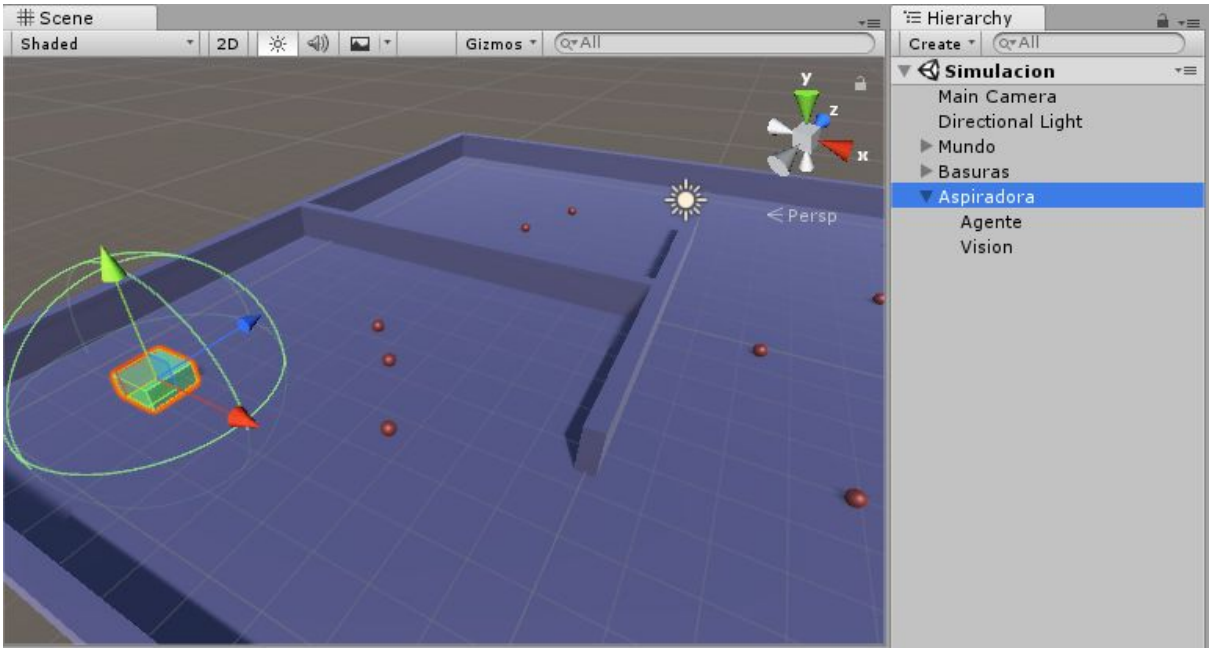
Facultad de Ciencias, UNAM
Inteligencia Artificial 2018-2
Proyecto con agentes
19 de febrero de 2018

Proyecto en Unity “Aspiradora”

En la plataforma Moodle se encuentra un archivo que contiene un proyecto en Unity con un prototipo del proyecto a trabajar para la materia de inteligencia artificial.

El objetivo del proyecto en Unity es utilizar elementos relativamente básicos de este framework como movimiento y detección de colisiones para programar un agente primordialmente mediante scripts que pueden comunicarse entre ellos y dotar de un comportamiento al agente.

Al abrir el proyecto desde Unity podemos notar que hay un *Scene* llamado **Simulacion** en la carpeta *Assets* > *Scenes* donde se encuentra todo el material.



La simulación consiste de un mundo simple con una “Aspiradora” actuando como un **agente**, algunas esferas desperdigadas y un par de paredes.

El aspecto más importante a considerar son los scripts del Agente:

1. Actuadores.cs
2. Sensores.cs
3. Comportamiento.cs

Como sus nombres lo indican, Actuadores.cs y Sensores.cs brindan la funcionalidad para que el agente pueda actuar sobre sí mismo y en el mundo, mientras que el segundo brinda la capacidad de percibir su entorno. Estos script solamente incluyen la firma de los métodos, en la sesión de laboratorio ya se han implementado los métodos necesarios:

- Actuadores
    - MoverAdelante: mueve el agente al frente.
    - MoverAtras: mueve al agente hacia atrás.
    - GirarDerecha: gira al agente en el sentido de las manecillas del reloj.
    - GirarIzquierda: gira al agente en contra del sentido de las manecillas del reloj.
- Sensores
    - TocandoBasura: indica si el agente está tocando una basura.
    - TocandoPared: indica si el agente está tocando una pared.
    - CercaDeBasura: indica si el agente está cerca de una basura.
    - CercaDePared: indica si el agente está cerca de una pared.

El script Comportamiento.cs ya está implementado y espera la intervención del usuario (espera Input desde el teclado para moverse). Pero en realidad solamente manda a llamar a los métodos de las clases anteriores. Por lo tanto, al implementar de manera correcta cada método de las clases Actuadores.cs y Sensores.cs, el usuario podrá interactuar y mover al agente; los sensores imprimirán textos en la consola si llega a ocurrir alguno de los eventos mencionados.

Visión del Agente

Collider esférico

Para dotar al agente un visión radial extendida es necesario incluir un *Collider* adicional que nos permita realizar detección de colisiones abarcando más espacio. Este nuevo *Collider* se integra como un gameObject nuevo que se mueve a la par del agente.

La necesidad de detectar estos diferentes eventos de colisión hace que el *GameObject* **Vision** contenga su propio *script* para detectar estas colisiones. A continuación se presenta parte del código:

```
private bool isNearBasura = false;

void OnTriggerStay(Collider other){
    if(other.gameObject.CompareTag("Basura"))
        isNearBasura = true;
}
void OnTriggerExit(Collider other) {
    if(other.gameObject.CompareTag("Basura"))
        isNearBasura = false;
}
public bool CercaBasura(){
    return isNearBasura;
```

```
}
```

Es decir, el script utiliza una bandera (variable isNearBasura) para guardar el estado de intersección con los objetos que representan la basura. El método CercaBasura nos indica el valor de la variable y es público para que otros *scripts* puedan hacer uso de él.

El *script* Sensores emplea esta información para informar correctamente cuando la visión del Agente está suficientemente cerca de una esfera que representa Basura. De modo que parte del script Sensores quedaría de la siguiente manera:

```
private VisionController radar;

void Start(){
    radar =
GameObject.Find("Vision").gameObject.GetComponent<VisionController>();
}

public bool CercaDeBasura(){
    return radar.CercaBasura();
}
```

Y de esta manera utilizamos métodos de otros scripts que a su vez son capaces de detectar eventos importantes como colisiones y brindamos sensores adicionales en la simulación que servirán para la toma de decisiones. La detección de una pared cercana es análoga.

Rayo

Otra forma de agregar visión al agente es mediante una línea que se “dispara” al frente y detecta los gameObjects con los que colisiona. Este comportamiento simula la vista al frente que podría otorgar un ojo o una cámara.
Unity permite agregar este tipo visión mediante el método **Raycast**. A continuación se presenta parte del código que se incluye en el script *Sensores.cs*:

```
private bool isTouchingBasura = false;
public float rayDistance;

void FixedUpdate(){
    RaycastHit raycastHit;
    if(Physics.Raycast(transform.position, transform.forward, out
raycastHit, rayDistance)){
        if(raycastHit.collider.gameObject.CompareTag("Basura"))
            isInFrontOfBasura = true;
    }else{
        isInFrontOfBasura = false;
    }
}
```

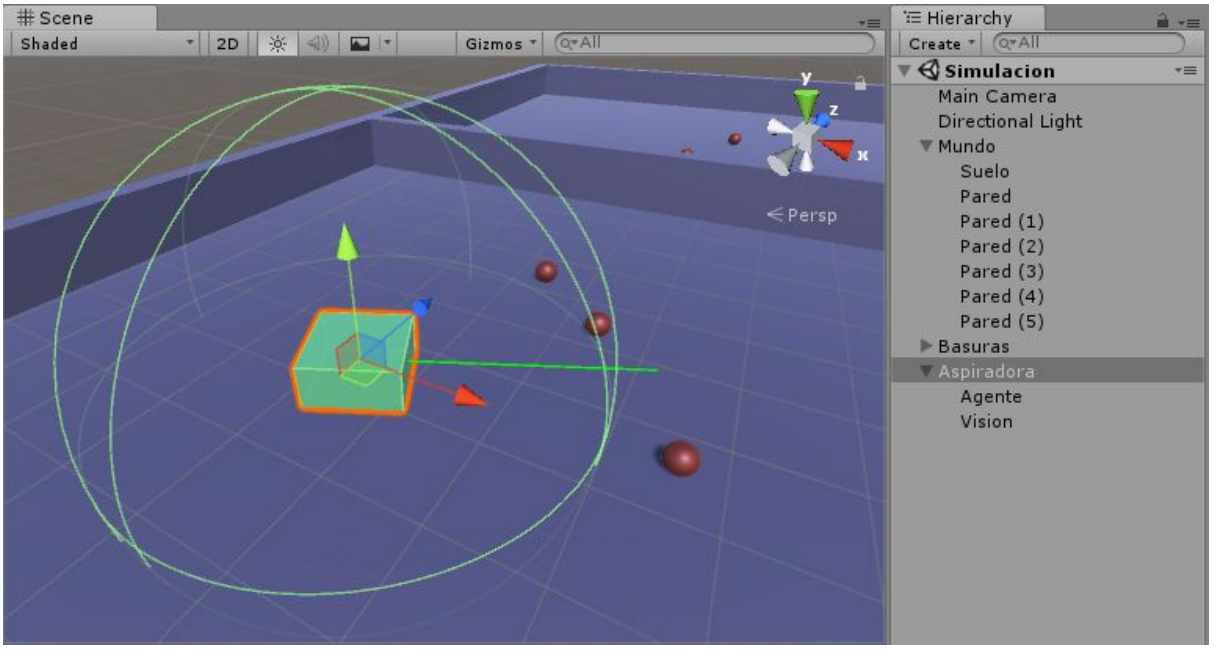
```
public bool FrenteBasura(){
    return isInFrontOfBasura;
}
```

Se define una variable para guardar el valor booleano si es que hay una basura frente a nuestro agente, así como una variable pública para controlar la longitud del rayo que se disparará.
Dentro del método *FixedUpdate* definimos una variable de tipo *RaycastHit* que guarda la información de la colisión del rayo con otros gameObjects. Usando el método **Physics.Raycast** indicamos el punto de origen del rayo (el agente), la dirección del rayo (al frente del agente), la variable con toda la información de las colisiones y el tamaño o longitud del rayo.

El resto del código consiste en verificar la etiqueta del gameObject para asegurar que es una basura o el uso de un método público para obtener dicha información.

Una consideración importante es que los rayos no se muestran en la pantalla de juego, pero pueden ser mostrados en la vista *Scene* para hacer *debug*. Esto ayuda a corroborar la información de las variables con un referente visual del comportamiento esperado. Logramos este dibujado en pantalla al seleccionar el gameObject agente e incluir el siguiente código en Sensores.cs:

```
void Update(){
    Debug.DrawLine(transform.position, transform.position +
transform.forward * rayDistance, Color.green);
}
```

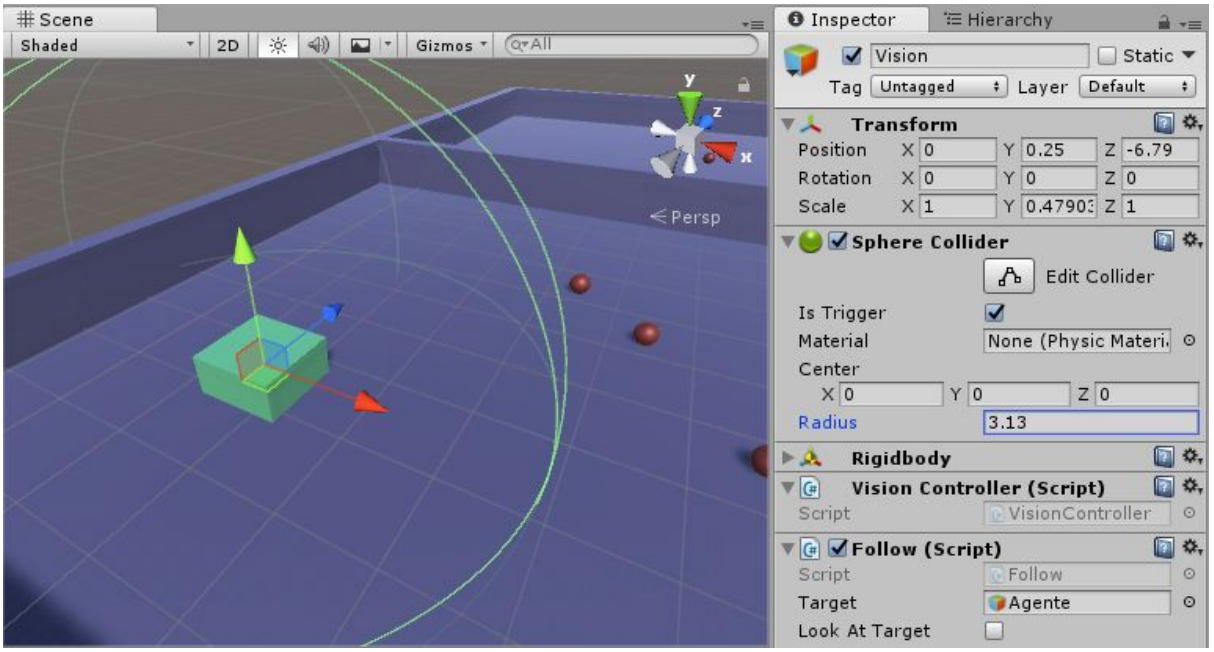


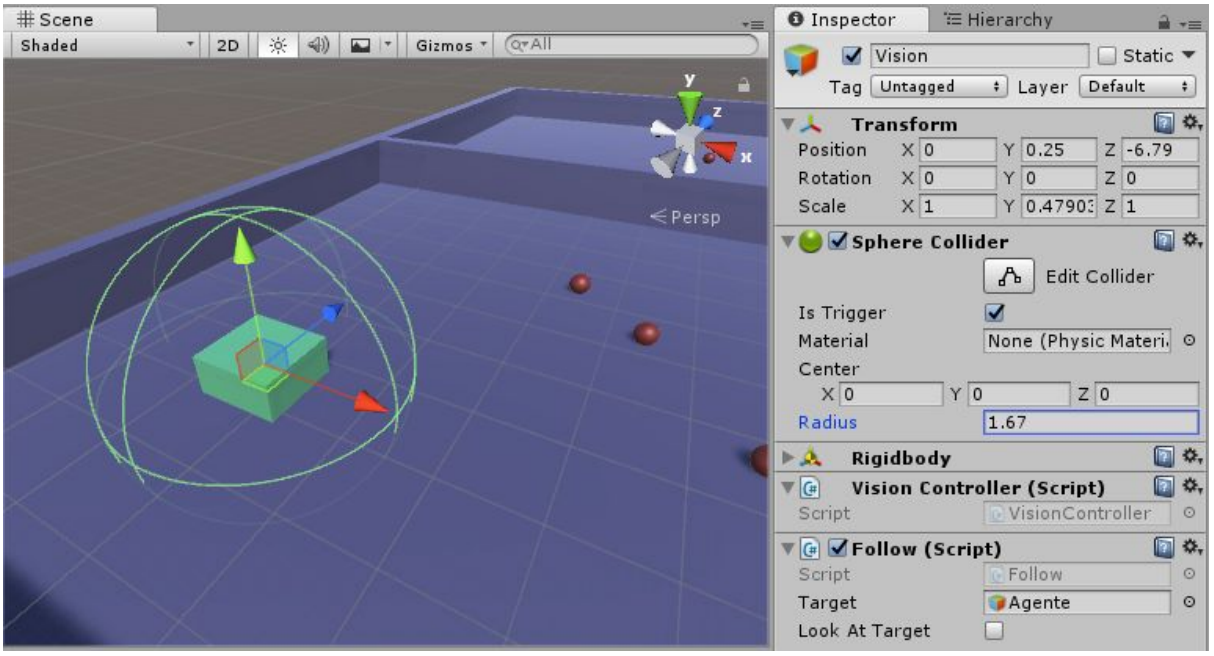
Notemos que el uso de FixedUpdate es casi el mismo que Update, se recomienda usar el método Update para cálculos numéricos y dibujado en pantalla; y usar FixedUpdate para comprobaciones y cambios del motor físico en la geometría del juego.

La comprobación para determinar que hay una pared frente al agente es análoga a la comprobación de una basura.

Configuraciones de visión

Es posible modificar el tamaño de la esfera que proporciona visión adicional al agente. Para cambiar este valor hay que seleccionar el gameObject **Vision** y en el menu del *Inspector* ubicar el componente **Sphere Collider** y modificar el valor **Radius**.





Similarmente se puede modificar la longittud del rayo seleccionando el gameObject Agente y en el menú del *Inspector* ubicar el componente **Sensores** y modificar el valor **Ray Distance**. NOTA: El rayo solo es visible en la pantalla de Scene, pero no en la vista Game.