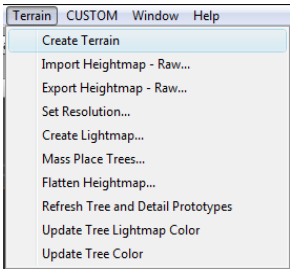
VI. Ejercicio 2: Creación de un FPS

Empezando un nuevo proyecto

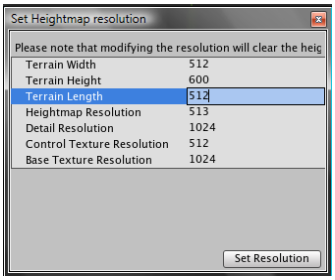
Crear un nuevo proyecto de la misma forma como se hizo anteriormente. Para no empezar de cero nuestro FPS, debemos descargar unas librerías para FPS que se encuentran en la página del Unity, las importamos a nuestro proyecto y empezamos a realizar el videojuego.

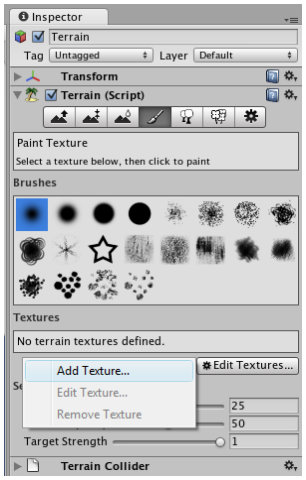
Estableciendo el ambiente de juego

Ahora creamos una escena con el nombre de "telada2" y estableceremos primeramente nuestro terreno. Este puede ser encontrado en ***Terrain-> Create Terrain***.

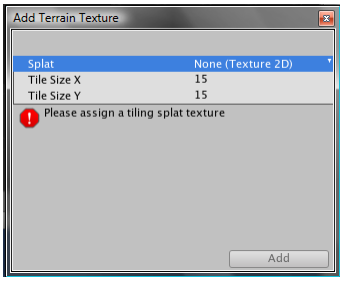


La selección de la resolución del terreno se realiza en ***Terrain-> Set Resolution***. Elegiremos en nuestro caso una resolución como se muestra en la figura.

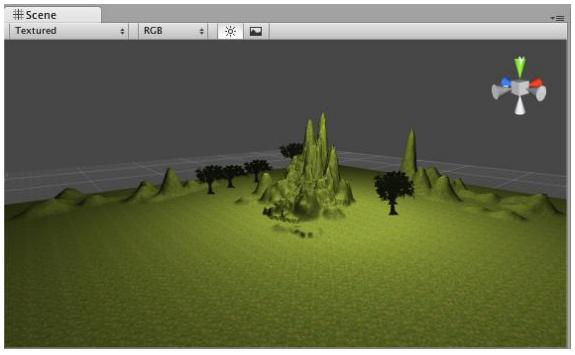


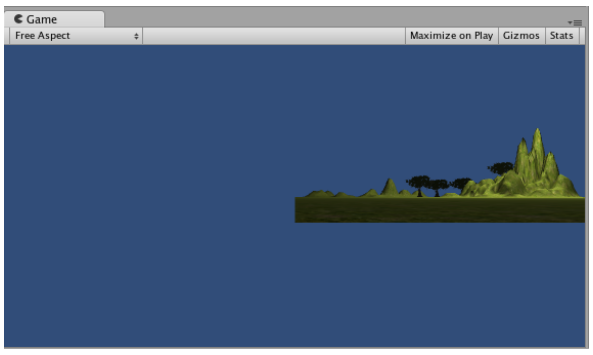


Ahora debemos agregar una textura al terreno creado. Para esto damos click a ***Edit Textures-> Add Textures***. Luego elegimos cualquiera de las texturas prediseñadas. Para nuestro caso será ***Grass (Hill)***.



Como se observará en la escena, el terreno está oscuro, para arreglar esto debemos agregar un ***GameObject*** llamado ***Point Light*** que se encuentra en ***GameObject-> Create Other-> Point Light***. Seleccionando este objeto, en el inspector le damos las coordenadas **256; 200; 256**, con una intensidad de **5** y un rango de **500**.
Ahora podemos modificar el terreno agregando elevaciones, depresiones, arboles, arbustos, montañas, etc. Para esto seleccionamos el objeto ***Terrain***, usamos cualquiera de los 7 bloques que se muestran en la figura anterior y seleccionamos el pincel que querramos.





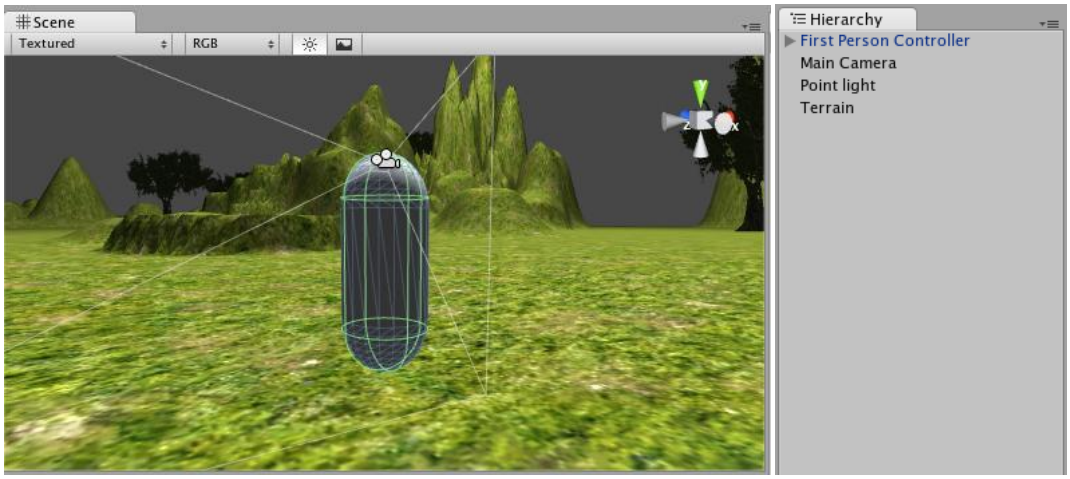
Por simplicidad dejaremos nuestro ambiente de juego como se muestra en las imágenes de arriba, tenga en cuenta que podemos agregar cualquier objeto en nuestro mapa.

Agregando el personaje principal

En las librerías que descargamos de la página web del Unity podemos extraer el ***First Person Controller***, que se encuentra exactamente en el ***Project View*** Standard ***Assets-> Prefabs-> First Person Controller***.

Arrastramos el objeto hacia el ***SceneView*** y le damos las coordenadas ***200; 2; 200***.

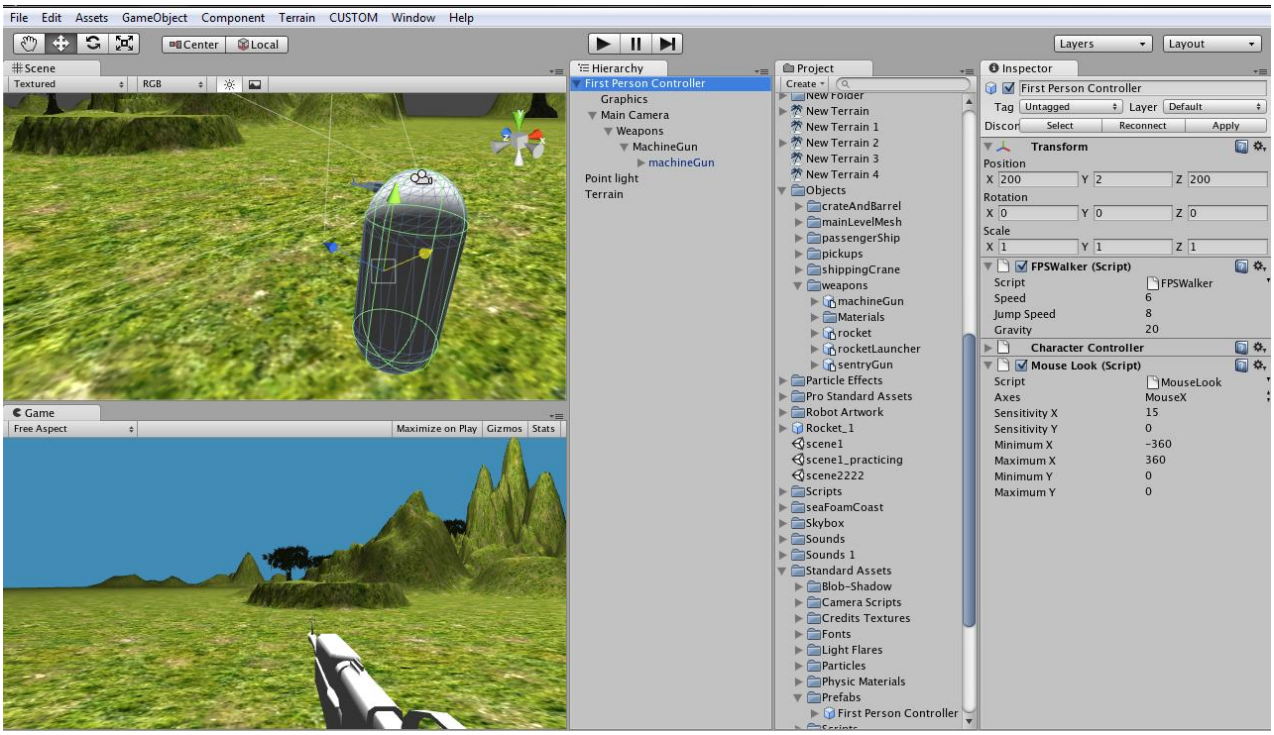
Nota: Se recomienda establecer la coordenada y con un valor mayor de 1 para que el objeto no caiga por efecto de la gravedad; x, z pueden tomar cualquier valor de acuerdo al terreno.



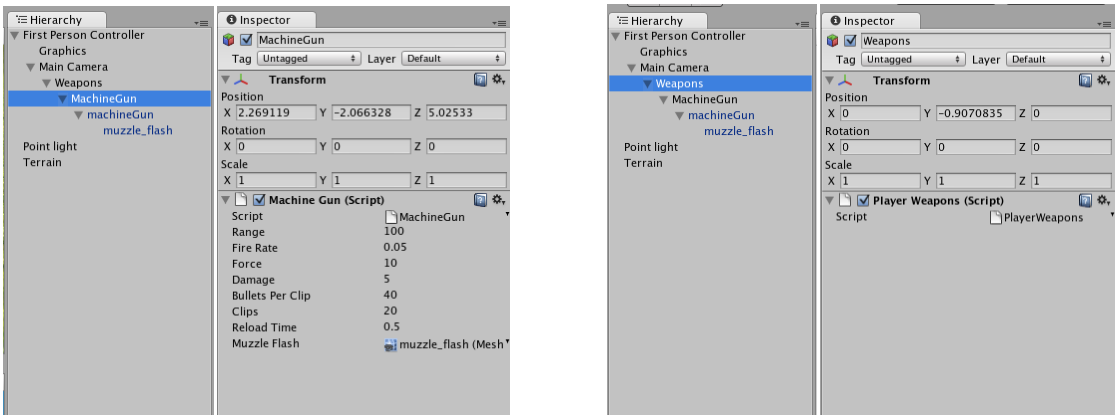
Como es un objeto prediseñado, este ya incluye el movimiento usando ***A, W, D, S*** y teclas direccionales, además del giro de la cámara con el mouse.

Agregando una arma

Para esto creamos un nuevo ***GameObject*** desde ***GameObject-> Create Empty*** y lo nombramos "***Weapons***" y lo convertimos en ***Child Object*** de ***Main Camera*** que está en el ***First Person Controller***.
Seleccionamos una arma prediseñada siguiendo la ruta ***Object-> weapons-> machineGun***. La insertamos en el ***Scene View*** y la ubicamos bien según nuestra preferencia. Para insertarlo debemos crear un nuevo ***GameObject*** que nombraremos "***MachineGun***", éste contendrá el objeto ***machinegun*** y a la vez será ***Child Object*** de "Weapons". El orden de los objetos es como se muestra en la figura.

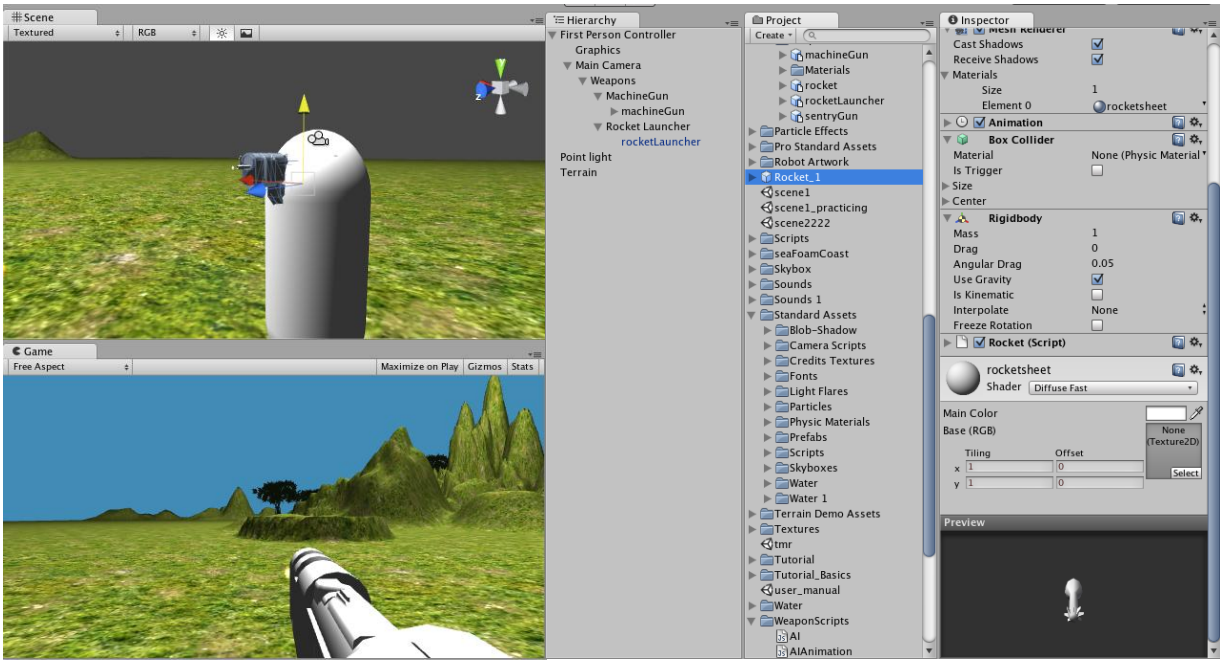


Ahora añadimos el script "***MachineGun.js***" que se encuentra en ***WeaponScripts***, lo arrastramos hasta el objeto ***MachineGun***. Luego, seleccionar el objeto ***MachineGun*** y arrastrar el objeto ***muzzle_flash*** hacia el Inspector View para vincularla a la variable ***Muzzle Flash*** del script añadido.
Este script permite el funcionamiento del arma creada, emitiendo disparos y generando una animación con partículas. Adicionalmente agregamos el script "***Player Weapons.js***" al objeto ***Weapons***. Se debe tener lo que se observa en la figura de abajo.



Luego agregamos una animación para el disparo de las balas del arma. Agregamos el prefabricado ***Sparks*** al ***Scene View***. Este se encuentra en ***Standard Assets-> Particles***. Ahora convertimos a ***Sparks*** en ***Child Object*** de ***machineGun***.

Añadimos ahora otra arma mediante un ***GameObject*** que llamaremos ***Rocket Launcher*** y lo convertimos en ***Child Object*** de ***Weapons***. Arrastramos el prefabricado ***rocketLauncher*** hacia el objeto ***Rocket Launcher*** que se encuentra ***Objects***-> ***weapons***. También arrastramos el script "***RocketLauncher.js***" que se encuentra en ***WeaponScripts***, lo arrastramos hasta el objeto ***Rocket Launcher***.
Como se trata de un lanza misiles, debemos agregar un prefabricado llamado ***rocket*** que se encuentra en ***Objects***-> ***weapons***. Agregamos también el script "***Rocket.js***". Por comodidad y poder hacer uso de este después, debemos hacer de este un prefabricado llamado ***Rocket_1***. Finalmente agregamos el prefabricado ***Rocket_1*** como variable del script "***Rocket Launcher.js***" del objeto ***Rocket Launcher***.

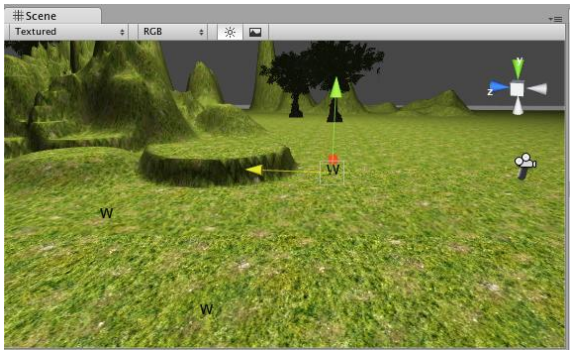


Podemos agregar si deseamos una animación para las explosiones. Arrastramos el prefabricado ***Small explosion*** hacia el script "***Rocket.js***" del prefabricado ***Rocket_1***, haciéndolo variable de entrada de éste.

Creando waypoints

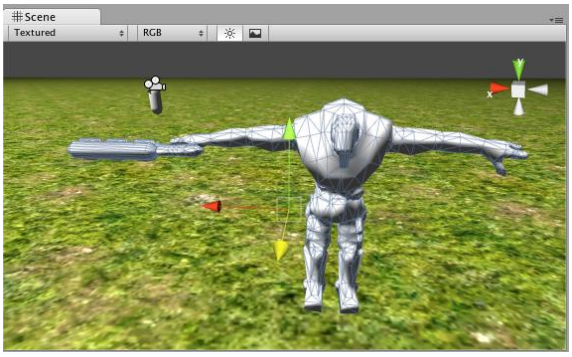
Creamos un GameObject y lo nombramos ***Waypoints***. Le agregamos el script "***AutoWayPoint.js***" al objeto mencionado y creamos 3 ó más de estos ***Waypoints***.

Nota: Colocar los ***Waypoints*** a 1 metro en el eje y. Colocarlos como se muestra en la figura de abajo.

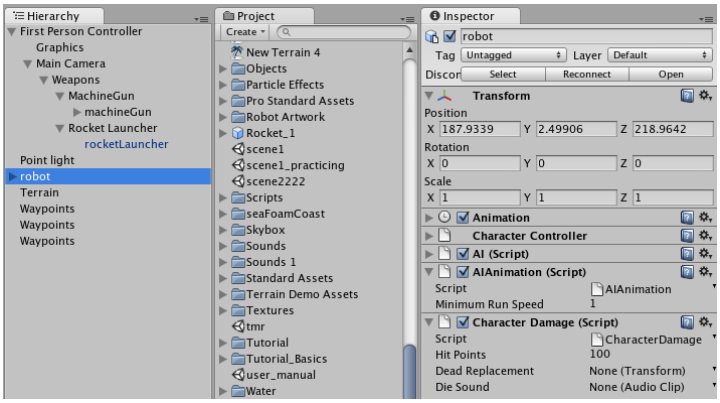


Agregando un robot AI

Arrastramos el prefabricado ***robot*** que se encuentra en ***Robot Artwork***.



El objeto creado debe ser ubicado de tal manera que los pies del robot no crucen el terreno de juego, ya que se caería por efecto de la gravedad. Podemos modificar sus dimensiones como queramos. Ver variables relacionadas con el objeto.



Como se observa en la figura de arriba, debemos agregar los scripts "***AI.js***", "***AIAnimation.js***", "***Character Damage.js***". Sin embargo, el robot no tiene un arma definida aún. Debemos crear un ***GameObject*** llamado ***gun_spawn***, hacemos de este un ***Child Object*** del objeto ***robot***.

Arrastramos el script "***Rocket Launcher.js***" hacia el objeto ***gun_spawn***. Las variables de estos scripts ya deben ser fáciles de agregar hasta este momento.

Aquí hemos mostrado los primeros pasos de cómo realizar un FPS, hay muchas cosas que se deben agregar como un interfase humano-maquina, más escenarios, menús, más armas, más enemigos, niveles de dificultad, etc.