Un juego de carreras

PID_00244517

Rafael González Fernández
Pierre Bourdin Kreitz

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Índice

Introducción

En este módulo dejaremos atrás el 2D y empezaremos a diseñar juegos puramente 3D.

La estructura del módulo seguirá la línea de los anteriores y se dividirá en dos partes, una de carácter más teórico y otra donde practicar lo aprendido. En la parte teórica veremos la potente herramienta «Terrain» de Unity, que nos ayudará a simplificar la creación de escenarios y a hacer nuestros juegos mucho más profesionales. Con esta herramienta podemos crear superficies montañosas, árboles, zonas de césped o rocas, y hacer incluso que estos se muevan acompasados con el viento.

Al tratarse de un juego totalmente 3D, profundizaremos en conceptos ya conocidos como las cámaras o los componentes relacionados con las físicas, como los «Colliders» o los «Triggers», pero siempre desde una perspectiva 3D.

En la parte práctica, haremos un juego de carreras sencillo en el que podremos correr por un circuito con físicas realistas y competir contra otros coches. Y gracias a que trabajaremos con varios *asset packages* predefinidos de Unity, podremos tener funcionalidades avanzadas ya implementadas que ahorran bastante trabajo a aquellos que aún están iniciándose en Unity y en la programación de videojuegos en general.

Por último, también aprenderemos a personalizar nuestros propios *scripts*, con lo que adaptaremos el entorno de Unity a nuestras necesidades con la ayuda de *gizmos* o creando nuestro propio Inspector personalizado en algunos *scripts*. Esta última parte es realmente útil, ya que la creación de herramientas personalizadas es algo muy habitual dentro del flujo de trabajo en el desarrollo profesional.

Objetivos

En este módulo didáctico presentamos los conocimientos necesarios para alcanzar los siguientes objetivos:

1. Entender las herramientas básicas para trabajar en juegos 3D.

2. Poder utilizar las físicas en entornos plenamente 3D.

3. Comprender cómo funciona una cámara 3D y sus tipos.

4. Dominar la herramienta «Terrain».

5. Crear un juego plenamente 3D aplicando los conocimientos previos ya adquiridos en anteriores módulos.

1. «Terrain»

El sistema «Terrain» de Unity es una herramienta muy potente que nos permite crear vastos bosques, desiertos o montañas de una manera fácil y rápida. Estos están optimizados internamente para su renderización, lo que hace que usar «Terrain» sea una opción mejor que la creación de *assets* propios para elaborar paisajes.

Ejemplo de terreno creado con Unity

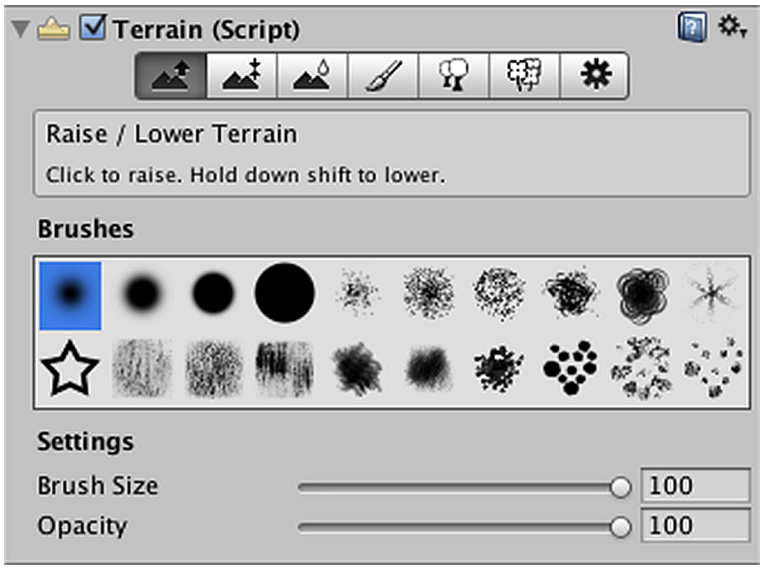
Para crear un terreno hay que ir a «GameObject > 3D Object > Terrain». Esto generará un plano en la escena completamente blanco, que nos servirá para empezar a darle forma.

1.1. El editor de terreno

El editor de terreno es la herramienta de Unity que nos permitirá editar las propiedades de nuestros terrenos. Podremos acceder a ella desde el Inspector una vez seleccionado el terreno.

Esta herramienta consta de un editor personalizado en cuya parte superior podremos elegir entre varias opciones, casi todas ellas pinceles (*brushes*) que nos servirán para modelar y pintar nuestros terrenos de una manera muy similar a como se trabajaría con un editor de imágenes como Photoshop.

Una vez seleccionada cualquiera de estas opciones principales, en la parte inferior aparecerán las características y propiedades de cada una de ellas.



Excepto en el caso de la opción de configuración, todas las demás opciones tienen en común las propiedades del «Brush Size» y de la «Opacity» de este.

Las tres primeras opciones sirven para moldear la orografía del terreno, mientras que las tres siguientes sirven para darle detalles mediante la textura y la generación de césped, rocas y árboles.

Nota

Al contrario de como se suele trabajar en Unity, si tocamos la tecla «F» mientras tenemos seleccionado un terreno, el foco no se centrará en el terreno completo, sino que la vista se centrará en la parte del terreno donde tengamos colocado el ratón.

1.2. Subir / bajar terreno

La primera opción que nos aparece sirve para «Raise / Lower Terrain». Con este pincel, mientras mantengamos apretado el ratón y lo desplacemos por la superficie del terreno, este empezará a crecer en altura con la forma del pincel que tengamos seleccionado en la sección «Brushes».

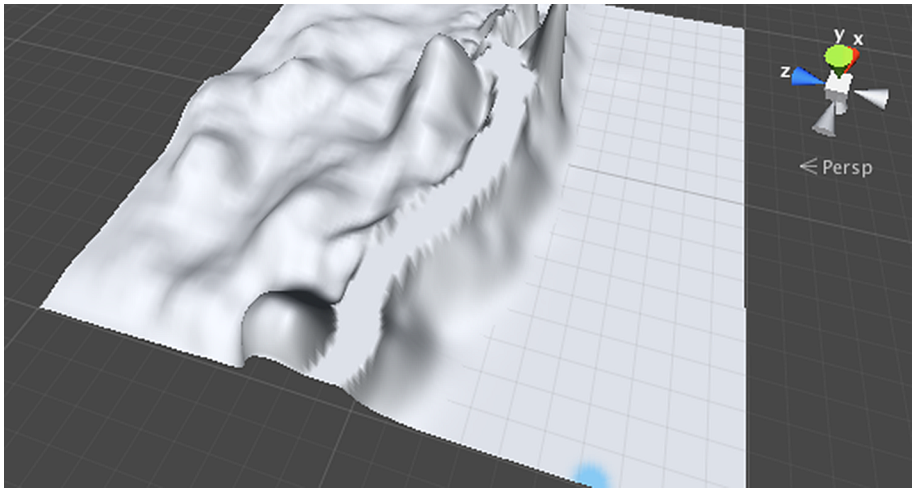
Si por el contrario quisiéramos reducir la altura, deberíamos hacer lo mismo pero manteniendo apretada la tecla «Shift».

El tamaño del pincel define la cantidad de superficie de terreno que se verá afectada por este, y su opacidad marcará «la caída» de esta intensidad. Alguna de las imágenes de los pinceles tiene un degradado. A menor opacidad, más suave será el peso de ese degradado al afectar al terreno.

1.3. Pintar alturas

La siguiente opción es «Paint Height», con la que podremos definir una altura concreta en el lugar donde pintemos.

Dentro de sus propiedades hay una opción extra llamada «Height», que nos marcará la altura objetivo. Si clicamos en un punto del terreno con una altura inferior a esta, el terreno subirá. Y si, por el contrario, el punto del terreno donde clicamos tiene una altura superior a esta, bajará.



Un terreno que se está aplanando

Esta vez la tecla «Shift» servirá para setear esa altura de manera automática. Allí donde cliquemos con la tecla «Shift», la altura que tenga ese punto es la que se asignará automáticamente a la variable altura de la herramienta.

Nota

El botón «Flatten» coloca todo el terreno por completo a la altura seleccionada.

1.4. Suavizar altura

La tercera opción es «Smooth Height». Nos servirá para reducir la diferencia de altura entre varias zonas, haciendo un promedio de estas y creando un terreno menos escarpado. Algo parecido a lo que haría la herramienta «Blur» de un editor de imágenes.

1.5. Pintar texturas

La cuarta opción es el «Paint Texture». Con ella podremos pintar la superficie del terreno para dar la sensación de que estamos encima de piedra, hielo, césped, barro, etc.

Inicialmente el terreno viene en blanco sin ninguna textura definida, por lo que tendremos que ir añadiendo todas aquellas con las que queramos pintar nuestro terreno.

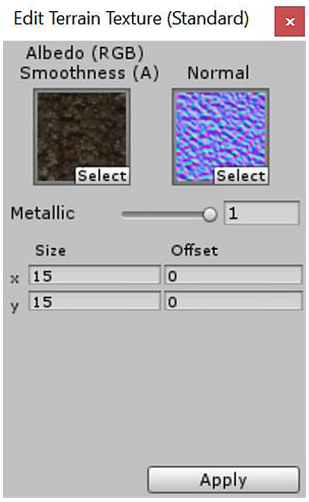
Nota

Unity viene con varias texturas y modelos de árboles y césped listos para utilizar en nuestros prototipos. Podemos instalarlos al crear el proyecto o hacerlo más tarde accediendo a «Assets > Import Package > Terrain Assets».

Para añadir las texturas clicaremos en el botón «Edit Textures > Add Texture». La ventana de añadir textura de terreno nos permite añadir una textura de difuso «Albedo» y su «Normal» para esa superficie.

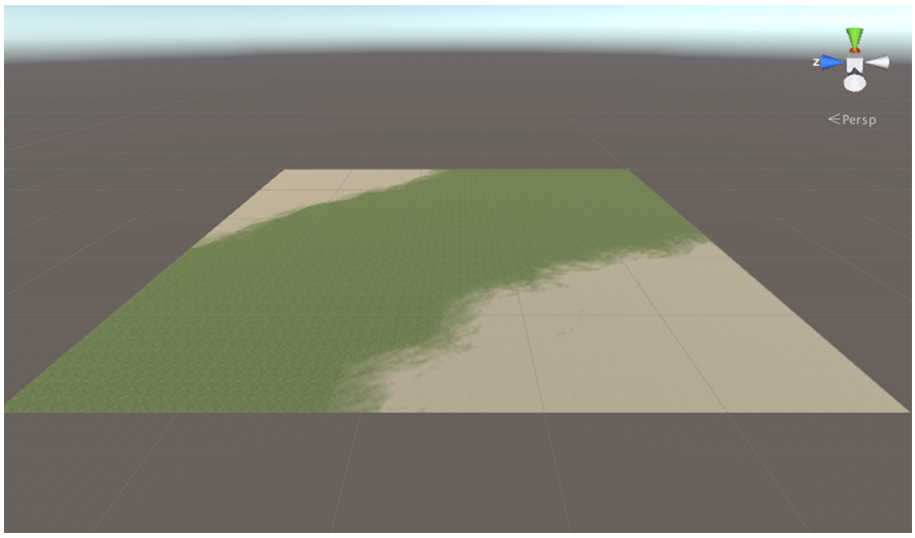
Podemos añadir varias texturas a la lista, pero la primera que tengamos en nuestra lista será la que se aplicará como base para todo el terreno.

Cada vez que queramos pintar con alguna textura en concreto, solo habrá que seleccionarla y decidir el tipo de pincel que queremos y su opacidad.



Ventana de añadir textura

Aquí nos aparecerá una propiedad nueva, «Target Strength», que marcará la intensidad que tendrá la textura al pintarse.



Pintando césped en el terreno

1.6. Árboles

La quinta opción es «Place Trees», con la que podremos poblar nuestro terreno con todo tipo de vegetación de una manera orgánica. Tan solo tendremos que añadir los modelos 3D de árboles a la lista de árboles que aparece, de una forma muy similar a como se hace con las texturas a la hora de pintar.

Nota

Si queréis crear un árbol con la herramienta de Unity, recordad que siempre podéis hacerlo accediendo a «GameObject > 3D Object > Tree».

Podéis encontrar más información sobre su uso en: https://docs.unity3d.com/Manual/tree-FirstTree.html.

Si el modelo de árbol que asignemos ha sido creado con el creador de árboles de Unity, la ventana de selección de *mesh* mostrará una propiedad llamada «Bend Factor», que definirá cuán afectado se verá este por la fuerza del viento.

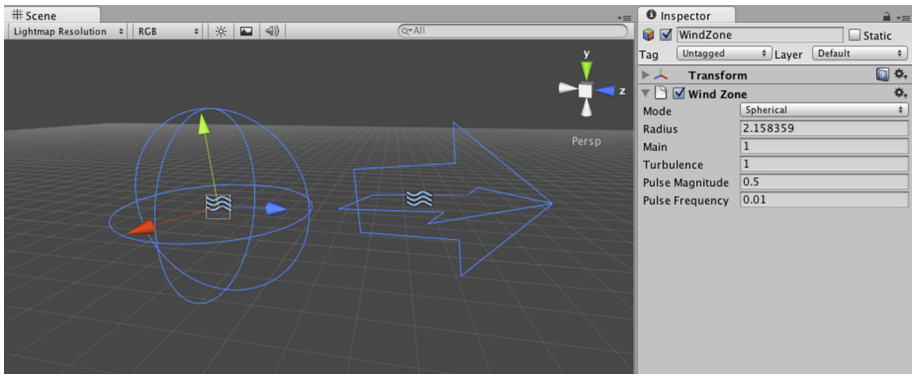
El funcionamiento de esta herramienta es prácticamente calcado al de las anteriores, pero en este caso dispondremos de muchas más opciones, como la densidad de árboles que queremos que se pinten dentro la zona de pintado, si queremos que tengan diferentes alturas/rotaciones, si queremos que estas sean *random* o que mantengan la ratio de aspecto original del árbol.

Presionando la tecla «Shift» o «Control» mientras hacemos clic borraremos los árboles existentes dentro de la zona de pintado que sean del tipo seleccionado en la herramienta.

Por otro lado, el botón «Mass place Trees» sirve para poblar de manera masiva todo el terreno con una mezcla aleatoria de árboles de los tipos que tengamos cargados.

1.7. Zonas de viento

Los árboles generados proceduralmente con Unity pueden verse afectados por el viento. Para ello, necesitamos incluir en nuestra escena el elemento «WindZone» accediendo a «Game Object > 3D Object > Wind Zone».



Propiedades de las zonas de viento

Su opción principal es el modo que define si este será «Directional» (el viento afectará a toda la escena) o «Spherical» (el viento solo afectará al interior de la esfera marcada por la variable «Radius»).

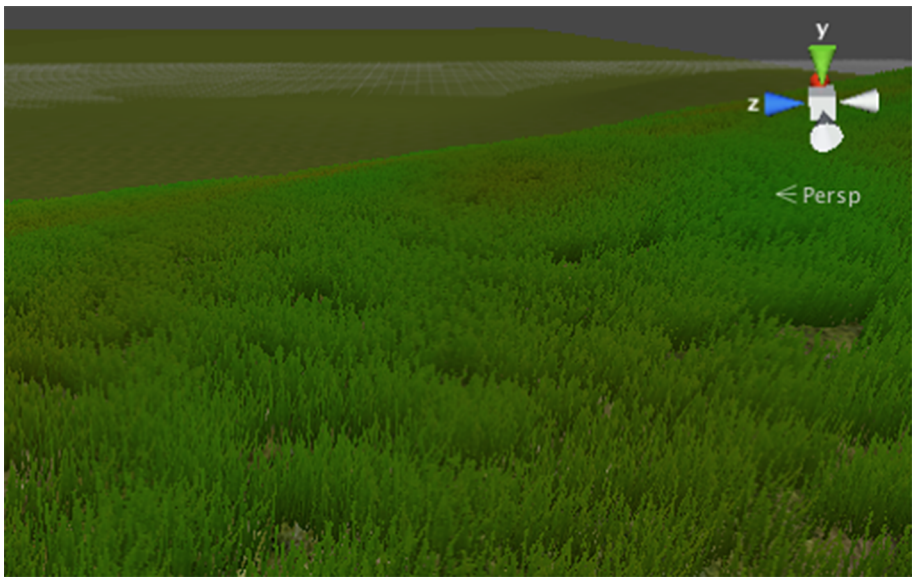
Nota

Podéis encontrar más información sobre cómo sacarle el máximo partido a las zonas de viento en: https://docs.unity3d.com/Manual/class-WindZone.html.

La propiedad «Main» indica la fuerza constante del viento, pero para darle un comportamiento más realista existen propiedades como «Turbulence», que modifica rápidamente la velocidad del viento, y «Pulse», con la que podremos definir la frecuencia de estos cambios y su intensidad.

1.8. Pintar detalles

La sexta opción es «Paint Details», con la que podremos generar césped, flores, rocas o elementos decorativos en general dentro del terreno.



Vista de las hierbas en el terreno

En esta categoría encontraremos dos opciones: «Add Grass Texture» y «Add Detail Mesh». El césped puede ser o texturas aplicadas a *billboards* que se mueven con el viento o texturas fijas con una orientación concreta, por lo que solo necesitamos una textura y no un modelo 3D. De esta manera, podemos utilizar una imagen con alpha de unas flores o incluso de un alambre de púas utilizando el mismo sistema.

Nota

Es un *quad* pintado con una textura que siempre se encuentra orientada hacia la cámara. Esta textura modifica su orientación según la posición de la cámara en tiempo real.

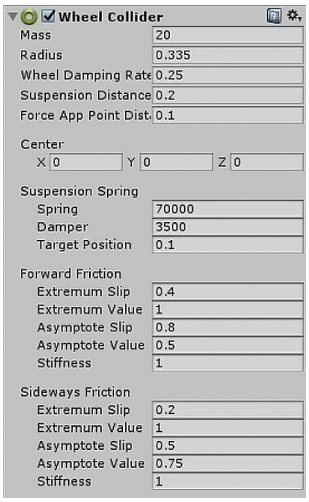
Como el césped se generará de manera aleatoria, definiremos unos valores mínimos y máximos, tanto para su altura como para su anchura, por cada tipo de «textura de césped» que tengamos.

También podremos definir un tintado de color sobre esa textura para darle un aspecto más «saludable» o más «seco» según el «ruido» que se genere. Podemos crear así zonas de césped más tupido junto a otras menos pobladas o secas.

En el caso de tener *meshes* de detalle, el sistema funciona exactamente igual pero seleccionando *meshes* que podrían ser, por ejemplo, rocas.

1.9. Ajustes del terreno

Y por último tenemos el botón de «Settings», con el que se configuran todos los valores genéricos del terreno seleccionado.



Las opciones más destacadas serían:

- «Draw»: Activa o desactiva el pintado del terreno.
- «Cast Shadows»: Activa o desactiva que el terreno proyecte sombras.
- «Tree / Detail Distance»: La distancia respecto a la cámara a partir de la cual los elementos de detalle no se renderizarán.
- «Tree / Tree Distance»: Lo mismo que la anterior, pero para los árboles.
- «Tree / Billboard Start»: A partir de qué distancia las *meshes* de los árboles se substituirán por *billboards* de estos.
- «Tree / Max Mesh Trees»: El número máximo de árboles representados por *meshes* 3D en pantalla antes de ser sustituidos por *billboards*.
- «Wind / Speed»: La velocidad del viento que moverá el césped.
- «Wind / Size»: El tamaño de las «oleadas» de viento del césped.
- «Wind / Bending»: La cantidad de inclinación que se aplicará al césped al moverse.

- «Resolution / Width-Height-Length»: Tamaño del objeto Terreno en cada uno de sus ejes.

Los botones «Import Raw» y «Export Raw» permiten exportar el mapa de alturas generado por el terreno como una imagen. Esto permite generar o modificar terrenos fuera de Unity y luego reeimportarlos.

2. Vehículos

Para adentrarnos un poco más dentro del mundo de las físicas de Unity y conocerlas un poco mejor, esta vez en su vertiente 3D, trabajaremos con los vehículos que vienen incluidos dentro de sus *assets packages*.

Nota

Internamente, Unity utiliza actualmente el SDK de PhysX3.3 de Nvidia como motor de físicas 3D.

Igual que con los *assets* del terreno, podremos instalarlos tanto al crear el proyecto, como hacerlo más tarde accediendo a «Assets > Import Package > Vehicles».

Antes de diseccionar cómo está formado internamente nuestro vehículo, insertemos uno en la escena. Podéis encontrar su *prefab* en «Assets > Standard Assets > Vehicles > Car > Prefabs > Car.prefab».

Los objetos con físicas dinámicas necesitan un suelo sobre el que sostenerse, ya que de lo contrario estarían cayendo eternamente, por lo que, en la misma escena que nuestro coche, deberíamos tener un *terrain* o un cubo o plano lo suficientemente grande y con «Colliders» para que nos sirva como suelo para nuestro vehículo.

El *prefab* ya está preparado para que podamos conducirlo con solo añadirlo a la escena y darle al «Play». Eso sí, viene sin cámaras, unos elementos que le añadiremos más adelante, pero ya podemos ver su movimiento si colocamos una cámara cenital a nuestro terreno o si directamente seguimos el coche en la vista de escena.

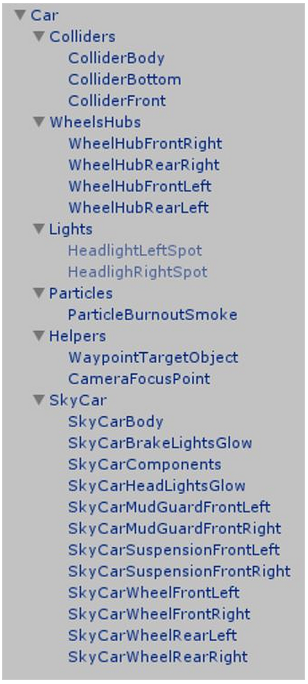
2.1. Físicas del vehículo

Las físicas en tres dimensiones son muy similares a las de dos dimensiones. Cuando uno ya está familiarizado con los componentes que nos ofrecía Unity para la gestión de físicas en 2D, el paso a las de 3D es muy sencillo.

Al pasar a 3D ampliamos la complejidad de las operaciones matemáticas que han de ejecutarse en el motor de físicas y añadimos algunos componentes que antes no existían, como el «Cloth» o el «Character Controller».

Sin más dilación, veamos cómo funciona internamente el vehículo.

Primero tenemos al padre de toda la estructura, llamado «Car», que contiene los *scripts* que gestionan el movimiento realista del vehículo, así como la rotación de las ruedas, sus efectos de sonido o el *input* del *player*.



Detalle de la estructura interna completa del coche

Tiene, además, un «Rigidbody» asociado. Un «Rigidbody» 3D funciona igual que uno 2D. Sirve para dotar al vehículo de una presencia física dentro del motor de físicas.

La parte decorativa del objeto la encontramos en el hijo «SkyCar», que contiene toda la geometría del coche dividida en partes para poder utilizarlas y modificarlas según nos interese.

Los tres hijos («Lights», «Particles» y «Helpers») están ahí para dotar de realismo al coche, como veremos más adelante, pero no afectan en nada a su comportamiento físico.

Las partes clave del coche son sus «Colliders» y las ruedas.

El primer hijo en la estructura es donde están contenidos sus «Colliders», que servirán para detectar las colisiones con el escenario o con otros elementos como coches u objetos dinámicos colocados por la escena.

Y por otro lado tenemos las ruedas. El objeto «WheelsHubs» contiene cuatro *GameObjects*, cada uno de ellos destinado a simular el comportamiento de cada una de las ruedas del coche.

Nota

Si con el juego ejecutándose seleccionáis una de las ruedas, veréis que su pivote no está en el mismo lugar que el centro de la rueda. Eso se debe a que hay una «Joint» desde el pivote hasta la rueda que hace de suspensión.

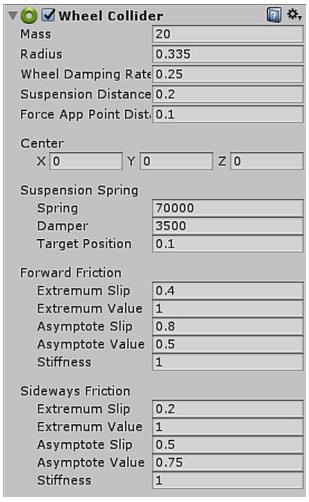
2.2. «WheelCollider»

El «WheelCollider» es un tipo de «Collider» especial creado únicamente para ser utilizado como rueda y que contiene unas propiedades específicas para simular el comportamiento de un neumático real.

Sus cálculos de fricciones son ejecutados aparte de los cálculos generales del motor de físicas, por lo que al colisionar con otro objeto no interactuará con ningún «Physic Material». Estos cálculos, aparte del motor, proporcionan un comportamiento más realista, ya que permiten simular la manera en la que la goma del neumático absorbe gran parte de las fuerzas.

Las propiedades más importantes del «WheelCollider» son estas:

- «Mass»: La masa de la rueda.
- «Radius: El radio de la rueda.
- «Wheel Damping Rate»: Es la ratio de la caída de amortiguación que habrá entre una oscilación del amortiguador y la siguiente.
- «Suspension Distance»: La distancia que habrá desde el pivote del *GameObject* que contiene este «Collider» hasta el centro de la rueda, que simulará ser la suspensión de esta.
- «Force App Point Distance»: La altura respecto al suelo en la que se aplicará la fuerza de giro a la rueda.



Inspector del «WheelCollider»

Los demás valores sirven para configurar la suspensión y los valores de fricción en los dos ejes de la rueda.

Por desgracia, este componente, por muy completo que sea, por sí solo no hará nada. Necesitamos añadir otro *script* que se encargue de modificar sus valores y hacer funcionar la rueda. Lo habitual es aplicarle un giro y una fuerza a la rueda, esta última, según la colisión con el suelo que tenga y sus características, aplicará una fuerza resultante al «Rigidbody» del que forma parte, con lo que desplazará el coche.

Una vez aplicada, miramos cuánto ha girado este «Collider» y aplicamos ese giro a la *mesh* visual que la representa.

```
1. public void FixedUpdate() {
2.
3.   // Recogemos del Input los valores de aceleración,
4.   //giro y freno.
5.   float motor = maxMotorTorque * Input.GetAxis("Vertical");
6.   float steering = maxSteeringAngle * Input.GetAxis("Horizontal");
7.   float brake = maxBrakeTorque * Input.GetAxis("Jump");
8.
9.    // Aplicamos a cada rueda estos valores
10.    leftWheel.steerAngle = steering;
11.    leftWheel.motorTorque = motor;
12.    leftWheel.breakTorque = brake;
13.
14.    rightWheel.steerAngle = steering;
15.    rightWheel.motorTorque = motor;
16.    rightWheel.breakTorque = brake;
17.
18.    Vector3 position;
19.    Quaternion rotation;
20.
21.    // Asignamos a la mesh de la rueda la posición
22.    // y el giro del collider
23.    // Estos valores de posición y rotación pertenecen
24.    // a los cálculos hechos en el ciclo anterior, por lo que
25.    // siempre irán una iteración por detrás.
26.    leftWheel.GetWorldPose(out position, out rotation);
27.    visualLeftWheel.transform.position = position;
28.    visualLeftWheel.transform.rotation = rotation;
29.
30.    rightWheel.GetWorldPose(out position, out rotation);
31.    visualRightWheel.transform.position = position;
32.    visualRightWheel.transform.rotation = rotation;
33.    }
```

Las variables y funciones clave de este componente son las siguientes:

- «steerAngle»: Define el ángulo de giro que hay que aplicar a la rueda.
- «motorTorque»: Es la cantidad de fuerza aplicada en ese instante de tiempo.
- «brakeTorque»: Lo mismo, pero para la fuerza de freno.
- «GetWorldPose()»: Devuelve por parámetros la posición y rotación de este «Collider» respecto al mundo.

Nota

Unity recomienda que para frenar no usemos una fuerza de motor negativa, sino que apliquemos correctamente su «brakeTorque».

3. Cámaras 3D

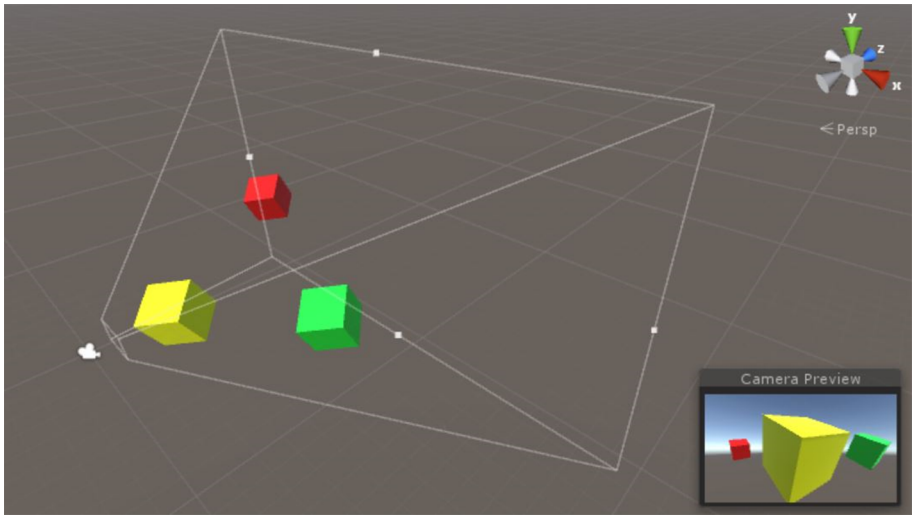
En un juego 3D las cámaras suelen tener mucho más protagonismo que en uno 2D. Ya no solo por la complejidad de los movimientos y acciones que hagan, sino también por lo que respecta a su configuración.

En este apartado profundizaremos un poco más en los detalles que las componen y veremos los dos tipos de cámara que podremos utilizar en un juego 3D.

3.1. Cámara en perspectiva

La cámara en perspectiva es la que se suele utilizar en la inmensa mayoría de los juegos 3D. Esta cámara simula nuestro campo de visión de una manera realista, en la que los objetos lejanos se ven más pequeños que los que están más cerca de la cámara.

El *frustum* es el volumen de la visión de la cámara. En la imagen es toda la zona interior delimitada por las líneas blancas. Todo aquel objeto que esté dentro del *frustum* será pintado. Si un objeto queda fuera, no se pintará. Si un objeto queda cortado mitad dentro mitad fuera, cuando se dibuje por pantalla se verá tan solo la parte que quede dentro.



El *frustum* de esta cámara tiene forma de una pirámide con la punta cortada.

Los valores que configuran el *frustum* vienen dados por:

- «Clipping Planes (Near/Far)»: Son los dos planos perpendiculares a la orientación de la cámara. Marcan dónde va a empezar a pintar y dónde va a terminar.

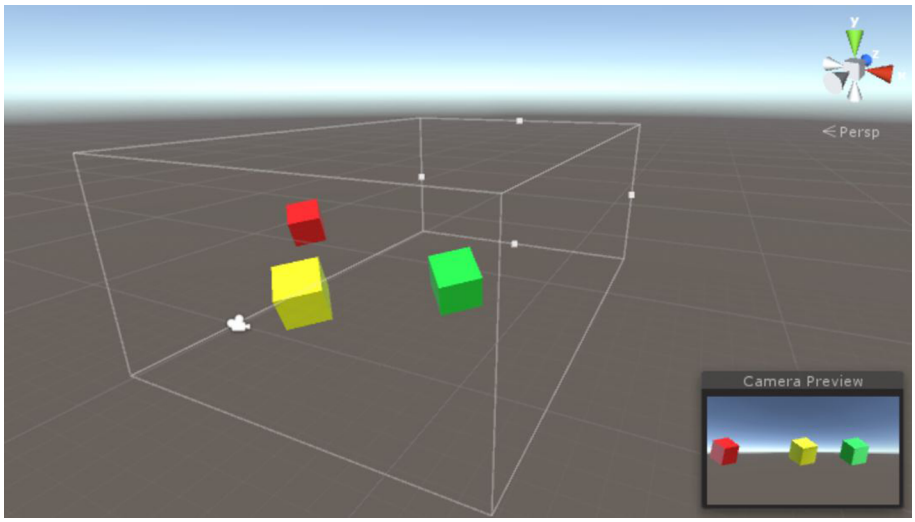
- «Field of View»: Este valor indica el ángulo de visión horizontal de esta cámara. Si es pequeño, simulará que miramos a través de una puerta. Si lo hacemos más grande, simulará un gran angular.

3.2. Cámara ortográfica

Este tipo de cámara es el que se suele usar en juegos 2D. Utiliza una proyección ortográfica, lo que permite dibujar todos los elementos de la escena de manera uniforme, sin perspectiva, tal como se dibujarían las diferentes vistas en un mapa o un plano de un arquitecto.

Todas las distancias se mantienen intactas, aunque los elementos en la escena estén más cerca o más lejos de la cámara.

Este tipo de cámaras, además de para juegos 2D, también se suele utilizar para generar minimapas.



Esta imagen muestra cómo el *frustum* de esta cámara tiene forma rectangular y no tiene *field of view*

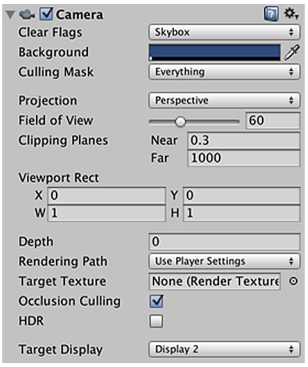
Los valores que configuran este *fustrum* vienen dados por:

- «Clipping Planes (Near/Far)»: Son los dos planos perpendiculares a la orientación de la cámara. Marcan dónde va a empezar a pintar y dónde va a terminar.

- «Size»: Este valor indica el ancho que tendrá la cámara. El ancho total de su *frustum* será «2*Size».

3.3. Propiedades generales

Sea cual sea la cámara que elijamos para nuestro juego, todas comparten ciertos valores comunes. Vamos a ver los más significativos:



- «Viewport Rect»: Estos cuatro valores indican en qué parte de la pantalla se pintará el contenido de la cámara. Los valores van de 0 a 1 en proporción a la pantalla. *X* e *Y* indican la posición inferior izquierda donde se pintará, y *W* y *H*, su anchura y altura en porcentaje de pantalla de 0 a 1.

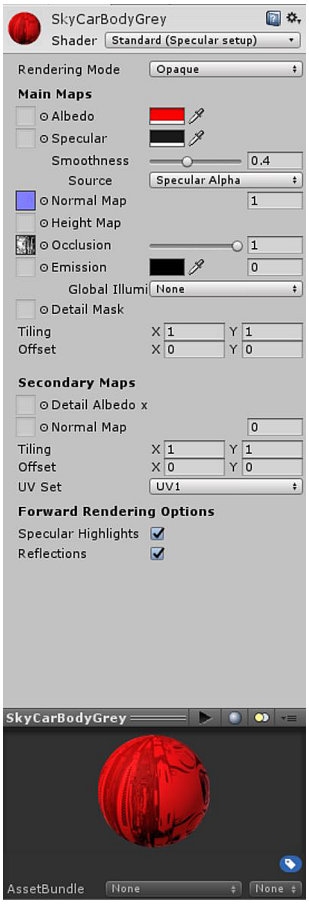
- «Depth»: Si hay varias cámaras activas a la vez, este valor indicará la prioridad a la hora de pintarse. Las cámaras de mayor valor se pintarán encima de las de menor valor.

4. Proyecto: Un juego de carreras

En esta sección vamos a describir cómo desarrollar un juego de carreras 3D sencillo. El juego consistirá en intentar ganar una carrera alrededor de un circuito contra varios coches. Este proyecto va a trabajar fundamentalmente con la física, las cámaras y una versión muy sencilla de una inteligencia artificial.

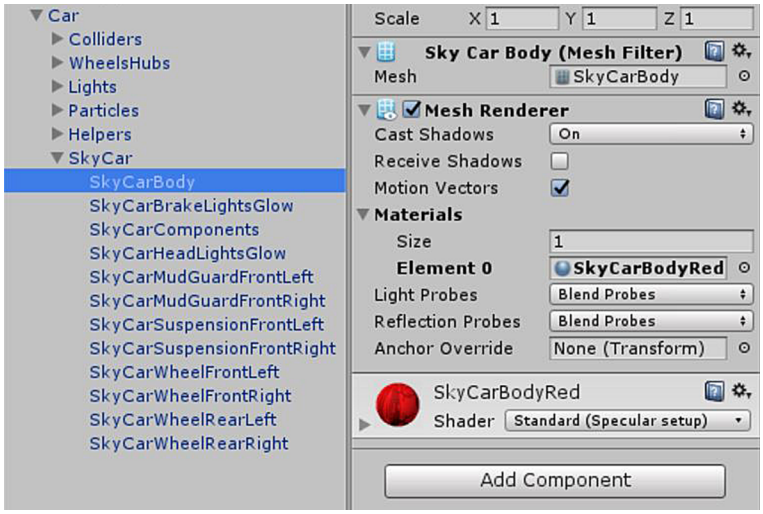
4.1. El coche

Ya tenemos nuestro coche estándar descargado en el *asset package* de Unity, pero viene sin textura de difuso y pintado de un gris muy apagado. Para darle un poco más de vida y variedad, cambiaremos los materiales de algunas zonas. Lo primero que haremos será duplicar el material «SkyCarBodyGrey» hasta crear tres más, uno con el tinte del «Albedo» en azul, otro en rojo y otro de un gris oscuro.



Inspector de materiales

Estos nuevos materiales se los asignaremos al componente «MeshRenderer» de las *meshes* del coche que queramos cambiar.



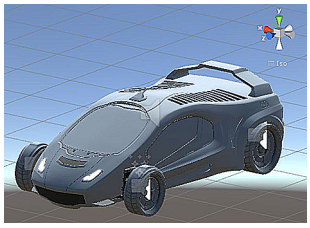
El componente «MeshRenderer» es el que se encarga de dibujar la geometría contenida en la *mesh* asignada al componente «Mesh Filter» de ese mismo objeto. Su propiedad «Materials» contiene una lista de todos los materiales que se van a utilizar para pintar esa geometría.

Los materiales contienen la información necesaria para pintar un triángulo de la geometría. Esa información suele contener la textura de difuso («Albedo») que definirá el color y el dibujo de ese triángulo, y su textura de normal que

Reto 1

Cambiad el color por defecto de la carrocería del coche, los cristales, las ruedas y los guardabarros.

sirve para darle un volumen ficticio a esa geometría. También incluye otras texturas que nos servirán para darle un resultado más realista al proceso de pintado o información sobre cómo deberían *tilearse* esas texturas.



Nuestro flamante coche gris

Ahora que ya tenemos nuestro coche personalizado vamos a crear un circuito por donde podamos circular y hacer la carrera.

4.2. El terreno

Un juego de carreras no está completo, por más coches que tenga, sin un escenario y una pista por la que correr. En nuestro ejercicio crearemos un terreno montañoso con un lago, un bosque y una gran explanada por la que serpenteará nuestra carretera.

Reto 2

Cread un terreno de 400 x 400 que contenga zonas montañosas, una gran explanada llena de césped, todo el terreno con bastantes árboles y vegetación, y una zona de bosque poblado. Que haya al menos dos tipos de césped y tres tipos de árboles. Y que las zonas montañosas no tengan la misma textura que el suelo y que sus picos estén nevados.

Para darle un poco más de vida a ese terreno, le añadiremos un pequeño lago. Para ello, utilizaremos el componente «Water», que viene con el *asset package Environment* que habremos instalado previamente.

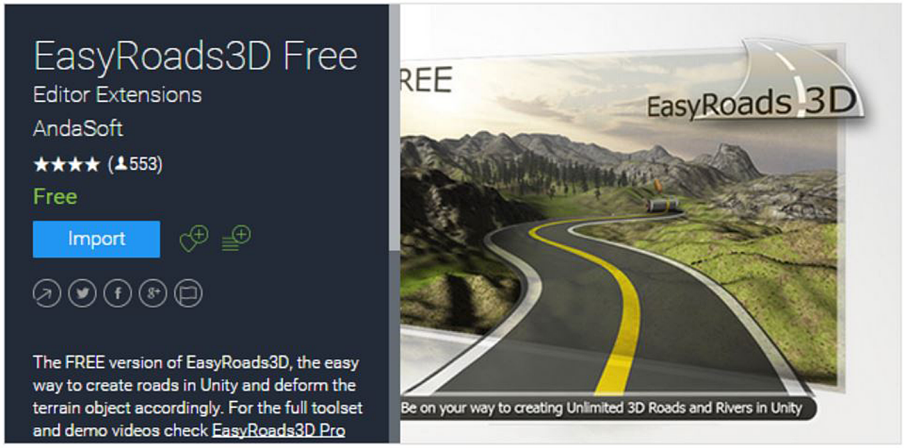
En este caso utilizaremos el *prefab* que se encuentra en «Assets/Standard Assets/Environment/Water/Water/Prefabs/WaterProDaytime.prefab».

Para este ejercicio no hará falta que retoquemos ninguno de sus valores más allá de su escala para adaptarlo a nuestro terreno y que no se vea el corte cuando termina.

4.3. El circuito

Vamos a añadirle a nuestro maravilloso terreno la guinda del pastel, el circuito. Unity no tiene una herramienta para este menester, y normalmente serían los artistas o diseñadores con herramientas personalizadas los que se encargarían de crear la geometría del circuito.

Como no es el objetivo de este módulo modelar 3D, utilizaremos una herramienta gratuita para crear carreteras disponible en la Asset Store.



Nota

Podéis descargar el *plugin* buscando su nombre en «Window > Asset Store», o desde el siguiente enlace:

https://www.assetstore.unity3d.com/en/#!/content/987.

Una vez descargado e instalado el paquete, insertaremos en nuestra escena el *prefab* «Assets/EasyRoads3D Free/Resources/EasyRoad3DObject.prefab». Este objeto contiene un componente llamado «Road Object Script» con un aspecto bastante similar al editor de terrenos.

Lo primero que os recomendamos hacer es ir al menú «GameObject > Create Other > EasyRoads3D > BackUp > All Terrain Data» para hacer una copia de seguridad de nuestro terreno, para que no perdamos lo que ya tenemos si luego queremos dar marcha atrás. Si fuera el caso, tan solo habría que ir a la opción «Restore > All Terrain Data» de ese mismo menú.

Os dejamos explorar con más tiempo todas las opciones de este interesante *plugin* de Unity, pero nosotros nos centraremos en su primer botón, «Add road markers», que nos permite crear nuestra carretera apretando «Shift» mientras clicamos en diferentes puntos del terreno. El último punto no cierra automáticamente la carretera; para ello, tendremos que ir al botón «Settings» y en la subcategoría «Object Settings» activar «Closed Track» para que se nos cierre el circuito.

No tengáis miedo de pasar por encima de montañas o entre árboles, ya que el circuito se adaptará a cualquier superficie (excepto el agua).

En «Settings» también podremos cambiar el ancho de la carretera con la opción «Road Width». Por desgracia y dado que se trata de una versión gratuita, si el circuito creado no es de nuestro agrado, tendremos que borrar el objeto y empezar de cero.

Una vez ya tengamos un circuito que nos guste, clicaremos en el tercer botón, «Procesar Terreno», para generar el circuito.

Como podéis ver, el *plugin* nos ha modificado el terreno para adaptarse a la orografía de este. En algunos puntos tendréis que ir con cuidado de no tener rampas muy pronunciadas o curvas muy cerradas porque en esos casos la geometría o el terreno no acaban de quedar bien.

Reto 3

Cread un circuito alrededor del terreno, que bordee el lago, cruce el bosque, pase por la explanada y serpentee las montañas, pero sin tener desniveles muy pronunciados o curvas extremadamente cerradas.

4.4. La cámara

Ahora que ya tenemos terreno, circuito y coche, nos queda colocar una cámara que lo persiga y nos permita poder darnos una vuelta por el terreno y por nuestro circuito.

Como cámara básica utilizaremos el *asset package Cameras*, que podremos importar al proyecto yendo al menú «Assets > Import Package > Cameras». Una vez instalado en el proyecto, utilizaremos el prefab que se encuentra en: «Assets/StandardAsset/Cameras/Prefabs/MultipurposeCameraRig.prefab».

Lo colocaremos detrás de nuestro coche a la distancia deseada (¡sin ser hijo!), y como el coche ya tendrá seguramente el *tag* «Player», el *script* de la cámara «Auto Cam» lo habrá seleccionado como el objeto que seguir.

Nota

Recordad que si no habéis borrado la *main camera* de la escena, tendréis dos cámaras a la vez y se pintará la que tenga el «Depth» más alto. Es recomendable tener solo una cámara activa a la vez.

Reto 4

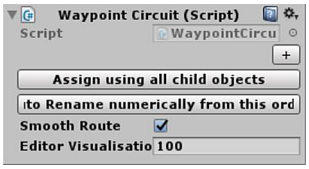
Modificad los valores del componente «Auto Cam» para que la cámara persiga al coche de una manera realista tal y como esperaríamos en un juego de coches normal.

4.5. Los rivales

Ahora que ya podemos dar vueltas nosotros solos por el circuito, ya solo nos queda crear a nuestros contrincantes en la carrera.

Para los coches de los enemigos utilizaremos un *prefab* similar al del coche del *player* pero que está preparado para seguir una serie de «Waypoints» que marcan la trazada del circuito.

Empezaremos pues añadiendo el *script* «Waypoint Circuit» a nuestro objeto «Road» que contiene la geometría del circuito. Y posteriormente iremos creando *GameObjects* vacíos como hijos suyos que vayan marcando toda la trazada del circuito.



Componente «WayPoint Circuit»

Mediante todos estos hijos, una vez tengamos la trazada completada, crearemos una trayectoria por la que circularán nuestras inteligencias artificiales.

Cuando tengamos todos los hijos creados, iremos al componente «WayPoint Circuit» de nuestro objeto «Road» y clicaremos encima del botón «Assign using all child objects».

Nota

Parece ser que el *script* tiene un problema al autoimportar y ordena los «Waypoints» por nombre y no por orden en el que fueron creados. Para que no tengamos problemas con ese orden, renombrad los nueve primeros *GameObjects* para que en vez de acabar en (1) acaben en (01) y así los coja antes que a los demás.

Una vez añadidos todos los «Waypoints», si modificamos la variable «Editor Visualisation Substeps», veremos cómo se ha creado una línea amarilla que será la trazada por la que intentarán circular nuestras inteligencias artificiales. A mayor valor, más suaves irán.

Reto 5

Cread toda la serie de «Waypoints» siguiendo la trazada de la carretera y a la misma altura que esta para que las inteligencias artificiales puedan circular por el mismo circuito que nosotros de manera automática.

Si ahora añadimos los *prefabs* de los coches enemigos desde el *prefab* «Assets/Standard Asset/Vehicles/Car/Prefabs/CarWaypointBased.prefab» y a la variable «Circuit» de su componente «Waypoint Progress Tracker» le pasamos nuestro objeto «Road» que contiene todos los «Waypoints», este ya sabrá cuál es la ruta que debe seguir.

Nota

Es importante cambiar el *tag* de «Player» a los coches de las inteligencias artificiales para que nuestra cámara no se vaya detrás de estas.

¡Listo! Ahora ya podemos correr contra diferentes coches en nuestro circuito.

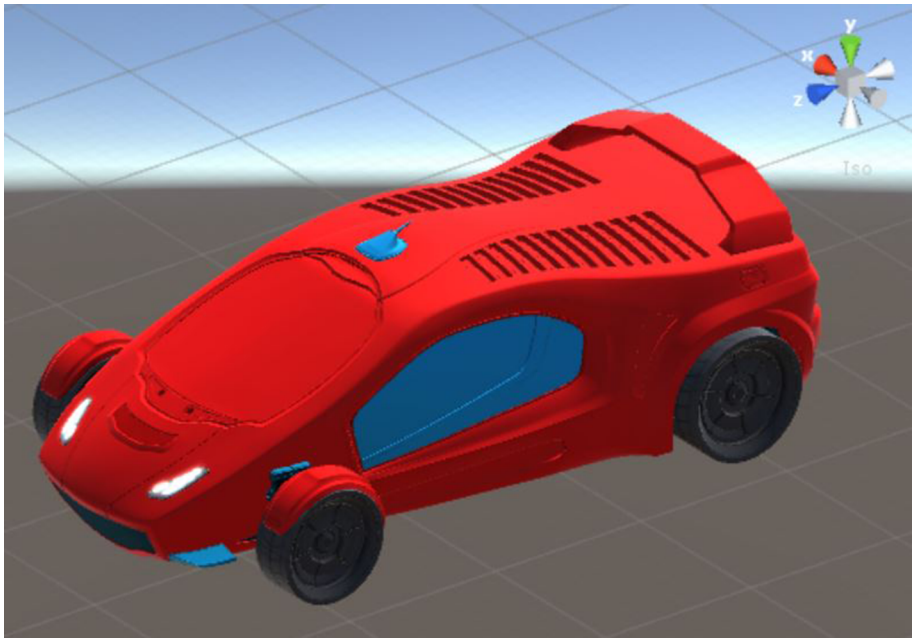
Reto 6

Cread diferentes inteligencias artificiales con coches de colores diferentes y que cada una empiece en un lugar distinto del circuito. Haced también que modificando el *script* del coche «Car AI Control» tengan un comportamiento distinto cada uno.

4.6. Soluciones a los retos propuestos

Reto 1

Es importante ver cómo la *mesh* «SkyCarComponents» tiene dos materiales asociados. Eso se debe a que ciertas partes de la geometría se pintarán con un material y otras, con el otro.

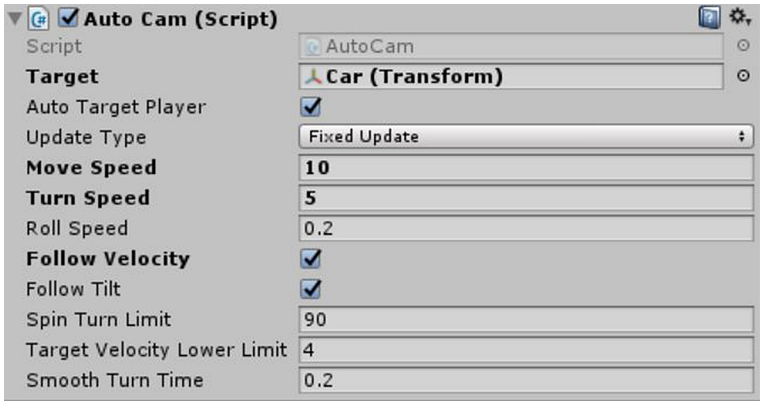


Retos 2 y 3

Para hacer la nieve de las montañas hay que utilizar una textura en blanco para pintarlas. Lo bueno de la flexibilidad del terreno es que permite crear cosas como la pequeña isla del lago de la imagen, poblada por palmeras.

Reto 4

Una de las posibles combinaciones de valores que funcionarían adecuadamente sería esta:



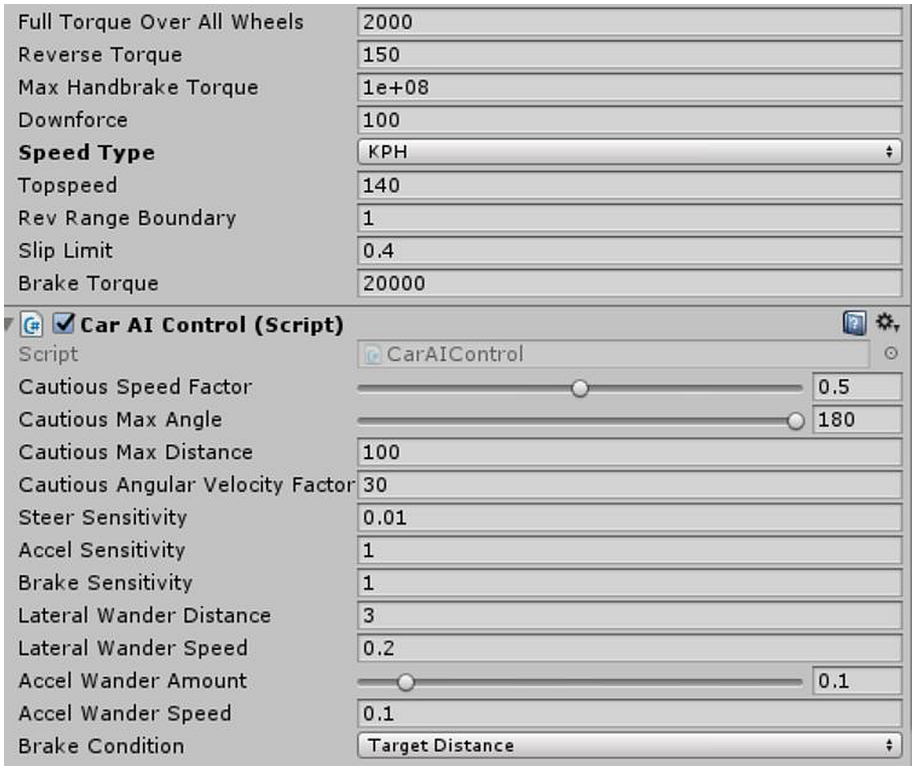
Reto 5

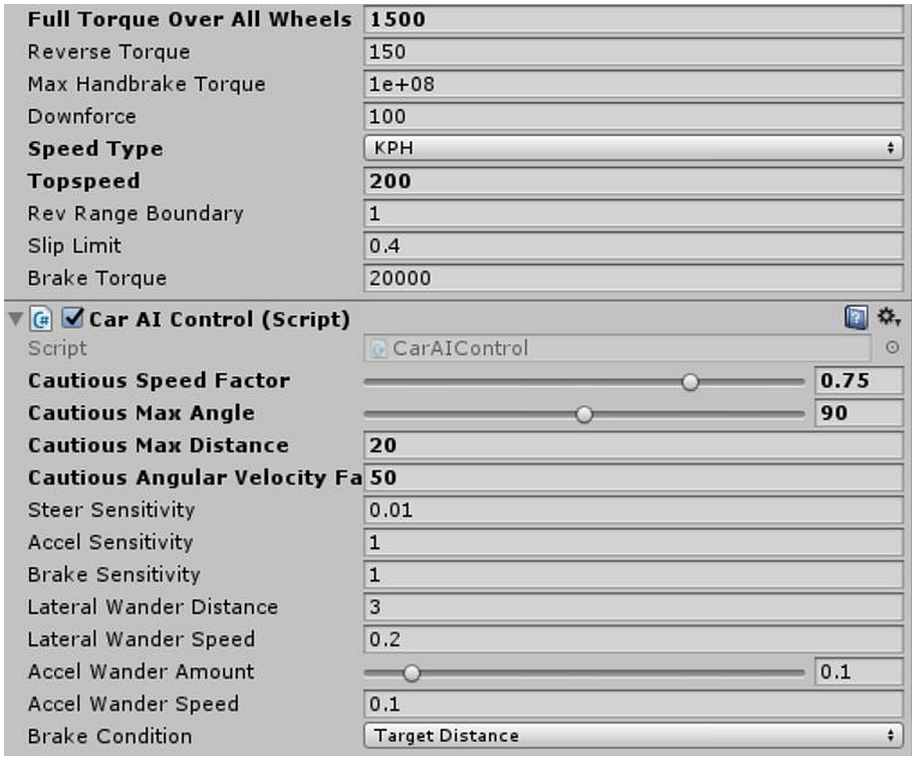
Es importante que cada uno de los *GameObjects* que marcarán los «Waypoints» esté situado en los puntos clave de giro de la trazada; cuantos más tengamos, más precisión tendremos. También es importante que estén a ras de pista.

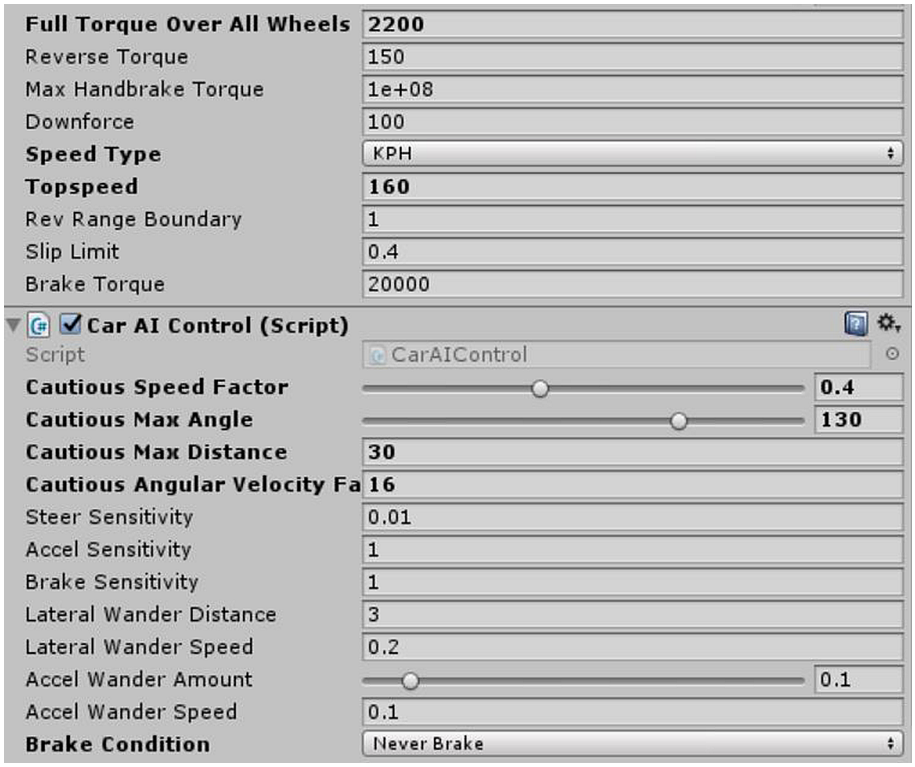
Reto 6

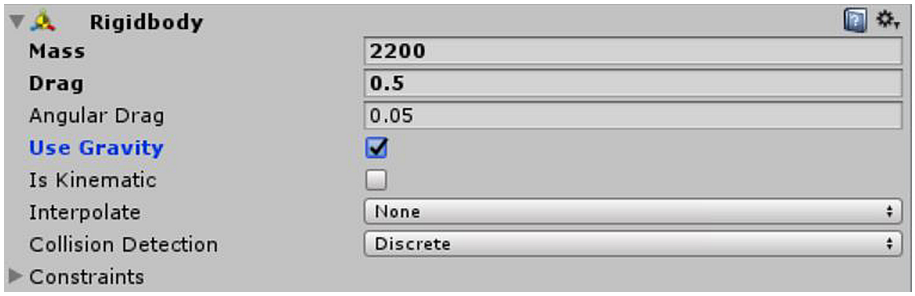
Aquí realmente tenéis mucha libertad para generar todo tipo de comportamientos. Pero los valores clave serían:

- Velocidad máxima.
- «Torque» (velocidad aplicada en cada ciclo).
- «Brake Torque» (lo mismo, pero de frenada).
- Sus niveles de cautela. Si gira antes, frena antes o después, etc.
- Si suele frenar o no frena nunca.
- Y, por último, su masa y su agarre en su «Rigybody».


Full Torque Over All Wheels 2000
Reverse Torque 150
Max Handbrake Torque 1e+08
Downforce 100
Speed Type KPH
Topspeed 140
Rev Range Boundary 1
Slip Limit 0.4
Brake Torque 20000
Car AI Control (Script)
Script CarAIControl
Cautious Speed Factor 0.5
Cautious Max Angle 180
Cautious Max Distance 100
Cautious Angular Velocity Factor 30
Steer Sensitivity 0.01
Accel Sensitivity 1
Brake Sensitivity 1
Lateral Wander Distance 3
Lateral Wander Speed 0.2
Accel Wander Amount 0.1
Accel Wander Speed 0.1
Brake Condition Target Distance


Full Torque Over All Wheels 1500
Reverse Torque 150
Max Handbrake Torque 1e+08
Downforce 100
Speed Type KPH
Topspeed 200
Rev Range Boundary 1
Slip Limit 0.4
Brake Torque 20000
Car AI Control (Script)
Script CarAIControl
Cautious Speed Factor 0.75
Cautious Max Angle 90
Cautious Max Distance 20
Cautious Angular Velocity Fa 50
Steer Sensitivity 0.01
Accel Sensitivity 1
Brake Sensitivity 1
Lateral Wander Distance 3
Lateral Wander Speed 0.2
Accel Wander Amount 0.1
Accel Wander Speed 0.1
Brake Condition Target Distance


Full Torque Over All Wheels 2200
Reverse Torque 150
Max Handbrake Torque 1e+08
Downforce 100
Speed Type KPH
Topspeed 160
Rev Range Boundary 1
Slip Limit 0.4
Brake Torque 20000
Car AI Control (Script)
Script CarAIControl
Cautious Speed Factor 0.4
Cautious Max Angle 130
Cautious Max Distance 30
Cautious Angular Velocity Fa 16
Steer Sensitivity 0.01
Accel Sensitivity 1
Brake Sensitivity 1
Lateral Wander Distance 3
Lateral Wander Speed 0.2
Accel Wander Amount 0.1
Accel Wander Speed 0.1
Brake Condition Never Brake


Rigidbody
Mass 2200
Drag 0.5
Angular Drag 0.05
Use Gravity
Is Kinematic
Interpolate None
Collision Detection Discrete
Constraints

Resumen

En este módulo didáctico hemos trabajado con algunos de los conceptos básicos del desarrollo 3D, pero encarándolos de una manera muy práctica y guiada. Primero hemos conocido el objeto «Terrain» y toda la potencia que nos ofrece. Con él podemos generar entornos con montañas, llanuras, bosques, lagos, césped, rocas, etc. Y todo con un sistema muy optimizado para su renderizado.

Seguidamente hemos visto el que seguramente sea el concepto más importante del módulo, que es el uso de las cámaras y sus propiedades. Las diferencias entre perspectiva y ortográfica son claves. Perspectiva es una visión muy similar a como vemos nosotros realmente el mundo y a como estamos acostumbrados a verlo tras una cámara, en la que los objetos, cuanto más alejados están, más pequeños se ven. Mientras que la ortográfica dibuja siempre las proporciones perfectas esté donde esté el objeto, perdiendo toda perspectiva.

Nos hemos adentrado en los comportamientos físicos en 3D, pero sin entrar en mucho detalle, ya que son los mismos que en 2D pero con una dimensión extra. Hemos visto los *prefabs* de vehículos que nos ofrece Unity y los *scripts* que los gestionan, pero con atención principal para el componente «Wheel Collider».

También hemos aprendido el concepto de material y cómo este va asociado a una *mesh* y a sus texturas.

Uno de los objetivos de este módulo es que el alumno investigue por sí solo cómo funcionan internamente los *scripts* que Unity nos va ofreciendo en sus componentes. De la manera como están creados se puede sacar mucha información valiosa. También podemos aprender mucho probando y haciendo nuestras propias modificaciones para conseguir que el *prefab* se comporte como nosotros queremos.

Hemos visto todos los pasos necesarios para diseñar un juego de carreras con algunos coches enemigos, creando nuestro propio circuito, las versiones de los coches y el comportamiento de la cámara.