Características básicas de Unity

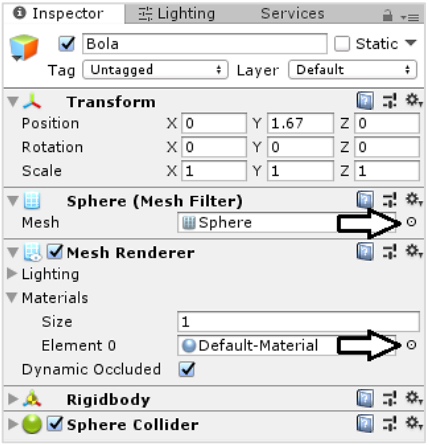
Unity permite creación de juegos 2D y 3D. Visualizaciones arquitectónicas o animaciones 3D. Es un motro gráfico similar a *Unreal* pero tiene una curva de aprendizaje más fácil, basado en el lenguaje de programación C# (más práctico que C++ de *Unreal*). En contrapartida, *Unity* para grandes juegos en 3D es algo lento.
Descarga e instalación: Desde *unity3d.com/es* o *unity3d.com/es/get-unity/download*. versión gratuita personal para < 1500 €
Interfaz gráfica:

- *Main Editor Window*: Es la ventana del editor principal. Contiene varias vistas (views)
- *Project View*: Panel de propiedades. Controlas los recursos del proyecto.
- *Hierarchy*: Panel que contiene los objetos de la escena. *GameObject*
- Scene View: ventana interactiva donde seleccionar y posicionar el jugador, la cámara, los enemigos, y otros *GameObjects*.
- *Game View*: Vista desde la cámara representando el producto final.
- *Inspector*: Panel con información detallada del objeto seleccionado *(GameObject).*

Scripting: define el comportamiento del juego.
Funciones básicas: Cuando se crea un nuevo Script, por defecto éste contiene la función Update(), unque tenemos otras opciones / eventos comunes disponibles, como:

- *FixedUpdate*(): se ejecuta en intervalos regulares.
- *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*, si el script es habilitado .
- *OnCollisionEnter*(): Cuando el objeto de juego colisiona con otro objeto de juego.
- *OnMouseDown*(): Cuando el ratón hace clic sobre un objeto de juego que contiene un 'GUIElement' o un 'Collider'.
- *OnMouseOver*(): Cuando el ratón se coloca sobre un objeto de juego que contiene un 'GUIElement' o un 'Collider'.

Escenas: Contienen los objetos del juego. Representan un nivel, con ambiente, obstáculos, y decoraciones propios. Para guardar la escena pulsa Control+S o File > Save escene. Son guardadas como assets y se muestran en la carpeta de Assets. Para abrir la escena, hacer doble click en el asset de la escena.
GameObjects: Son objetos del juego. Tienen propiedades y pueden contener componentes que los dieferencian. Un GameObject siempre tiene el componente *Transform* adjunto (para representar la posición y orientación) y no es posible quitar esto. Los otros componentes que le dan al objeto su funcionalidad pueden ser agregados del menú *Component* del editor o desde un *script*. También hay muchos objetos útiles pre-construidos (figuras primitivas, cámaras, etc) disponibles en el menú *GameObject* > *3D Object* , ver Primitive Objects.
Por ejemplo, un objeto de cubo sólido tiene un componente Mesh Filter y Mesh Renderer, para dibujar la superficie del cubo, y un componente Box Collider para representar el volumen sólido del objeto en términos de física.
Componentes: Transform: Siempre adjunta a un *GameObject* y define su posición, rotación, y escala. Otros componentes como Camera Component, GUILayer, Flare Layer, Audio Listener, Rigidbodies, Colliders, Particles proporcionan funciones adicionales al GameObject.
Ejercicio:



- Crear un objeto vacío: Gameobject > *Create empy*. Ponle el nombre: *Bola* →
- Añádele el componente: Component> Mesh> *MeshFilter* para añadir una malla con forma 3D. Para seleccionar la forma, selecciona la malla: *sphere*.
- Para que se vea esta malla en la escena, hay que añadir un componente de renderización o visualización del modelado. Escoge del menú: Componente > Mesh > *MeshRenderer*. Para poder apreciar su volúmen, en la sección *Materials*, escoge en *Element*: el material Default.
- Añadir un componente rigibody (sólido) al objeto: Escoge del menú: Component->Physics->Rigidbody
- Pulsa Play, y observa cómo la posición Y va disminuyendo, y el objeto cae por efecto de la gravedad.
- Añadir un componente *Collider* al objeto: el Collider permite que el Rigidbody se choque e interactúe con otros Colliders. Escoge del menú: Component->Physics-> *Shere Collider*.
- Guradar el trabajo: Para guardar estos objetos, sólo es necesario guardar la escena que los contiene: Control+S.

Entradas input: Representa la entrada de control del juego, mediante teclado, joystick o gamepad.
Cada proyecto tiene el siguiente eje input por defecto cuando es creado:

- Horizontal y Vertical son asignados a las teclas w, a, s, d y las teclas de flecha.
- Fire1, Fire2, Fire3 están asignadas a las teclas Control, Option (Alt), y Command, respectivamente.
- Mouse X y Mouse Y están asignados al movimiento delta del mouse.
- Window Shake X y Window Shake Y es asignado al movimiento de la ventana.

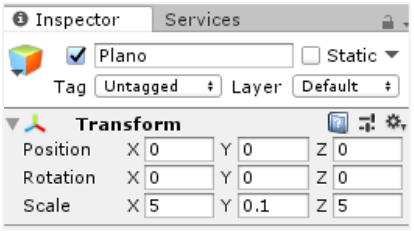
Para calibrar, añadir del menú Edit->Project Settings->Input y cambiar los ajustes para cada eje 3D X,Y,Z.
Usando Ejes de Input (Input Axes) desde los Scripts: value = Input.GetAxis ("Horizontal");
Un eje tiene un valor entre –1 y 1. La posición neutral es 0
Para identificar las teclas son los mismo de la interfaz del scripting: value = Input.GetKey ("a");
En dispositivos móviles, la clase *Input* ofrece acceso al input de la pantalla táctil, acelerómetro y geográfico/ubicación.

Ejercicio 1. Mover un objeto en el juego.

- Creamos un nuevo proyecto: ***+ New*** *Proje*ct o del menú: *File – New Project*.
  Seleccionamos la localización del proyecto y pulsamos en Create project. La nombramos "***ejercicio1***".
- Creamos una escena. *File->New Scene*. La nombramos "***escena1***

Entorno del jugador:

Para situar la escena, crearemos una superficie en la que camine el usuario con forma de cubo.



- Crea un cubo (Create – 3DObject – Cube)
- Desde el *inspector*, *escala* x, y, z a 5, 0.1, y 5 respectivamente, para ampliar y aplastar el cubo. (Equivaldría a un suelo de 5x5 metros)
- Desde el inspector, cambia el cuadro del nombre del objeto por: *Plano*.
- Crea una esfera (Create – 3DObject – Sphere) y colócala en el centro del plano. Si no puedes ver los objetos desde la Vista de Juego (***Game View***), cambia la cámara principal para que esté todo visible. Renombra elobjeto como *Cubo1*.
- Orbita y encuadra, manteniendo el botón derecho del mouse.
- También deberías crear un punto de luz (Create – Light – Point light) y colocarlo encima de los objetos de forma que sean visibles más fácilmente. (Tirando del gizmo verde vertical hacia arriba).

Escribir el Script

Vamos a mover la esfera. Para esto vamos a escribir un script que leerá la dirección desde el teclado y luego adjuntamos (asociamos) el script a la esfera.

- Comenzar creando un script vacío. Elije: ***Assets->Create->C#Script*** (Si instalaste el complemento de *Microsoft Visual Studio C#)* y renombra este script como *Movimiento1* en el *Panel Project.*
  Otra opción es crear el componente sobre el objeto: Con la esfera seleccionada, desde el inspector escoger: ***Add component – New script*** *Movimiento1.*
- Si instalaste el complemento de *Microsoft Visual Studio C# script*, haz *doble click* sobre el Script *C#.* Se abrirá en C# con las funciones *Start()* y **`Update()`** ya insertadas. Utilizaremos la función ***Update*** que es el comportamiento por defecto para cualquier código que queremos que se ejecute a cada frame del juego.
- Para mover un objeto necesitamos cambiar la propiedad de *posición* de su *transform*: **`transform.Translate(x,y,z)`** con sus 3 parámetros los movimientos x, y, z. Si queremos mover el objeto con las teclas del cursor, simplemente asociamos código de teclas del cursor para los respectivos parámetros:

```
void 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.
En Unity, el eje "Y" es el vertical, de ahí 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.`** Estos nombres pueden ser fácilmente cambiados en **`Edit->Project Settings->Input`**.
-Abre el C#Script *Movimiento1* y escribe el código superior, presta atención a las mayúsculas.

Adjuntar o asociar el script al objeto:
Si no has creado el script en el objeto, debes asociarselo: haz click en la esfera desde la *Vista de Jerarquía* (**`Hierarchy View)`** o desde la *Vista de Escena* (**`Scene View`**).
Con la esfera seleccionada, elige del menú: **Components->Scripts**: Movimiento1. En la Vista de Inspector (Inspector View), verás que aparece el apartado: ☑ Movimiento 1 (script).
Truco: puedes también asignar un script a un objeto arrastrando el script desde la Vista **`Project`** sobre el objeto en la **`View Scene`**.
-Pon en marcha el juego (presiona el icono 'play' en la parte superior), deberías ser capaz de mover la esfera con las teclas del cursor o W, S, A, D.
Como probablemente, la esfera se mueva demasiado rápido, vamos a buscar una mejor forma de controlar su velocidad.

Como el anterior código estaba dentro de la función Update(), la esfera 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 por la función `Input.GetAxis()` por el *tiempo Delta* y también por la velocidad a la que queremos movernos por segundo:

```
float speed = 5.0f;   //variable decimal flotante
void Update() {
        float x = Input.GetAxis("Horizontal") * Time.deltaTime * speed;
        float z = Input.GetAxis("Vertical") * Time.deltaTime * speed;
        transform.Translate(x, 0, z);
    }
```

-Actualiza el script Movimiento1 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) o pública, y aparecerá en el `Inspector View` para cualquier objeto de juego al que esté adjunto el script
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

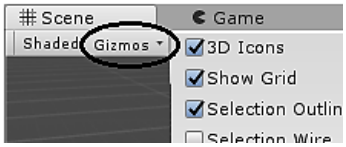
Conectar variables a través de GUI es una característica muy poderosa de Unity. Permite que las variables que normalmente estarían asignadas en código puedan ser hechas mediante el drag and drop (arrastrar y tirar) en la GUI de Unity.
Necesitaremos exponer una variable en nuestro código de script para poder asignar el parámetro en el **`Inspector View`**.
Vamos a demostrar el concepto de conectar variables creando un foco de luz que seguirá al objeto 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 Script y nómbralo 'Follow' (Seguir).
Para que nuestro nuevo foco de luz apunte a un objeto, hay una función en Unity que lo hace: ***transform.LookAt().***
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`** script debería parecerse a esto:

```
Transform target;         //variable expuesta del tipo 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 esfera desde la Vista de Jerarquía (**`Hierarchy View`**) hasta la variable "**`target`**" en la Vista de Inspector (**`Inspector View`**).
Pon en marcha el juego. Si miras la Vista de Escena (**`Scene View`**) deberías ver el foco de luz siguiendo a la esfera. Puedes que quieras cambiar la posición del foco para mejorar el efecto.
Guarda el proyecto: *File – Save Project*.

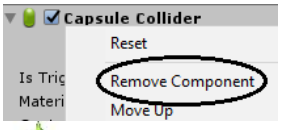
Tutorial de objetos. Game Objects y Scripts. Crear y animar un reloj

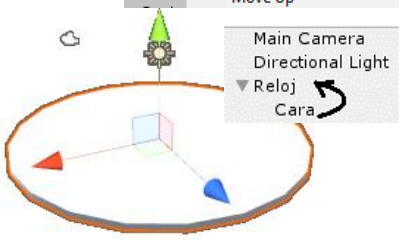
- Abre *Unity* y crea un nuevo proyecto en 3D llamado *Reloj*, con el diseño predeterminado.
- Desactiva ✓ *Show Grid* en la ventana *Scene*, a través de su menú desplegable *Gizmos*.
  En el panel *Hierachy* / *Jerarquía*, puedes ver los dos objetos predeterminados: la cámara principal y la Luz Direccional.



- Escoge del menú: *GameObject* - ***Create Empty*** para agregar un nuevo objeto.
  En el *inspector* sólo se verán las propiedades de *Trasnform* y el objeto es inerte.
  Cambia el nombre y llámale ***reloj***.

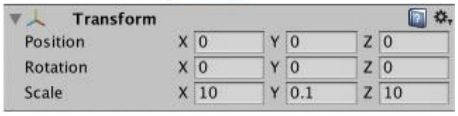
- Agrega un cilindro a la escena: *GameObject* - *3D Object* - ***Cylinder***.
  Este objeto tiene un *Mesh filter/ filtro de malla* para la forma, un *Capsule Collider*, que lo puedes quitar ya que no necesitamos colisión, y un *Mesh Renderer* para la visualización.





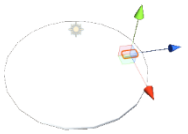
- Cambia el nombre del cilindro a ***Cara***, ya que representará la cara del reloj. Como es un subobjeto secundario del reloj, arrastra la cara sobre el reloj en la ventana de *Hierachy* / *Jerarquía*.

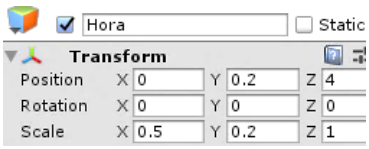
- Aplana el cilindro, disminuyendo el componente **Y** de su escala a 0.1 y aumenta X y Z a 10.



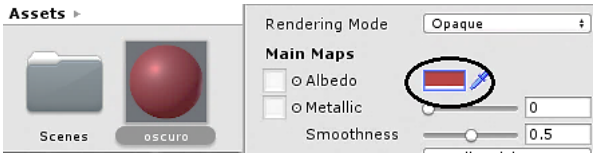
Para poner las horas:

- Agrega un cubo a la escena: *GameObject* - *3D Object* - ***Cube***.
  Cambia su escala a (0.5, 0.2, 1) para hacerlo largo y estrecho.
  Nombrarlo ***hora***. Establece su posición en (0, 0.2, 4).
  Arrastra la hora sobre la cara del reloj, en *Hierachy*.
  Rota la vista [Alt + ratón] para que corresponda a las 12.



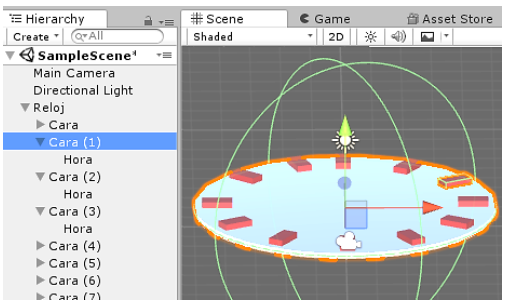


- Para verlo mejor le añadiremos un material, desde: Assets - Create - ***Material***. Ponle el nombre: *Oscuro*. Y cambia el color *Albedo* a azul oscuro. *Albedo* es una palabra latina que significa *blancura*. Es simplemente el color de un material, también llamado *difusa*.
  Arrastra el material sobre el objeto.



- Para ponerlo en la una, entemos que mover el indicador a lo largo del borde de la cara para alinearlo con la hora 1. Es más fácil girar la cara, que es el objeto padre que lo contiene, a 30 grados: Selecciona la cara y cambia su propiedad Trasform.**Rotation** Y a **30º**.

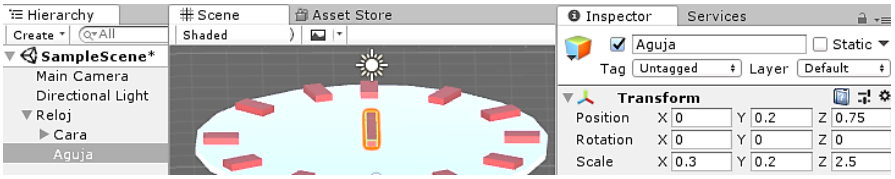
- Como es más fácil rotar la *cara* (el contenedor o padre) que el objeto *hora*, Duplica el objeto *cara*, con Control o con el menú contextual (botón derecho del mouse) en Hierachy: ***Duplicate***. Aumenta la rotación **Y** de la cara duplicada en otros **30** °. Sigue haciendo esto hasta que termines con un indicador por hora.



- Una vez terminado, arrastra los indicadores de hora sobre la primera cara original y borra el resto de las caras temporales (tecla Mayusc – Delete).

Para crear las agujas del reloj:

- Crea otro cubo (*3D Object- Cube*) llamado *Aguja* y ponle el mismo material oscuro que las horas. Establece su escala y posición como en la imagen, para que se encuentre en la parte superior de la cara y señale las 12.



- Para hacer que el brazo gire alrededor del centro del reloj, crea un objeto vacio que será el padre o contenedor de la aguja, con su transformación de 0 para posición y rotación y 1 para escala. Debido a que rotaremos el brazo más tarde, conviértelo en un padre del reloj y nómbrelo *AgujaHoras*. Así que Aguja termina como un nieto de Reloj.

- Duplica (*Duplicate*) *AgujaHoras* dos veces para crear *AgujaMinutos* y *AgujaSegundos*. El brazo de minutos debe ser estrecho y más largo que el brazo de horas, por lo tanto, establezce la escala de la aguja hija en (0.2, 0.15, 4) position (0,2,1). Y la aguja hija de *AgujaSegundos*: escala (0.1, 0.1, 5) y posición z la subes a 1.25
- Para diferenciar aún más el segundero, crea un material (*Create – Material*) nombre: *RojoPuro* con los valores RGB de su *albedo* configurado en (255, 0, 0) y asígnaselo a la aguja de *AgujaSegundos*.
- Guarda la escena (*Save Scenes*). Se almacenará como un *asset* / *activo* en el proyecto.

Animar el reloj

- Los componentes se definen a través de scripts. Agrega un nuevo recurso de script al proyecto (*Assets - Create - C # Script*) y asígnale el nombre Clock. Pulsa en el botón *Open*.. o *doble clic* en el Asset Clokc para abrir el editor.
- El archivo de script contendrá un código predeterminado en C# (*Nota*: el otro lenguaje de programación compatible, generalmente conocido como JavaScript, pero su nombre real es UnityScript, desaparecerá en próximas versiones).
- Borra todo el código predefinido para empezar desde cero.

Un componente se identifica en un script. Un script vacio no es válido ya que debe tener, al menos, la definición del componente. Nuestro componente es el reloj. No definiremos nuestro reloj particular, sino que creamos una clase o tipo general de relojes conocido como class="type". Así podríamos crear varios componentes de este tipo en Unity.
En C #, definimos el class="type" diciendo primero que estamos definiendo una clase, seguido de su nombre: **`class reloj`**
Como no queremos restringir el acceso a nuestro componente desde un archivo anexo o dll, le daremos un dominio publico: **`public class Clock`** `para declaralo.` El bloque de intrucciones, debe ir entre llaves: `{ }`
Por consiguiente, con este renglón es suficiente para que el script sea válido: **`public class Reloj { }`**
Guarda el archivo y vuelve a Unity. El editor de Unity detectará que el activo del script ha cambiado y activará una recompilación. Una vez hecho esto, selecciona nuestro script. El inspector nos informará que el activo no contiene una secuencia válida para Unity ya que un componente necesita unos eventos o comportamientos básicos de unity que se encuentran en la clase **MonoBehaviour** guardada en la librería `UnityEngine`. Es la clase base a partir de la cual se deriva cualquier script para Unity.

> ¿Qué significa **Mono-Behaviour**? mono-comportamiento.
> La idea es que podemos programar nuestros propios componentes para agregar un comportamiento personalizado al juego objetos. La parte “mono” se refiere al modo de uso de la plataforma del framework .NET. para compatibilidad con versiones anteriores.

Para convertir nuestra clase **`class reloj`** en un subtipo de `MonoBehaviour` tenemos que cambiar nuestra declaración:

```
public class Reloj : UnityEngine.MonoBehaviour { }
```

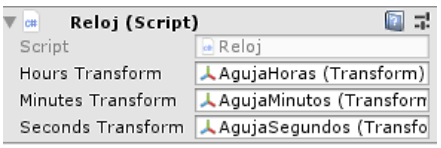
Para evitar tener que usar siempre el prefijo *UnityEngine* cuando accedemos a tipos y funciones de Unity, ponemos al principio del código las librerías y/o los namespaces. Los Namespaces permiten utilizar librerías sin conflictos con los nombres. El código viene con Unity.

```
using UnityEngine;
public class Reloj : MonoBehaviour {}
```

Ahora agregamos nuestro componente *script* a nuestro objeto de reloj arrastrando el elemento del script al objeto o mediante el botón *Add component* en la parte inferior del inspector de objetos.
Para cambiar la rotación de las agujas, usaremos tres variables: horasTransform, minutosTransform, segundosTransform. Serán del tipo Transform y mejor hacerlas públicas para que cualquiera puede cambiar sus contenidos.

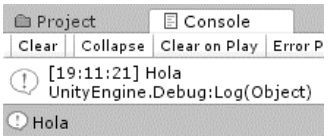
```
using UnityEngine;
public class Reloj : MonoBehaviour {
public Transform hoursTransform, minutesTransform, secondsTransform;
}
```

Actualiza el código. Al hacer las variables públicas, se convierten en un campo editable desde el *Inspector de objetos*. Para conectar el valor de la variable con cada objeto de aguja, arrastra cada aguja desde *Hierachy* al campo del *Inspector* o pulsa en el selector del campo ⊙ para asignar el objeto.



Vamos a añadir un bloque de código a la clase, conocido como método o función, al evento *Awake*, sugiriendo que el código se ejecute cuando el componente se despierte: `void Awake () { }` No recogemos ningún parámetro (void)
Para probar si esto funciona, vamos crear un mensaje de depuración, tipo test, que es una simple cadena de texto:

```
using UnityEngine;
public class Reloj : MonoBehaviour {
    public Transform hoursTransform, minutesTransform, secondsTransform;
    void Awake () {
                    Debug.Log("hola");
                }
}
```



Al poner en modo de reproducción, verás que la cadena de prueba aparece en la barra de estado inferior del editor.

Si queremos mostrar la hora, necesitamos la la función **DateTime** que se encuentra en la librería del sistema **System**.

```
using System;
using UnityEngine;
public class Clock : MonoBehaviour {
     public Transform hoursTransform, minutesTransform, secondsTransform;
     void Awake () {
             Debug.Log(DateTime.Now);
             }
}
```

En este caso se mostrará en la consola y en la barra de estado: 

Para mover las agujas:

Las rotaciones se almacenan en Unity como *cuaterniones*. Los indicadores de horas están establecidos en intervalos de 30 °. Para que la rotación coincida con eso, tenemos que multiplicar la hora del sistema por 30 (decimal flotante)

```
        Quaternion.Euler(0f, DateTime.Now.Hour * 30f, 0f);       //giro en Y
```

Crearemos una constante que cambia la cantidad de grados por hora. Las constantes suelen ponerse antes que las variables.

```
    const float gradosHora = 30f;  // constante decimal flotante
    public Transform hoursTransform, minutesTransform, secondsTransform;
    void Awake () {
         Quaternion.Euler(0f, DateTime.Now.Hour * gradosHora, 0f);
    }
```

Para aplicarlo a la aguja de horas, asignaremos localRotation propiedad de su componente de transformación.

- **localRotation:** rotación real de un componente de transformación, independientemente de la rotación de sus padres.
- **rotation**: rotación final de transformación en el espacio mundial, teniendo en cuenta las transformaciones de sus padres.

```
    void Awake () {
        hoursTransform.localRotation =
        Quaternion.Euler(0f, DateTime.Now.Hour * gradosHora, 0f);
    }
```

Aplicamos los mismo a los minutos y segundos que se mueven cada 30/5 = 6 grados

```
    const float gradosHora = 30f, gradosMinuto = 6f, gradosSegundo = 6f;
    public Transform hoursTransform, minutesTransform, secondsTransform;
    void Awake () {    //  --> al ejecutarse
       hoursTransform.localRotation = Quaternion.Euler(0f, DateTime.Now.Hour * gradosHora, 0f);
       minutesTransform.localRotation = Quaternion.Euler(0f, DateTime.Now.Minute * gradosMinuto, 0f);
       secondsTransform.localRotation = Quaternion.Euler(0f, DateTime.Now.Second * gradosSegundo, 0f);
    }
```

Para no repetir la función Datetime.Now tres veces, es mejor crear una variable de tiempo y asignarle el valor de Datetime.Now. Comprueba el código completo. Recuerda tener asociadas las tres variables a los tres objetos de aguja.

```
using System;
using UnityEngine;
public class Clock : MonoBehaviour {
   void Update () {  //--> al actualizarse - cambiamos Awake por Update que ejecute continuamente
      DateTime tiempo = DateTime.Now;
      hoursTransform.localRotation =Quaternion.Euler(0f, tiempo.Hour * gradosHora, 0f);
      minutesTransform.localRotation =Quaternion.Euler(0f, tiempo.Minute * gradosMinuto, 0f);
      secondsTransform.localRotation =Quaternion.Euler(0f, tiempo.Second * gradosSegundo, 0f);
   }
}
```

Guarda la escena y el proyecto.

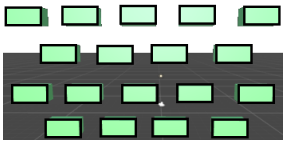
Generar y empaquetar la aplicación

- Escoge del menú: *File - Build&Run*
- Para reproducir escoge el tipo de gráfico y pulsa en *Play*
- Para parar, pulsa: Alt+F4
- Comprueba que en la carpeta de Unity/Reloj se ha creado un archivo llamado: Reloj.exe

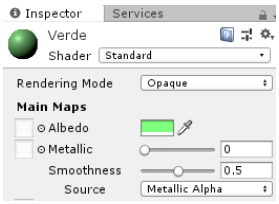
Para empaquetar, distribuir o copiar la aplicación en otro equipo, necesitarás de los archivos:
Reloj.exe (el ejecutable), Unityplayer.dll (la librería de reproducción) y las carpetas: Reloj_data y Mono
La carpeta *Assets*, no la necesitas para distribuir, contiene los archivos con los activos o herramientas para el programador (código, materiales, etc..)
Para más Info, véase: catlikecoding.com/unity/tutorials/basics/game-objects-and-scripts/

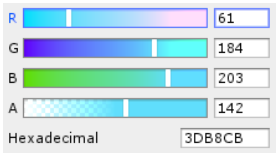
Crear un juego de estilo Breakout.

- Crear nuevo proyecto en la carpeta de Proyectos de Unity llamado ***BreakOut***. Recuerda que cada nuevo proyecto se crea en una carpeta nueva.
- Crear un cubo para la pala: (*Create – 3DObject – Cube*)
  Scale: 3,2,1 Nombre: ***Pala***
- Crear 1 cubo alragado para la pared izquierda
  Nombre: ***pared1*** (Pos: -10,0,0 Scale: 1,20,1)
- Duplicar (*Edit – Duplicate*) para la pared derecha y desplazar a la posición: 10,0,0 Nombre: ***pared2***
- Duplicar (*Edit – Duplicate*) y rotar el cubo sobre el eje z para el techo:
  Nombre: ***Techo*** (Position: 0,10,0, Rotation: 0,0,90, Scale: 1,24,1)
- Crear el ladrillo (*Create – 3DObject – Cube*) Brick
  Scale 2,1,1 Duplicar 17 veces en 4 filas = 18 total
  Agrupar todos los ladrillos, arrastrándolos sobre el primero desde el explorador *Hierachy*.



- Crear esfera para la bola: (*Create – 3d object – Sphere*) Scale: 1,1,1 Nombre: ***Bola***
- Crear un cubo para el suelo: (Pos: 0, -22, 0 - Scale: 50, 20, 50) Nombre: ***Suelo*** *o DeathZone*
- Crear una carpeta en Assets llamada *materiales* (*Assets – Create – Folder*)
  Dentro de la carpeta, crear 3 materiales (Assets - *Create – Material*) llamados: Rojo, suelo y verde. Escoger el color desde el inspector: Rojo, azul y verde. Asignar cada material al objeto correspondiente, arrastrando el material encima del objeto.



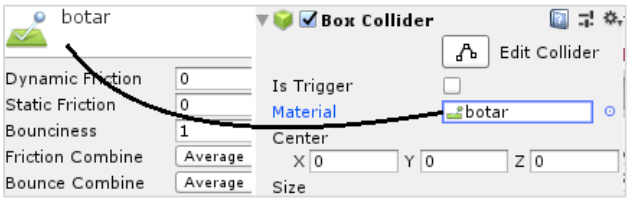


Al suelo le podemos dar una sensación de azul líquido variando el valor alfa de transparencia.

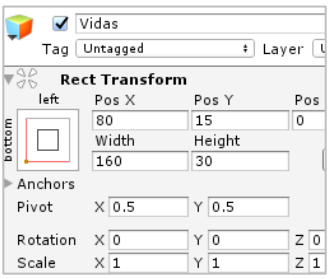
- Seleccionar la pala y añadir el componente *RigidBody* (*Component – Physics – **Rigibody***) para convertirla en cuerpo sólido o cuerporígido. Añadir también este componente a la Bola.
- Añadir a la pala el Script de movimiento con las flechas: *Add component - New Script* o *Assets – Create – C#Script:*

```
public class Pala : MonoBehaviour {
    public float velocipala = 1f;  //variable pública para la velocidad de la pala del tipo numérico decimal
    void Update () {        // es llamado en cada impulso
        float Posx = transform.position.x + (Input.GetAxis("Horizontal") * velocipala); //coge variabe Posx
        transform.position = new Vector3(Posx, -9f, 0f); //movemos la pala a la nueva posición del objeto
     }
```

  Arrastrar este Script sobre la pala. Al correr el juego (ejecutar), debería de moverse la pala hacia los lados.
- Añadir efecto físico de rebote*:*
  En la carpeta de *Materiales* añadir material físico (Assets - Create – Physics material) Nombre: botar. Establecer los valores de la imagen.
  Añadir a la pala el efecto de colisión: (*Component – Physis – **Box collider***) y arrastar *botar* hasta el cuadro *Material* de *Box Collider*
  Añadir el mismo *efecto* de colisión a las dos paredes, el techo y a todos los ladrillos.



  Añadir a la bola el efecto de colisión esférica: (Component – Physis – ***Sphere collider***) y arrastar *botar* hasta *Material*.
  Radius: 0.5
- Añadir texto (*GameObject – UI – **Text***). Aparecerá en canvas con un eventsystem
  Para la puntuación: Nombre: Vidas, Text: Lives: 3. Poner tamaño/ posición de la imagen →
  Añadir otro texto. Nombre: GameOver, Text: GAME OVER, fuente: Arial 40 bold negrita
  Desactiva la casilla junto al nombre ✓ para que no se muestre al principio.
  Añadir otro texto. Nombre: YouWon texto: GANASTE - YOU WON !! desactiva disable ✓



- Añadir el efecto de sistema de partículas para la caída (*GameObject – Effects – Particle Systems*)
  Nombre: *ParticulasBrick.* Rotation x: 90 (hacia abajo). sin ✓ loop, duración: 1
  Activar: rotación y tamaño mientras dure: ✓ Rotation over lifetime ✓ Size over lifetime
  Puedes añadir un nuevo material en la carpeta de materiales (*Create – Material*) para asignar a las particulas el color rojo oscuro.
- Añadir el objeto vacío (Game object – Create empty) **GM** en la posición 0,0,0. Para el control del juego Game Management

Extracto de: unity3d.com/es/learn/tutorials/modules/beginner/live-training-archive/creating-a-breakout-game?playlist=41639

Uses: using UnityEngine;

public class **Pala** : MonoBehaviour {
```
  public float velocipala = 1f;
  private Vector3 posicionjugador;
  void Update () {
  float Posx = transform.position.x +
(Input.GetAxis("Horizontal") * velocipala);
  posicionjugador = new Vector3(Posx, -9.5f, 0f);
  transform.position = posicionjugador;
  }
}
```

public class **Bola** : MonoBehaviour {
```
  public float velocidadinicial = 600f;
  private Rigidbody rb;
  private bool ingame;
  void Awake () {// Use this for initialization
    rb = GetComponent<Rigidbody>();
    }
    void Update () {
    if (Input.GetButtonDown("Fire1") && ingame ==
false)
        {
        transform.parent = null;
        ingame = true;
        rb.isKinematic = false;
        rb.AddForce(new Vector3(velocidadinicial,
velocidadinicial, 0));
        }
    }
}
```

public class **Bricks** : MonoBehaviour
```
{
    public GameObject brickParticle;
    void OnCollisionEnter(Collision other)
    {
     Instantiate(brickParticle, transform.position,
Quaternion.identity);
     GM.instance.DestroyBrick();
     Destroy(gameObject);
    }
}
```

public class **Suelo** o DeadZone : MonoBehaviour
```
{
    void OnTriggerEnter(Collider col)
    {
        GM.instance.LoseLife();
    }
}
```

public class **GM** : MonoBehaviour
```
{
    public int lives = 3;
    public int bricks = 18;
    public float resetDelay = 1f;
    public Text livesText;
    public GameObject gameOver;
    public GameObject youWon;
    public GameObject bricksPrefab;
    public GameObject paddle;
    public GameObject deathParticles;
    public static GM instance = null;
    private GameObject clonePaddle;
    void Awake()    // Use this for initialization
    {
        if (instance == null)
            instance = this;
        else if (instance != this)
            Destroy(gameObject);
        Setup();
    }
    public void Setup()
    {
     clonePaddle = Instantiate(paddle,
transform.position, Quaternion.identity) as
GameObject;
     Instantiate(bricksPrefab, transform.position,
Quaternion.identity);
    }

 void CheckGameOver()
    {
        if (bricks < 1)
        {
           youWon.SetActive(true);
           Time.timeScale = .25f;
           Invoke("Reset", resetDelay);
        }

        if (lives < 1)
        {
           gameOver.SetActive(true);
           Time.timeScale = .25f;
           Invoke("Reset", resetDelay);
        }
    }
    void Reset()
    {
     Time.timeScale = 1f;
     Application.LoadLevel(Application.loadedLevel);
    }

    public void LoseLife()
    {
     lives--;
     livesText.text = "Lives: " + lives;
     Instantiate(deathParticles,
clonePaddle.transform.position, Quaternion.identity);
     Destroy(clonePaddle);
     Invoke("SetupPaddle", resetDelay);
     CheckGameOver();
    }
    void SetupPaddle()
    {
    clonePaddle = Instantiate(paddle,
transform.position, Quaternion.identity) as
GameObject;
    }
    public void DestroyBrick()
    {
        bricks--;
        CheckGameOver();
    }
}
```