Diseño de niveles 3D en Unity

PID_00247900

David León Molero

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

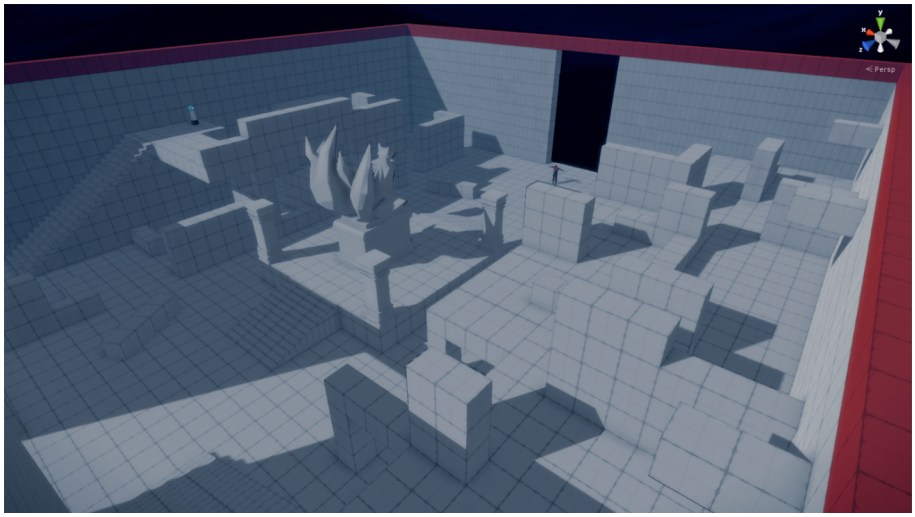
Índice

Introducción

Diseñar un nivel o escenario para un juego 2D es una tarea relativamente fácil; una simple cuadrícula de una libreta puede servir como herramienta de edición de mapas improvisada en la que colocar *tiles* cuadro a cuadro y hacer un diseño fácilmente trasladable a un juego 2D. Al aplicar una tercera dimensión, el diseño de niveles se complica de manera exponencial. Diferentes alturas, ángulos de cámara y profundidades no se llevan bien con el papel, y hay que buscar soluciones alternativas.

En esta unidad, vamos a repasar los elementos que definen la creación de niveles en un entorno 3D. Nos centraremos en aprender a hacer nuestro diseño de nivel sobre Unity y en crear *whiteboxes*: diseños provisionales que formarán la base para un futuro nivel de nuestro juego.

Imagen 1.



Todos los conceptos de esta unidad se irán explicando mediante su implementación en un caso de uso real. Utilizaremos un proyecto de aprendizaje de Unity, Survival Shooter, y lo adaptaremos para convertirlo en un nuevo juego que hará de base para nuestros experimentos como diseñadores de nivel. Durante la creación de este *mod*,irán surgiendo diversos problemas y veremos qué soluciones aplicar paso a paso.

Todo el código utilizado y cualquier otro contenido serán aportados en este documento o en los materiales asociados de la asignatura. El repositorio con el proyecto completo lo podéis encontrar en: https://github.com/xDavidLeon/uoc_leveldesign_m3

Las soluciones a los retos se hallan en el último apartado.

1. Configuración del proyecto

1.1. Software utilizado

Para realizar este unidad, utilizaremos el siguiente software:

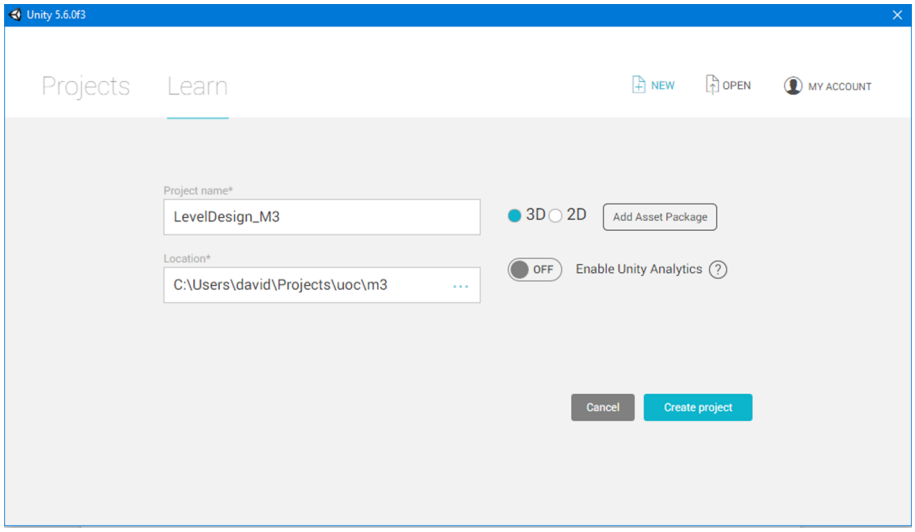
- Unity 5.6.0. Una versión anterior también nos servirá, aunque quizás algunos elementos de interfaz y código de los ejemplos pueden cambiar.
- Visual Studio Community. Se instalará automáticamente junto con Unity. MonoDevelop o XCode también deberían servir.
- SourceTree (opcional). Siempre recomendamos crear un repositorio de GIT para todo proyecto en el que se trabaje. SourceTree es una interfaz gratuita para gestionar nuestros repositorios de GitHub o similares.

Los ejemplos y capturas de pantalla incluidos en esta unidad han utilizado el software anteriormente descrito en un entorno con Windows 10. Es totalmente plausible reproducir esta unidad en OS X o Linux, y utilizando otras herramientas para edición de código o control de versiones.

1.2. Importando Survival Shooter

Lo primero de todo es crear un nuevo proyecto de Unity en alguna carpeta de nuestro ordenador. En nuestro caso, hemos llamado al proyecto LevelDesign_M3.

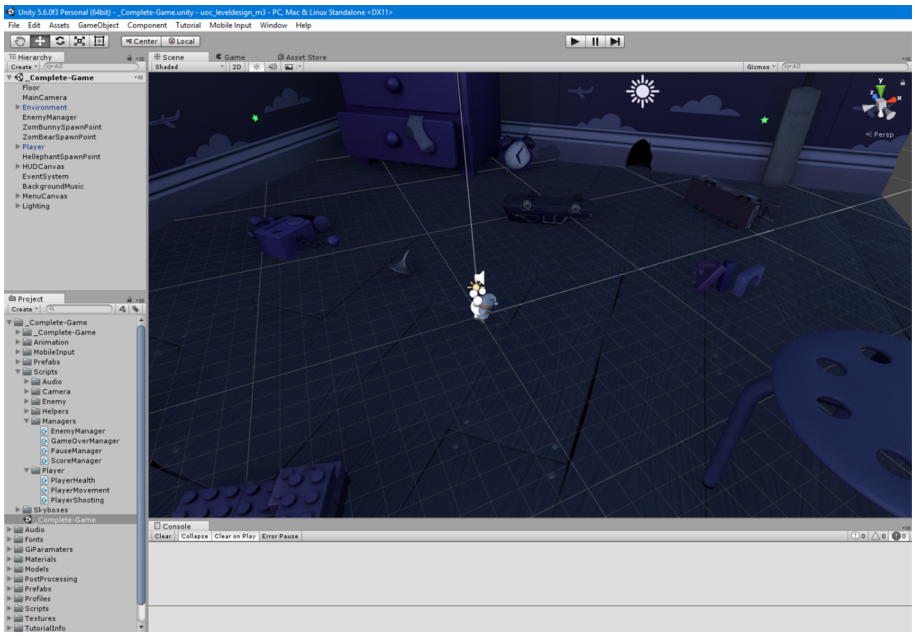
Imagen 2.



Una vez creado el proyecto, podemos ir a Asset Store (Ctrl + 9) y buscar Survival Shooter. Descargamos el proyecto gratuito y lo importamos.

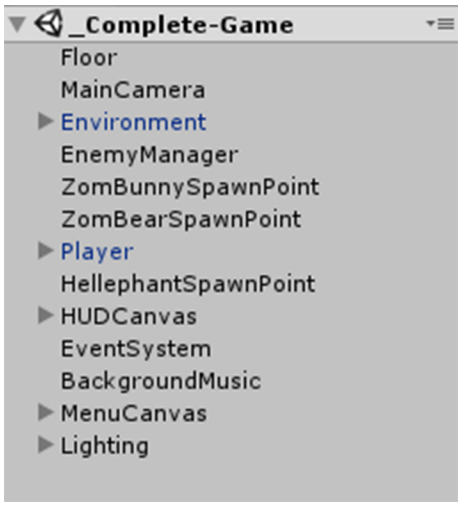
A estas alturas, deberíais poder jugar al proyecto sin problemas mediante el *play* en la escena _Complete-Game.

Imagen 3.



Analizando Hierarchy, nos encontramos con:

Imagen 4.



Antes de empezar a construir nuestra modificación, debemos aprender con qué material contamos. Dedicad unos minutos a ver en qué consiste cada *game object* de la escena. Tenemos la cámara, el jugador, interfaces, música, etc. Todos estos elementos los mantendremos relativamente intactos para nuestro *shooter* personalizado. Pero podemos ya prever que el ***environment*** (elementos que conforman el escenario) y los ***spawn points*** de enemigos (que instancian un tipo específico de enemigos cada pocos segundos) los tendremos que alterar para comenzar nuestro diseño.

2. Diseño de nivel

2.1. *The UOC Gauntlet*

Antes de empezar a crear nuestros niveles, vamos a definir el juego que queremos prototipar. *Gauntlet* fue una serie de videojuegos que ayudaron a definir el género del *hack and slash* y los *third-person shooters*. Crearemos nuestra propia rendición de *Gauntlet* con Unity, a la que llamaremos *The UOC Gauntlet*, y lo haremos modificando el proyecto de Survival Shooter para definir el *gameplay* y construir varios niveles. Podéis mirar vídeos de *Gauntlet* en YouTube a fin de obtener referencias e inspiración a la hora de montar vuestra versión de *The UOC Gauntlet*.

Nuestra pequeña lista de ***features*** será la siguiente:

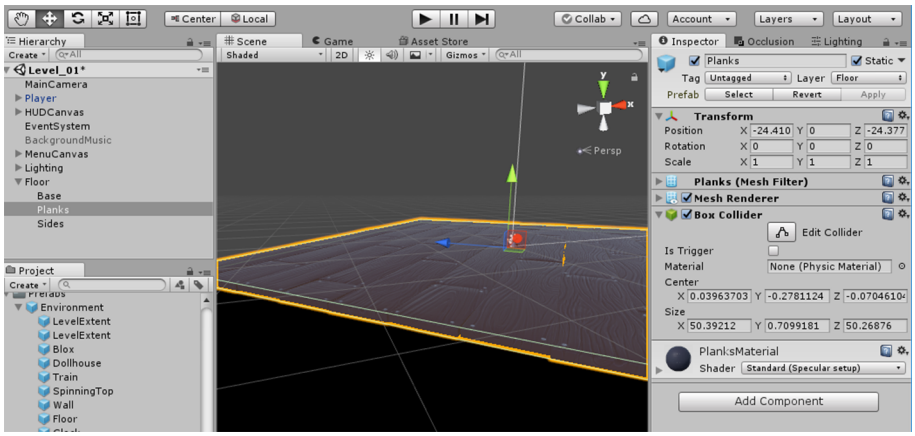
1) Cámara con plano en picado para controlar al personaje desde arriba.
2) Posicionamiento de enemigos en el nivel.
3) Tipos de enemigos más diferenciados.
4) Posibilidad de encontrar llaves que abran puertas o zonas del nivel.
5) Creación de *triggers* que nos permitan instanciar enemigos al instante (a modo de trampa) o que tengan otros efectos.
6) Colocación de ítems que regeneren la salud del personaje.

2.2. Creando nuestra escena de trabajo (*whitebox*)

Para empezar a trabajar sin peligro de sobrescribir material importante, crearemos una nueva escena. Utilizando la escena _Complete-Game, vamos a File -> Save Scene As y guardamos nuestra nueva escena en una carpeta de nuestra elección. Ahora eliminamos Environment, Floor, EnemyManager y Spawn Points.

Puesto que nos hemos quedado sin suelo, podríamos crear un *cube* de Unity y darle algún material vistoso. Otra opción es aprovechar el *floor* que hay dentro del *game object* Environment y adaptarlo para hacerlo modular. Para ello, arrastramos el *floor* fuera de Environment en Hierarchy, y luego expandimos Floor y añadimos un componente BoxCollider a Planks. También le ponemos la *layer* Floor al *game object* Planks:

Imagen 5.



Podemos arrastrar nuestro nuevo *floor* desde Hierarchy a una carpeta del Project para convertirlo en un *prefab* y poder utilizarlo de manera modular siempre que necesitemos más metros de suelo.

También necesitaremos muros. Para poder empezar a prototipar niveles lo antes posible, crearemos un *prefab* Whitebox_Wall a partir de un cubo al que le pondremos una *layer* Shootable (para que las balas puedan colisionar con el objeto) y lo marcaremos con el *toggle* de Static (para que el motor de renderizado pinte todas las paredes con un solo *draw call*).

Nota del autor

Idealmente, cuando nos encargamos de prototipar un nivel construyendo una *whitebox* (o en el prototipado general de un juego) no interesa perder tiempo embelleciendo los *props* usados ni añadiendo ningún tipo de *eye-candy*. El objetivo de un prototipo es conseguir conclusiones rápidas acerca de la viabilidad de un diseño, por lo que todo el tiempo invertido en algo que no esté directamente relacionado con este objetivo, es tiempo perdido.

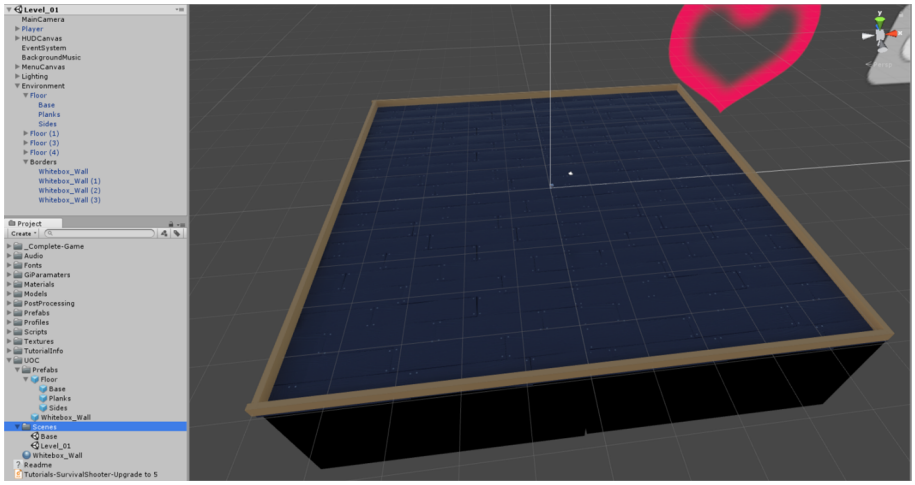
Dicho esto, habrá veces en las que añadir *assets* de arte «finales» a un prototipo o *whitebox* puede ser útil para encontrar mayor inspiración en el diseño de nivel o para comunicar mejor nuestro diseño a personas ajenas a su construcción (como el resto de artistas u otros programadores). Si estamos construyendo un juego de miedo, siempre es más fácil imaginarse cómo se sentirá un jugador si añadimos algunas tumbas y un poco de niebla...

Por otro lado, como diseñadores debemos cuidar nuestras herramientas. Si prevemos que vamos a utilizar repetidas veces una serie de *assets*, no es mala idea dedicar algún tiempo a crear unos materiales o texturas adecuados que nos faciliten el trabajo de cara al futuro.

Podemos fácilmente crear una zona amplia y vacía para reutilizar en el futuro como base de nuestros niveles. Para este ejemplo, colocaremos cuatro *floors* bordeados por una hilera de muros como límite de nivel. Recordemos algunos atajos del teclado útiles para colocar *props* en el escenario:

- Q: mover cámara.
- W: herramienta de traslación.
- E: herramienta de rotación.
- R: herramienta de escalado.
- T: herramienta de UI, muy útil para escalar objetos de manera asimétrica.
- F: centrar cámara en objeto seleccionado.
- Ctrl + Shift + F: colocar *prop* en posición y orientación de la vista de escena.
- Ctrl + Alt + F: colocar *prop* delante de la vista de escena.
- Ctrl + Shift + arrastrar *prop* en escena: mover el *prop* «pegándolo» a superficies.
- V: si aguantamos la tecla V mientras arrastramos un objeto en la escena, este se pegará a vértices cercanos.

Imagen 6.



2.3. Modificando la cámara

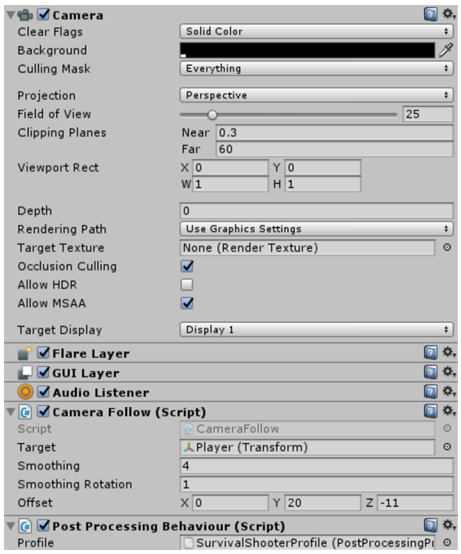
Actualmente, la cámara utiliza una perspectiva ortográfica que es bastante funcional, pero queremos modificarla a nuestro gusto. Para tener más flexibilidad, cambiamos el *script* CameraFollow de manera que la cámara utilice perspectiva y mire automáticamente hacia el jugador.

Reto 01

Cambiad las preferencias del componente Camera para que utilice una vista no ortográfica (en perspectiva). Modificad el *script* CameraFollow para asignar a la cámara un *offset* respecto a su posición normal mediante el *inspector* de Unity. Aseguraos también de que la cámara siempre mire al jugador utilizando la función LookAt.

En nuestro caso, los parámetros utilizados son los siguientes:

Imagen 7.



Y el resultado:

Imagen 8.

Nota del autor

Modificar la cámara no es, obviamente, un requisito para nuestro diseño de nivel, y ni siquiera es un requisito para crear nuestro clon de *Gauntlet*. De todos modos, va bien practicar la creación de *scripting* para modificar cosas tan críticas para el diseño de nivel como el ángulo o el comportamiento de cámara.

Decisiones como cuáles son las propiedades de la cámara o el tamaño de un nivel están íntimamente ligadas al diseño de juego, y en muchos equipos el diseñador de niveles y el diseñador de juego son personas totalmente diferentes. En el «mundo real», este tipo de decisiones, al igual que las mecánicas de juego, los elementos interactivos del nivel o los tipos de enemigos, deberán estar decididos por el diseñador de juego **antes** de empezar el diseño de nivel, ya que un buen diseño depende de estos elementos. Si diseñamos un nivel teniendo en cuenta una mecánica que no está cerrada al 100% y durante el desarrollo esta cambia o se cancela su implementación, el resultado puede ser desastroso.

Dicho esto, debido a la inexistencia de un documento de diseño de juego para este proyecto y a su naturaleza didáctica, iremos «inventándonos» las mecánicas de juego a medida que diseñemos el nivel, e implementaremos las nuevas *features* nosotros mismos.

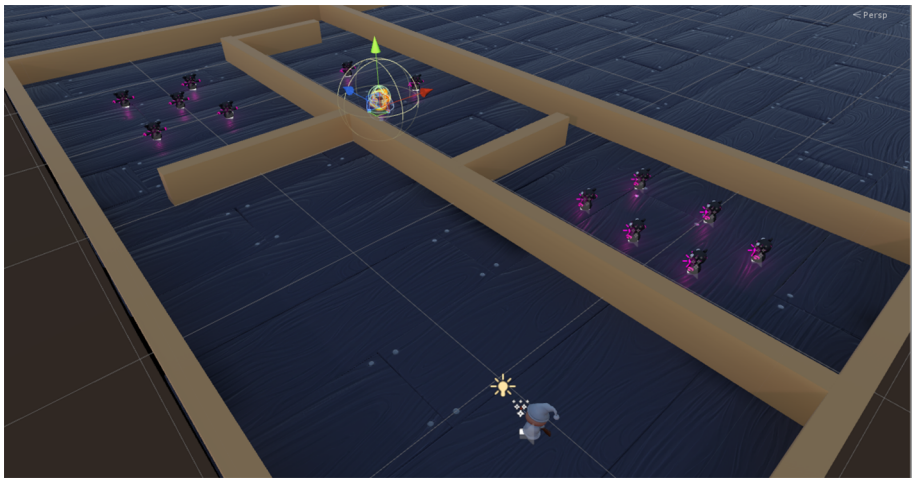
2.4. Colocación de enemigos

Tenemos a un personaje moviéndose por un escenario vacío y con una cámara que nos convence. Ahora falta algo a lo que disparar.

En _Complete-Game/Prefabs tenemos varios *prefabs* de enemigos que podemos utilizar. Soltándolos en la escena, los colocaremos estratégicamente, y al activar el *play*,empezarán a perseguir al jugador buscando un camino hacia él.

Crearemos una primera serie de habitaciones que contengan diferentes «desafíos» a modo de oleadas de enemigos encerrados. Aún no contamos con medios para cerrar salas con puertas con llave o tipos de enemigos más interesantes, pero a estas alturas lo que buscamos es definir el *layout* general de nuestro primer nivel.

Imagen 9.



Utilizando unos pocos muros, hemos definido varias habitaciones introductorias del nivel. Para enseñar al jugador cuál es el comportamiento de los enemigos, hemos puesto a un conjunto de enemigos a la vista del mismo, pero detrás de un muro, de manera que el jugador pueda ver cómo se mueven sin estar en peligro.

Actualmente los enemigos utilizan *pathfinding* para encontrar una ruta hacia el jugador. Si queremos imitar el comportamiento de la IA de *Gauntlet*, necesitamos unos enemigos mucho más simples.

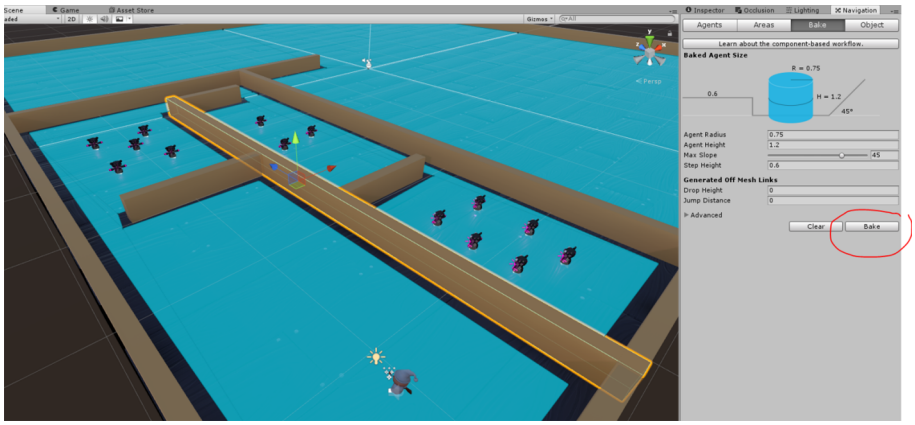
Reto 02

Adaptad el *script* EnemyMovement para que el enemigo solo siga al jugador cuando este se encuentre cerca. Para adaptarlo a una IA del estilo de *Gauntlet*, el enemigo no debe saber sortear obstáculos, por lo que no tendría que utilizar *pathfinding*.

En nuestro caso, hemos modificado EnemyMovement.cs de manera que los enemigos ya no utilicen la *navigation mesh* por defecto y simplemente vayan en la dirección del jugador cuando este se encuentre cerca.

Si queremos seguir apoyando el uso de *navigation meshes* y *pathfinding* en el futuro, deberemos actualizar la *nav mesh* cada vez que haya cambios en el nivel. Vamos a Window -> Navigation, y en la pestaña Bake hacemos clic sobre el botón Bake. Recordad que los muros deben tener una *layer* Shootable para que la malla de navegación los reconozca como obstáculos.

Imagen 10.



2.5. Enemigos más variados

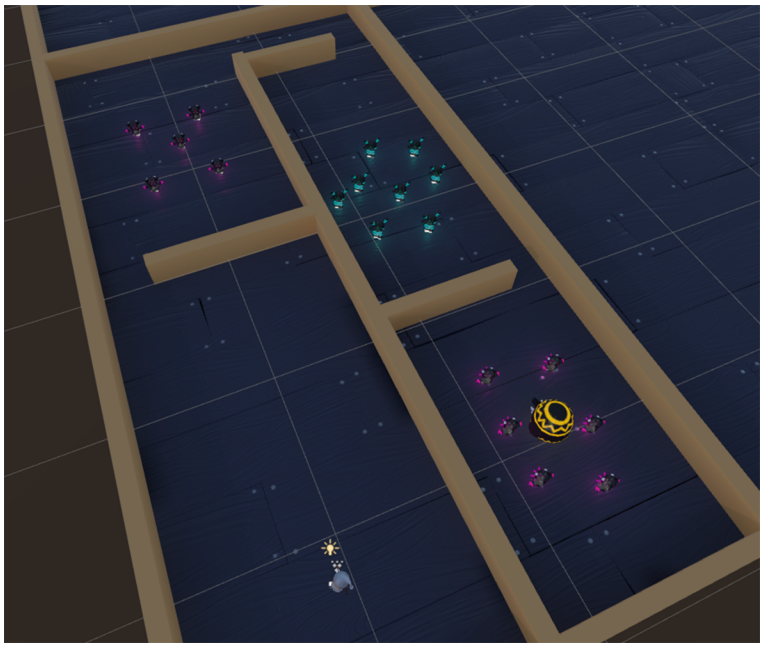
Aprovechando que ya disponemos de unos enemigos funcionales, vamos a modificar los tres tipos de enemigos que tenemos (Hellephant, ZomBear y ZomBunny) para hacerlos más interesantes. Convertiremos a ZomBunny en un enemigo muy ágil, pero con muy poca vida; a ZomBear, en el enemigo básico, y a Hellephant, en un enemigo lento, pero con gran cantidad de vida. Podemos definir estos valores cambiando las variables públicas de EnemyHealth y EnemyMovement de los *prefabs* de los enemigos que hemos de modificar.

Reto 03

Modificad los *prefabs* de Hellephant, ZomBear y ZomBunny para conseguir una distinción interesante desde el punto de vista del *gameplay*. También se pueden crear nuevos *prefabs* aprovechando los modelos disponibles y cambiando colores en los materiales, de manera que podamos aumentar nuestra fauna.

Actualizamos el *layout* de nivel para introducir a los nuevos tipos de enemigos de manera más progresiva.

Imagen 11.



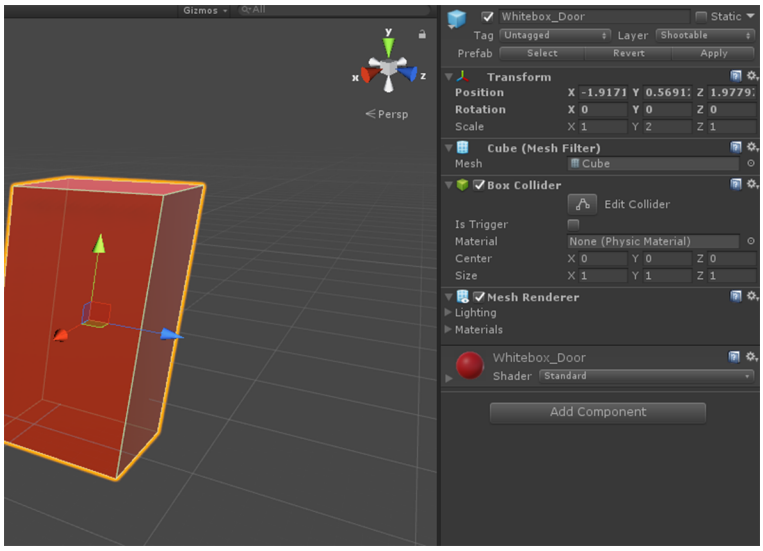
3. *Triggers* y eventos de *scripting*

3.1. Puertas interactivas

Queremos tener la opción de poder cerrar habitaciones con puertas, y que estas se abran con llaves que colocaremos por el nivel.

Las puertas serán simples cubos que se desactivarán al acercarse a ellos con una llave. Podemos crear una «puerta» cogiendo un cubo de Unity o duplicando el *prefab* de Whitebox_Wall y cambiando su aspecto. También desactivamos la casilla Static del *inspector*, ya que serán objetos que podrán activarse y desactivarse en tiempo real. Hemos llamado a este nuevo *prefab* Whitebox_Door.

Imagen 12.



Si queremos poder «abrir» puertas, deberemos utilizar *triggers*. Un *trigger* es simplemente una zona o área en el mundo que ejecutará una función de código o *scripting* al entrar o salir de ella.

Para crear nuestro primer *trigger*,generaremos un cubo de Unity. En su componente BoxCollider, activaremos la opción isTrigger. Podemos desactivar el componente de MeshRenderer para que no se pinte el cubo en el mundo, ya que este será un *trigger* invisible. Al entrar en esta área, desactivaremos el muro o puerta objetivo. Esto se consigue mediante un *script* al que llamaremos TriggerDisableTargets.

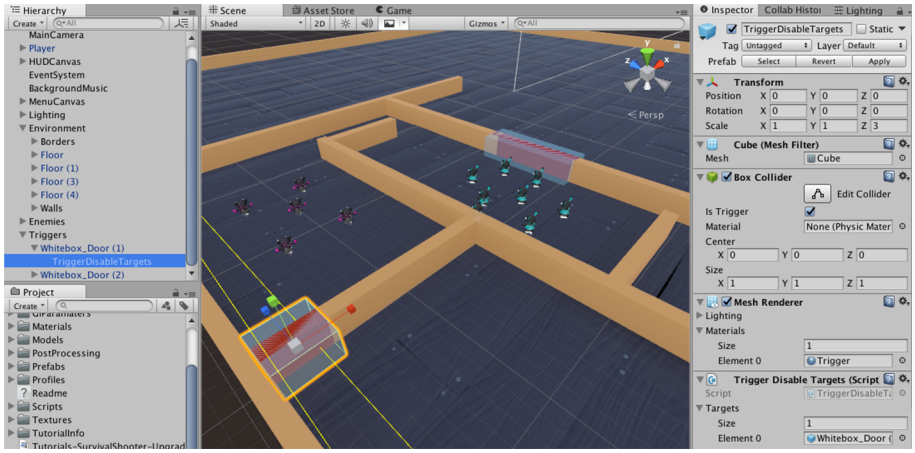
Reto 04

Cread el *script* TriggerDisableTargets. El *script* deberá consistir en una función OnTriggerEnter que, al detectar al *player*, desactive uno o varios *game objects*.

Nuestro nuevo *trigger* está terminado, de manera que le cambiamos el nombre a algo apropiado, como TriggerDisableTargets, y lo arrastramos a nuestra ventana de Project para convertirlo en *prefab*.

Mediante la colocación de este tipo de *triggers* por el nivel, podremos hacer que puertas o zonas secretas se abran al «pisar» la zona del *trigger*. Si no queremos tener que colocar un TriggerDisableTargets por cada *door* del nivel, podemos simplemente hacer que TriggerDisableTargets sea «hijo» del *prefab* Whitebox_Door y hacer un *apply* al *prefab*. De esa manera, cada Whitebox_Door instanciada tendrá un *trigger* ya asociado.

Imagen 13.



Si activamos el *play*,podremos ver cómo, al acercar a nuestro personaje a una puerta, esta desaparecerá. No queremos que las puertas se abran simplemente con acercarse a ellas, sino que el jugador necesite encontrar una llave para abrir una puerta.

3.2. Creando llaves

Antes de crear una llave, necesitaremos un sitio donde almacenar la información de cuántas llaves tiene el jugador. Crearemos un *script* PlayerInventory y lo añadiremos al *prefab* del *player*:

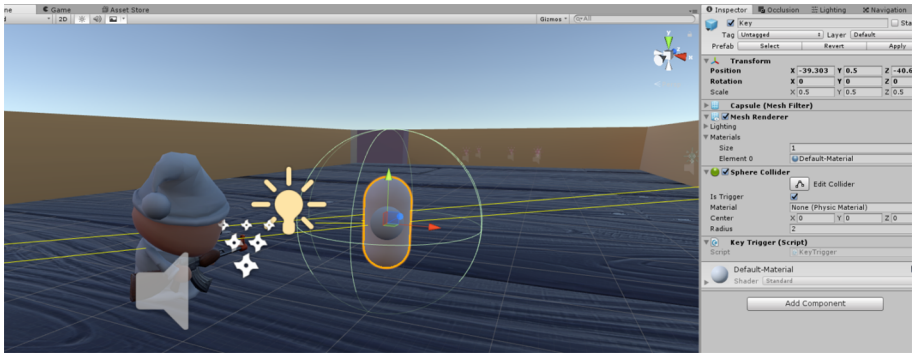
Imagen 14.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerInventory : MonoBehaviour {
    public int keysObtained = 0;

       public void AddKey()
    {
        keysObtained++;
        Debug.Log("Key added. Number of keys: " + keysObtained);
    }

    public bool UseKey()
    {
        if (keysObtained == 0) return false;
        keysObtained--;
        Debug.Log("Used key. Number of keys: " + keysObtained);
        return true;
    }

    public int GetKeyAmount()
    {
        return keysObtained;
    }
}
```

El *script* lleva la cuenta de cuántas llaves tiene el jugador en su poder.

Para crear una llave, usaremos una simple cápsula de Unity como *placeholder*, le añadiremos un *sphere collider* con el *flag* de isTrigger (eliminando el *sphere collider* original) y crearemos un *script* KeyTrigger. Este *script* será muy similar en estructura al TriggerDisableTargets.

Imagen 15.



Reto 05

Cread el *script* KeyTrigger, por el que el *player*, al tocar la llave, incrementará su inventario en +1 y la llave se autodestruirá.

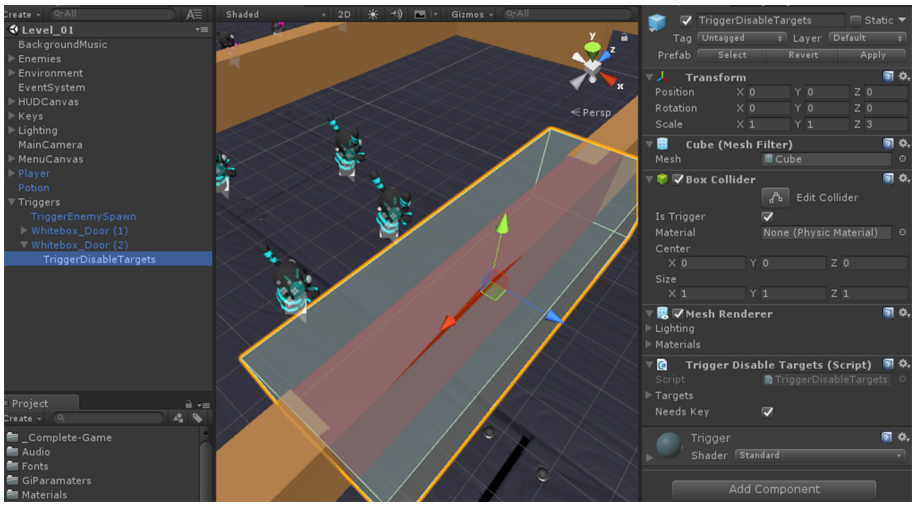
Ahora que ya tenemos llaves que el jugador puede adquirir, podemos modificar nuestras puertas (y los *triggers* que «abren» las puertas) para que exista la posibilidad de que requieran llaves.

Reto 06

Modificad TriggerDisableTargets para que su ejecución requiera que el jugador posea al menos una llave en su inventario.

A estas alturas, deberíamos poder tener la libertad de crear distintas habitaciones cerradas por puertas que requieran (o no) llave para abrirse. También podremos colocar llaves repartidas por el escenario (posiblemente en los rincones más peligrosos).

Imagen 16.



3.3. *Power-ups* de salud

La quintaesencia del *power-up* es el botiquín de salud, poción curativa o equivalente.

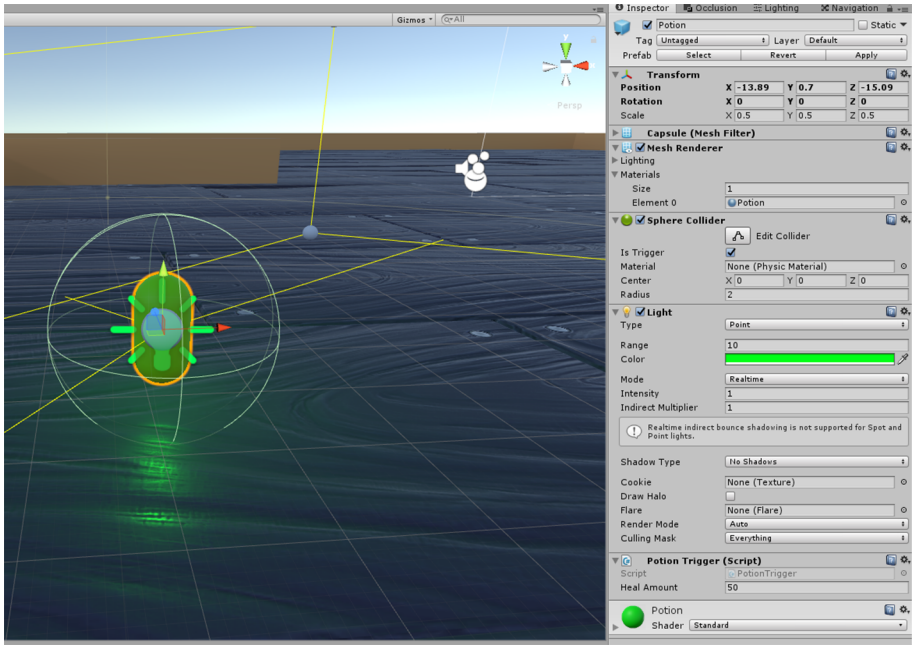
Copiamos el *prefab* de Key y le cambiamos el nombre a Potion. Cambiamos también el color o cualquier elemento visual que queramos, y sustituimos su *script* KeyTrigger por un PotionTrigger. Nuestro nuevo *script* deberá, simplemente, acceder al componente PlayerHealth del *player* y regenerar su vida.

Reto 07

Cread un *script* PotionTrigger que forme parte de un nuevo *prefab* Potion y que regenere la vida al *player* al tocar Potion.

Podéis colocar el *power-up* de salud al final de habitaciones complicadas o instanciarlo de manera aleatoria (con cierto porcentaje de probabilidad) al matar a un enemigo. Una buena idea sería modificar el *script* de EnemyHealth para instanciar pociones al morir en la función Death().

Imagen 17.



3.4. Emboscadas enemigas

Uno de los últimos elementos interactivos que necesitamos para tener nuestro clon de *Gauntlet* completo es una manera de instanciar grupos de enemigos al instante, de manera que podamos crear habitaciones trampa y situaciones de tensión.

Generaremos una nueva clase de *trigger* que instanciará grupos de enemigos del tipo del deseado en el momento en que el jugador entre en el *trigger*.

Para ello, volveremos a crear un cubo con BoxCollider e isTrigger. Añadiremos un *script* llamado TriggerEnemySpawn:

Imagen 18.

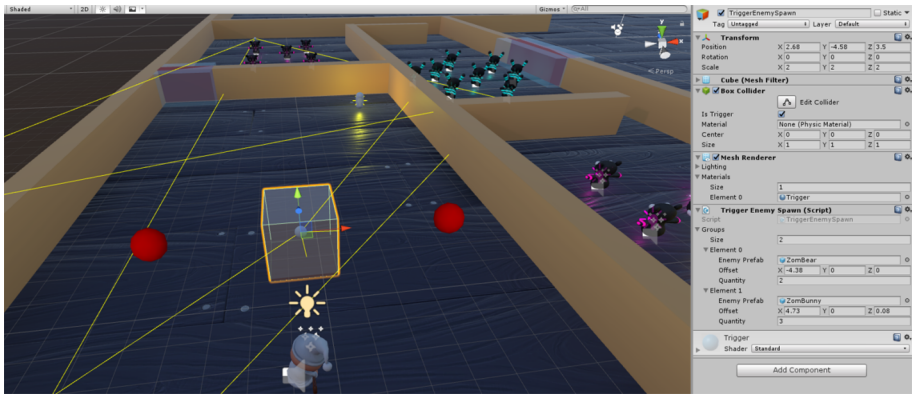
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerEnemySpawn : MonoBehaviour {

    [System.Serializable]
    public struct EnemySpawn
    {
        public GameObject enemyPrefab;
        public Vector3 offset;
        public int quantity;
    }

    public List<EnemySpawn> groups;

        void OnTriggerEnter(Collider c)
    {
        if (c.CompareTag("Player") == false) return;

        foreach (EnemySpawn spawn in groups)
        {
            for (int i = 0; i < spawn.quantity; i++)
            {
                GameObject.Instantiate(spawn.enemyPrefab, transform.position + spawn.offset,
Quaternion.identity);
            }
        }

        GameObject.Destroy(this.gameObject);
    }

    void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        foreach (EnemySpawn spawn in groups)
        {
            Gizmos.DrawSphere(transform.position + spawn.offset, 0.5f);
        }

    }
}
```

El *trigger* recorre una lista de EnemySpawn e instancia a los enemigos definidos en dichos *spawns*. Un EnemySpawn es un *struct* que hemos creado para contener la información de nuestro punto de *spawn*: tipo de enemigo, posición y cantidad de enemigos que hay que instanciar.

A fin de facilitarnos la tarea al ver dónde aparecerán los enemigos, utilizamos la función OnDrawGizmos para mostrar un *gizmo* en forma de esfera en la escena por cada punto de *spawn* creado.

La siguiente captura de pantalla muestra un ejemplo de TriggerEnemySpawn con dos grupos EnemySpawn:

Imagen 19.



Reto de solución abierta

Desarrollad el nivel creando nuevas habitaciones y pasillos. Utilizad los elementos de *gameplay* y los eventos de *trigger* que hemos explicado para crear un *gameplay* interesante. Dejad espacio para una habitación final desde la que se pueda pasar al siguiente nivel.

Acordaos de utilizar los atajos de teclado introducidos durante los primeros capítulos. Activad la visualización del *grid* (Gizmos -> Show Grid) para cuadrar mejor vuestros diseños.

3.5. Carga de niveles

Cuando contemos con más de un nivel diseñado, nos interesará poder pasar de un nivel a otro sin tener que salir del *play* de Unity 3D ni estar cambiando manualmente entre escenas.

Una carga básica de escena se realiza mediante la llamada a:

UnityEngine.SceneManagement.SceneManager.LoadScene («Nombre de la escena»)

Así pues, si queremos que el juego cargue otra escena al llegar a un punto concreto del nivel, necesitaremos crear un *trigger* que llame a esa función.

Imagen 20.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerLoadLevel : MonoBehaviour {
    public string sceneToLoad;

    void OnTriggerEnter(Collider c)
    {
        if (c.CompareTag("Player") == false) return;

        UnityEngine.SceneManagement.SceneManager.LoadScene (sceneToLoad);
    }
}
```

Para poder cargar una escena, esta debe estar incluida en los Build Settings (File -> Build Settings).

La función LoadScene acepta un segundo parámetro para cargar aditivamente escenas. «Cargar aditivamente» significa que en vez de destruir la escena actual y pasar a la escena cargada, se cargará la nueva escena de manera que todo su contenido (*game objects*) se añadirá a la escena actual. Esto es extremadamente útil en juegos con elementos *open world*, donde no queremos que existan pantallas de carga, o donde el mundo se va cargando de manera modular y queremos ir cargándolo «por partes».

Reto de solución abierta

Cread un segundo nivel únicamente con elementos de escenario y enemigos, y cargadlo de manera aditiva de tal modo que el jugador pueda pasar de un nivel a otro sin darse cuenta de que se ha «cargado» un segundo nivel.

Por último, para evitar que el juego se «cuelgue» durante unos instantes o varios segundos al cargar un nuevo nivel, existe la carga asíncrona de niveles. Mediante la función SceneManager.LoadSceneAsync, podemos cargar una escena en segundo plano, y activarla solo cuando esta esté disponible. En los últimos años, la mayoría de juegos utilizan carga asíncrona y aditiva de niveles a la vez para conseguir simular un mundo abierto y sin pantallas de carga.

4. Conclusiones y notas sobre el Mundo Real™

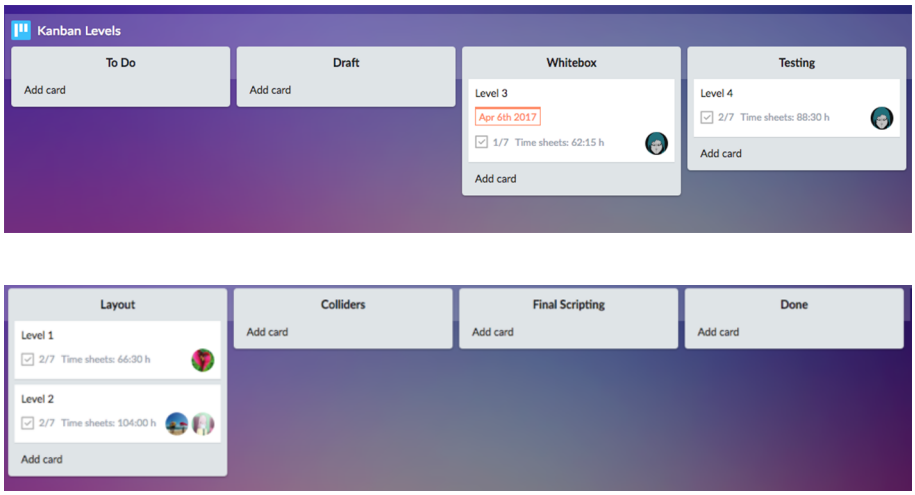
Durante el transcurso de este unidad, hemos cogido un prototipo básico y lo hemos convertido en un juego «real» con niveles diseñados desde cero y eventos de *scripting*. El hecho de ir añadiendo mecánicas y eventos a medida que avanzábamos en el diseño de nivel nos ha obligado a jugar y a rejugar en el nivel y a iterarlo, habilidades esenciales para un buen diseñador de niveles.

El proceso de diseño de niveles depende de manera extrema del tipo de juego: un juego de plataformas requerirá un exhaustivo detalle en las distancias de salto; un juego *shooter* necesitará una buena colocación de coberturas, medidas de cada elemento del escenario y situación de los enemigos; un juego de horror se basará en el *scripting* y en el juego con los ángulos de cámara para conseguir crear momentos de tensión y «sustos» en los puntos adecuados. De todos modos, lo aprendido en esta unidad servirá como una buena introducción para empezar en el mundo del diseño de niveles.

Dicho esto, en el mundo real cada equipo tiene su propia metodología para diseñar niveles, adaptada al equipo de trabajo y al tipo de juego.

Por poner un ejemplo de caso de uso real, en el estudio donde trabaja el autor de este material, tienen el siguiente *pipeline* para el proceso de creación de niveles:

Imagen 21.



La creación de un nivel comienza con un *draft* o borrador, un diseño en papel, documento escrito o *whitebox* de muy alto nivel que define cómo será el nivel y qué lo caracterizará a grandes rasgos.

Cuando el diseño se aprueba, se comienza a trabajar en una *whitebox* mucho más detallada y que pueda aproximarse lo máximo posible al nivel final, por mucho que no haya arte final todavía. A partir de ese momento, el nivel siempre deberá ser jugable de inicio a fin, con su eventos de nivel, enemigos y demás elementos para que pueda hacerse *playtesting* del mismo del principio al final.

Tanto el diseñador como otros miembros del equipo empiezan a testear el nivel para comprobar si es divertido y si se han conseguido los objetivos planteados. Si no es así, el nivel vuelve a la fase de *whitebox*, o incluso a la de *draft* si es necesario. Si pasa el testeo, los artistas 3D pueden empezar a crear el arte 3D final (proceso de *layout*) y colocar los *colliders* finales, y luego los programadores, *level designers* o *game designers* harán las pasadas de *scripting* y pulido general que sean necesarias antes de dar por cerrado un nivel.

Es importante asumir que cualquier problema en el diseño de nivel se ha de «cazar» lo antes posible en el *pipeline*. De lo contrario, uno se arriesga a perder semanas de trabajo de otros miembros del equipo, si tiene que echar para atrás el vestido (*layout*), el *scripting* u otros elementos de la creación del nivel. Un *playtesting* exhaustivo desde el minuto cero solventa en la mayoría de casos este problema.

5. Soluciones a los retos

5.1. Reto 01

Imagen 23.

```
using UnityEngine;
using System.Collections;

namespace CompleteProject
{
    public class CameraFollow : MonoBehaviour
    {
        public Transform target;  // The position that that camera will be following.
        public float smoothing = 5f;  // The speed with which the camera will be following.
        public float smoothingRotation = 5f;   // The speed with which the camera will be
rotating to aim at the player.

        public Vector3 offset;  // The initial offset from the target.

        void Start ()
        {
            // Calculate the initial offset.
            transform.position = target.position + offset;
        }

        void FixedUpdate ()
        {
            // Create a postion the camera is aiming for based on the offset from the target.
            Vector3 targetCamPos = target.position + offset;

            // Smoothly interpolate between the camera's current position and it's target
position.
            transform.position = Vector3.Lerp (transform.position, targetCamPos, smoothing *
Time.deltaTime);

            // Smoothly interpolate rotation to look at the player position
            Vector3 targetPosition = target.position - transform.position;
            Quaternion newRotation = Quaternion.LookRotation(targetPosition); // Create a new
rotation that aims at the player position
            transform.rotation     =     Quaternion.Lerp(transform.rotation,     newRotation,
smoothingRotation * Time.deltaTime);
        }
    }
}
```

5.2. Reto 02

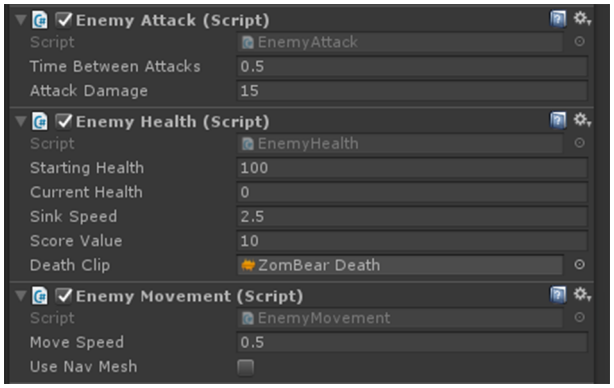
Imagen 24.

```
using UnityEngine;
using System.Collections;

namespace CompleteProject
{
    public class EnemyMovement : MonoBehaviour
    {
        public float moveSpeed = 0.5f;
        public bool useNavMesh = false;

        Transform player;               // Reference to the player's position.
        PlayerHealth playerHealth;      // Reference to the player's health.
        EnemyHealth enemyHealth;        // Reference to this enemy's health.
        UnityEngine.AI.NavMeshAgent nav;                    // Reference to the nav mesh
agent.

        void Awake ()
        {
            // Set up the references.
            player = GameObject.FindGameObjectWithTag ("Player").transform;
            playerHealth = player.GetComponent <PlayerHealth> ();
            enemyHealth = GetComponent <EnemyHealth> ();
            nav = GetComponent<UnityEngine.AI.NavMeshAgent>();
        }

        void Update ()
        {
            // If the enemy and the player have health left...
            if(enemyHealth.currentHealth > 0 && playerHealth.currentHealth > 0)
            {
                if (useNavMesh)
                {
                    // ... set the destination of the nav mesh agent to the player.
                    nav.SetDestination(player.position);
                }
                else
                {
                    // Not using navmesh, just go towards the player if he's close
                    float     distanceToPlayer     =     Vector3.Distance(player.position,
transform.position);
                    if (distanceToPlayer < 10.0f)
                    {
                        Vector3 dirToPlayer = player.position - transform.position;
                        transform.position += dirToPlayer * Time.deltaTime * moveSpeed;
                        transform.LookAt(player);
                    }
                }
            }
            // Otherwise...
            else
            {
                if (useNavMesh)
                {
                    // ... disable the nav mesh agent.
                    nav.enabled = false;
                }
            }
        }
    }
}
```

5.3. Reto 03

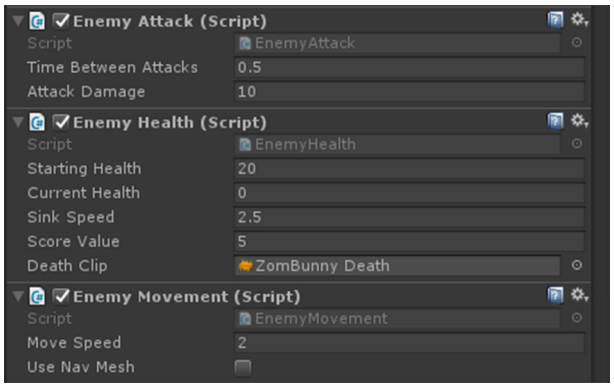
ZomBear

Imagen 25.



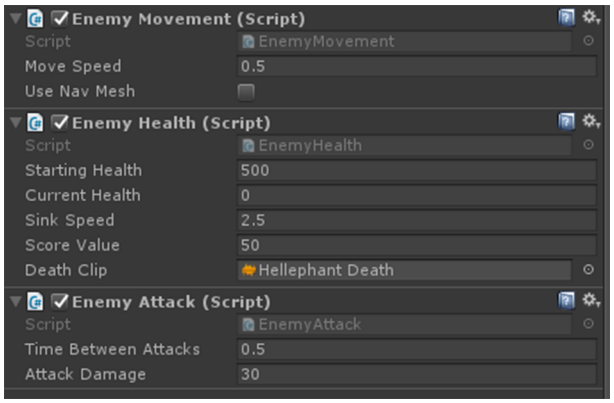
ZomBunny

Imagen 26.



Hellephant

Imagen 27.



5.4. Reto 04

Imagen 28.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerDisableTargets : MonoBehaviour {

    public GameObject[] targets;

    void OnTriggerEnter(Collider c)
    {
        // Ignorar todo lo que no sea el jugador
        if (c.CompareTag ("Player") == false)
            return;

        foreach (GameObject g in targets) {
            g.SetActive (false);
        }
    }
}
```

A la función OnTriggerEnter se la llama cada vez que un objeto con *rigid body* o *character controller* entra dentro del volumen del *trigger*. En nuestro *script*, comprobamos que el objeto que ha entrado es el jugador, y luego desactivamos todos los *game objects* objetivo.

5.5. Reto 05

Imagen 29.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class KeyTrigger : MonoBehaviour {

        void OnTriggerEnter(Collider c)
    {
         if (c.CompareTag("Player") == false) return;

         PlayerInventory inventory = c.GetComponent<PlayerInventory>();
         if (inventory == null) return;

         inventory.AddKey();
         Destroy(this.gameObject);
    }
}
```

El *script* es un simple OnTriggerEnter que incrementa el contador de llaves del jugador y destruye el objeto.

5.6. Reto 06

Imagen 30.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerDisableTargets : MonoBehaviour {

        public GameObject[] targets;
    public bool needsKey = false;

        void OnTriggerEnter(Collider c)
        {
                // Ignorar todo lo que no sea el jugador
                if (c.CompareTag ("Player") == false)
                        return;

         PlayerInventory inventory = c.GetComponent<PlayerInventory>();
         if (inventory == null) return;

         if (needsKey && inventory.UseKey() == false) return; // use key returns false if
I don't have keys

                foreach (GameObject g in targets) {
                        g.SetActive (false);
                }
        }
}
```

El código es bastante autoexplicativo. Si la opción de needsKey está en *true*, comprobaremos que el jugador tenga alguna llave en su poder para destruir la puerta.

5.7. Reto 07

Imagen 31.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PotionTrigger : MonoBehaviour {
    public int healAmount = 50;

       void OnTriggerEnter(Collider c)
    {
        if (c.CompareTag("Player") == false) return;

        PlayerHealth health = c.GetComponent<PlayerHealth>();

        int hp = health.currentHealth;
        hp += healAmount;
        if (hp > 100) hp = 100;

        health.currentHealth = hp;

        GameObject.Destroy(this.gameObject);
    }
}
```