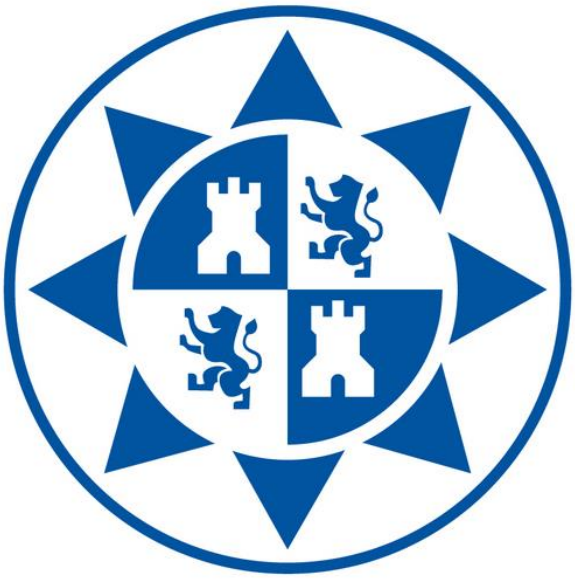
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA



Trabajo Fin de Grado

SIMULADOR HÍBRIDO DE REDES Y TRÁFICO PARA LA SIMULACIÓN DE SITUACIONES DE RIESGO DE ACCIDENTE

AUTOR: Elena Martín Seoane

DIRECTOR: Esteban Egea López

Mayo 2017

Autor	Elena Martín Seoane
E-mail del autor	elenamseoane@gmail.com
Director	Esteban Egea López
E-mail del director	esteban.egea@upct.es
Título del TFG	Simulador híbrido de redes y tráfico para la simulación de situaciones de riesgo de accidente
Descriptores	Simulación, Unity, redes vehiculares

Resumen

El propósito de este trabajo es el desarrollo de un simulador híbrido de redes y tráfico para proveer de una herramienta adecuada en la simulación de situaciones de riesgo de accidente. El objetivo es que el resultado de este proyecto pueda ser utilizado para demostraciones de situaciones de riesgo en carretera y con fines académicos y de investigación. Nos centraremos en la creación de dicha herramienta de simulación de forma extensible y en la ejemplificación de algunos de los escenarios de riesgo más comunes.

Titulación	Grado en Ingeniería Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de presentación	Mayo - 2017

ÍNDICE

1 Introducción 9
2 Tecnologías utilizadas 11
2.1 Unity3D 11
2.1.1 UnityCar Pro 12
2.1.2 EasyRoads 3D 12
2.2 MonoDevelop 12
2.3 OMNeT++ 12
2.3.1 Boost C++ 13
2.4 FlatBuffers 13
3 Desarrollo 15
3.1 Interconexión 15
3.2 Parte 1: Simulador en tiempo real (Unity) 16
3.2.1 Escenas del simulador 16
3.2.1.1 Escena principal 16
3.2.1.2 Escenas de riesgo: escenarios 1 y 2 18
3.2.2 Detalles del código empleado 19
3.2.2.1 Script AICarController 19
3.2.2.2 Script InstantiateAICars 20
3.2.2.3 Script InterpWp 21
3.2.2.4 Script ExternalSimClient 21
3.2.2.5 Script MessageManager 21
3.2.2.6 Script CommunicationSupport 22
3.3 Parte 2: Simulador en tiempo discreto (OMNeT++) 22
3.3.1 Clase ExternalClockScheduler 22
3.3.2 Módulo ExecutionServer 22
3.3.3 Módulo VenerisServer 23
3.3.4 Red de simulación 23
4 Manual de usuario 25
4.1 Creación de un nuevo proyecto 25
4.2 Creación de la escena y el terreno 26
4.3 Creación de la calzada 26
4.4 Colocación de los objetos en la escena 27
4.5 Parámetros de configuración 30

4.5.1 Configuración de parámetros de InstantiateAICars.cs 30
4.5.2 Configuración de ExternalSimClient.cs 31
4.6 Configuración del proyecto OMNeT++ 32
5 Resultados 35
5.1 OMNeT 35
5.2 Unity 36
6 Conclusiones y líneas futuras 37
7 Referencias y bibliografía 39

ÍNDICE DE FIGURAS

IMAGEN 1. LOGO DE UNITY3D 11
IMAGEN 2. LOGO DE MONODEVELOP 12
IMAGEN 3. LOGO DE OMNET++ 12
IMAGEN 4. LOGO DE FLATBUFFERS 13
IMAGEN 5. ESTRUCTURA BÁSICA DEL SIMULADOR 15
IMAGEN 6. DETALLE DE LA INTERCONEXIÓN DEL SIMULADOR 15
IMAGEN 7. ESTRUCTURA DE LAS ESCENAS (SCENES) EN UNITY 16
IMAGEN 8. JERARQUÍA DE OBJETOS EN LA ESCENA MAINMENU 17
IMAGEN 9. VISUALIZACIÓN DE LA ESCENA MAINMENU 17
IMAGEN 10. ESCENA REPRESENTANDO EL PRIMER ESCENARIO DE RIESGO. 18
IMAGEN 11. ESCENA REPRESENTANDO EL SEGUNDO ESCENARIO DE RIESGO 19
IMAGEN 12. ESQUEMA DE CONTROLADORES DE LOS VEHÍCULOS 20
IMAGEN 13. CONTENIDO DE LA CARPETA FLATBUFFERS 25
IMAGEN 14. PROPIEDADES DE UN OBJETO 'PLANE' DE EJEMPLO. 26
IMAGEN 15. MENÚ DE LA RED DE CARRETERAS DE EASYROADS 27
IMAGEN 16. JERARQUÍA DE OBJETOS EN UNA ESCENA DE SIMULACIÓN DE EJEMPLO 28
IMAGEN 17. EJEMPLO DE ESCENA CON CAMINO DE WAYPOINTS VISIBLE 28
IMAGEN 18. COMPONENTES DE UN BRAKINGPOINT 29
IMAGEN 19. EJEMPLO DE OBJETOS EN UN WAYPOINTCONTAINER 30
IMAGEN 20. CONFIGURACIÓN DEL OBJETO CONTROLLERS 31
IMAGEN 21. CONFIGURACIÓN DE EXTERNALSIMCLIENT 31
IMAGEN 22. CONFIGURACIÓN DE INCLUDES EN EL PROYECTO OMNET++ 32
IMAGEN 23. CONFIGURACIÓN DE LIBRARY PATHS EN EL PROYECTO OMNET++ 32
IMAGEN 24. VENTANA DE SIMULACIÓN EN OMNET++ 35
IMAGEN 25. RESULTADO DE LA EJECUCIÓN EN UNITY. 36
IMAGEN 26. CONSOLA DE OMNET++ CON MENSAJES DE LA SIMULACIÓN. 36

1 INTRODUCCIÓN

El objetivo de este proyecto es el desarrollo de un simulador híbrido de redes y tráfico para la simulación de situaciones de riesgo de accidente.

El proyecto se encuentra enmarcado en el ámbito de la seguridad vial y la aplicación de las nuevas tecnologías a la reducción del número y gravedad de accidentes en carretera. La problemática principal de este campo de estudio es la falta de herramientas adecuadas para simular tanto la física de vehículos y calzada como las comunicaciones entre ellos. Mientras que para el primer caso la mejor opción son los motores de desarrollo 3D como los empleados en el diseño de videojuegos, para el segundo suelen emplearse diversas herramientas de simulación de tecnologías de comunicación y diseño de redes. Estas dos opciones son muy diferentes y emplean modelos temporales totalmente opuestos: los entornos de desarrollo visuales en 3D trabajan con sistemas de renderizado en tiempo continuo mientras que los simuladores de redes trabajan en tiempo discreto.

Se hace necesario, por tanto, un simulador adecuado que sincronice ambos modelos y provea de una herramienta útil a investigadores y particulares, de forma que pueda ser empleado en demostraciones de situaciones de riesgo en carretera y contextos de aprendizaje e investigación sobre posibles modos de prevenir y evitar peligros al volante.

Con dicho objetivo, este proyecto se centrará en desarrollar una herramienta de simulación extensible a diversos escenarios de situaciones de riesgo en carretera y ejemplificará algunos de los más comunes. Para ello se empleará el motor gráfico Unity3D junto con el simulador de redes en tiempo discreto OMNeT++, empleando las librerías de FlatBuffers para la interconexión de ambas plataformas.

2 TECNOLOGÍAS UTILIZADAS

En primer lugar se van a detallar las diferentes tecnologías que se han empleado durante el desarrollo de este trabajo, así como los motivos de su elección y su función en proyecto.

2.1 UNITY3D

Es un motor de videojuegos multiplataforma en tiempo real muy utilizado en todo el mundo. Permite el desarrollo en lenguajes como C#, Boo y JavaScript y se integra de forma nativa con Visual Studio y MonoDevelop. Además, la versión básica incluye todas las funcionalidades necesarias para nuestro caso y es de licencia gratuita.

Unity se encuentra disponible para Windows y MacOS y permite la exportación a casi cualquier sistema de escritorio, web, móvil, consola o TV, entre los que se incluyen Windows, Linux, MacOS, Android, Windows Phone, iOS o sistemas de realidad virtual como Gear VR, Oculus Rift o PlayStation VR entre otros.

Imagen 1. Logo de Unity3D

En este proyecto se encarga de la simulación de la física, tanto de vehículos como de terreno, y además lleva el peso de la simulación en general, pues es quien se encarga de iniciar la simulación de los distintos escenarios de riesgo diseñados.

Se ha escogido este motor frente a otras posibles opciones como Shiva 3D, GameMaker Studio o Torque porque es el más sencillo de aprender a utilizar. Además, es el más extendido en la comunidad de desarrolladores, por lo que se puede encontrar mucho material disponible de forma gratuita en su Asset Store y existen grandes cantidades de manuales y comunidades de usuarios dedicadas a la producción de documentación, lo que simplifica el aprendizaje y desarrollo.

Respecto a los posibles lenguajes de programación, se ha escogido C# por varias razones. La primera se basa en sus similitudes con Java, el cual ha sido estudiado en varias asignaturas a lo largo del Grado. La segunda, consiste en su mayor predictibilidad frente a JavaScript, que en ocasiones confunde al programador con los tipos de variables dinámicas que emplea.

En la AssetStore de Unity podemos encontrar gran cantidad de paquetes que podemos incorporar a nuestro proyecto para simplificar el desarrollo.

2.1.1 UnityCar Pro

Implementa una física muy realista de la conducción y permite integrar diversos modelos de vehículos en nuestro proyecto sin necesidad de diseñarlos desde cero. Se ha empleado UnityCar para este propósito y simplificar así el proceso de desarrollo.

Como contraprestaciones se encuentran la complejidad del código que emplea internamente, lo que dificulta su extensión o modificación y el hecho de no ser software libre como el resto de tecnologías empleadas.

2.1.2 EasyRoads 3D

Es el paquete empleado para el diseño de las calzadas en los escenarios ejemplificados. Incorpora un editor para crear las carreteras directamente sobre el terreno y varios modelos de asfalto para simular diferentes vías.

2.2 MONODEVELOP

Se trata de un entorno de desarrollo de libre distribución multiplataforma utilizado para la programación de los scripts de Unity en C#.

Imagen 2. Logo de MonoDevelop

Aunque viene predeterminado Visual Studio como entorno de desarrollo, se ha decidido optar en su lugar por MonoDevelop dada su mayor eficiencia y su mejor conexión con Unity para efectuar pruebas y realizar *debugs* del código.

2.3 OMNET++

OMNeT++ es un simulador de redes en tiempo discreto empleado ampliamente en propósitos de investigación y educación. Se encuentra disponible tanto para sistemas Unix como para Windows y se utiliza habitualmente para modelar tráfico de redes de comunicaciones y simular protocolos distribuidos.

Emplea un entorno de desarrollo basado en Eclipse y utiliza como lenguaje de programación C++.

Imagen 3. Logo de OMNeT++

Ha sido escogido principalmente por la experiencia previa del grupo de investigación con este simulador y por su gran eficacia a la hora de simular el comportamiento de cualquier tipo de red de telecomunicaciones. Además, los desarrollos previos a este trabajo se encontraban realizados en una versión anterior de este programa por lo que sólo ha sido necesario efectuar los cambios oportunos para exportar el código ya programado a la versión actual.

2.3.1 Boost C++

Se trata de un conjunto de librerías para C++ que dan soporte para tareas de álgebra lineal, generación de número pseudoaleatorios y procesamiento multihilo entre otras.

Estas librerías son de licencia libre y están diseñadas para ser compatibles con cualquier programa o aplicación en C++. Aunque existen más de ochenta librerías incluidas en Boost, para este proyecto solo se emplea la librería *Boost.Asio*, que hace posible el procesamiento de datos de manera asíncrona y gestiona la programación de bajo nivel de E/S asíncrona.

2.4 FLATBUFFERS

Es una librería de serialización entre plataformas que soporta lenguajes como C++, C, C#, Java, JavaScript, Python, PHP o Go. Fue creada originalmente por Google para el desarrollo de videojuegos y otros sistemas en los que el rendimiento y tiempo de ejecución son críticos.

Imagen 4. Logo de FlatBuffers

Provee de acceso a datos serializados sin necesidad de conversiones y se ha seleccionado por su especial eficiencia de memoria, rapidez y flexibilidad. Esta librería es utilizada para solventar el problema que se genera cuando el entorno de simulación de redes (OMNeT++) emplea C++ mientras que el entorno gráfico (Unity) utiliza C#. De esta forma podemos realizar el intercambio de datos entre las plataformas sin demasiados inconvenientes.

3 DESARROLLO

La estructura principal del proyecto involucra dos simuladores interconectados entre sí. Por una parte, tenemos el simulador de redes OMNeT++ trabajando en la simulación de las comunicaciones mientras que por otra tenemos al motor gráfico de Unity que llevará la física de los vehículos y la parte visible del proyecto. Para facilitar la comprensión se desarrollan a continuación las partes individualmente, detallando el contenido y función de cada una de ellas.

Imagen 5. Estructura básica del simulador

3.1 INTERCONEXIÓN

En primer lugar se detalla a continuación la forma de interconexión entre ambos sistemas. Dado que uno es un simulador en tiempo real y otro funciona por eventos en tiempo discreto, además de ejecutarse en diferentes lenguajes, son necesarias una serie de librerías y la sincronización se realiza como sigue:

Unity funciona en tiempo real, ejecutando actualizaciones de estado a un determinado 'framerate'. Esto se mide en Hz, es decir, s^{-1}, por lo que la simulación se actualiza cada 1/framerate segundos. Dado que OMNeT++ realiza su ejecución en tiempo discreto utilizando una lista de eventos futuros o FEL (Future Event List), ambos sistemas no son compatibles en primera instancia. Es por ello que Unity, dado que es el sistema con mayor frecuencia de actualización, debe informar a OMNeT del tiempo de simulación actual. De esta forma, el simulador de redes puede extraer los eventos correspondientes al instante temporal real.

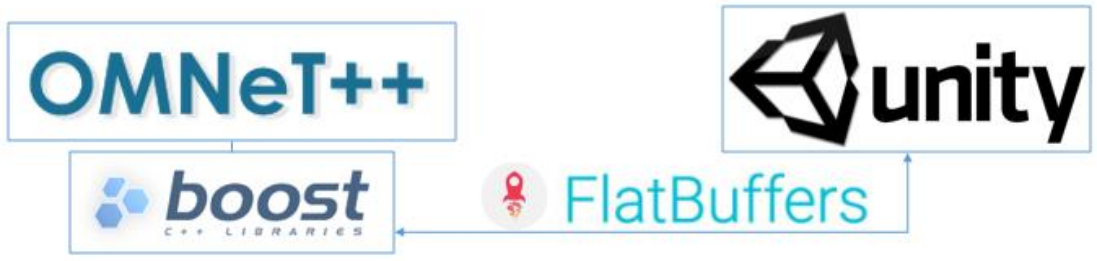


Imagen 6. Detalle de la interconexión del simulador

El simulador de redes OMNeT++ emplea la librería de Boost C++ Asio para simplificar el uso de los sockets de comunicación que se emplearán entre un sistema y otro. Para comunicarse con el simulador Unity se ha empleado la librería FlatBuffers, que realiza una serialización de los mensajes, convirtiéndolos a un lenguaje común fácilmente entendible por la mayoría de sistemas (ya sean C++, C#, Java, etc.).

3.2 PARTE 1: SIMULADOR EN TIEMPO REAL (UNITY)

Para realizar el simulador se ha empleado el paquete UnityCar Pro, de la AssetStore de Unity, en conjunto con una serie de scripts y modificaciones que serán detalladas en posteriores apartados. Además, para el intercambio de información con OMNeT se emplean las clases generadas por FlatBuffers en C#.

3.2.1 Escenas del simulador

Para describir cómo se ha implementado este proyecto es necesario saber que Unity divide su funcionamiento interno en *escenas*. Cada escena es completamente independiente del resto y debe contener sus propias instancias de los scripts u objetos que se deseen emplear.

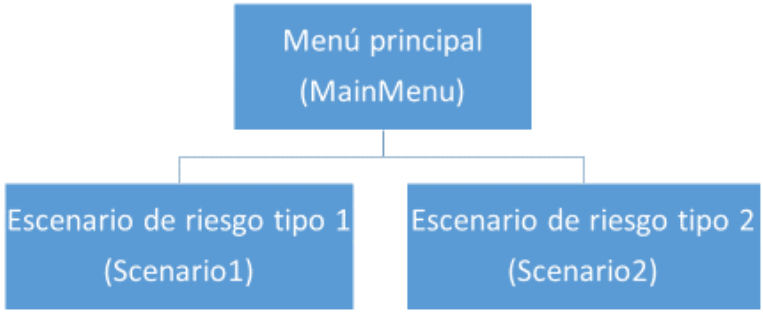


Imagen 7. Estructura de las escenas (scenes) en Unity

En nuestro caso, el proyecto consta de tres escenas. La primera de ellas, con la que se arranca el simulador, es simplemente un menú de bienvenida donde podemos escoger entre los dos escenarios de riesgo de ejemplo que se han desarrollado en este trabajo, cada uno implementado en una escena diferente dentro de Unity.

3.2.1.1 Escena principal

Esta primera escena es un sencillo menú de bienvenida en el que el usuario del simulador puede elegir entre algunos de los distintos escenarios implementados durante el desarrollo de este trabajo. Si fuesen necesarios parámetros adicionales para realizar la ejecución, esta sería la pantalla donde se solicitarían.

En este caso, la escena consta únicamente de elementos visuales como el título, el texto de bienvenida o los botones de selección de escenario. Se ha realizado empleando el sistema de UI de Unity, con lo que la estructura de objetos de la escena queda como se muestra en la Imagen 8.

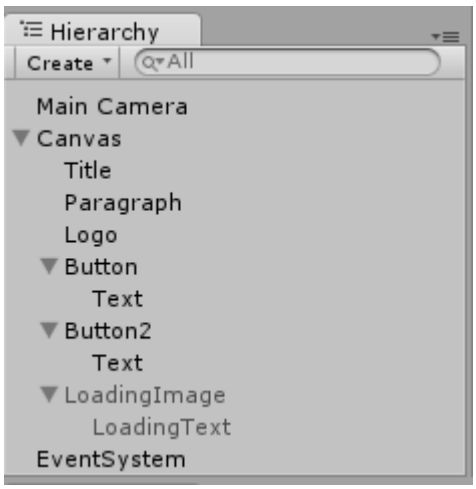


Imagen 8. Jerarquía de objetos en la escena MainMenu

Dado el fuerte carácter jerárquico del modelo de UI que se emplea, es importante destacar que el objeto que representa la imagen de carga que aparece como transición entre esta escena de selección y el escenario que se haya escogido debe estar siempre en la última posición dentro del objeto *Canvas*, pues de otra forma se solaparían unas imágenes con otras.

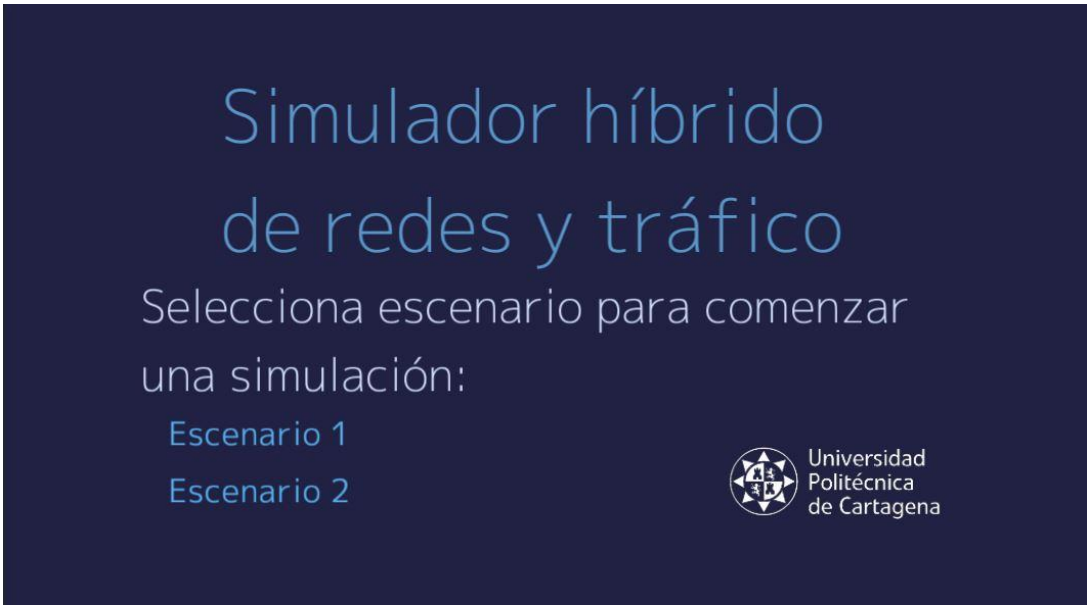


Imagen 9. Visualización de la escena MainMenu

3.2.1.2 Escenas de riesgo: escenarios 1 y 2

De acuerdo a las estadísticas, la mayoría de accidentes en carretera tienen lugar en intersecciones, especialmente en las de cuatro direcciones. Siguiendo esta línea, grupos de investigación han especificado cuáles son las situaciones más comunes en las que se producen accidentes entre vehículos[1]. Es por ello que se ha decidido implementar dos de ellas para ejemplificar el funcionamiento de este simulador.

El primer escenario de riesgo se corresponde con la escena número uno. En él, el vehículo principal se aproxima a la intersección cuando un vehículo aparece repentinamente por su izquierda.

En la imagen se ha representado con una flecha gruesa en color naranja el punto por el que aparecerá el vehículo con el que se colisiona y con una flecha más fina en azul, la dirección en la que circula el vehículo principal.

Imagen 10. Escena representando el primer escenario de riesgo.

Por otra parte, en el segundo escenario de riesgo, el vehículo con el que se produce la colisión está situado delante del vehículo principal y se prepara para girar a la derecha. Es al hacer este giro cuando frena de forma inesperada para el conductor principal, que tiene pocos segundos para evitar la colisión.

Al igual que en la imagen anterior, la flecha naranja representa al vehículo de la colisión, que en este caso se encontrará inmediatamente a continuación del coche

[1] Véase el capítulo *8.4.6 – Intersection scenarios* sobre las situaciones de riesgo en intersecciones definidas en el proyecto de investigación europeo TRACE (Traffic Accident Causation in Europe) [4].

principal en la misma calzada. Ambos se dirigen en la misma dirección como indica la flecha azul del conductor.

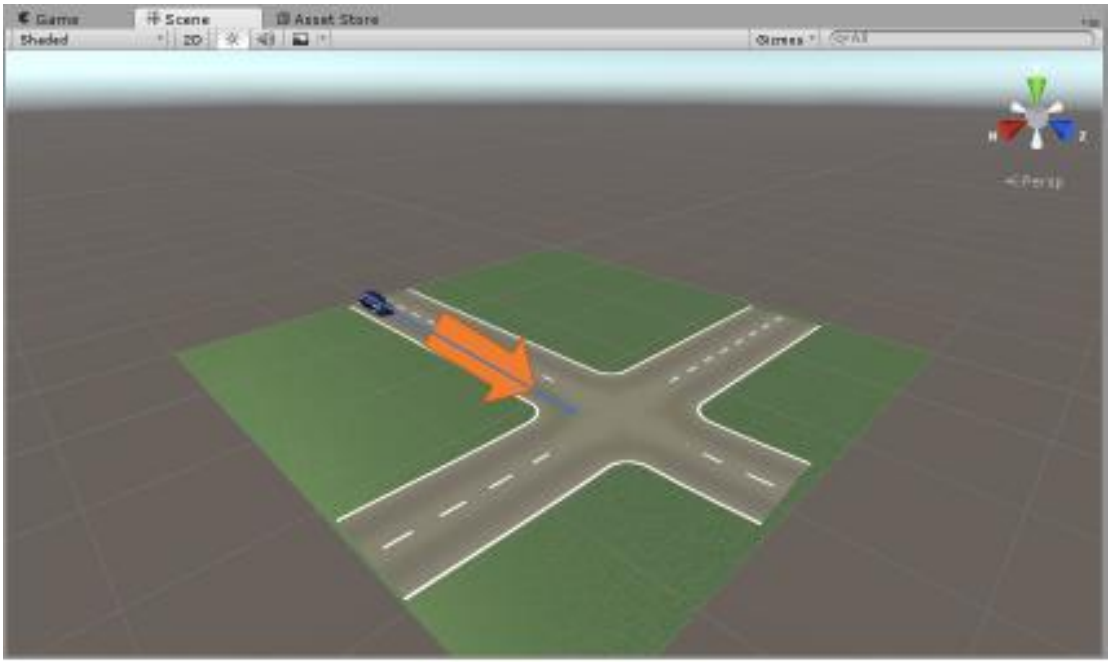


Imagen 11. Escena representando el segundo escenario de riesgo

3.2.2 Detalles del código empleado

Para el desarrollo del simulador se ha utilizado el paquete UnityCar Pro, que contiene modelos de vehículos y sus características, así como controladores para manejarlos y simular la física asociada.

Estos controladores, sin embargo, solo funcionan para el vehículo que maneja el usuario, por lo que sería imposible realizar un simulador de situaciones de riesgo sin otros vehículos en la calzada que imiten el tráfico real. Es por ello que se han desarrollado unos scripts que controlan los vehículos de la inteligencia artificial, de forma que sigan un camino preestablecido. Además, se han desarrollado otras clases para la automatización del simulador como la generación de los vehículos de tráfico.

3.2.2.1 Script AICarController

Será el encargado del movimiento autónomo del vehículo. Esta clase hereda de la clase abstracta CarController perteneciente a UnityCar, de forma que la estructura de programación no se vea excesivamente alterada y el sistema pueda continuar funcionando apropiadamente. Para ello se ha sobreescrito el método GetInput de dicha clase, donde deben fijarse los parámetros adecuados para que el paquete UnityCar genere el movimiento deseado del vehículo. Dichos parámetros consisten en el ángulo de giro del volate, la presión de los pedales de aceleración y freno y otras variables de un coche real como pueden ser el freno de mano o el arranque de motor.

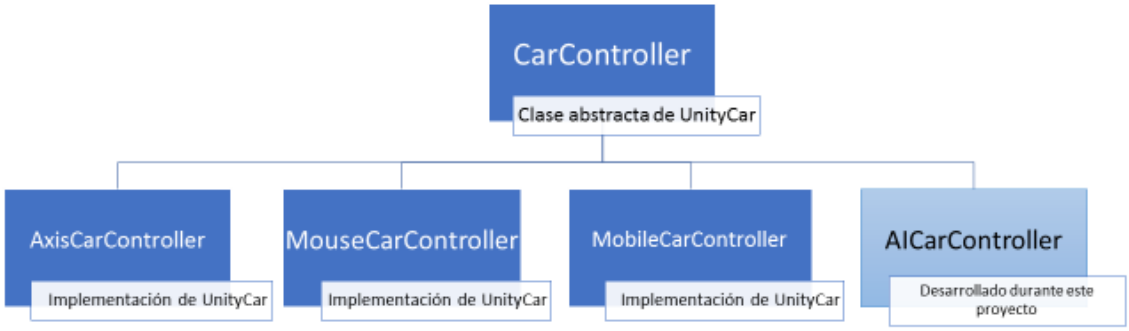


Imagen 12. Esquema de controladores de los vehículos

Inicialmente el método *Awake()* obtiene instancias de los scripts de UnityCar referentes al funcionamiento del motor y el vehículo, para poder posteriormente modificar los atributos necesarios y hacer que el coche se mueva o frene según nuestro diseño.

El método *GetInput* es el que contiene todo el código para el funcionamiento de la AI. Mientras tenga puntos que seguir, obtendrá el vector dirección relativo para indicar el ángulo del volante y presionará los pedales para acelerar o frenar según convenga. Cuando llegue al último punto, frenará, colocando el freno de mano y apagando el motor, dando por concluido el viaje.

Este script contiene, además, un método denominado *setCarPath*, que será invocado desde el script InstantiateAICars para asignar el camino de puntos a seguir.

3.2.2.2 Script InstantiateAICars

El script contiene un método *Start()* que será invocado tras la creación del mismo. En él se inicializan el generador de números aleatorios y las variables necesarias para la comprobación del número de vehículos que se instanciarán. Por último, se realiza una llamada a la función que preparará el sistema para la creación de los AICars, eligiendo el modelo de vehículo desde la lista para posteriormente simplemente crear la instancia de dicho modelo.

Para la creación de los vehículos, el método *Update()* comprobará si se ha generado el número de vehículos deseado y llamará a la función *InstantiateAI* que genera los AICars. Dicha función sólo es invocada si han transcurrido 7 segundos desde la anterior para evitar que todos los vehículos se generen a la vez y provoquen un colapso. En este método también se comprueban el número de frames que han transcurrido desde la última transmisión de un mensaje temporal y, cuando llega a 10, envía uno y resetea el contador.

Por último, el método *InstantiateAI* es el encargado de generar cada vehículo. Instancia el prefab seleccionado anteriormente, lo sitúa en la posición indicada por el programador y le añade el script AICarController, que será el encargado del movimiento autónomo del vehículo. Posteriormente se accede a la función *setCarPath* de dicho script, pasando como argumento el contenedor de los puntos del camino que deseamos que recorra. Por último, se le asigna un ID al coche y se le añade un fichero de configuración que contiene las características para la simulación física del vehículo. Antes de salir de la

función, se incrementa la variable que indica el número de coches instanciados hasta el momento y se envía un mensaje de aviso de creación de un AICar a OMNeT.

3.2.2.3 Script InterpWp

Inicialmente, con su método *Start()* obtiene todos los puntos contenidos en el objeto de entrada y los transforma en vectores de tres coordenadas para que el sistema pueda manejarlos. A continuación llama a la función *DPH*, a la que pasa el array de vectores que acaba de obtener. Es entonces cuando se realiza la interpolación con el método Catmull-Rom para generar un camino más suave con gran densidad de puntos por el que el vehículo pueda moverse de forma natural, tal como lo haría en una situación de la vida real.

El resultado de esta interpolación será un gran contenedor de puntos con los que el vehículo guiará su trayectoria. Es por esto que al instanciar el objeto AICar debemos pasarle en el apartado correspondiente al camino, el objeto vacío con el script InterpWP, de manera que no se confunda y seleccione el camino previamente trazado a mano.

3.2.2.4 Script ExternalSimClient

En su inicio, es el encargado de establecer la comunicación con el proyecto OMNeT. Establece la dirección IP y número de puerto de la máquina en la que se ejecuta y trata de inicializar un socket de comunicaciones con dichos parámetros.

Este script será añadido a un objeto cualquiera de la escena, aunque se recomienda que sea siempre al objeto 'Controllers' dado que de esta forma se facilita la comprensión y se evita mezclar propiedades de objetos.

Su método *Update*, ejecutado cada frame, va comprobando si tiene nuevos mensajes para enviar y, de ser así y estar establecida la conexión, los transmite mediante el uso de las funciones *sendHeader* y *sendMsg*, también implementadas en esta clase.

Por último, contiene un método *OnApplicationQuit* que se ejecutará, como su nombre indica, al finalizar la simulación y enviará un mensaje de tipo 'End' a OMNeT y cerrará el socket de la comunicación.

3.2.2.5 Script MessageManager

Es una clase estática muy sencilla cuyo único propósito es proporcionar las herramientas adecuadas para implementar la cola de mensajes. Contiene métodos para encolar, extraer de la cola, comprobar si hay mensajes acumulados para enviar y resetear la FIFO.

3.2.2.6 Script CommunicationSupport

Por último, encontramos el script encargado de la generación de la estructura de los mensajes, existiendo un método correspondiente a cada tipo de mensaje que enviaremos. Esto se realiza gracias a las clases generadas por FlatBuffers, simplemente se crea una instancia del tipo de mensaje pertinente y se le pasan todos los datos que debe contener.

Para finalizar, antes de salir del método, llama a la función de encolar proporcionada por el MessageManager para añadir el mensaje a la cola de salida.

3.3 PARTE 2: SIMULADOR EN TIEMPO DISCRETO (OMNET++)

Para realizar la simulación de la red de comunicaciones se empleará el simulador OMNeT++. Para el funcionamiento este proyecto se han desarrollado una serie de clases y archivos necesarios para ejecutar la simulación.

3.3.1 Clase ExternalClockScheduler

En esta clase se encuentra el código relativo a la planificación y gestión de los eventos de la cola.

Contiene los métodos *startRun()* y *endRun(),* que se ejecutan al inicio y final de la simulación y se encargan de inicializar y cerrar las variables necesarias, respectivamente. Ambos métodos son heredados de la clase 'cScheduler' junto con *getNextEvent()*, encargado de extraer el siguiente evento de la cola cuando corresponda.

Por último, contiene un último método denominado *setExternalTime(simtime_t)* que actualiza la variable del tiempo que recibe de Unity para que el sistema conozca en todo momento el tiempo de la simulación.

3.3.2 Módulo ExecutionServer

Se encarga de establecer la comunicación con Unity gracias a las librerías de Boost C++. Hereda de 'cSimpleModule' e implementa todos los métodos necesarios para manejar la comunicación.

Para realizar la simulación se emplean en primer lugar los métodos *initialize(), initializeServer()* y *launchServer(),* que se encargan de arrancar el servidor e inicializar los sockets.

Otros métodos relevantes implementados en esta clase son *handleMessage*, *readMsg* y *processMessage*, los encargados de determinar el tipo de los mensajes que se reciben y de procesarlos, mientras que gracias a *ExternalTime* procesa los mensajes del mismo tipo.

3.3.3 Módulo VenerisServer

Hereda de la clase explicada anteriormente, ‘ExecutionServer’ e implementa toda la funcionalidad de Veneris, procesando cualquier mensaje recibido. Contiene un método para cada tipo de mensaje que se puede recibir, que será invocado a su llegada.

3.3.4 Red de simulación

Para ejecutar el servidor y comprobar el funcionamiento del simulador en su conjunto, existe una red de prueba denominada ‘test1’ que se ejecutará al hacer el *run* del proyecto OMNeT y contará con los módulos apropiados para poder verificar la recepción de los mensajes en el sistema.

4 MANUAL DE USUARIO

A continuación se detallan los pasos a seguir para poder diseñar un escenario de riesgo y ejecutar una simulación propia basada en el simulador desarrollado en este proyecto.

4.1 CREACIÓN DE UN NUEVO PROYECTO

En primer lugar deberemos crear un nuevo proyecto en Unity e importar los paquetes de UnityCar y de EasyRoads o el equivalente con el que deseemos crear el trazado de la carretera.

Para poder conectar con el simulador OMNeT++, se necesita incluir al proyecto la librería FlatBuffers y las clases correspondientes a los mensajes de dicho simulador. Primero, se debe crear una carpeta dentro de los Assets del proyecto llamada "_Scripts". De esta forma, tanto el programa como el usuario sabrán encontrar y colocar los scripts para las simulaciones. En ella también deberán insertarse los scripts generados a lo largo de este proyecto para el correcto funcionamiento del simulador.

Una vez creada la carpeta, añadimos una subcarpeta llamada "FlatBuffers" donde copiaremos todo el contenido de la carpeta "FlatBuffers/net" de la librería disponible para bajar online del repositorio GitHub. Además, añadiremos también todos los archivos compilados de los mensajes que intercambiarán OMNeT++ y Unity.

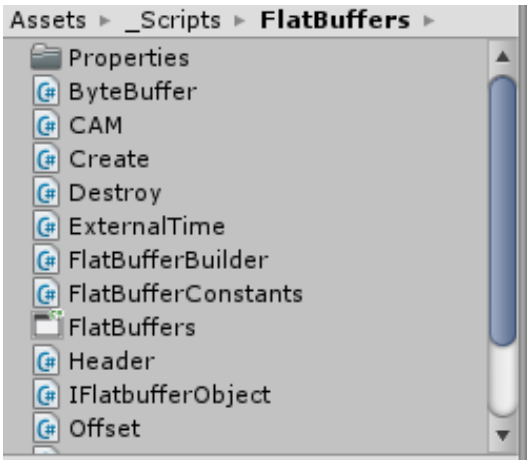


Imagen 13. Contenido de la carpeta FlatBuffers.

También deberemos copiar dentro de Assets, la carpeta llamada "_myPrefabs" generada durante la realización de este proyecto, puesto que contiene *prefabs* que deberemos instanciar más adelante durante la creación de la escena de simulación.

4.2 CREACIÓN DE LA ESCENA Y EL TERRENO

Una vez listo, creamos una nueva escena, añadimos el terreno y le asignamos un mapa de textura. También debemos asegurarnos de que se haya creado la luz direccional que iluminará nuestro mapa o no podremos ver los objetos una vez se inicie la simulación.

También es aconsejable añadir barreras en los bordes del terreno. Sin este paso, cualquier vehículo que se acerque demasiado al final de la calzada caerá por el borde del terreno y no podremos recuperarlo. Para ello basta con añadir objetos de tipo *plano*, colocarlos en modo vertical y situarlos de manera que formen un rectángulo en los cuatro bordes del mapa. Para que no dificulten la visión de la simulación, se puede desactivar el componente 'Mesh Renderer', de forma que el objeto seguirá siendo susceptible a colisiones, pero el usuario no será capaz de verlo y así no cerrará el paso a la luz.

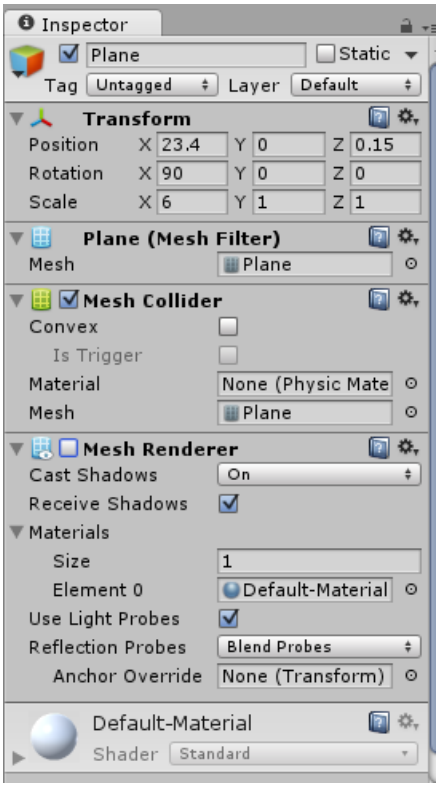


Imagen 14. Propiedades de un objeto 'Plane' de ejemplo.

4.3 CREACIÓN DE LA CALZADA

El proyecto de Unity constará de una escena con la calzada en la que tendrá lugar la situación de peligro. Dicha calzada, podrá ser generada de diversas formas. En este caso, han sido creadas con el paquete de la AssetStore de Unity, 'EasyRoads 3D'.

Una vez se instala el paquete en nuestro proyecto, aparecerá un nuevo apartado en el menú 'GameObject' > '3D Object' llamado EasyRoads3D. Pinchamos dentro de él y pulsamos sobre 'New Road Network'. Esto generará los objetos necesarios sobre nuestra

escena para que podamos dibujar el trazado de la carretera. Es importante que este paso tenga lugar *después* de haber finalizado la edición del terreno, pues si se realizan cambios en él posteriormente se pueden generar errores en la escena.

Una vez dispongamos del objeto de la red de carreteras, podremos acceder a su menú de creación en el Inspector y a partir de entonces podremos modificar los datos de la calzada desde él para adecuarla al escenario de riesgo que se haya decidido representar.

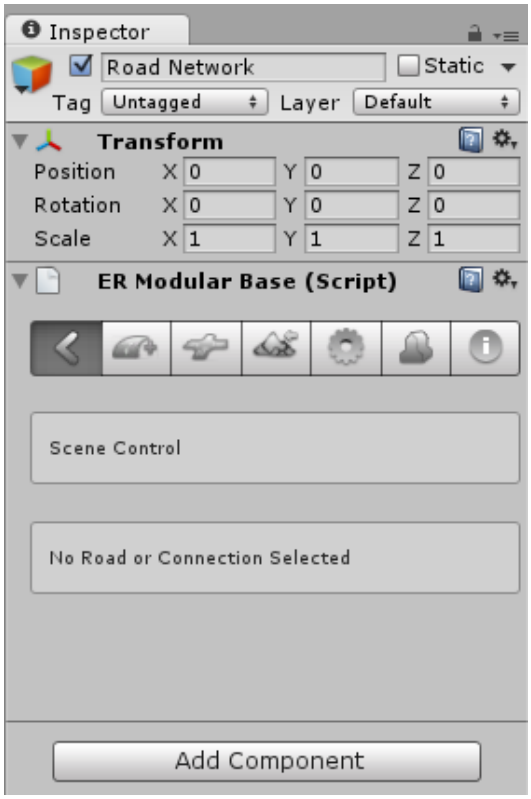


Imagen 15. Menú de la red de carreteras de EasyRoads

4.4 COLOCACIÓN DE LOS OBJETOS EN LA ESCENA

Gracias a apartados anteriores, ya dispondremos de un terreno con la calzada preparada. A continuación, pasaremos a añadir los elementos restantes: el coche principal, un contenedor de objetos con las posiciones por las que deseamos que pasen los vehículos de inteligencia artificial y un objeto vacío al que asignaremos el script InstantiateAICars.cs, que será el encargado de poner en marcha la simulación.

Además, deberemos incluir objetos vacíos para la posición de *spawn*[2] de vehículos de AI y un objeto al que se le haya asociado el script InterpWP.cs, que será el encargado de obtener el camino dibujado y trazar uno con mayor densidad de puntos, de manera que el coche se mueva de forma más natural.

[2] En la industria de los videojuegos, se denomina *spawn* o *punto de respawn* al lugar desde el que aparecerán los personajes o al que volverán una vez sean eliminados de la escena por otro jugador.

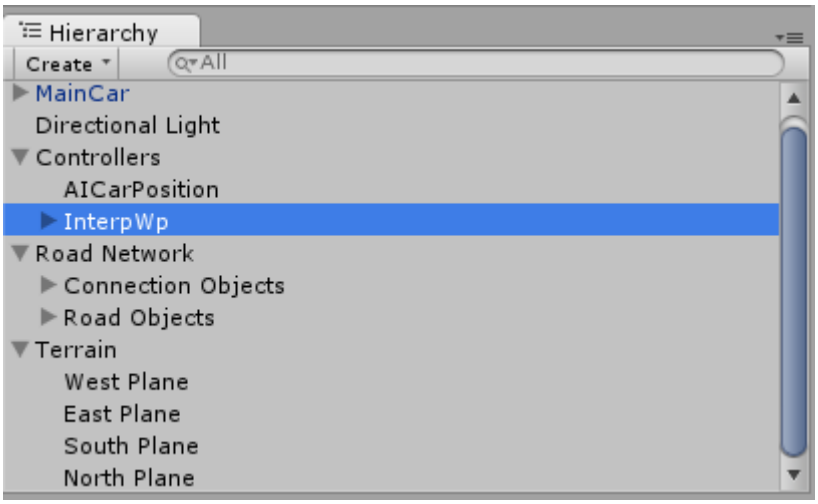


Imagen 16. Jerarquía de objetos en una escena de simulación de ejemplo

Para la creación del coche principal basta con arrastrar el vehículo desde la carpeta 'Assets/UnityCar/Samples/Prefabs/Cars' y colocarlo en el lugar desde el que deseamos empezar la simulación.

En cuanto a la creación del camino de puntos (waypoints) que seguirá, antes de añadir los puntos debemos crear un objeto vacío para que todos los puntos del camino aparezcan como hijos de dicho objeto. Una vez preparado, podemos usar cualquier objeto como referencia del punto por el que debería pasar el coche siempre que tenga componente *Transform*. Para nuestro caso, se ha decidido usar cubos 3D de forma que se pueda visualizar el recorrido.

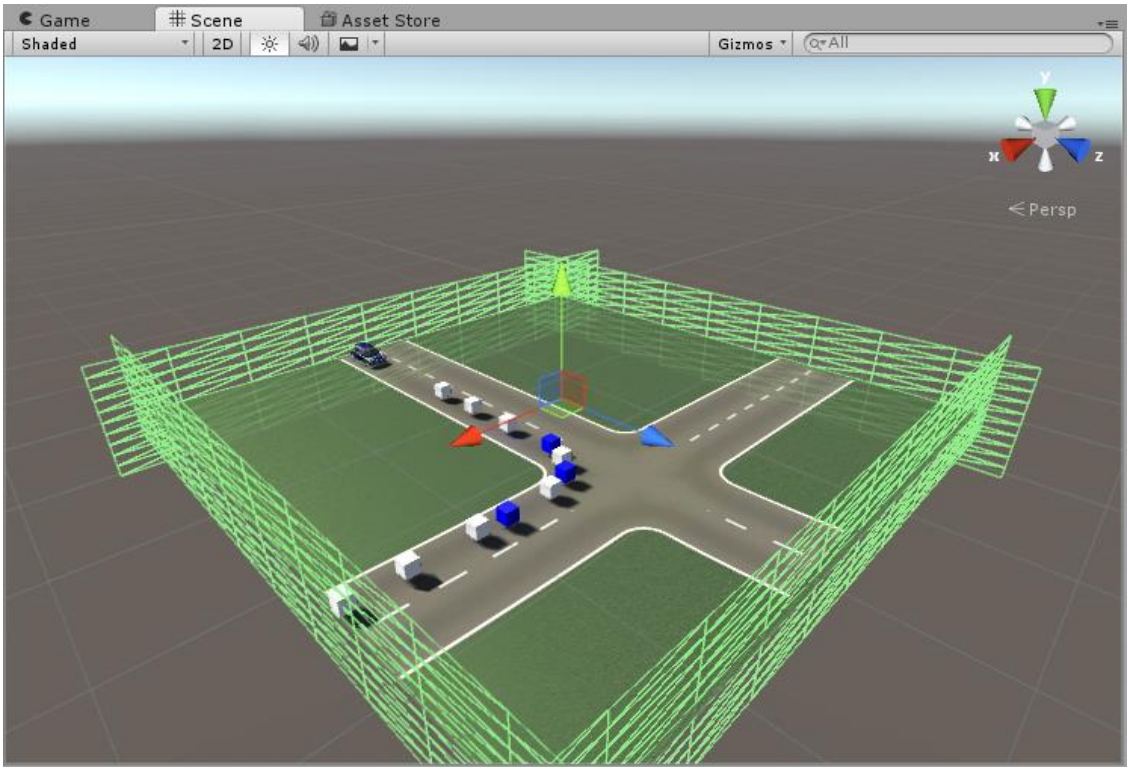


Imagen 17. Ejemplo de escena con camino de Waypoints visible

Para indicar al simulador los puntos en los que el coche de AI deberá frenar, se utilizan instancias del objeto *brakingPoint* contenido en la carpeta 'Assets/_myPrefabs'. Estos puntos son representados en la escena como cubos azules para facilitar su distinción.

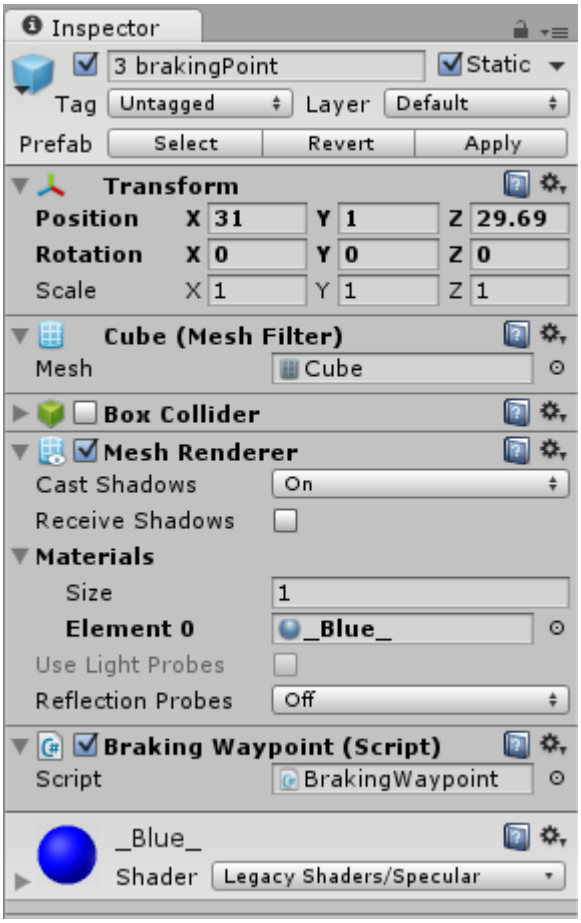


Imagen 18. Componentes de un brakingPoint

Una vez dibujado el camino completo, es importante comprobar el estado de los componentes 'Box Collider' y 'Mesh Renderer' de cada objeto. El primero deberá estar en todo momento desactivado pues de no ser así, nuestro vehículo colisionaría con los objetos como si se tratase de obstáculos en la carretera, ya sean visibles o no. El segundo podrá ser activado o desactivado al gusto, lo que provocará que los puntos sean visibles o invisibles. Para realizar el diseño es aconsejable mantenerlos visibles, pero una vez terminado lo mejor es ocultarlos para que no obstaculicen la visión del conductor una vez iniciada la simulación.

Cuando finalice la creación del camino, el objeto resultante deberá tener una forma parecida a la indicada en la siguiente imagen. El orden debe ser siempre el que aparece en la lista. Independientemente de la ubicación sobre el mapa, el programa tomará los puntos en el orden indicado dentro del objeto InterpWp.

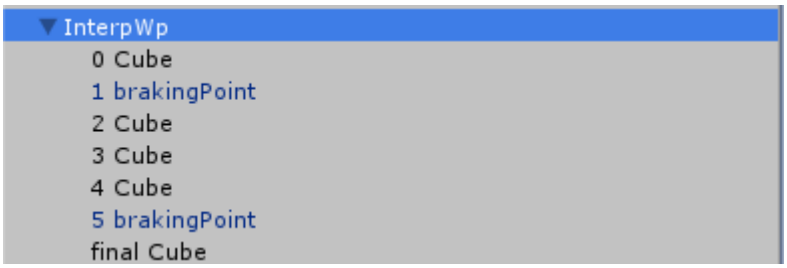


Imagen 19. Ejemplo de objetos en un WaypointContainer

Si deseamos que la ruta sea un bucle y el vehículo nunca pare, deberemos duplicar el primer punto y colocar la réplica en la última posición del waypointContainer sin moverlo ni realizar cambios sobre él. De esta forma el script podrá ver que ha vuelto al punto de partida y reiniciar el camino.

Para generar los vehículos que circularán por la ruta que acabamos de crear, necesitaremos añadir dos objetos en la escena: el punto desde el que queramos que salgan (por ejemplo, con un cubo 3D o una esfera; bastará con que tenga componente *Transform*, al igual que durante la creación de los waypoints y que sus componentes 'Box Collider' y 'Mesh Renderer' se encuentren desactivados) y el script que los genera.

Para asignar los scripts que iniciarán la simulación necesitaremos dos objetos. Con el objetivo de mantener una estructura clara y ordenada, en los escenarios de riesgo uno y dos explicados en esta memoria se ha creado un objeto vacío al que se ha llamado 'Controllers' y otro objeto como *hijo* de este al que se ha llamado InterpWp (ver Imagen 16). Al primero se le deben asignar los scripts InstantiateAICars.cs y ExternalSimClient.cs haciendo clic en 'Add Component > Scripts' y buscando los que tienen ese nombre. El segundo deberá contener InterpWp.cs, añadido de la misma forma.

En resumen, el usuario deberá encargarse, una vez dibujada la carretera, de colocar objetos en los puntos desde los que desea que aparezcan los vehículos de la AI, posicionar el coche principal desde donde se deberá iniciar la simulación y trazar a grandes rasgos el camino que se desea que sigan los AICars. Con todos estos objetos en escena, se asignarán a los scripts los objetos correspondientes y se configurarán los parámetros restantes según se desee, tal como se explica en el apartado posterior.

4.5 PARÁMETROS DE CONFIGURACIÓN

4.5.1 Configuración de parámetros de InstantiateAICars.cs

En el script InstantiateAICars.cs se deben añadir desde el editor de Unity las variables: *position*, *waypointsObject*, *prefabs* y *numberOfAICars*. Position indica el punto desde el que se generarán los vehículos, *waypointsObject* será un objeto vacío que contendrá una lista de objetos correspondientes a los puntos del camino por los que queremos que el pase el vehículo de inteligencia artificial generados por el script asociado (*InterpWP*), *prefabs* contendrá los diferentes modelos de vehículo que podrán instanciarse y de los que luego se elegirá uno de manera aleatoria.

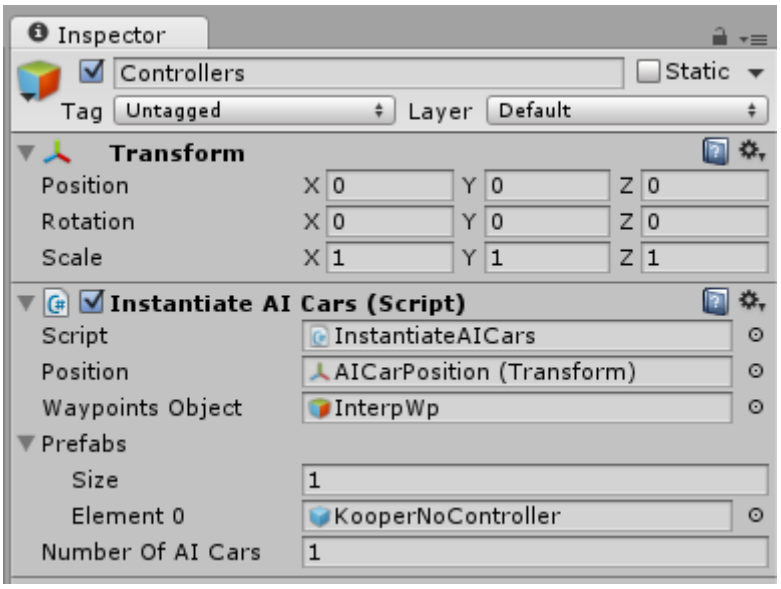


Imagen 20. Configuración del objeto Controllers

Respecto a *position* y *waypointsObject*, lo único necesario para configurar es arrastrar hasta la casilla correspondiente el objeto desde el panel *Hierarchy*. Sin embargo, para indicar cuáles son los modelos de vehículos que pueden aparecer en escena, tendremos que rellenar los datos de *prefabs.* 'Size' indicará el número de posibilidades entre las que el script elegirá el modelo y, al cambiar su valor, aparecerán debajo tantos huecos como hayamos indicado. Dichos espacios deberán rellenados con los prefabs de vehículos modificados contenidos en la carpeta 'Assets/_myPrefabs' para que no interactúen con el resto de controladores.

Por último, *numberOfAICars* indicará el número de coches que deseamos que se generen desde dicho punto con dicha trayectoria, para poder crear un tráfico fluido.

4.5.2 Configuración de ExternalSimClient.cs

Será colocado en la escena y tendrá como variable de entrada un String de caracteres con la dirección IP de la máquina donde se está ejecutando el simulador OMNeT++.

En este caso, la configuración es mucho más sencilla. Simplemente tendremos que la dirección en el hueco correspondiente. Por defecto se encuentra preparada para 'localhost', en caso de que OMNeT se ejecute en el mismo PC que Unity. En este caso, dado que se hace en una máquina Linux independiente, se debe configurar la dirección IPv4 del ordenador.

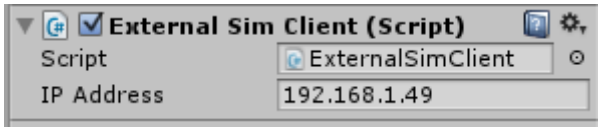


Imagen 21. Configuración de ExternalSimClient

4.6 CONFIGURACIÓN DEL PROYECTO OMNET++

En primer lugar, una vez instalado OMNeT, iniciamos el programa y nos preguntará si deseamos importar el paquete ‘inet’, que contiene todos los módulos necesarios para simular cualquier tipo de comunicación. Debemos marcar esta opción y dar a continuar.

Tras un tiempo de importación, podremos acceder a la jerarquía del proyecto creado. Para poder compilar y ejecutar el simulador, deberemos acceder a la pestaña ‘Project’ y en ‘Properties’, ir al submenú ‘C/C++ General > Paths and Symbols’. En él encontraremos las pestañas ‘Includes’ y ‘Library Paths’.

En ‘Includes’ se deberán añadir las dos entradas correspondientes a la carpeta de Flatbuffers y de Boost C++, mientras que en ‘Library Paths’ deberán incluirse el subdirectorio de librerías de Boost.

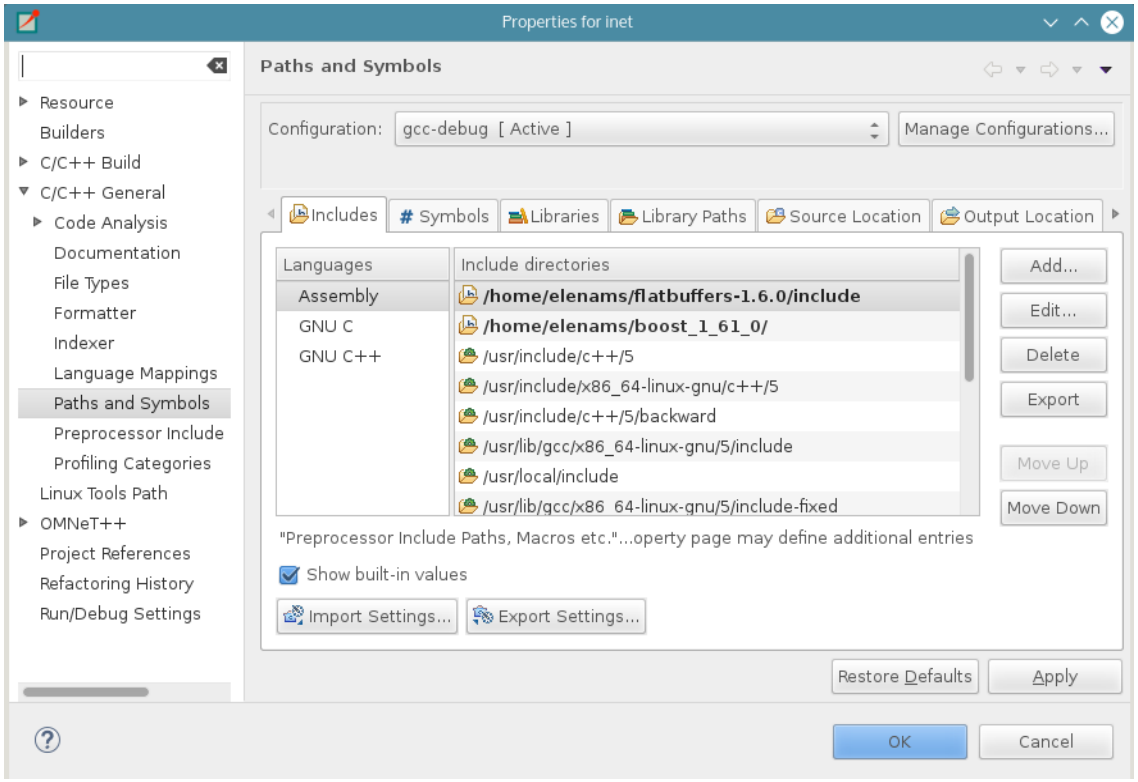


Imagen 22. Configuración de Includes en el proyecto OMNeT++

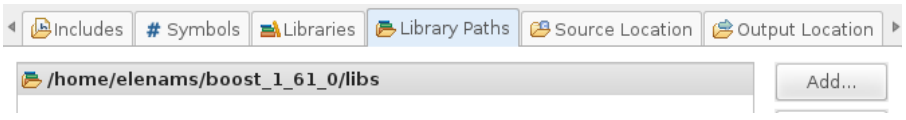


Imagen 23. Configuración de Library Paths en el proyecto OMNeT++

Una vez tenemos preparado el proyecto, se deberá crear una carpeta “veneris” bajo el directorio ‘inet/src/inet/common’, en la que situaremos todos los archivos del simulador.

Para añadir los archivos necesarios se puede probar simplemente a intentar copiar y pegar todos en la carpeta, si bien esto no suele funcionar. Otra opción es importar los archivos desde el *Wizzard* de OMNeT, aunque en ocasiones también puede acarrear

problemas. La forma más segura de incluir el código es crear los módulos con el nombre que tengan los ficheros mediante clic derecho sobre la carpeta > New > Simple Module.

Realizar esto para todos los archivos que tienen extensión '.h' y '.cpp'. Para los archivos que además contengan extensión '.ned', en lugar de SimpleModule se debe emplear Network.

Una vez creados los ficheros, simplemente hay que copiar y pegar el código de cada uno de ellos en su archivo correspondiente.

Para finalizar, se debe incluir el archivo de red que simula el servidor en el proyecto. Para ello, en 'inet/examples' añadimos una carpeta llamada 'veneris' y creamos una nueva red de la forma indicada anteriormente. Copiamos el código de los dos archivos de test y ya tendremos OMNeT preparado para ejecutar.

5 RESULTADOS

En la ejecución de la aplicación generada durante el desarrollo de este proyecto, se pueden observar los siguientes resultados si todo ha transcurrido con normalidad.

5.1 OMNET

Una vez hemos importado correctamente todo el proyecto, únicamente hay que ejecutar la red de test que se ha incluido bajo el directorio 'examples/veneris' haciendo clic en *run*. Aparecerá una ventana como la que se muestra a continuación.

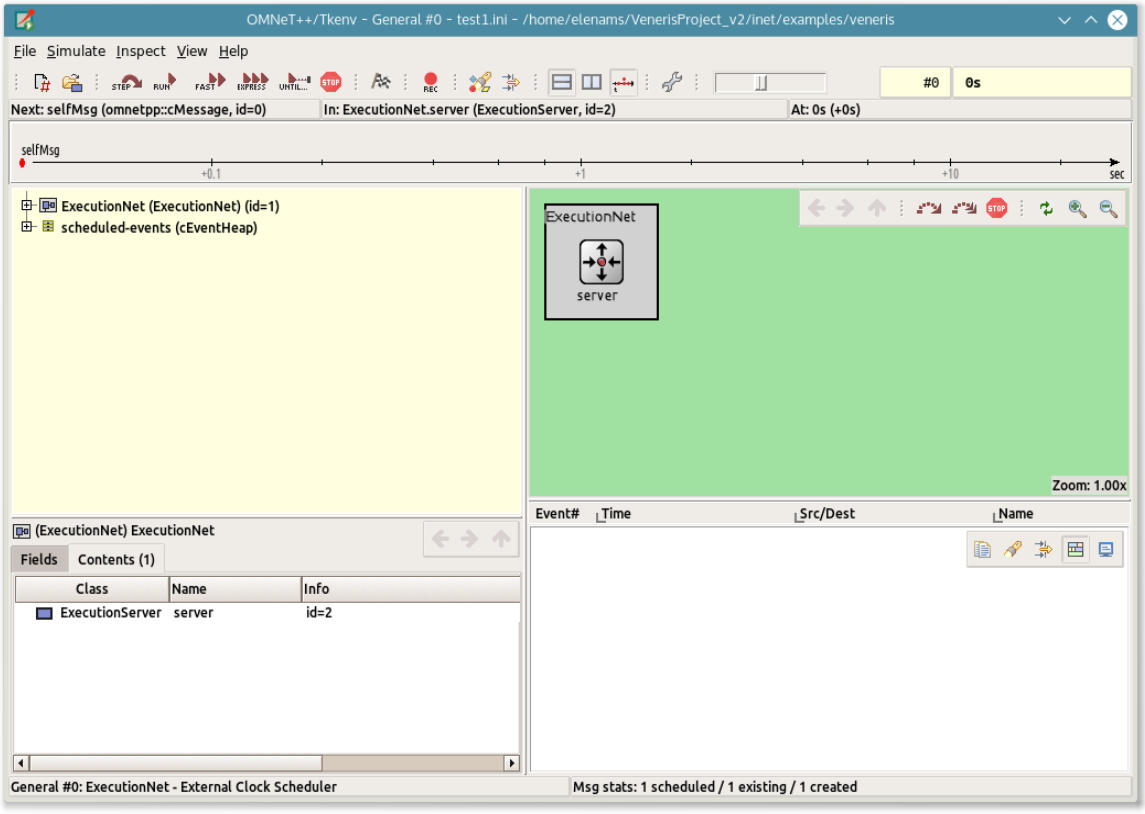


Imagen 24. Ventana de simulación en OMNeT++

Cuando estemos preparados para lanzar la ejecución de Unity, damos al botón de Start de la parte superior y el sistema quedará a la espera de la conexión entrante. Parecerá bloqueado, aunque simplemente está a escuchando nuevas peticiones de conexión, por lo que no se aprecia ningún cambio en esta ventana.

5.2 UNITY

Una vez OMNeT está preparado para escuchar los mensajes, ejecutamos el proyecto de Unity y cargamos una escena.

Deberá entonces iniciarse y obtener una vista como la mostrada en la siguiente imagen.

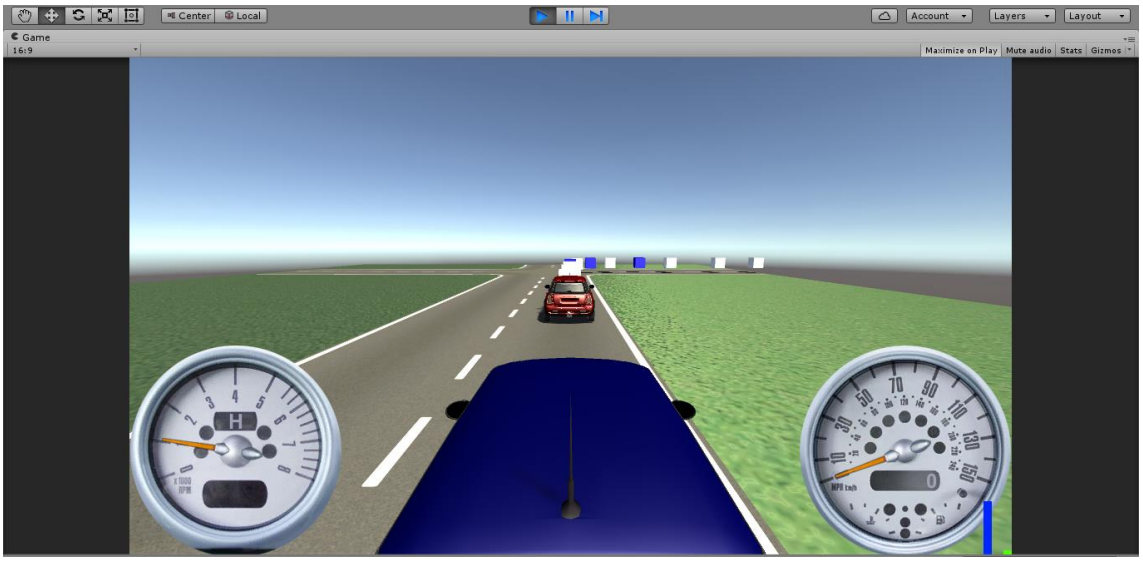


Imagen 25. Resultado de la ejecución en Unity.

A partir de este punto, Unity se encargará de ir enviando los mensajes oportunos y de simular la física de la carretera y los vehículos, tanto de tráfico como el del usuario, que podremos conducir. Mientras que en OMNeT++ podremos ver los mensajes que van llegando en la consola.

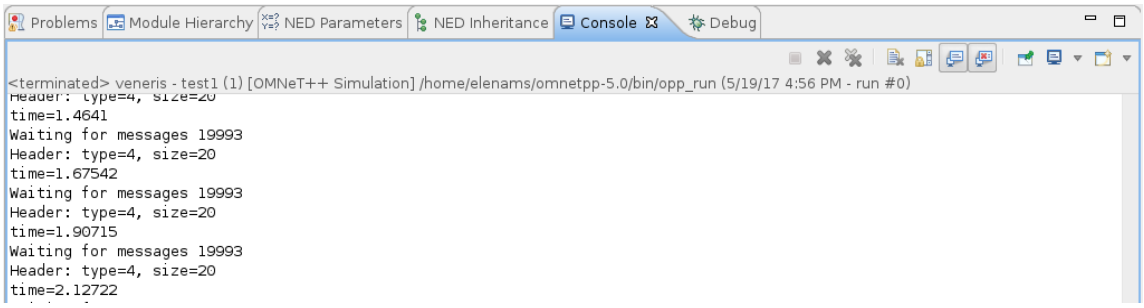


Imagen 26. Consola de OMNeT++ con mensajes de la simulación.

6 CONCLUSIONES Y LÍNEAS FUTURAS

El objetivo de este proyecto es la realización de un simulador para situaciones de riesgo en carretera que pueda ser utilizado con fines educativos mediante la integración de dos simuladores, uno de redes y otro de videojuegos. Además, se desarrolla un prototipo de aplicación del simulador, con dos de los escenarios de riesgo de accidente más comunes.

En cuanto a posibles líneas futuras y continuaciones, este proyecto tiene innumerables derivados.

En primera instancia, una posible mejora sería realizar un módulo de simulación de física de coches realista como UnityCar propietario, de forma que no fuera necesaria la licencia y el código esté más adaptado a lo que realmente necesita un proyecto de estas características. De igual forma, generar el equivalente a EasyRoads también sería útil, dado que es ligeramente complicado de aprender a utilizar.

La vertiente más importante a continuar sería, sin embargo, la ampliación del servidor de simulación Veneris para que trabaje con más tipos de mensajes y la implementación de nuevas librerías para generar y manejar datos estadísticos que ayuden a la comprensión de los resultados del simulador.

7 REFERENCIAS Y BIBLIOGRAFÍA

[1] J. W. Murray, C# Game Programming Cookbook for Unity 3D, CRC Press, 2014.

[2] J. Hocking, Unity in action, Manning Publications Co, 2015.

[3] Unity Spain, *Empezando en Unity3D.*

[4] A. Molinero Martinez, E. Carter, C. Naing, M. Simon y T. Hermitte, «Accident causation and pre-accidental driving situations,» 2008.

[5] «Flatbuffers for Unity,» Septiembre 2015. [En línea]. Available: http://exiin.com/blog/flatbuffers-for-unity-sample-code/.

[6] «How should I decide if I should use C#, JavaScript or Boo for my project?,» 2009. [En línea]. Available: answers.unity3d.com/questions/7528/how-should-i-decide-if-i-should-use-c-javascript-u.html.

[7] «UnityCar Wiki,» 2015. [En línea]. Available: http://unitypackages.net/unitycar/wiki/index.php/Main_Page.

[8] FPL, «FlatBuffers Documentation,» Agosto 2016. [En línea]. Available: https://google.github.io/flatbuffers/index.html.

[9] Unity Technologies, «Unity Documentation,» 2016. [En línea]. Available: https://docs.unity3d.com/Manual/index.html.

[10] «Omnet++ User Guide Documentation,» 2016. [En línea]. Available: https://omnetpp.org/doc/omnetpp/UserGuide.pdf.

[11] «Comparativa de motores gráficos para el desarrollo de videojuegos,» 2012. [En línea]. Available: http://www.vidaextra.com/listas/4-motores-graficos-para-perder-el-miedo-y-lanzarse-al-desarrollo-de-videojuegos.

[12] N. Gómez Bueno, «Simulador de redes vehiculares mediante integración de motores de videojuegos y simulador de redes,» 2015.

[13] A. Fernández Arroyo, «Implementación de situaciones de riesgo en un simulador de accidentes de tráfico mediante morot de videojuegos Unity,» 2015.