



UCA

Pontificia Universidad Católica Argentina

Facultad de Ingeniería y Ciencias Agrarias

Ingeniería en informática

Trabajo final

Project R: Videojuego basado en Unreal Engine 4

23 de junio de 2021

Alumno: Alberto Mikulan

Tutor: Ricardo Di Pasquale

Resumen

Se realizará un videojuego de carreras futurista en 3D sobre el motor de juegos Unreal Engine 4.25. Dicho videojuego tendrá soporte para multijugador (local y en línea). Se dará un entendimiento razonable de la

arquitectura del motor que sirva para la comprensión del desarrollo del videojuego en cuestión. Se mostrarán los pasos durante el desarrollo, problemas encontrados y soluciones utilizadas.

Índice de Contenidos

Resumen	2
Índice de Contenidos	3
Motivación	6
Introducción	7
Estado del Arte	8
Objetivos	9
Unreal Engine 4	10
Introducción	10
Blueprints	11
Terminología	12
Gameplay Framework	16
Ciclo de Vida de Actores	17
Desarrollo del Trabajo	21
Configuración	21
Test Driven Development	21
Continuous Integration	22
Jenkins	22
Ngrok	23
Cobertura de Código	26
Sistema de Automatización de UE4	27
Pruebas Automáticas	28
Notación	31
Módulos de Pruebas	31
Organización del Código	31
Clases del Juego y Lógica	33
UProjectRGameInstance	34
ARaceGameMode	34
AProjectRGameState	34
ALobbyGameState	34
ALapManager	34
ALapPhase y Clases Derivadas	35
AProjectRPlayerController	35

ARacePlayerState	35
USessionManager	36
ATrackManager	36
ATrackGenerator	36
Interfaces de Usuario	37
AJet	42
Replicación	50
Client-Side Prediction	50
Server Reconciliation	51
UDeloreanReplicationMachine	52
Problemas y Soluciones por Pruebas (y problemas de pruebas)	56
Ticks, Mundos y Fuerzas	56
Clics en Editor	61
Pulsación de Botones	63
Pruebas de Replicación	64
Utilización del Entorno	66
Limitación del cuadro de Imagen	66
Análisis de GPU	67
Análisis de CPU	70
Análisis de la Red	71
Arquitectura	74
1 - Contexto del Proyecto	75
1.1 - Unreal Engine 4	75
1.1.1 - Arquitectura basada en Eventos	75
1.1.2 - Sistema de Propiedades	76
1.2 - Diseño	78
2 - Requerimientos de Arquitectura	79
2.1 - Descripción breve de los Objetivos Clave	79
2.2 - Casos de Uso de Arquitectura	79
2.3 - Requerimientos de Arquitectura de los Actores	80
2.4 - Restricciones	81
2.5 - Requerimientos No Funcionales	81
2.6 - Riesgos	81
3 - Solución	83
3.1 - Patrones Arquitectónicos Relevantes	83
3.1.1 - Patrones de diseño	83

3.1.2 - Patrones arquitectónicos	85
3.2 - Breve Descripción de la Arquitectura	86
AMotorStateManager y UMotorStates (State)	86
AJet y Componentes (Composition Over Inheritance)	87
ALapManager y ALapPhases (Eventos)	89
ARacePlayerState y AProjectRGameState (Servidor-Cliente)	90
3.3 - Vistas Estructurales	92
3.3.1 - Vista Lógica	92
3.3.2 - Vista de Despliegue	93
3.4 - Vistas de Comportamiento	95
3.4.1 - Secuencia de Aceleración en AJet	95
3.4.2 - Secuencia de Actualización de Posiciones	97
3.4.3 Secuencia de actualización de vueltas	98
3.5 - Temas de Implementación	100
4 - Análisis de la Arquitectura	101
4.1 - Análisis de Escenarios	101
Conclusiones y Aportes al Tema	102
Temas Abiertos	104
Referencias Bibliográficas	105

Motivación

El autor ha elegido este tipo de desarrollo particular ya que es su deseo dedicarse profesionalmente en esta área de la informática, por lo que considera oportuno poder cerrar el ciclo de carácter formativo de la universidad dando al mismo tiempo un paso hacia adelante en su carrera profesional futura. Se eligió especialmente el uso del motor Unreal Engine por su uso aceptado en la industria de videojuegos desde pequeños emprendedores a grandes empresas del sector.

Introducción

Se presentará el desarrollo de un videojuego que utiliza el motor Unreal Engine 4 como base. Para ello, primero se dará una breve situación actual de la industria para establecer un contexto global.

A continuación, se presentarán los objetivos del trabajo final, dando a conocer así la meta del proyecto.

Luego se profundizará en el desarrollo del proyecto, donde:

Primero se dará una introducción al motor Unreal Engine, su terminología y clases comunes (como UObject, Actor, Pawn, PlayerController, etc.).

Más adelante se mostrará la configuración utilizada, que incluye el entorno de integración continua y desarrollo dirigido por pruebas. También se dará un vistazo a las herramientas adoptadas para lograr los entornos mencionados: Jenkins, ngrok, OpenCppCoverage y la plataforma de repositorios altamente conocida, GitHub.

También se mostrará cómo es el sistema de automatización de Unreal Engine 4, los tipos de pruebas que soporta y una explicación de las pruebas simples y complejas que vamos a utilizar.

Más adelante se muestran las distintas clases de objetos creados para el proyecto, cómo se abordó la replicación (envío de copias de objetos mediante la red), problemas y soluciones mediante pruebas automatizadas y uso de recursos por el proyecto.

Cabe mencionar que se tiene a continuación de lo anterior al informe de arquitectura del proyecto, que tiene un enfoque más técnico en el desarrollo del proyecto.

En las últimas secciones se dará una conclusión al proyecto, que incluye las lecciones aprendidas durante la realización del proyecto y por último, se tendrán los temas abiertos, donde se notan cuestiones no abarcadas por el proyecto y de interés para el futuro de desarrollo en videojuegos.

Se dejará en esta sección el enlace a la carpeta que contiene este y otros documentos, formularios, programa y código para que sea posible su acceso y uso.

Estado del Arte

En Argentina se encuentran pocas instituciones que ofrecen educación en el ámbito de los videojuegos y por lo general son ofertas para estudios terciarios. En el mundo ya se encuentra recientemente instaurado como una carrera en países como Estados Unidos, Japón y Europa por lo que se encuentra más formalizado.

La mayoría de las grandes empresas utilizan sus propios motores de videojuegos, los cuales pocas veces son accesibles al público. No obstante, existen ciertas compañías que se dedican a ofrecer sus motores a cambio de un porcentaje de las ventas que se hagan por los juegos que se basen en sus motores (si es que se venden).

Existen dos motores muy conocidos que se basan en este esquema: Unity y Unreal Engine. Solo Unreal Engine permite acceder a todo su código (Unity solo lo permite para su parte en C#) una vez que uno se hace miembro de la plataforma sin costo alguno (lo cual es sencillo), por lo que es posible realizar un análisis exhaustivo del funcionamiento y diseño del motor, lo que produce un mayor entendimiento de la arquitectura, facilita la resolución de conflictos que puedan surgir en el desarrollo, así como también la posibilidad de personalizar el motor a las necesidades de cada desarrollo. Esto último debe realizarse con mucho cuidado a fin de evitar la incompatibilidad con futuras versiones de los motores.

Unity utiliza C# como lenguaje base de desarrollo, mientras que Unreal Engine utiliza C++ y un lenguaje de scripting visual propio (Blueprints).

Se decidió utilizar Unreal Engine 4 por ser C++ un lenguaje estudiado en la carrera, poder analizar su funcionamiento y ser, el motor, más estable que Unity.

Unreal Engine 4 es un motor de videojuegos altamente aceptado en la industria de los videojuegos. Grandes empresas con pie en el sector de videojuegos los han realizado sobre este motor (como Microsoft, Warner Bros. Interactive Entertainment, Sony Interactive Entertainment, Nintendo, Bandai Namco Entertainment, Square Enix, Capcom, etc.). Actualmente se usa en las consolas de videojuegos de octava generación (PlayStation 4, Xbox One, Nintendo Switch) y en PC. Cuenta además con soporte para iOS, Android y realidad virtual.

A partir de 2020 incluyó soporte para las consolas de novena generación (PlayStation 5, Xbox Series X) que se lanzarán en noviembre del mismo año. Se tiene planeado el lanzamiento del motor Unreal Engine 5 para fines de 2021, lo que traerá consigo gran cantidad de tecnologías novedosas. Esto no quita que Unreal Engine 4 deje de ser soportado, sino lo contrario ya que la introducción de nuevos motores en la industria tiene un período de asimilación de unos años.

Unreal Engine 4 ha sido utilizado en títulos de gran éxito como Fortnite (creado por Epic Games, la misma empresa que construye el motor), Dragon Ball FighterZ, Final Fantasy VII Remake, Gears 5, Street Fighter V, Yoshi's Crafted World, The Outer Worlds, Sea Of Thieves, y Borderlands 3 entre otros^{1 2}.

Objetivos

- Crear un videojuego de carreras futurista en 3D con soporte multijugador local y en línea.
- Preparar el sistema para convertirlo en un producto comercial luego de ser presentado como trabajo final.
- Desarrollar el sistema con metodologías y herramientas adecuadas y aceptadas.
- Proporcionar un buen entendimiento sobre el framework Unreal Engine 4 (utilizado como base para el juego).
- Prestar atención al uso de la red en el modo en línea y los recursos utilizados por el sistema.

Unreal Engine 4

Introducción

Unreal Engine 4 es una plataforma de creación de 3D en tiempo real³. Su propósito original es la de ser un motor de videojuegos⁴. Esto significa que es una plataforma que ofrece las funciones básicas (y avanzadas en el caso de este motor) necesarias para construir un videojuego, lo que implica tener módulos de renderización de video, soporte de modelos en 3D, sistema de audio y físicas, sistema de navegación y de inteligencia artificial, soporte de entradas (teclado, mouse y controles), sistema de iluminación, sistema de multijugador, simulación de juego, etc.⁵.

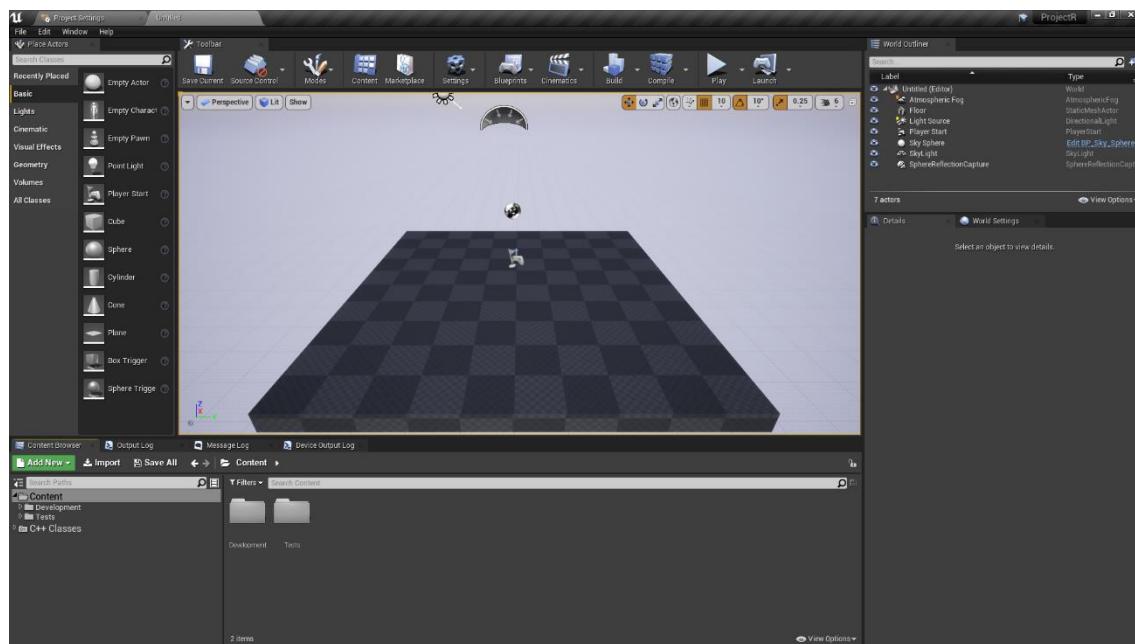


Ilustración 1. Editor de Unreal Engine 4.

Una de las mayores ventajas es que todo el código del motor se encuentra a disposición del cliente una vez que se registra en la plataforma, lo cual es gratis⁶.

El motor permanece gratis hasta que las ventas en el juego que lo use lleguen a un millón de dólares estadounidenses. Luego, se cobra un 5% de cada venta que se realice.

El motor cuenta con un sistema de reflexión propio que le permite tener información del código en C++ (cosa que C++ no tiene por defecto al ser fuertemente tipificado, las clases deben conocerse en tiempo de compilación). También permite tener recolección de basura (garbage collection), serialización y replicación en red⁷.

Blueprints

El sistema visual guionizado de Blueprints⁸ (Blueprints Visual Scripting) en Unreal Engine es un sistema completo que posee el concepto de usar interfaces basadas en nodos para crear elementos de jugabilidad dentro del editor.

Como muchos lenguajes guionizados, es usado para definir (dentro del motor) clases u objetos del paradigma de objetos. Estos objetos o clases se los llama simplemente Blueprints.

Este sistema es altamente flexible y poderoso ya que provee a los diseñadores la habilidad de usar virtualmente todo el rango de conceptos y herramientas generalmente solo disponibles para programadores. Adicionalmente, existen propiedades de marcado de código en C++ del motor que permite a los programadores crear sistemas base para luego ser usados y extendidos por diseñadores.

Los Blueprints funcionan como adiciones secuenciadas al juego. Al conectar Nodos, Eventos, Funciones y Variables con Cables, es posible generar elementos complejos de jugabilidad. Es un sistema de grafos con nodos multipropósito, como construcción de objetos, funciones, eventos, que son usados poder implementar comportamiento y otras funcionalidades específicamente para cada instancia del Blueprint⁹.

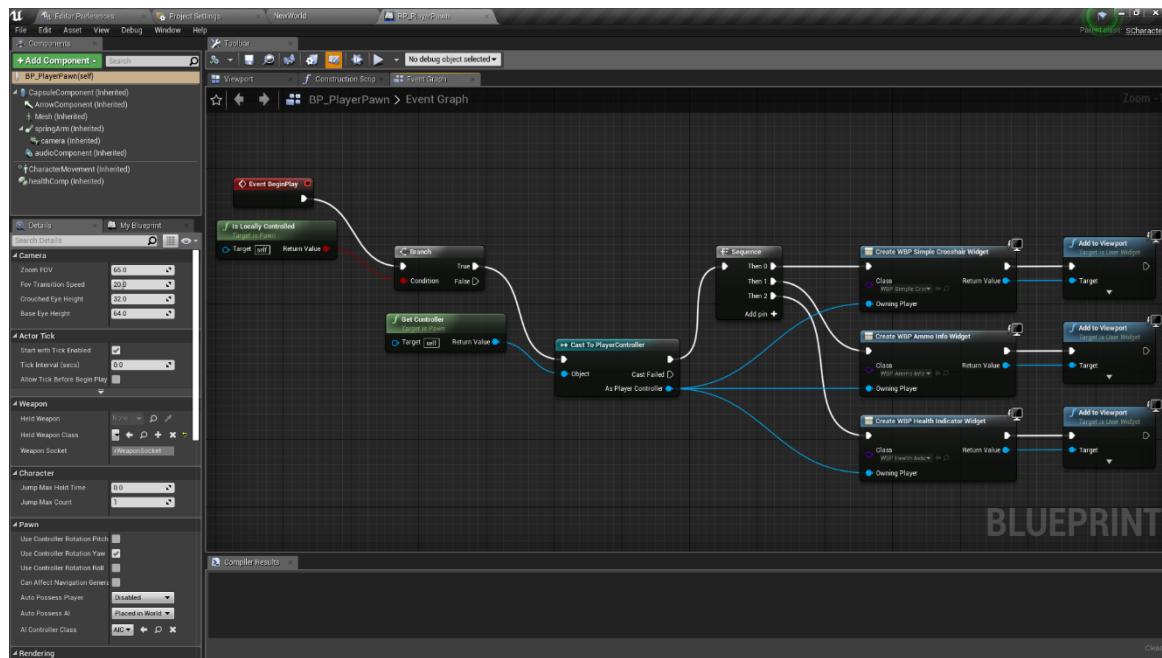


Ilustración 2. Vista general de Un Blueprint. En este caso este Blueprint toma el control del peón y crea una interfaz de usuario (HUD) para el jugador.

Terminología

UObject

es la clase base de todos los objetos. Implementa características como recolección de basura, soporte de metadatos para exponer variables al editor (con UProperty) y serialización para carga y guardado^{10 11}.

Actores

Un actor (Actor¹²) es cualquier objeto que puede ser colocado en un nivel. Los actores son clases generales que soportan transformaciones en 3D, como traslación, rotación y escala.

Pueden ser creados (aparecer en niveles) y destruidos por medio de código de juego (C++ o Blueprints).

En C++, AActor¹³ es la clase base de todos los actores y existen varios tipos de ellos, como StaticMeshActor, CameraActor, PlayerStartActor, etc.

Componentes

Un componente (Component^{14 15 16}) es una pieza funcional que puede añadirse a un actor. Los componentes no pueden existir por sí solos, por lo que solo se accede a su funcionalidad un Actor cuando se agrega ese componente.

Por ejemplo, un Rotating Movement Component hace que un actor pueda girar por sí mismo, un Audio Component hace que un actor sea capaz de emitir sonidos, y así sucesivamente.

En C++ la clase es UActorComponent¹⁷.

Peones

Un peón (Pawn^{18 19}) es una subclase de un actor que sirve como un personaje o avatar en el juego (por ejemplo, personajes o vehículos en un juego). Los peones son controlables por jugadores o por inteligencia artificial, por medio de NPCs (Non-Player Characters) o personajes no controlados por jugadores en inglés.

Cuando un peón es controlado por un jugador humano o inteligencia artificial se lo considera poseído. En cambio, cuando no está controlado se le considera desposeído/despajado.

En C++ la clase es APawn²⁰.

Personajes

Un personaje (Character^{21 22 23}) es una subclase de un Peón que tiene como objetivo el ser usado como un personaje humano. Incluye por defecto una configuración de colisiones, entradas asociadas a movimiento bípedo y código adicional para movimiento controlado por un jugador.

En C++ la clase es ACharacter²⁴.

Controles humanos

El control humano (PlayerController²⁵) es una clase usada para tomar entradas de un jugador (presión de un botón, movimiento de un stick, acelerómetros, toques, etc.) y transformarlas en interacciones en el juego.

Frecuentemente un control humano posee peones o personajes como representación del jugador dentro del juego. De esta forma, toda voluntad de un jugador está representada por un control humano.

Cada juego tiene por lo menos un control humano.

Un control humano también es el punto de interacción de red primario en juegos multijugador. Durante un juego multijugador, el servidor tiene una instancia de un control humano por cada jugador en juego, ya que tiene que ser capaz de realizar llamadas de función a través de la red a cada jugador. Cada cliente tiene solamente un control humano que corresponde a su jugador y solamente puede usar su control para comunicarse con el servidor.

En C++ la clase es APlayerController²⁶.

Control de Inteligencia Artificial

Así como el control humano representa la voluntad de un jugador, el control de inteligencia artificial (AIController²⁷) posee un peón representando un NPC en el juego.

Por defecto, peones y personajes tienen un control de inteligencia artificial poseyéndolos, a menos que sean poseídos específicamente por un control humano o se les diga que no creen un control de inteligencia artificial para ellos.

En C++ la clase es AAIController²⁸.

Pinceladas

Una pincelada (Brush^{29 30}) es un actor que describe un volumen 3D que se encuentra en un nivel para poder definir geometría de nivel (referido como BSP) y volúmenes de jugabilidad.

Típicamente se usan pinceladas BSP para prototipar o bloquear niveles para probar la jugabilidad del juego.

Los volúmenes en cambio tienen varios usos dependiendo de los efectos asociados a ellos, como volúmenes bloqueantes (que son invisibles, pero sirven para que un actor no los atraviese), volúmenes de dolor (que causan daño al Actor cuando uno solapa con ellos), volúmenes activables (que se utilizan como una forma para activar eventos cuando un actor entra o sale de ellos), etc.

En C++ existen varias clases para pinceladas: ABrush³¹, FSlateBrush³² y UBrushComponent³³.

Niveles

Un nivel (Level^{34 35 36}) es un área definida para jugabilidad. Los niveles son creados, vistos y modificados principalmente ubicando, transformando y editando propiedades de actores que contienen.

En el editor, cada nivel se guarda como un archivo “.umap”, por lo que es normal que también se los conozca como mapas (Maps).

Mundo

Un mundo (World³⁷) contiene una lista de niveles que se cargan en él. Maneja la presentación de niveles y la aparición (creación) de actores.

La interacción directa con un mundo no es necesaria, pero provee ayuda en que da un punto referencial dentro de la estructura del juego.

Cada mundo tiene un modo de juego asociado a él.

En C++ la clase es UWorld³⁸.

Instancia de Juego

Una instancia de juego (GameInstance³⁹) es una administradora de alto nivel de los objetos del juego. Se crea cuando el juego es ejecutado y solo se destruye cuando se cierra el juego. Existe durante toda la ejecución del juego. En C++ su clase es UGameInstance.

Modos de Juego

Un modo de juego (GameMode⁴⁰) es responsable de establecer las reglas de juego que se está jugando.

Las reglas pueden incluir cómo se unen jugadores al juego, si un juego puede ser pausado o no, transiciones de niveles, así como también comportamiento específico de juego, como las condiciones para ganar.

Se puede establecer un modo de juego general en las opciones del proyecto, pero también se puede especificar para cada nivel. No importa cómo se quieran establecer los modos de juego, siempre hay solamente uno por cada nivel.

En un juego multijugador, el modo de juego solo existe en el servidor y las reglas son replicadas (enviadas) a cada cliente conectado.

En C++ la clase es AGameModeBase⁴¹, pero generalmente se usa una derivada, AGameMode⁴², que tiene soporte multijugador.

Estados de Juego

El estado de juego (GameState⁴³) contiene información que se quiere replicar a cada cliente del juego. Es simplemente el estado del juego para todo aquel conectado.

A menudo contiene información sobre puntajes, si una partida comenzó o no, cuántos NPC aparecer en base al número de jugadores, y otra información específica del juego.

Para juegos multijugador, solo hay una instancia de estado de juego por cliente, siendo la instancia del servidor la que controla al resto (envía información al resto de clientes).

En C++ la clase es AGameState⁴⁴.

Estados de Jugador

Un Estado de jugador (PlayerState⁴⁵) es el estado de un participante del juego, como un jugador humano o de inteligencia artificial. Solo inteligencia artificial que controla a un peón tiene un estado de jugador.

Datos que son apropiados para un estado de jugador pueden ser el nombre del jugador, su puntaje, nivel actual o vida restante, o si están llevando algún ítem perteneciente al modo de juego.

En juegos multijugador, los estados de jugador de todos los jugadores se encuentran presentes en todos los clientes y pueden replicar datos desde el servidor para mantenerse sincronizados.

En C++ la clase es APlayerState⁴⁶.

Gameplay Framework

Las clases mencionadas anteriormente son las bases funcionales para el entorno de jugabilidad⁴⁷.

El gráfico a continuación representa cómo se relacionan estas clases:

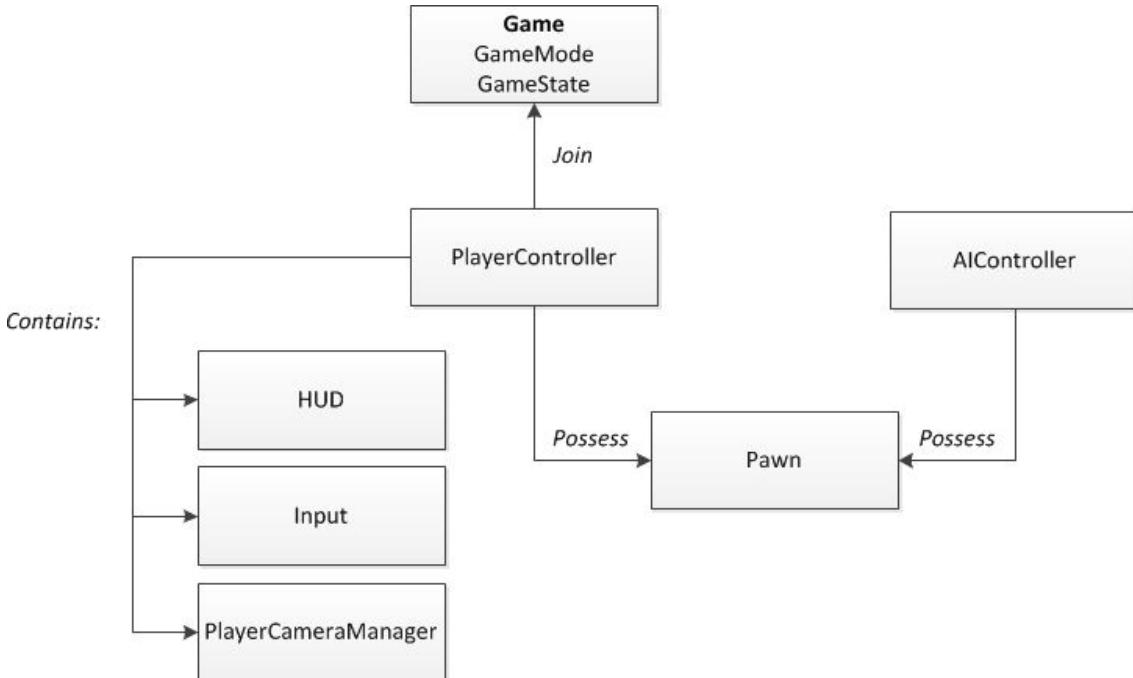


Ilustración 3. Relación entre clases de Jugabilidad.

Un juego está hecho de un modo de juego (GameMode) y estado de juego (GameState). Jugadores humanos que se unen al juego se asocian a controles humanos (PlayerController). Estos controles humanos permiten a los jugadores humanos poseer peones (Pawn) en el juego así pueden tener representaciones físicas en el nivel^{48 49}. Cada control humano tiene su propio estado del jugador (PlayerState) que es modificado por el modo de juego y leído por el control.

Los controles humanos también pueden dar a los jugadores, entre otras cosas, controles de entradas (Input), una interfaz (Heads-Up Display, HUD) y un administrador de cámaras de jugador (PlayerCameraManager) para controlar vistas de la cámara.

Ciclo de Vida de Actores

A continuación, se mostrará cómo es el ciclo de vida de los actores en un juego⁵⁰.

Creación

-Cargado desde el Disco

Esta vía ocurre cuando algún actor está actualmente en un nivel, como cuando se realiza un LoadMap o cuando se llama a AddToWorld.

1. Los actores de un nivel (o paquete) son cargados en disco.
2. PostLoad:
 - a. Se llama por medio de un actor serializado, después de que se cargaron.
 - b. Cualquier versionado personalizado y arreglos debería ir en este momento.
 - c. Es mutuamente excluyente con PostActorCreated.
3. InitializeActorsForPlay inicializa a los actores para que puedan jugar.
4. RouteActorInitialize para actores que no se inicializaron (fueron transferidos desde otro nivel).
 - a. PreInitializeComponents se llama antes de que InitializeComponent sea llamado en cada componente de actor.
 - b. InitializeComponent es un método que ayuda en la creación de cada componente definido en un actor.
 - c. PostInitializeComponents es llamado luego de que se hayan inicializado los componentes de los actores.
5. BeginPlay se hace cuando el nivel empieza el juego.

-Jugada desde el Editor

Este camino es muy parecido al de cargado desde el disco, la diferencia es que los actores nunca son cargados desde el disco, se copian desde el editor.

1. Los Actores en el editor son duplicados en un mundo nuevo.
2. Se llama a PostDuplicate.
3. Se sigue el mismo camino que cargado desde el disco, desde el punto 3.

-Aparición de Actores

Se sigue este camino cuando se hace aparecer (instanciar) un actor en un mundo/nivel.

1. Se llama a SpawnActor que inicializará y colocará un actor dentro de un mundo.
2. Después se utiliza PostSpawnInitialize.
3. Luego, PostActorCreated es llamado cuando se terminan de crear los actores. El comportamiento que posee es de tipo constructor, por lo que ese tipo de comportamiento debería especificarse en este lugar. Es mutuamente excluyente con PostLoad.

4. Comienza ExecuteConstruction:

- a. OnConstruction es la construcción del actor propiamente dicha. En este momento es donde se crean los componentes de actores que provienen de Blueprints y se inicializan las variables contenidas en Blueprints.

5. Se utiliza PostActorConstruction:

- a. Se realizan los pasos a, b y c dentro del cuarto punto de cargado desde el disco. (PreInitializeComponents, InitializeComponent y PostInitializeComponents).

6. Se hace un broadcast de OnActorSpawned en el mundo (UWorld).

7. Se llama a BeginPlay.

-Aparición de Actores (Diferido)

Un actor tiene su aparición diferida cuando alguna de sus propiedades se establece como “Expose on Spawn”.

1. En vez de usar SpawnActor, se utiliza SpawnActorDeferred, que sirve para aparecer actores creados con procedimientos. Permite una configuración adicional antes de que se construya el Blueprint.
2. Se trabaja igual que en SpawnActor, pero luego de que se llame a PostActorCreated ocurre lo siguiente:
 - a. Se hace la configuración/llamado de varios “métodos de inicialización” con una instancia de actor válida, pero incompleta.
 - b. Se llama a FinishSpawningActor, que es utilizada para finalizar al actor.
3. Continúa con ExecuteConstruction dentro del camino de aparecer actores.

Fin de la Vida

Los actores pueden ser destruidos de diferentes formas, pero la manera en que se termina su existencia es siempre la misma, la de recolección de basura.

-Durante el Juego

Estas formas son completamente opcionales, ya que muchos actores no van a morir durante el juego.

- Destroy se llama manualmente por un juego en cualquier momento que se desee remover un actor, pero el juego debe continuar. El actor se marca como pendiente para matar y se remueve del arreglo de actores que tenga el nivel.
- EndPlay se llama en varios lugares para garantizar que la vida de un actor está por terminarse.

Las llamadas que ocasionan un EndPlay son las siguientes:

- Llamada explícita a Destroy.
- La simulación desde el editor se termina.
- Transición de nivel (cargado de un mapa o una transición a otro).
- Se remueve un nivel presentado que contiene a un actor.

- Si la vida de un actor expiró.
- La aplicación se cierra.

Sin importar la manera en cómo sea llamado, el actor se marcará como RF_PendingKill por lo que en el próximo ciclo de la recolección de basura (garbage collection) será sacado de memoria.

- OnDestroy es una respuesta heredada (legacy) de Destroy. Todo lo que se encuentra en este lugar debería ser movido a EndPlay.

-Recolección de Basura

O Garbage Collection. Se realiza un tiempo después de que un actor se marcó para eliminarse. La recolección va a remover al actor de la memoria, liberando así a los recursos que estaba utilizando.

Los métodos llamados en la destrucción de un objeto son los siguientes:

1. BeginDestroy es llamado para darle la oportunidad al objeto de liberar la memoria y manejar otros recursos que se encuentran en ejecución en paralelo. La mayoría de la funcionalidad del juego relacionada a ser destruido debería haber sido manejada antes por EndPlay.
2. IsReadyForFinishDestroy será procesada por la recolección de basura para determinar si el objeto está listo para removese de la memoria permanentemente. Cuando retorne false, este método puede posponer la destrucción del objeto hasta el próximo ciclo de recolección de basura.
3. FinishDestroy será llamado finalmente cuando el objeto sea verdaderamente destruido y es otra chance para liberar estructuras de datos internas del objeto. Es la última llamada realizada antes de que la memoria del objeto se libere.

En la siguiente página se muestra un gráfico del ciclo de vida de un actor, desde los distintos caminos de creación de un objeto hasta el fin de su vida.

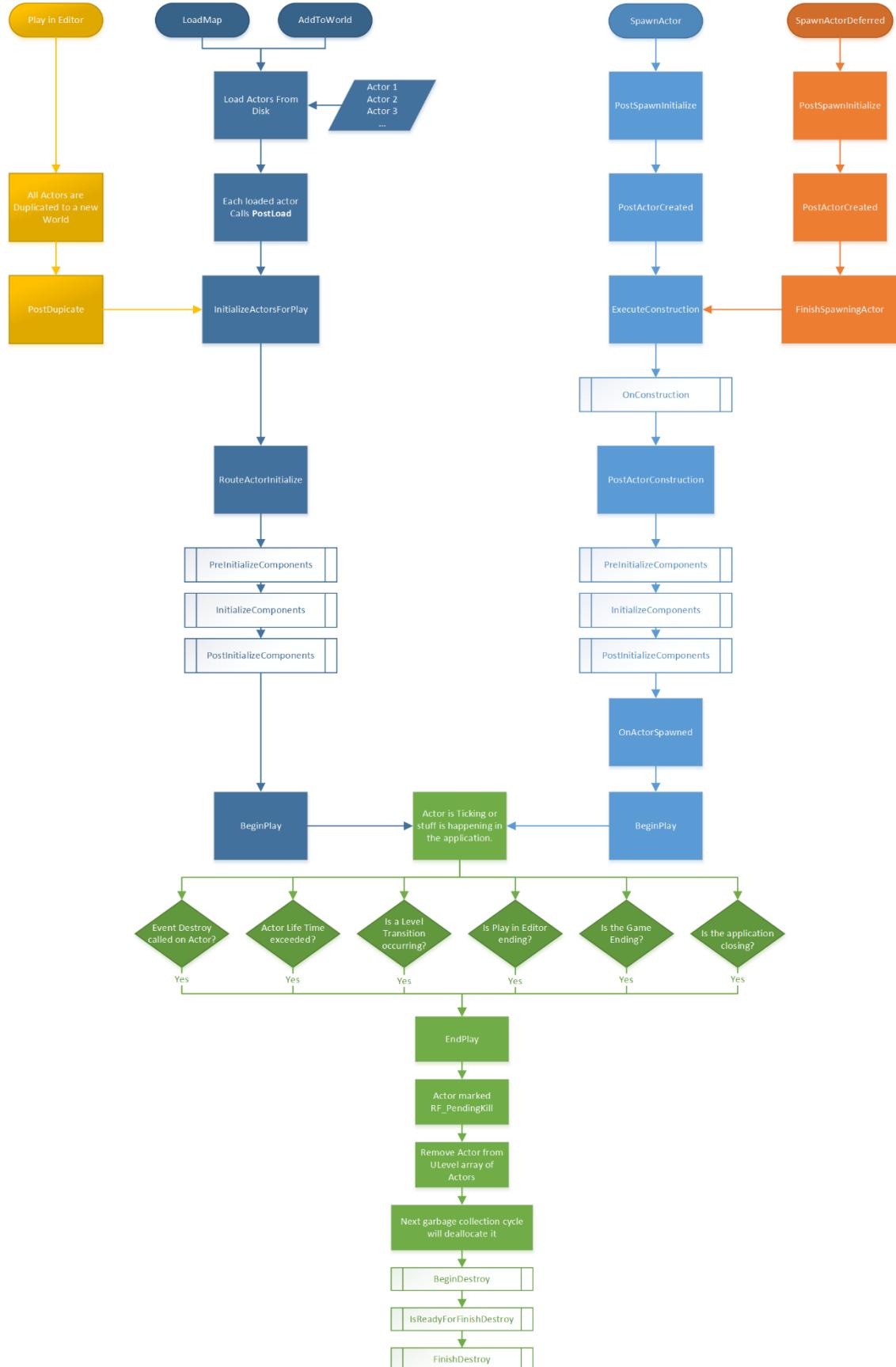


Ilustración 4. Ciclo de vida de un actor.

Desarrollo del Trabajo

Configuración

El proyecto se basa en un enfoque de pruebas (Test First) y código autoprovable (self testing code), donde se tiene un enfoque en aceptar los cambios solamente cuando se ejecutan las pruebas con éxito y donde las pruebas son lo primero que se desarrolla, con lo que se construye cada una antes de que exista la implementación. Para esto, se utilizan dos metodologías recomendadas y eficaces: Test Driven Development y Continuous Integration.

Test Driven Development

O desarrollo dirigido por pruebas⁵¹, se basa en los siguientes pasos y está enfocado en la programación:

1. Escribir la prueba para probar una sola característica (antes de que exista).
2. Correr la prueba, que fallará porque la característica no está implementada.
3. Implementar la característica.
4. Correr la prueba, que esta vez será exitosa. Si no lo es, volver al paso 3.
5. Mejorar la implementación y volver al paso 4.
6. Una vez que se tiene una implementación que cumpla la prueba y sea aceptada, repetir los pasos para la próxima característica.

Se quiere ir realizando pruebas para las partes más pequeñas del programa. De esta forma, las nuevas funcionalidades que se basen en ellas deberán probar solamente lo que ellas deben cumplir, porque su base ya está probada.

Otro beneficio es que se enfoca solamente en cumplir la prueba, por lo que se ahorra el escribir código innecesario y se escribe lo mínimo para ejecutar la prueba con éxito.

Se podría objetar que tener que escribir pruebas para cada funcionalidad hace que el código sea mucho mayor y se pierda tiempo en desarrollo de pruebas cuando se podría utilizar para realizar más trabajo.

Es cierto que el código sea mayor, pero el gran beneficio de esta técnica es que se tiene certeza en que lo que se hace funciona y los errores que surgen pueden exponerse con más pruebas y ser detectados muy fácilmente al momento que falle una o varias de ellas, por lo que a largo plazo reduce el trabajo necesario en arreglar errores y el costo de mantenimiento. Además, nuevas funcionalidades deben seguir el mismo proceso, por lo que todo lo que se ingrese al programa se está seguro de que funciona como se espera.

Es una técnica que requiere un cierto tiempo para acostumbrarse a pensar de esa forma, pero que a largo plazo incrementa la confiabilidad del código y reduce los tiempos de incertidumbre cuando algo falla.

Unreal Engine provee un sistema de pruebas de todo tipo, tanto unitarias (como las mencionadas en los párrafos anteriores) como de integración (probar la interacción entre funcionalidades distintas que deben relacionarse) y rendimiento.

Más adelante se detallará el sistema de automatización de UE4.

Fue difícil encontrar la manera de realizar las pruebas unitarias con el motor del juego. La documentación del motor es poco clara y no hay mucha información de la aplicación de la técnica en el motor, ya que la mayoría de la industria de videojuegos es escéptica de las mejoras que se pueden lograr.

Otro problema fue poder disminuir el tiempo en realizar la prueba y probarla, ya que se debe crear la prueba, compilar el proyecto, abrir el editor del motor (el paso más lento) y correr las pruebas. Para solucionar este problema, se recurrió a la integración continua.

Continuous Integration

Es tedioso que cada vez que se hace un cambio, se tenga que compilar el proyecto y luego ejecutar las pruebas para aprobar el cambio (todo esto realizado por el desarrollador) lo que genera una pérdida de tiempo en tareas repetitivas y un desgaste mental en quien lo hace.

La solución, Integración Continua⁵², que se basa en automatizar los procesos de aprobación de cambios para tener siempre disponible un entorno que funcione y cumpla con lo requerido.

La idea es la siguiente: cada vez que se realiza un cambio, este cambio debe cumplir una serie de requerimientos para que sea aceptado (que no destruya el resto del proyecto al implantarse) y aprobado (cumpla con su funcionalidad). Como la realización manual de estas tareas puede ser un proceso largo y repetitivo, se utilizan automatizadores, que son programas encargados de realizar tareas automáticamente luego de ser activados por algún tipo de evento. Jenkins es uno de ellos.

Jenkins

Jenkins⁵³ se eligió por tener una gran capacidad de personalización, ser muy reconocido en la industria de software y ser un programa que se encuentra disponible hace años.

Una de las formas en que se puede activar Jenkins es a través de webhooks, “ganchos” que enlazan un servicio web a otro, donde el primer servicio web “avisa” a otro que hubo un cambio y debe actuar. En el caso del trabajo, se utiliza un webhook entre GitHub (donde se va a guardar el código del proyecto) y Jenkins, el cual está configurado de tal forma que GitHub activa a Jenkins en el caso que se actualice el repositorio con un push (aceptar cambios a nivel remoto) o con un pull request (unir ramas del historial de versiones).

Jenkins lo que hará es recibir la notificación y comenzar un proceso similar a una línea de ensamble, donde en este caso se realiza el armado del proyecto (building) y realización de pruebas (testing). Esto lo hace siguiendo los pasos declarados en un archivo que se encuentra en el mismo repositorio de GitHub, el cual el desarrollador declara cuáles serán los pasos por realizar.

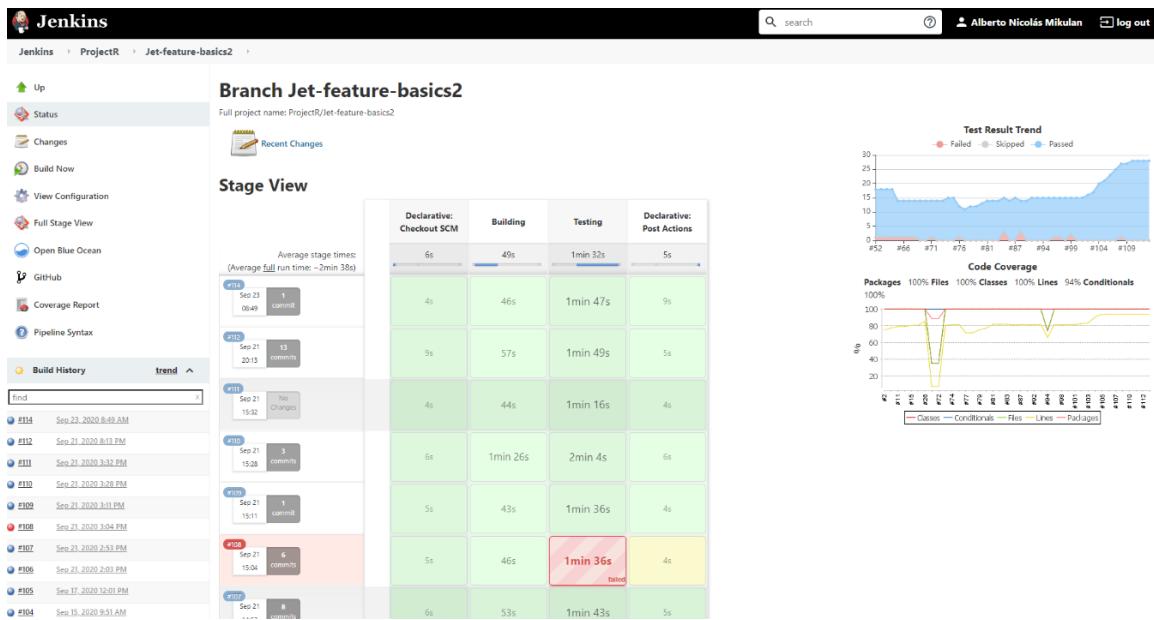


Ilustración 5. Vista de armados (builds) realizados por Jenkins en la rama Jet-Feature-basics2. A la izquierda, los armados hechos. En el centro, resultados de cada etapa de armado (verde: éxito; amarillo: advertencias; rojo: fallas). A la derecha, tendencia de resultado de pruebas y debajo de ella la tendencia de la cobertura de código.

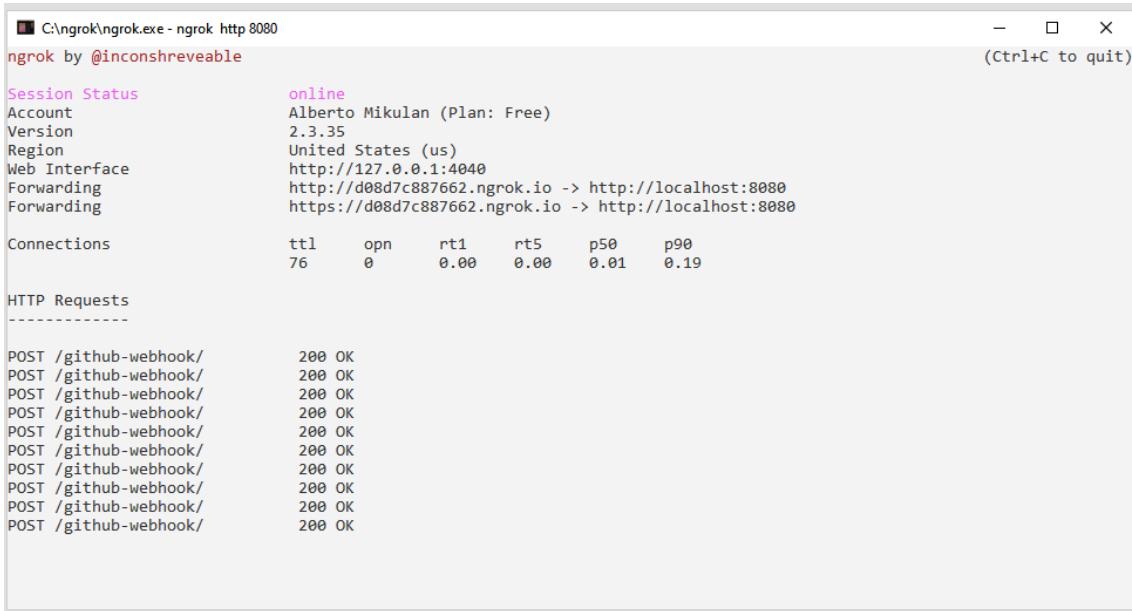
Ahora bien, Jenkins, al estar trabajando con un servicio web tiene que ser capaz de ser notificado de cambios a través de la red. Como el autor no posee un dominio el cual pueda redirigir a un servidor que contenga a Jenkins para procesar los cambios, tuvo que recurrir a utilizar ngrok.

Ngrok

Ngrok⁵⁴ es un programa que permite “abrir” los puertos de una computadora. En realidad, hace lo siguiente: Uno se descarga el cliente ngrok que se va a comunicar con el servidor ngrok cada vez que se quiera abrir un puerto. Para ello, en la consola del cliente se especifica qué puerto se desea abrir.

Luego, el cliente ngrok se comunica con el servidor ngrok y pide que se genere un dominio nuevo (bajo el dominio general de ngrok). El servidor lo crea, lo habilita y notifica al cliente la dirección que se habilitó. El cliente imprimirá por pantalla la dirección habilitada y el puerto al que se asoció. Lo que hace ngrok es lo siguiente: cuando el servidor recibe una solicitud a la dirección habilitada, le notifica al cliente ngrok y le reenvía la solicitud recibida. Ngrok reenvía esa solicitud al puerto asociado a ella.

Entonces, ¿por qué es necesario este proceso? Jenkins necesita escuchar ciertos puertos para poder recibir activaciones de las distintas líneas de ensamble que posea. GitHub se comunica a través de webhooks con otros servicios web. Entonces, se necesita utilizar la dirección que provee ngrok para ingresarla en el webhook del repositorio de GitHub y que el puerto al que redirija sea el que escuche Jenkins. Así, se logra comunicar a GitHub con Jenkins cuando no se posee un servidor propio.



```

C:\ngrok\ngrok.exe - ngrok http 8080
ngrok by @inconshreveable
Session Status: online
Account: Alberto Mikulan (Plan: Free)
Version: 2.3.35
Region: United States (us)
Web Interface: http://127.0.0.1:4040
Forwarding: http://d08d7c887662.ngrok.io -> http://localhost:8080
Forwarding: https://d08d7c887662.ngrok.io -> http://localhost:8080

Connections: ttl opn rt1 rt5 p50 p90
76 0 0.00 0.00 0.01 0.19

HTTP Requests:
-----
POST /github-webhook/ 200 OK

```

Ilustración 6. Vista de ngrok en ejecución. Los colores están invertidos.

Retomemos Jenkins. Se dijo que este programa recibe activaciones y activa “líneas de ensamble”. Se dijo también que los pasos de las líneas de ensamble se encuentran declarados en un archivo, dentro del repositorio. Lo que no se dijo es que existe un paso adicional entre que se recibe la activación y se activa la línea de ensamble. En este paso, Jenkins se comunicará con GitHub y recibirá todos los cambios generados en el repositorio (realiza un pull). De este modo, Jenkins recibirá también (dentro de un área de trabajo especial para cada línea de ensamble) la línea de ensamble a procesar por lo que, a partir de este momento, lo único que debe hacer es leerla y seguir los pasos. Cabe mencionar, y no es algo mínimo, que Jenkins debe tener instalados todos los programas que sean utilizados por la línea de ensamble en el servidor donde Jenkins esté instalado.

Para este proyecto, los pasos declarados en la línea de ensamble son:

1. Dentro de la etapa de armado:
 - a. Notificar a la aplicación Slack que se inició la línea de ensamble.
 - b. Ejecutar un archivo BATCH (el proyecto requiere Windows) que ejecutará la herramienta Unreal Automation Tool para que arme el proyecto en base a los archivos del repositorio.
2. Dentro de la etapa de realización de pruebas:
 - a. Ejecutar un archivo BATCH que llamará al editor de Unreal Engine 4 (en modo headless) para que ejecute el módulo de pruebas que trae consigo y ejecute las pruebas especificadas en los archivos del proyecto.

3. Al finalizar la línea de ensamble (se realiza en todo caso):
 - a. Se transforma el reporte generado por Unreal Engine desde el formato JSON a JUnit para que Jenkins sea capaz de comprenderlo y mostrar los resultados de las pruebas.
 - b. Se notifica a la aplicación Slack la cantidad de pruebas realizadas, cuántas fallaron y cuántas fueron exitosas, y si la línea de ensamble realizó los pasos declarados con éxito o no.
 - c. Se limpia el entorno de trabajo de la línea de ensamble para que no haya archivos que puedan entrar en conflicto la próxima vez que se active la línea de ensamble.
 - d. Jenkins muestra el resultado de la ejecución de la línea de ensamble.

Se dice que Jenkins muestra el resultado de la ejecución de la línea de ensamble, pero ¿a qué se refiere?

Esto quiere decir que solo se va a tener éxito en la ejecución de la línea de ensamble si no hubo error en ninguno de los pasos. O sea, que ningún programa haya fallado. Los programas pueden fallar porque no se les dan los parámetros correctos o están defectuosos.

En el caso de la realización de pruebas, el editor de Unreal Engine falla si por lo menos alguna prueba falla, lo que es lógico porque significa que el proyecto no cumple con los requisitos necesarios. Por lo que se rompe la integración continua.

¿Qué ocurre cuando Jenkins falla en alguna de sus etapas? Jenkins abortará la ejecución de la línea de ensamble, realizará los pasos que se realizan al finalizar la línea de ensamble y marcará esa activación de la línea de ensamble como fallida.

Para finalizar y tener una idea general se muestran los pasos que ocurren cuando se actualiza el repositorio en GitHub:

1. Se recibe el cambio del repositorio (pull request o push).
2. GitHub utiliza su webhook para comunicarse con la dirección del servidor ngrok y enviar la notificación del cambio del repositorio.
3. El servidor ngrok notifica y reenvía al cliente ngrok el cambio.
4. El cliente ngrok redirige la notificación al puerto donde escucha Jenkins.
5. Jenkins recibe la notificación y descarga los cambios del repositorio de GitHub.
6. Jenkins activa la línea de ensamble.
7. Jenkins guarda el resultado de la activación de la línea de ensamble.

De esta forma, se logra tener una integración continua, donde todo el proceso de cambios al proyecto se encuentra automatizado y disponible para todo desarrollador. En todo cambio, el proyecto se enfrenta a todas las pruebas y es aceptado únicamente cuando todas las pruebas son satisfactorias.

El autor construyó un pequeño tutorial para la configuración de un entorno simple de Continuous Integration en Unreal Engine 4 (aborda también la cobertura de código). El tutorial se puede acceder mediante:

<https://www.ue4community.wiki/jenkins-ci-amp-test-driven-development-6912tx0c>

Cobertura de Código

Se realiza también una cobertura de código de las pruebas para saber qué código no se prueba y analizar si es necesario incluirlo en pruebas. No se busca cumplir un porcentaje de cobertura, sino que se utiliza para ayudar a saber que se está probando lo suficiente.

Se utiliza OpenCppCoverage⁵⁵ para analizar la cobertura mientras se ejecutan las pruebas con el editor de Unreal Engine. OpenCppCoverage se une al editor mientras corre las pruebas, para así poder realizar la cobertura de código. Luego, generará un informe de cobertura que será leído por un plugin de Jenkins llamado Cobertura, el que mostrará los resultados del informe en la pantalla de Estado de la ejecución de Jenkins.

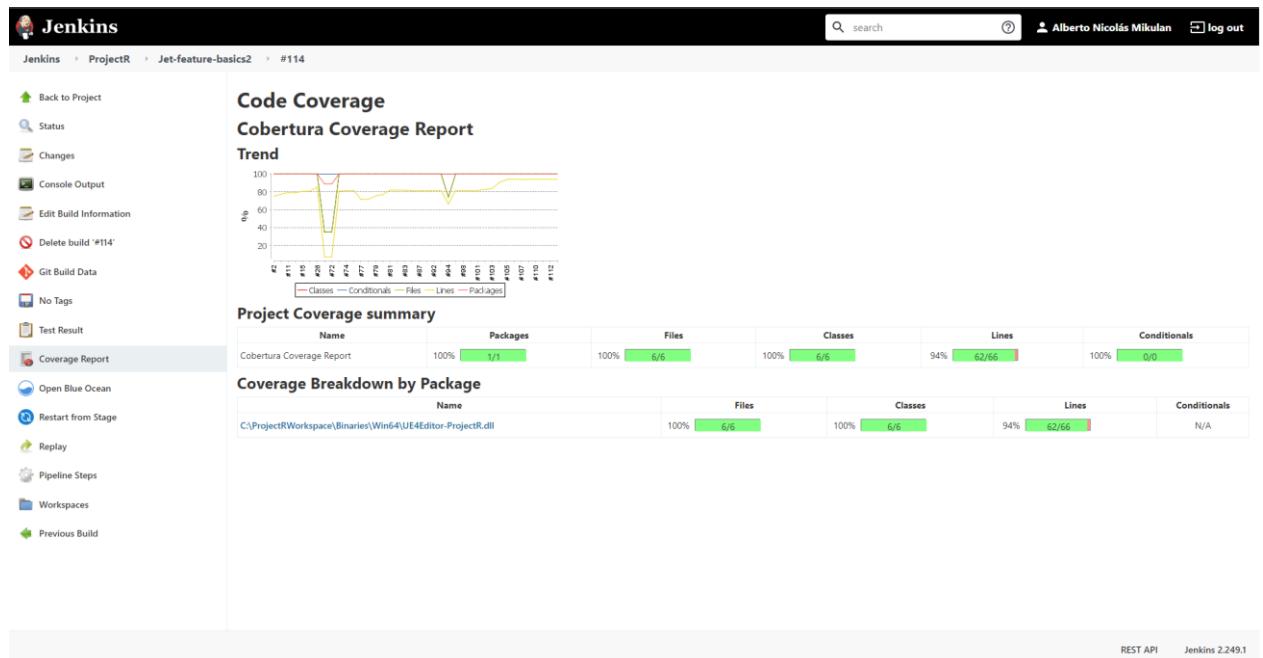


Ilustración 7. Cobertura de código en Jenkins.

Sistema de Automatización de UE4

El sistema de automatización⁵⁶ de Unreal Engine 4 se construye sobre el Framework de pruebas funcionales⁵⁷ (usado para Blueprints), que está diseñado para realizar pruebas a nivel de juego y funciona realizando una o varias pruebas automatizadas. La mayoría de las pruebas que se hagan serán funcionales, de bajo nivel o del editor que necesitan del uso del sistema de automatización.

Estas pruebas se pueden desprender en las siguientes categorías, según su propósito:

- Unitarias:
 - Son pruebas a nivel de APIs.
- De Características:
 - Pruebas a nivel del sistema que verifican PIE (Play In Editor, la simulación del juego), estadísticas a nivel del juego o cambios en resolución.
- De verificación de armado (Smoke):
 - Pruebas rápidas que se verifican cada vez que se inicie el editor, juego o comandos. Solo pueden ser unitarias o de características que sean rápidas, ya que se deben ejecutar en un segundo.
- De Estrés:
 - Pruebas que exigen a un sistema, para evitar fallos masivos.
- Comparación de imágenes:
 - Pruebas que identifican problemas de renderización entre diferentes versiones o armados del programa⁵⁸.

También se tienen pruebas para verificar modelos en formato FBX⁵⁹.

Nosotros nos vamos a enfocar en las pruebas unitarias y de características. Para eso, Unreal Engine tiene lo que se llama Automation Testing, o pruebas automáticas.

Pruebas Automáticas

Las pruebas automáticas⁶⁰ son el nivel más bajo de pruebas y existe por fuera del ecosistema UObject (no deriva de él), por lo que no es visible a los Blueprints ni al sistema de reflexión del motor (UE4 posee un sistema de reflexión de clases propio para tener la habilidad de meta clases).

Son pruebas codificadas y pueden ser ejecutadas por el editor o línea de comandos de la consola del editor.

Se pueden separar las pruebas en dos tipos: simples y complejas. Los dos tipos son clases derivadas de FAutomationTestBase⁶¹ y su diferencia radica en que las pruebas complejas son utilizadas cuando se quiere correr la misma prueba sobre un conjunto de objetos del mismo tipo (como realizar la prueba para todos los niveles del proyecto).

Creación de Pruebas Simples/Complejas

Las pruebas se declaran con macros y se implementan sobrescribiendo métodos virtuales de la clase FAutomationTestBase. Estas macros, para pruebas simples y complejas respectivamente son:

IMPLEMENT_SIMPLE_AUTOMATION_TEST

IMPLEMENT_COMPLEX_AUTOMATION_TEST

Las dos Macros llevan los mismos tres parámetros:

- TClass: El nombre de clase que se desea para la prueba.
- PrettyName: El nombre en String que se mostrará en la interfaz de usuario.
- TFlags: Una combinación (separada por |) de valores de EAutomationTestFlags, usados para especificar requerimientos y comportamientos para la prueba (si se puede ejecutar en el editor, en modo servidor, en el juego, etc).

Una vez que se declara la Macro, se puede llevar a cabo la definición de la prueba, para ello se sobreescreiben:

- bool RunTest(const FString& Parameters);
 - RunTest tendrá el cuerpo de la prueba y devolverá true si la prueba pasa o false si falla. Los argumentos de RunTest, Parameters solo van a ser usados en caso de que se tenga una prueba compleja.
- void GetTests(TArray< FString>& OutBeautifiedNames, TArray < FString>& OutTestCommands);
 - GetTests es usado únicamente si es una prueba compleja y sus dos argumentos se usan en conjunto para agregar los múltiples objetos a probar.

Por ejemplo:

```
IMPLEMENT_SIMPLE_AUTOMATION_TEST(FOneShouldBeOneTest,
"ProjectR.Unit.PlaceholderTest.OneShouldBeOne", EAutomationTestFlags::ApplicationContextMask | EAutomationTestFlags::ProductFilter)

bool FOneShouldBeOneTest::RunTest(const FString& Parameters)
{
    //One should be one.
    {
        TestEqual(TEXT("one is one"), 1, 1);
    }
    return true;
}
```

Que prueba si uno es uno, una prueba trivial. Es importante destacar que existe una serie de métodos que inician con Test, como TestEquals, que establecen el resultado de la prueba comparando valores y emiten un mensaje si la prueba falla. Si la prueba fallase en este caso, el mensaje sería: *"one is one" expected to be true.*

Comandos Latentes (Latent Commands)

Existen determinados métodos que es necesario que se lleven a cabo en varios cuadros (frames), para ello se utilizan Latent Commands que se ejecutarán en cada cuadro hasta que el método termine. Estos comandos latentes se llaman, como a una clase, dentro de RunTest.

Para declarar estos comandos se utiliza nuevamente una Macro:

- **DEFINE_LATENT_AUTOMATION_COMMAND**, la cual toma un solo parámetro que será el nombre de la clase del comando latente.

Para finalizar con el comando latente, se debe sobrescribir el método **Update()** (que es el método que se ejecutará en cada cuadro) e implementar la lógica del método dentro de él.

Por ejemplo:

```
DEFINE_LATENT_AUTOMATION_COMMAND(FWaitForEditor);

bool FWaitForEditor::Update()
{
    If ( GEditor->IsPlayingSessionInEditor() )
    {
        //hacer algo
        return true;
    }
    return false;
}
```

Que va a esperar al editor a que esté en una sesión de juego para hacer algo. Luego devuelve true para finalizar el método. Si el editor no está en una sesión, retorna false cada cuadro hasta que lo esté.

Para construirlo y que comience a funcionar su método `Update()` en una prueba, se debe utilizar:

- `ADD_LATENT_AUTOMATION_COMMAND(FWaitForEditor());`

Los comandos latentes también pueden ser definidos para recibir parámetros. En esos casos, se debe usar una macro distinta que especifica la cantidad de parámetros (que serán usados como variables pertenecientes a la clase creada), como por ejemplo `DEFINE_LATENT_AUTOMATION_COMMAND_ONE_PARAMETER`, donde los parámetros serán el nombre del comando, el tipo de variable a usar y el nombre que va a llevar esa variable en la clase comando, todo separado por comas.

Para llamar comandos latentes que tienen parámetros, se los llama de la misma forma en las pruebas, solo que esta vez se pasan los argumentos.

Por ejemplo:

```
DEFINE_LATENT_AUTOMATION_COMMAND_ONE_PARAMETER(FLatentCommandCounterCommand, int, latentCounter);

bool FLatentCommandCounterCommand::Update()
{
    ++latentCounter;

    If ( latentCounter > 5 )
    {
        return true;
    }

    return false;
}

IMPLEMENT_SIMPLE_AUTOMATION_TEST(FLatentCounterTest, "tests.LatentCounterTest",
EAutomationTestFlags::ApplicationContextMask | EAutomationTestFlags::ProductFilter)

bool FLatentCounterTest::RunTest(const FString& Parameters)
{
    int latentCounter = 0;

    ADD_LATENT_AUTOMATION_COMMAND(FLatentCommandCounterCommand(latentCounter));
    return true;
}
```

En este caso, se ejecutará el comando latente por cinco cuadros y en el sexto cuadro se terminará de correr.

Existe, además, soporte para Behaviour-Driven Development⁶². El autor todavía no está seguro de implementarlo al proyecto porque ya está en una etapa avanzada, pero lo tendrá presente para futuros proyectos o cuando finalice la etapa del trabajo final.

Otra posibilidad es implementar pruebas personalizadas, que son objetos que derivan de `FAutomationTestBase`. Se las implementa usando `IMPLEMENT_CUSTOM_SIMPLE_AUTOMATION_TEST` o `IMPLEMENT_CUSTOM_COMPLEX_AUTOMATION_TEST` y especificando la clase de prueba que se utilizará.

Notación

UE4 tiene un estándar de codificación⁶³ y en él un estándar de notación, pero el autor cree que tener prefijos para objetos/variables ('b' para bool, 'A' si desciende de un actor, 'U' si deriva de UObject, etc) traba la lectura de código y limita el poder de expresión de un programador.

El autor utilizará una nomenclatura sencilla que se utilizó en clases de Ingeniería de Software II:

- lowerCamelCase para objetos, variables, métodos.
- UpperCamelCase para clases.
- Prefijo 'a/an' y 'at' para argumentos de métodos, excepto cuando son colecciones.

Hay prefijos a los que no se puede escapar cuando se crean clases y métodos derivados que usan UpperCamelCase, por lo que se ignorará la nomenclatura especificada en esos casos y se utilizará la dada por Unreal Engine 4.

Módulos de Pruebas

A partir de un momento en el desarrollo, se modificó la organización del código por medio de módulos de código en el proyecto, para mantener una separación entre la implementación del programa y las pruebas que se hacen sobre él. Esto crea una mejor organización del proyecto y la posibilidad de excluir el código de las pruebas de ser empaquetado en la versión final del programa.

Crear un módulo para el proyecto es simple y se realiza creando archivos de configuración para módulos y modificando el archivo de construcción del módulo (módulos de los que depende), del editor y del proyecto.^{64 65 66 67 68 69 70}

Organización del Código

Durante el proyecto, la organización del código fue modificándose a medida que la cantidad de archivos dejaba de representar la forma lógica que se deseaba.

La organización de la implementación siempre se mantuvo durante el proyecto. Se tiene dos partes: la declaración de clases y la implementación de ellas. Cada parte tiene su carpeta y dentro de ella hay carpetas que engloban objetos o dominios de objetos.

La organización de las pruebas fue evolucionando a medida que crecía su contenido. Primero se tenía una carpeta para pruebas y otra para comandos latentes.

A medida que fue creciendo la cantidad de pruebas y clases, se llegó a un punto donde era difícil seguir el camino de cada prueba y comandos latentes que utilizaba.

Se decidió crear una carpeta por cada clase del proyecto. Dentro de esa carpeta se encuentran los cuatro archivos mencionados en el párrafo anterior, que representan el conjunto de pruebas para esa clase.

A continuación, se muestra una captura de la organización del código:

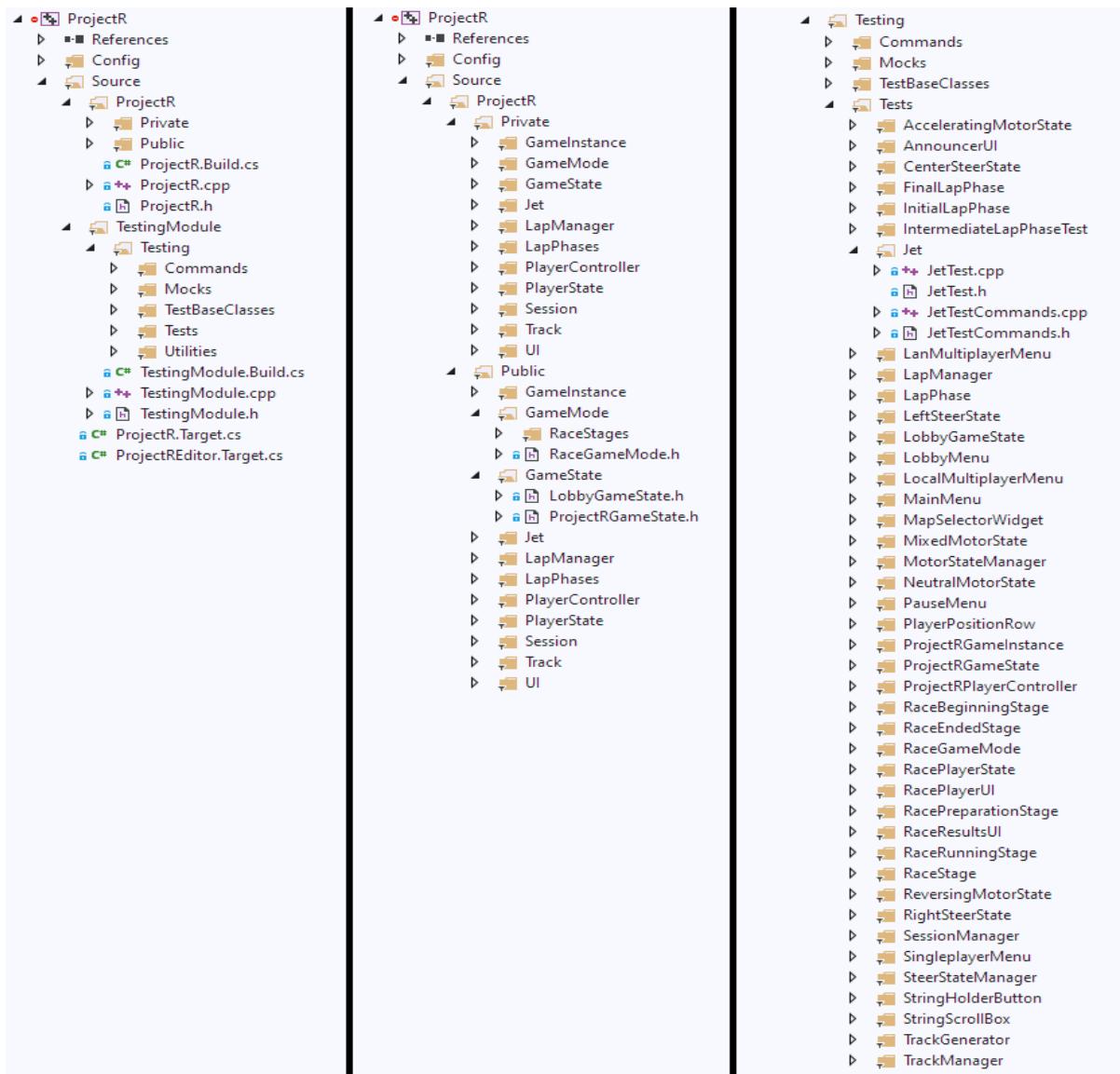


Ilustración 8. Vistas de la organización actual del proyecto. De izquierda a derecha: vista general del proyecto, ampliación del área de implementación, ampliación del área de pruebas.

Clases del Juego y Lógica

Como es un videojuego futurista de carreras que se puede jugar en multijugador y en red, se necesitará las reglas para una carrera, una pista, una nave, un control, interfaces de usuario, un sistema de replicación de clases sobre la red y otras clases para sustentarlo.

Las reglas del juego son:

- Para ganar la carrera, una nave tiene que dar 3 vueltas a la pista y salir primera.
- La pista tiene un único sentido para poder ganar.
- Solo se toma como una vuelta si se da una vuelta completa a lo largo de la pista.

Los requisitos para una nave son:

- Debe levitar, para dar un toque futurista al juego.
- Se tiene que poder controlar, para ser capaz de girar hacia ambos lados, acelerar, frenar y retroceder.
- Debe ser posible replicarla sobre la red (que se reflejen los cambios en otros usuarios conectados a la misma sesión).

El control debería:

- Poder controlar la nave.
- Pausar el juego.

Las interfaces de usuario tienen que:

- Permitir acceder a las distintas funciones (carrera multijugador y en red).
- Mostrar información de la carrera y la nave.

El sistema de replicación tendría que:

- Crear, buscar, unirse y destruir sesiones en red.
- Permitir replicar la nave en toda la red (cuando se modifique) a los usuarios que se encuentren en la misma carrera que ella.
- Transmitir información de la carrera a todos los usuarios, brindando información general y personalizada.

A continuación, se muestran las clases creadas para lograr lo mencionado anteriormente. Se enfocó durante todo el desarrollo en tratar de extender las clases dadas por el motor en vez de crear clases propias desde cero.

UProjectRGameInstance

Es la instancia del juego. La encargada de cargar el menú principal, ser un nexo entre menús y administrar sesiones del juego. En ella también se guarda variables comunes al juego. Deriva de ***UGameInstance***.

ARaceGameMode

Deriva de ***AGameModeBase***. En ella se especifica toda la lógica de reglas de una carrera: cuándo iniciarla, dónde posicionar las naves para la carrera, instanciar el administrador de vueltas (*ALapManager*), calcular posiciones, mostrar la cuenta regresiva para que se inicie la carrera, registrar jugadores entrantes a la pista, registrar ganadores y terminar la carrera.

Se decidió crear etapas de la carrera (derivadas de ***ARaceStage***) para controlar mejor el flujo de la carrera. Cada vez que una etapa termina, dispara un evento al que ***ARaceGameMode*** está suscrito para que cambie de etapa. El ***ARaceGameMode*** administra todas esas etapas (preparación, inicio, ejecución y finalización de la carrera). Se utiliza en mundos con pistas, mundos para correr carreras.

AProjectRGameState

Deriva de ***AGameStateBase***. El ***ARaceGameMode*** lo utiliza para compartir a todos los clientes la cuenta regresiva al inicio de la carrera. Guarda el tipo de menú de resultados de carrera y del anunciante. Se utiliza junto con ***ARaceGameMode*** en mundos para correr carreras. Para ello utiliza eventos a los que las clases interesadas se suscriben.

ALobbyGameState

Deriva de ***AGameStateBase***. Comunica a todos los clientes el mapa de la sesión LAN que seleccionó el anfitrión, así como también la cantidad de jugadores conectados en la sesión. Para ello utiliza eventos a los que las clases interesadas se suscriben.

ALapManager

Deriva de ***AActor***. Es la encargada de administrar las vueltas que tiene cada jet. Tiene registradas las distintas fases de una vuelta (inicial, intermedia y final; todas derivadas de ***ALapPhase***) y está suscrita a los eventos de solapado a las fases (por ejemplo, si una nave cruza esa fase, se comunica a la administradora de vueltas que se cruzó esa fase).

Controla si verdaderamente una nave cruza una vuelta o no (puede ser que una nave esté yendo en sentido contrario a la dirección de la pista e igual solaparse con las fases) guardando en una estructura la cantidad de vueltas de cada nave, la fase actual en la que está cada una y la última fase que cruzó. Esto es así ya que tener la última fase que se cruzó evita que se den varias vueltas en sentido reverso a la pista para contar vueltas.

ALapPhase y Clases Derivadas

Deriva de **AActor**. Una fase de vuelta es un muro invisible y atravesable en el juego. Se las ubica a lo largo de la pista y sirven para controlar el sentido de la pista.

Si fuese una sola fase, no se podría tener control del sentido de la pista (se podría ir en cualquier sentido y eso se tomaría como una vuelta), además que no haría falta dar toda la vuelta alrededor de la pista para contar una vuelta.

Si fuesen dos fases, se tendría que dar toda una vuelta para contar una vuelta, pero no se tendría control del sentido (se podría dar media vuelta en un sentido y media en el otro o dar la vuelta en el sentido contrario al deseado e igual se tomaría como una vuelta).

Con tres fases se puede lograr tener un control del sentido de la vuelta: para cruzar una vuelta, se tienen que haber cruzado las tres fases y en orden, lo que da el sentido.

Las fases son: inicial (**AInitialLapPhase**), intermedia (**AIntermediateLapPhase**) y final (**AFinalLapPhase**), y cada fase sabe cuál es la fase que le sigue y la que la precede.

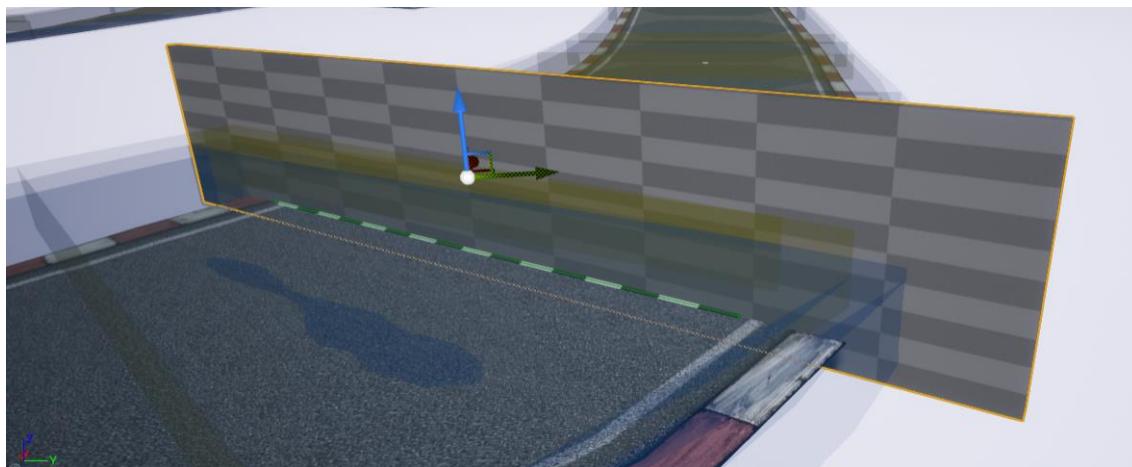


Ilustración 9. Vista de InitialLapPhase en el editor.

AProjectRPlayerController

Deriva de **APlayerController**. Se encarga de enviarle las acciones realizadas a la nave que controla, de crear y mostrar u ocultar el menú de pausa.

Tiene una referencia a la instancia de las interfaces de usuario de cuenta regresiva a la carrera, posición, vuelta y resultados de la carrera. **ARaceGameMode** le dice cuándo cargarlas.

ARacePlayerState

Deriva de **APlayerState**. **ARaceGameMode** la utiliza para enviar información de la carrera individualmente a cada jugador. Para ello le carga la vuelta, posición y el total de vueltas del jugador en ella. Las clases interesadas en estos datos se suscriben mediante eventos que **ARacePlayerState** dispara.

USessionManager

Deriva de **UObject**. Es utilizada por **UProjectRGameInstance** para administrar sesiones. Se encarga de crear, buscar, listar, unirse y destruir sesiones en red. Para ello, utiliza la interfaz de sesiones del motor y se subscribe a los eventos que ella dispone.

ATrackManager

Deriva de **AActor**. Es el administrador de pistas. Es creado por el generador de pistas (**ATrackGenerator**) y es utilizado para mantener atraídas las naves hacia la pista: en todo momento, cancela la fuerza de gravedad y la reaplica a cada nave hacia el vector normal de la pista en donde se encuentre esa nave para que no caiga si la pista se encuentra boca abajo.

ATrackGenerator

Deriva de **AActor**. Es la clase encargada de formar pistas. Tiene un componente que es una curva paramétrica. Esta curva se representa por puntos en el espacio y se unen por aproximación, usando el método de Newton-Raphson para lograrlo.

Lo que hace el generador es tomar secciones (entre dos puntos) y a lo largo de ellas colocar cortes de pista, aproximando el modelo de pista a la curva (inclinación, altura, ancho, etc.) y configura colisiones para cada sección. Cada vez que se modifica un punto, el administrador regenera la pista tomando en cuenta los cambios realizados.

Además de colocar cortes de pista, el generador de pistas agrega otro modelo, invisible, que cubre la pista para evitar que las naves salgan de ella. También se agrega un modelo invisible y atravesable a lo largo de la pista. Este modelo avisará al administrador de pistas que una nave lo está atravesando. Esto sirve para que el administrador sepa en qué pista está cada nave.

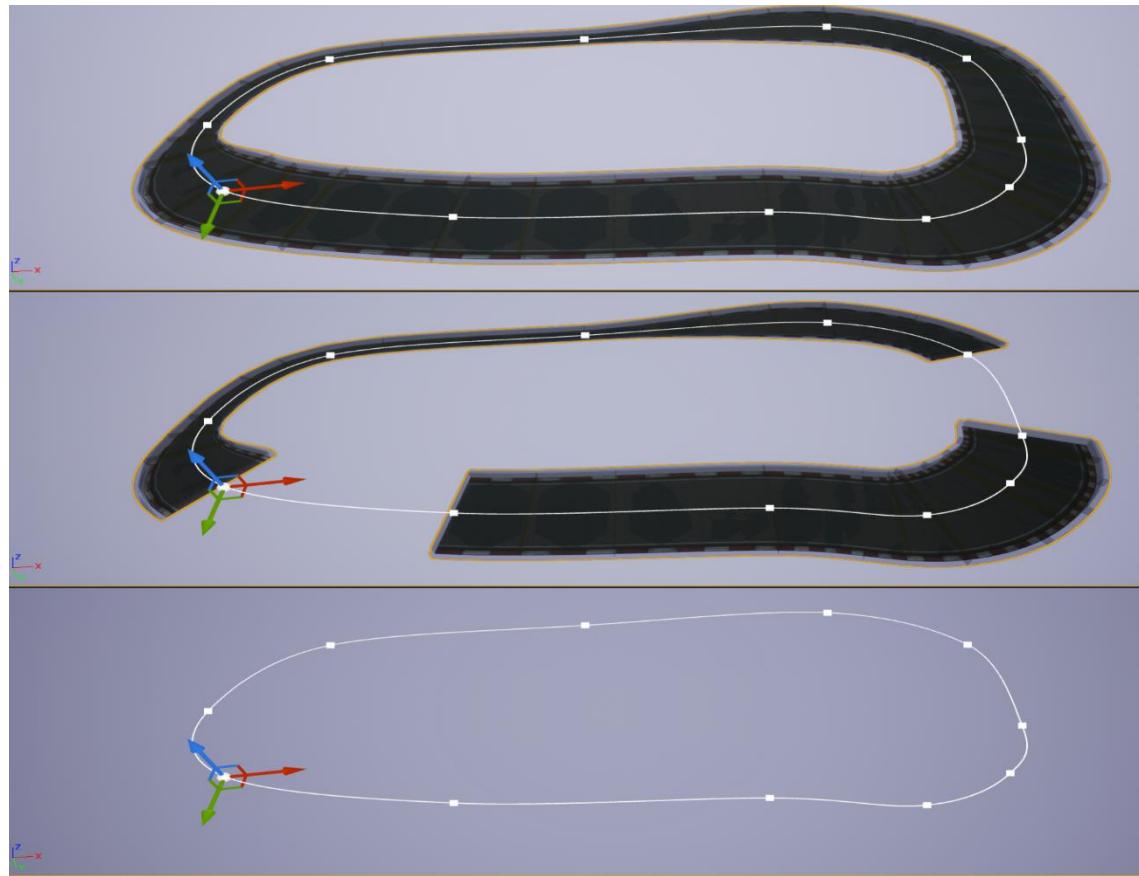


Ilustración 10. De arriba hacia abajo: descomposición de un *TrackGenerator* en el editor. Nótese cómo es posible sacar los modelos utilizados para el tramo de la pista que se deseé.

Interfaces de Usuario

Todas las interfaces de usuario derivan de **UUserWidget**⁷¹, que es una clase utilizada para contener y configurar elementos de UI.

UAnnouncerUI

Es cargada por **AProjectRPlayerController** una vez que se encuentra en una carrera. El control subscribe la interfaz a **AProjectRGameState** para que cualquier cambio en la cuenta regresiva del comienzo de la carrera sea mostrado por **UAnnouncerUI**.



Ilustración 11. Vista del AnnouncerUI en una sesión del juego. Remarcado en óvalo azul.

ULanMultiplayerMenu

Es el menú para crear, buscar y unirse a partidas en LAN. Llama a **UProjectRGameInstance** para buscar sesiones y actualizar la lista de sesiones que muestra a los usuarios. Para mostrar la lista de sesiones utiliza **UStringButtonScrollBox**.

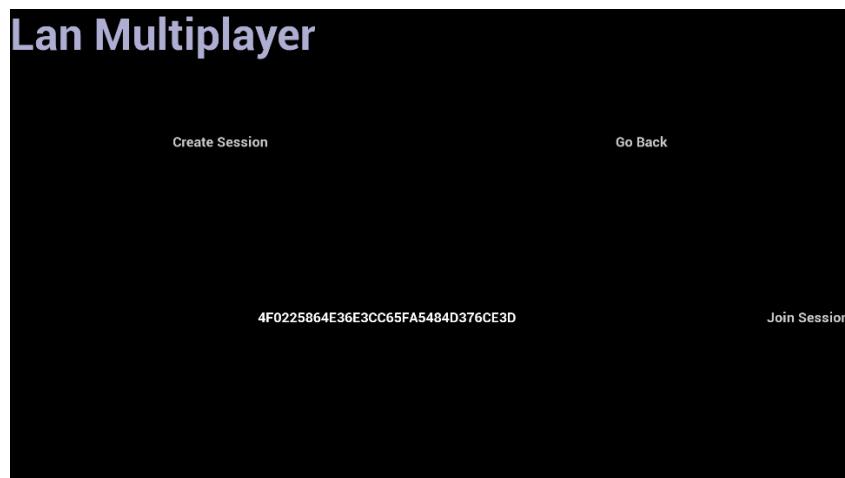


Ilustración 12. Vista del menú de multijugador LAN con una sesión encontrada.

UStringButtonScrollBox

Recibe un arreglo de texto y genera botones que contienen cada elemento del texto dentro. Cada vez que se hace clic en algún elemento de la lista, guarda el texto que contiene el botón seleccionado, para utilizarlo en el futuro.

UMapSelectorWidget

Permite buscar mapas (archivos terminados en '.umap') en un directorio especificado, filtrar el nombre del archivo y utilizar **UStringButtonScrollBox** para mostrar cada mapa seleccionable.

ULobbyMenu

Es el menú del lobby. Se utiliza como punto de reunión para las sesiones. Tiene distintas vistas según el usuario sea el anfitrión de la sesión o un invitado. Un anfitrión puede elegir la pista en la que se correrá la carrera y cuándo empezarla (utiliza **UMapSelectorWidget** para mostrar los mapas). Un invitado solo puede ver la pista elegida y la cantidad de jugadores en el lobby. **ULobbyMenu** se utiliza en conjunto con **ULobbyGameState** en un mismo mundo para mostrar la cantidad de jugadores conectados y la pista seleccionada.



Ilustración 13. Vista del menú de lobby (anfitrión).

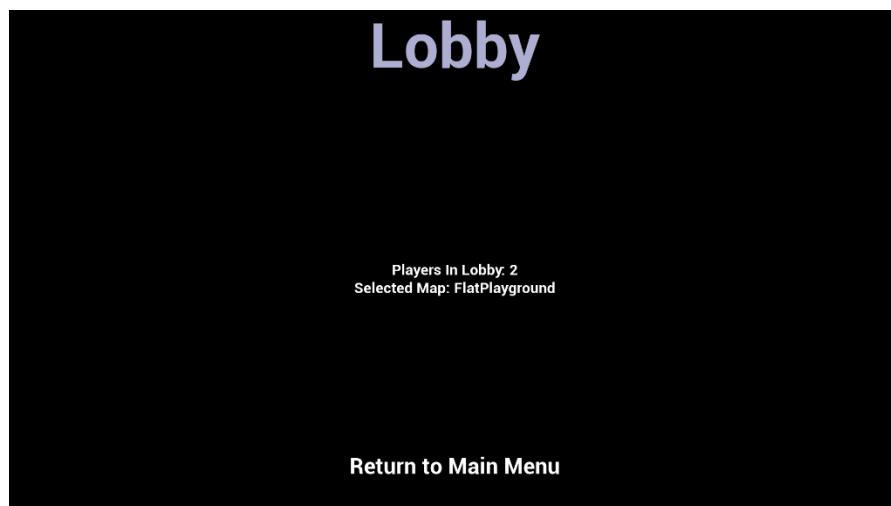


Ilustración 13. Vista del menú de lobby (invitado).

ULocalMultiplayerMenu

Es el menú para el multijugador local. Permite seleccionar la cantidad de jugadores que jugarán en la carrera, así como también lista las pistas disponibles, utilizando **UMapSelectorWidget** para ello.

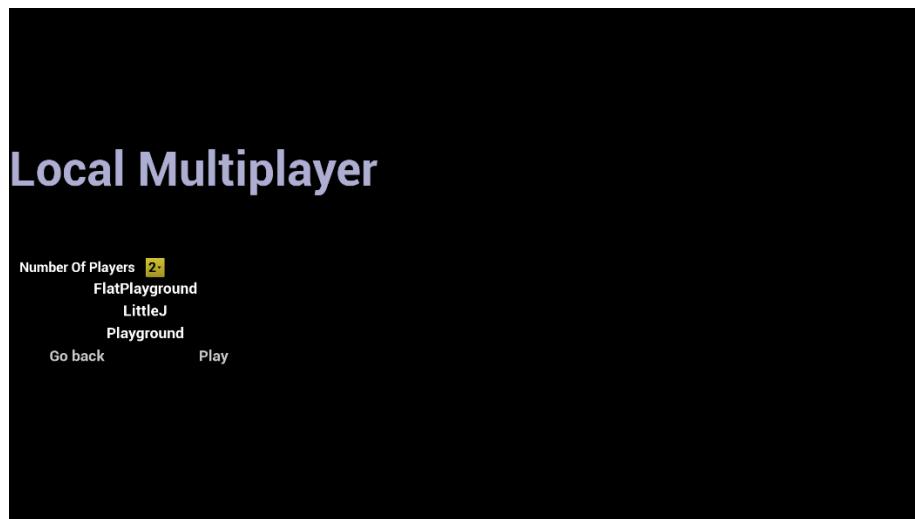


Ilustración 14. Vista del menú de multijugador local.

UPauseMenu

Es un menú para realizar una pausa durante una carrera. Es cargado, mostrado y ocultado por el **ProjectRPlayerController**. Permite resumir la carrera o regresar al menú principal (**UMainMenu**). Solo pausa una carrera si es que se está jugando una carrera localmente. Impide la pausa si es que se está jugando una carrera en red.



Ilustración 15. Vista del menú de pausa en una carrera.

URacePlayerUI

Es una interfaz ubicada en la esquina inferior derecha de la cámara en una carrera. Muestra la posición y vuelta actuales, así como también el total de vueltas. Es cargada por **ProjectRPlayerController** cuando **ARaceGameMode** le ordena hacerlo. En el cargado, se suscribe esta interfaz a los eventos del **ARacePlayerState**.



Ilustración 16. Vista de la interfaz URacePlayerUI en la esquina inferior derecha. Remarcado en azul.

URaceResultsUI

Es otra interfaz que se muestra una vez que el jugador termina una carrera. Muestra la lista de jugadores y sus posiciones. Al inicializarse, una referencia a **AProjectRGameState** para obtener así todos los **ARaceGameState** de los jugadores. Actualiza la lista de jugadores cada cierto tiempo, gracias a un temporizador. Es cargada por **AProjectRPlayerController** cuando **ARaceGameMode** le ordena hacerlo.

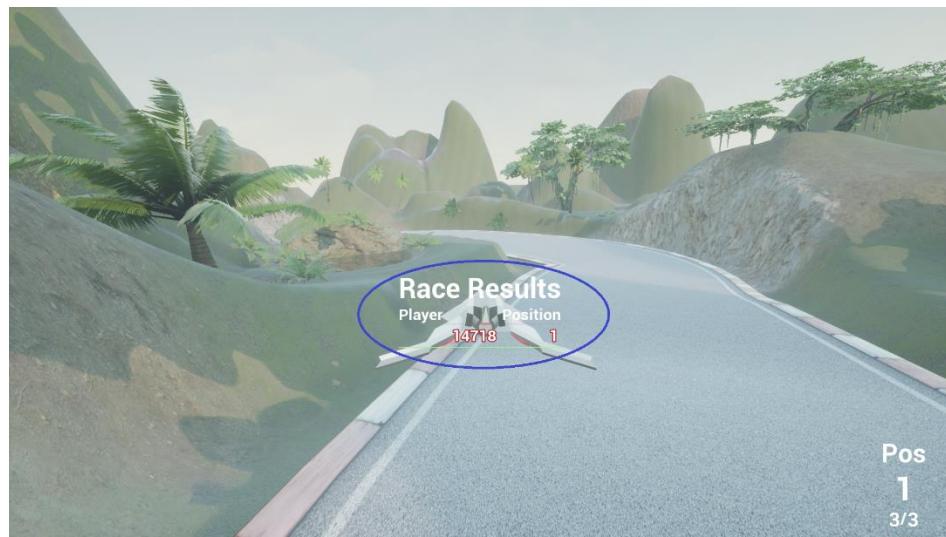


Ilustración 17. Vista del menú de resultados URaceResultsUI. Ubicado en el centro, remarcado en azul.

USingleplayerMenu

Es un menú utilizado para jugar carreras en solitario. Permite jugar una carrera y elegir la pista que se correrá gracias al **UMapSelectorWidget**.

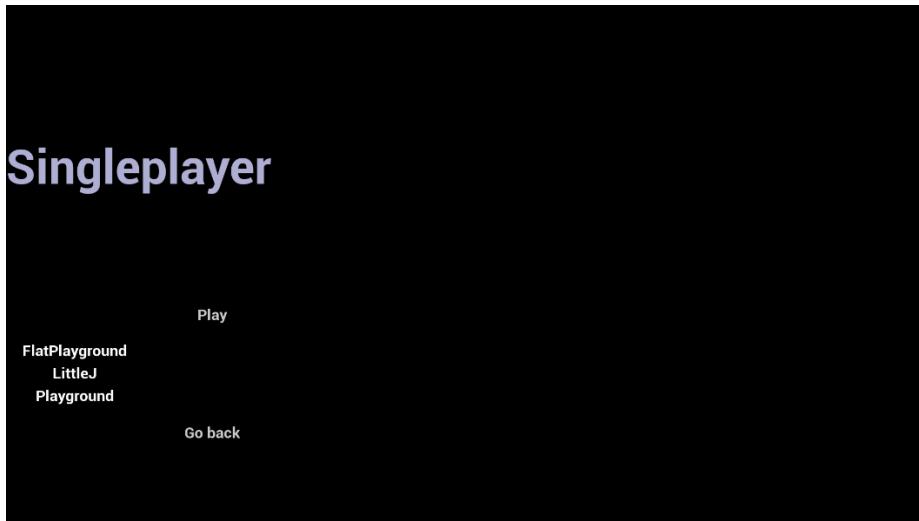


Ilustración 18. Vista del menú USingleplayerMenu.

UMainMenu

Es el menú principal del juego. Es cargado por el **ProjectRGameInstance** al inicio del juego y es al que se vuelve cuando se termina una sesión en red. Permite elegir cargar **USinglePlayerMenu**, **ULocalMultiplayerMenu** y **ULanMultiplayerMenu**. Tiene una función que automáticamente corre una serie de pruebas en la computadora que ejecuta el juego para poder elegir las opciones gráficas que más se adecúen.



Ilustración 19. Vista del menú principal.

AJet

Deriva de **APawn**. Es la clase más compleja de todo el proyecto. Representa las naves que se utilizan como vehículos en una carrera.

Es un objeto que utilizará extensivamente el sistema de fuerzas que utiliza Unreal Engine 4 (que es una implementación de NVIDIA PhysX).

Se debe aclarar que cuando uno habla de aplicar una fuerza o un torque, esto no sucede inmediatamente. En cambio, la fuerza/torque se efectuará en el próximo cuadro. Esto es así porque cuando uno aplica una fuerza/torque lo que en realidad está haciendo es agregar un elemento a la sumatoria de fuerzas/torques y al final del cuadro, el sistema de físicas aplica la fuerza/torque neta obtenida.

Para tener una reacción controlada a fuerzas y colisiones, se decidió dividir el modelo de la nave en dos partes: una visual (modelo del jet) y otra para físicas (física del jet).

Estas dos partes utilizan **UStaticMeshComponent**^{72 73}, que permite asociar un modelo 3D a un objeto y aplicarle propiedades físicas.

La parte física se llama internamente *physicsMeshComponent* y la visual es *jetModelMeshComponent*.

physicsMeshComponent es una tabla invisible con puntos destacados para aplicarle fuerzas y calcular distancias. Todos los componentes de **AJet** lo utilizarán para aplicar fuerzas o proyecciones.

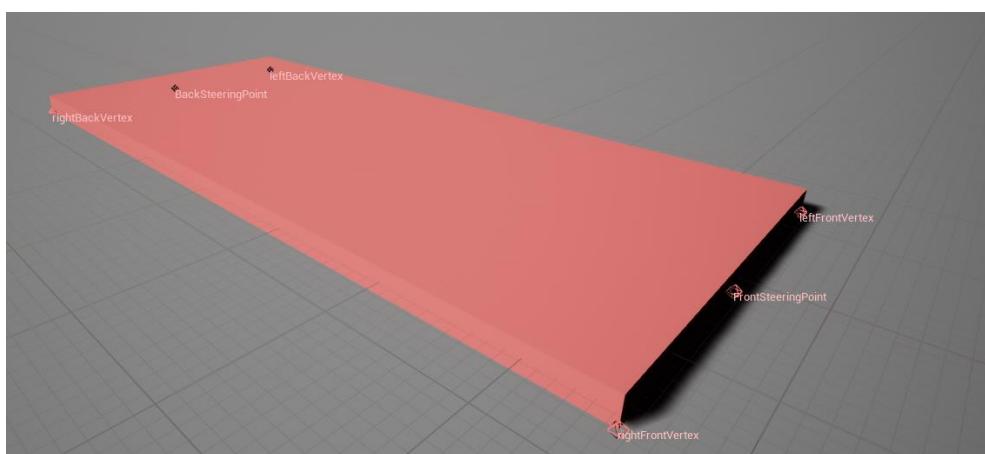


Ilustración 20. Vista del modelo usado para el *physicsMeshComponent* junto con sus puntos destacados. De atrás hacia adelante y de izquierda a derecha: *rightBackVertex*, *BackSteeringPoint*, *leftBackVertex*, *rightFrontVertex*, *FrontSteeringPoint* y *leftFrontVertex*.

jetModelMeshComponent es solo un modelo visual que se usará para tener una representación de la nave en la carrera.



Ilustración 21. Vista del modelo visual de **AJet** utilizado en *jetModelMeshComponent*.

Además de los modelos, AJet cuenta con un componente para simular una cámara, lo que permite al jugador tener una vista desde atrás del modelo de la nave en la carrera. El componente es una instancia de **UCameraComponent**.^{74 75}

Levitación en AJet

La levitación se logra por el **UAntiGravityComponent**, que es un componente creado y derivado de **UActorComponent**. En cada cuadro realiza lo siguiente:

1. Consigue la ubicación de cada punto destacado del *physicsMeshComponent* (excepto del BackSteeringPoint y FrontSteeringPoint).
2. Por cada punto hace una proyección hacia abajo del AJet.
 - a. Si choca con algo, se calcula la distancia a lo que chocó.
 - b. Se aplica una fuerza en dirección del vector normal de AJet y proporcional a la distancia calculada para poder distanciarse del piso una altura especificada.

De esa forma, se consigue que *physicsMeshComponent* se alinee (preferentemente) al piso de una pista y mantenga una distancia deseada del piso.

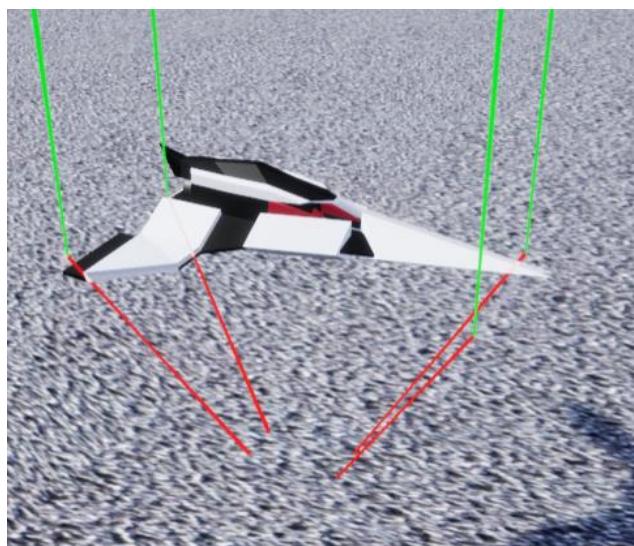


Ilustración 22. Vista de la levitación de una nave. Las líneas rojas representan la proyección hacia debajo de AJet, mientras que las verdes representan la dirección de la fuerza para levantar la nave.

Maniobrabilidad en AJet

Para lograr maniobrabilidad se creó el **USteeringComponent** y **UMotorDriveComponent** (también derivados de **UActorComponent**).

USteeringComponent habilita a **AJet** a poder girar en ambos sentidos

Giros

1. Se pide a **AJet** el vector normal de la pista en ese momento (más Adelante se explicará cómo logra conseguirlo **AJet**).
2. Se busca la ubicación del punto destacado del *physicsMeshComponent* llamado "FrontSteeringPoint".

3. Se calcula la aceleración centrípeta que provocaría doblar. Para ello, se pide la velocidad hacia delante de **AJet** combinada con la fórmula v^2/r donde v será la velocidad hacia delante y r será el radio de giro (modificable por el editor).
 4. Se multiplica la aceleración centrípeta obtenida por el vector de la derecha de **AJet** proyectado en el piso de la pista (de esa forma, siempre se dobla ortogonalmente al piso en el que esté **AJet**). A ese resultado se lo multiplica por el sentido en el que se quiera doblar (-1 para la izquierda y 1 para la derecha).
 5. Al resultado del punto anterior se lo utiliza para usarlo como fuerza a aplicar en la ubicación obtenida en el punto 1. Esta fuerza en ubicación es aplicar una fuerza y un torque (tomando la ubicación como el radio del torque).
- Cabe mencionar que no se multiplica la aceleración por la masa de **AJet** ya que el motor tiene una característica que permite obviar la masa del objeto para aplicar una fuerza.
6. Luego, se despacha un método que se aplicará en el próximo cuadro para alinear la velocidad y evitar que la nave “resbale” por la pista. Se despacha para el próximo cuadro ya que en ese cuadro nuestra fuerza va a ser efectivamente aplicada.

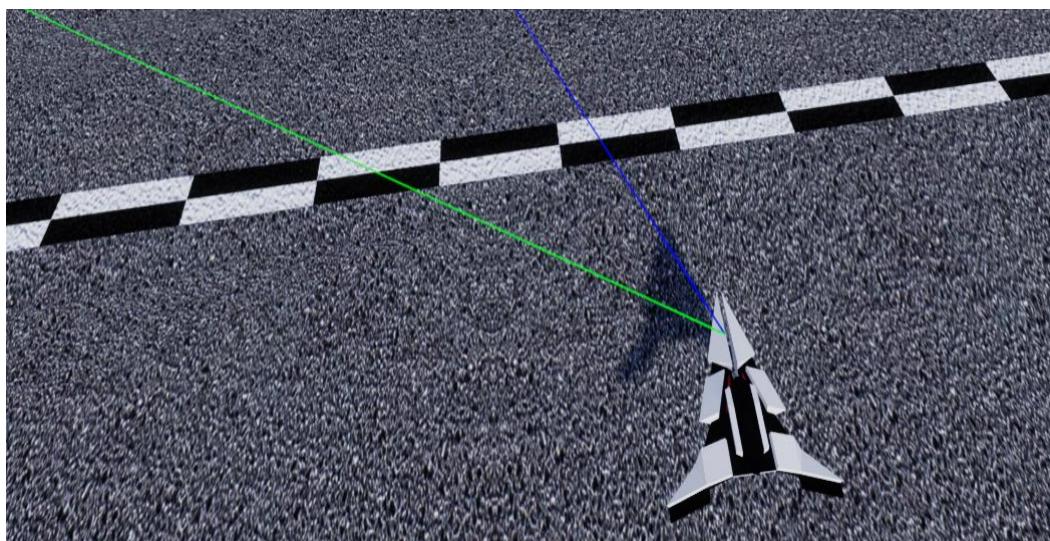


Ilustración 23. Vista de girar a la izquierda. La línea azul representa el radio del torque. La línea verde representa tanto la fuerza del torque como la fuerza que se aplicarán a la nave en el próximo cuadro.

Alineamiento de la Velocidad

Se debe cancelar la velocidad actual (que en este momento ya no está en dirección hacia delante de **AJet**, sino para un costado) y volver a aplicarla hacia delante de **AJet**.

Para cancelar la velocidad actual, se debe encontrar una fuerza que permita parar por completo a **AJet**. Para eso, se debe calcular la fuerza que hubiese sido necesaria desde el cuadro anterior hasta el actual para mover a **AJet**.

Lo único que se necesita del cuadro anterior será la ubicación de la nave y su vector hacia adelante en ese cuadro. Estos son los parámetros con los que se despacha el método del punto 6 mencionado anteriormente para poder doblar.

Para calcular la fuerza necesaria, se utiliza la fórmula de trabajo $W = F \cdot \Delta S$ ⁷⁶ y $\Delta K = \frac{m \cdot v^2}{2}$ ⁷⁷.

Igualando, se tiene:

$$F \cdot \Delta S = \frac{m \cdot v^2}{2}$$

$$m \cdot a \cdot \Delta S = \frac{m \cdot v^2}{2}$$

$$a \cdot \Delta S = \frac{v^2}{2}$$

$$a = \frac{v^2}{2 \cdot \Delta S}$$

v será la velocidad actual.

ΔS será la distancia entre la ubicación actual y la del cuadro anterior.

a será la aceleración de la fuerza necesaria para llegar desde el cuadro anterior al actual.

Entonces la fuerza que permita parar por completo a la nave será $\bar{F} = -m \cdot a \cdot \bar{v}$ (donde \bar{v} es el vector unitario hacia delante, proyectado al piso, del cuadro anterior).

La fuerza que se reaplica para alinear la nave será $\bar{F} = m \cdot a \cdot \bar{p}a$ (donde $\bar{p}a$ es el vector unitario que resulta de la proyección del vector hacia delante del cuadro actual de **AJet** sobre el piso de la pista).

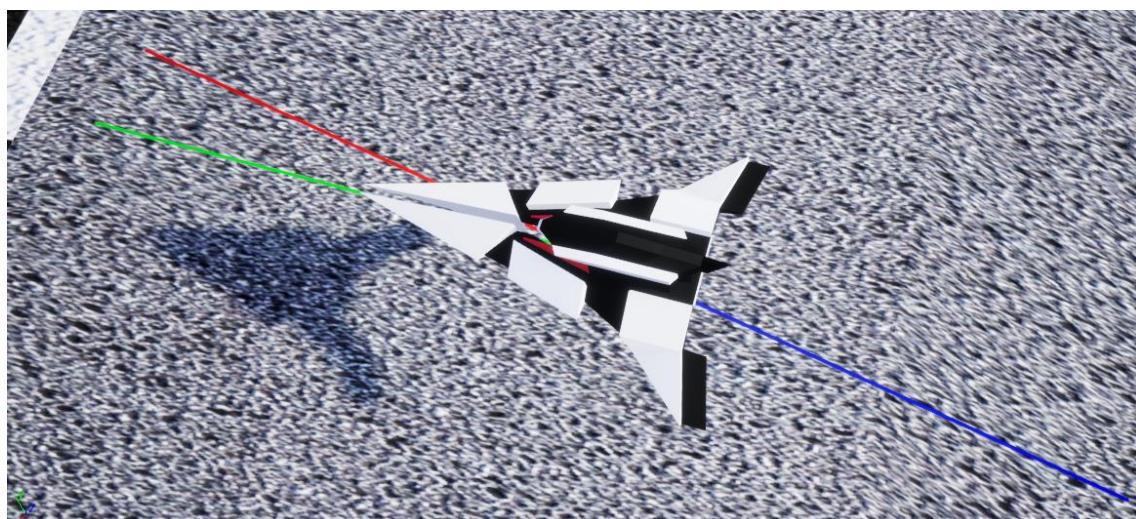


Ilustración 24. Vista representativa del alineamiento de velocidad (giro a izquierda). La línea roja representa la dirección de la velocidad actual. En azul está la fuerza a aplicar para frenar completamente la nave (desde el cuadro anterior). En verde se encuentra la dirección de la fuerza alineada a aplicar sobre la nave. Véase que la línea verde ahora se encuentra alineada con la punta de la nave.

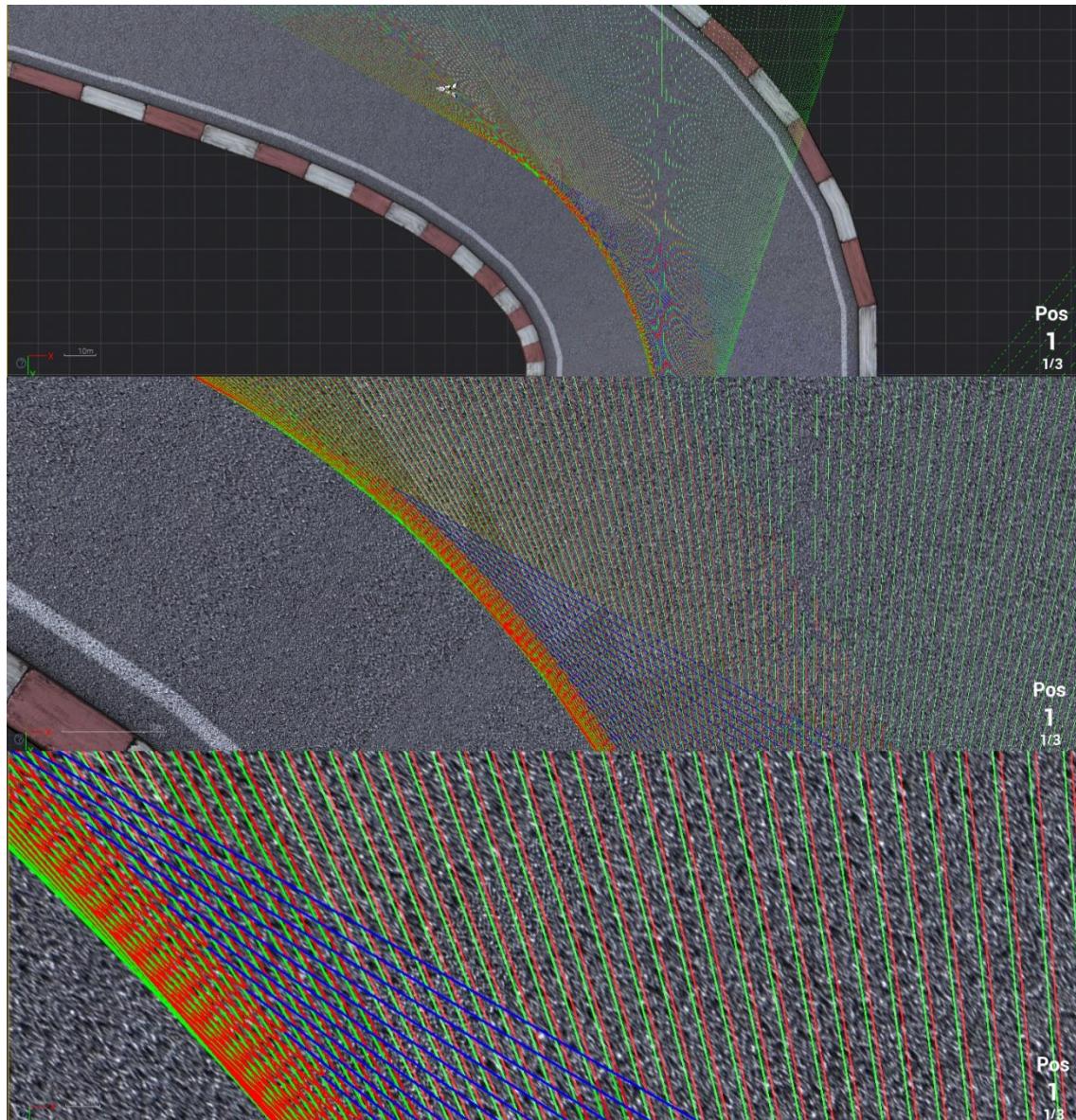


Ilustración 25. Vistas reales del alineamiento de velocidad (giro a izquierda) dentro de un laps. De arriba hacia abajo se muestra un acercamiento hacia las líneas. Las líneas en la izquierda de la última captura son más recientes que las de la derecha. Véase cómo en las de la derecha se nota un desalineamiento entre la dirección de la velocidad actual y la dirección de la fuerza que se aplica para alinear.

Aceleración, Frenado y Retroceso

Se toma el vector unitario hacia delante de **AJet**, se lo proyecta en el plano ortogonal al vector normal del piso de la pista y se lo multiplica por una fuerza que tendrá como aceleración la deseada para acelerar o retroceder. Para frenar, solamente se aplica la fuerza para retroceder. **UMotorDriveComponent** se encarga de realizar los cálculos anteriores.

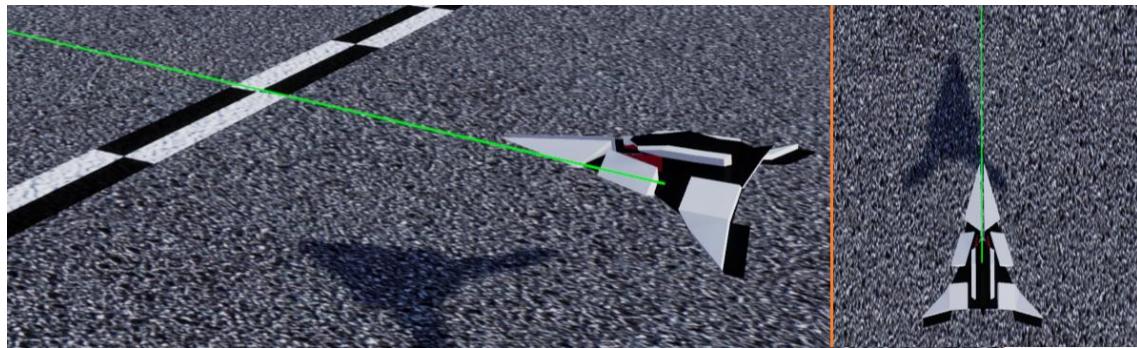


Ilustración 26. Vista de aceleración. La línea verde representa la dirección de la fuerza que se aplicará en el próximo cuadro a la nave para acelerarla.

Cálculo del Vector Normal al Piso

En cada cuadro, **AJet** realiza lo siguiente para actualizar su vector normal al piso (que se encuentra como su vector hacia arriba al inicio de su creación):

- Si existe una pista en el mundo en el que se encuentra, le pide al generador de pista que le dé el punto más cercano a la ubicación donde está **AJet**.
Realiza una proyección desde la ubicación de la nave hasta ese punto del generador de pista y obtiene el vector normal al choque, que será utilizado como el vector normal a la pista.
- Si no existe un generador de pista,
 - Proyecta hacia abajo hasta chocar con un piso. Toma el vector normal de ese piso como el vector normal a la pista.
 - Si no llega a chocar con nada, se utiliza el vector hacia arriba del jet como el vector normal a la pista.

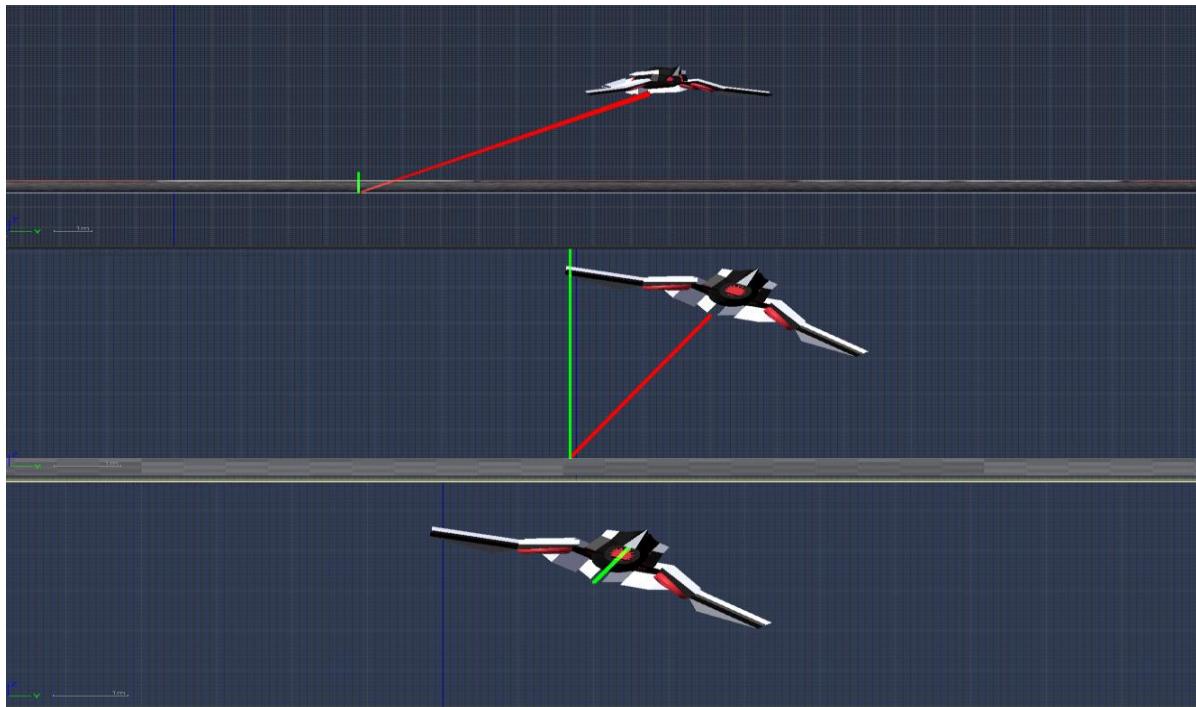


Ilustración 27. Vistas del cálculo normal al piso. De arriba hacia abajo se muestran los puntos del cálculo: a), b.1) y b.2). La línea roja representa la proyección realizada mientras que la verde representa el vector normal al piso conseguido.

Administradores de Estado

Se mencionó los componentes necesarios para la maniobrabilidad de **AJet**, pero eso no es lo único que se utiliza para lograrlo. Se construyeron dos actores que sirven para administrar el proceso de maniobrabilidad de una nave. Se agrega un agente intermedio que agrega una capa más de separación entre la nave y sus componentes. Los dos administradores utilizan el patrón de diseño State.

Estados del Motor

UMotorState representa los estados del motor de una nave y deriva en **UAcceleratingMotorState**, **UNeutralMotorState** y **UReversingMotorState** para representar aceleración, neutralidad (no acelera ni frena) y frenado respectivamente. Existe un cuarto estado que es **UMixedMotorState** y se utiliza para representar el caso en que se esté acelerando y frenando en el mismo instante.

AMotorStateManager se utiliza para administrar estos estados derivados de **UMotorState**. Una vez que se pide acelerar al administrador, este primero verifica que el estado actual no sea el pedido. Si el estado pedido no es el actual, cambia el estado por el deseado.

AJet en cada cuadro pide a **AMotorStateManager** que se active. Este a su vez le pide al **UMotorState** actual que se active (**UAcceleratingMotorState** llamaría al método *accelerate* del **UMotorDriveComponent**).

Estados de Giro

USteerState es la base para representar los estados de giro a la izquierda, derecha y centrado cuando no se está girando en ninguna dirección. **ULeftSteerState**, **URightSteerState** y **UCenterSteerState** son las clases que derivan para representar estos estados.

ASteerStateManager es la clase administradora de estos estados y actúa de la misma forma que **AMotorStateManager**. En cada cuadro, **AJet** le pide que se active.

Cuando un usuario quiere maniobrar un **AJet**, lo hace por medio de presionar y mantener un botón. Para acelerar, si la nave no tuviese administradores de estado, se tendría que pedir en todo momento que se acelere. En cambio, de esta forma solo se establece el estado una vez por cuadro y se active la aceleración en cada cuadro. Este método de administrar los estados de la nave será muy útil a la hora de manejar la replicación por medio de la red.

Replicación

Unreal Engine utiliza el modelo de cliente-servidor para comunicarse a través de la red. Esto quiere decir que solo el servidor tiene el estado real de las cosas y los clientes solo tienen una representación de la realidad.

Existe un grave problema para comunicar videojuegos entre la red: el tiempo que hay entre enviar una solicitud al servidor y recibir los resultados para mostrar al usuario hacen que la información enviada y recibida estén desfasadas temporalmente. Esto es mucho peor si se considera que varios clientes están conectados a un servidor y el servidor les envía información de los demás a cada uno. La información enviada estará desfasada porque al servidor le llega información que no es la actual del cliente. La información recibida también estará desfasada porque no representa el estado actual del servidor.⁷⁸

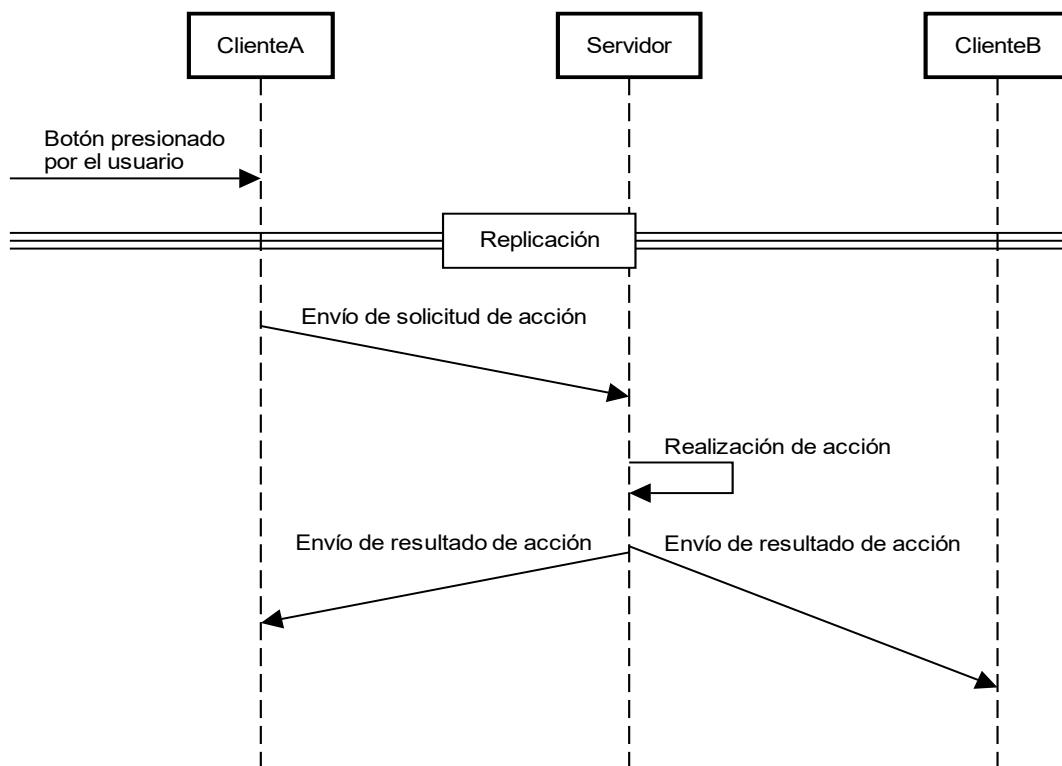


Ilustración 28. Replicación simple en videojuegos.

Para solucionar estos dos problemas, se recurre a dos métodos conocidos en la industria de los videojuegos como “Client-Side Prediction” y “Server Reconciliation”.⁷⁹

⁸⁰

Client-Side Prediction

En vez de que un cliente realice una solicitud a un servidor y espere la respuesta, lo que se hace es que el cliente simule la solicitud hasta que llegue la respuesta “real” del servidor.⁸¹

Server Reconciliation

Ya que la información recibida desde el servidor no es actual, lo que se hace desde el cliente cada vez que se quiere enviar algo al servidor es marcar cada solicitud y tener un registro de solicitudes enviadas. Cuando llega una respuesta desde el servidor, se mira la marca recibida, se aplica y se reaplican las marcas siguientes que se habían enviado y aún no tienen respuesta.

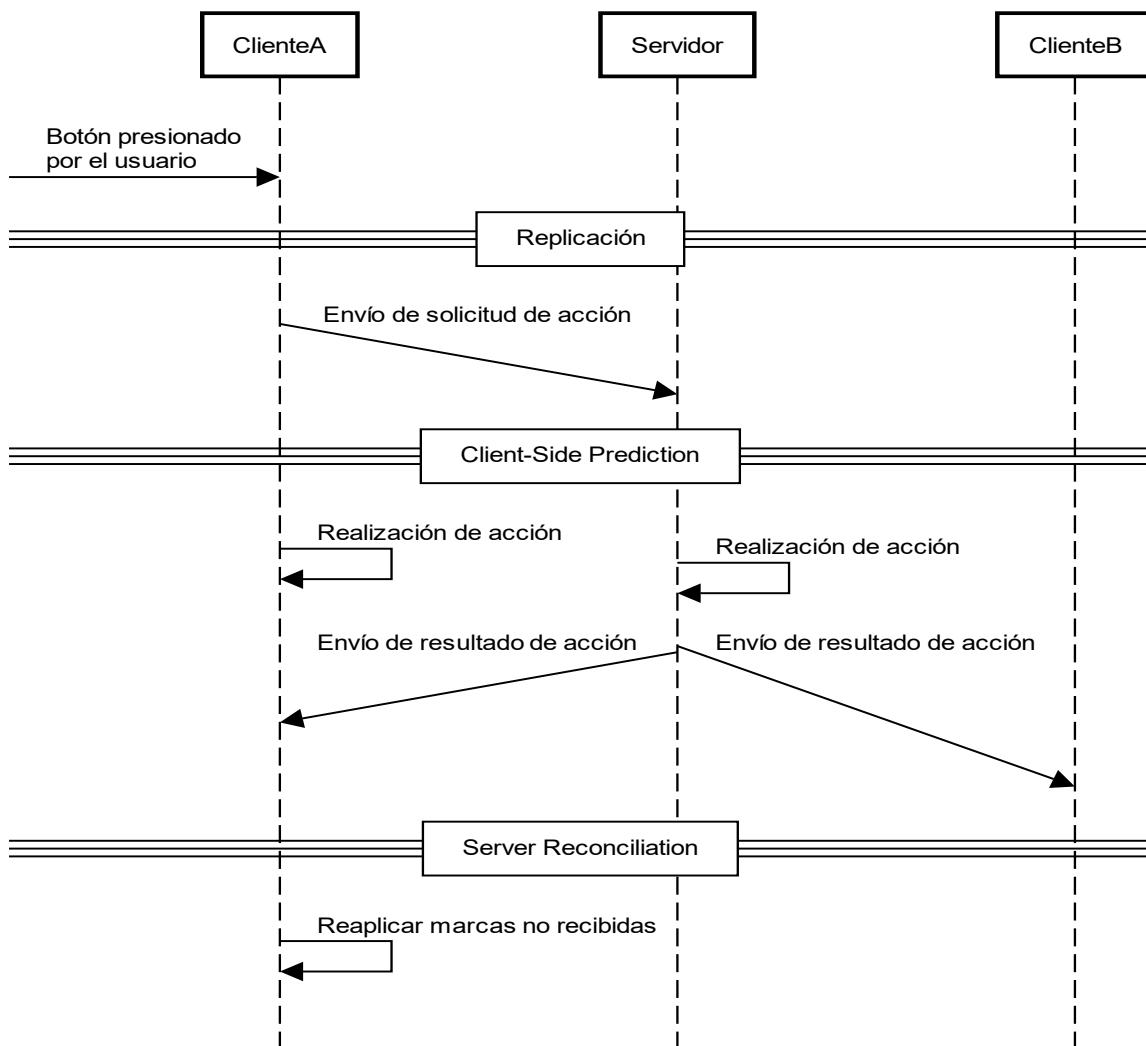


Ilustración 29. Replicación con Client-Side Prediction y Server Reconciliation.

Estos dos métodos solo se enfocan en el cliente, pero falta la parte en que el servidor recibe información atrasada por parte del cliente. Para ello, se explicará la solución utilizada en ProjectR para solventarlo, haciendo uso de marcas de tiempo.⁸¹

UDeloreanReplicationMachine

Es la clase que representa una máquina de replicación y se encarga de replicar **AJet** por la red. Cuenta con un sistema para preparar y recibir estados, así como también para actualizar un **AJet** a los valores actuales.

Cada **AJet** posee una instancia de ***UDeloreanReplicationMachine***, que tiene un registro del movimiento final en cada cuadro. Este movimiento tiene ubicación, rotación, velocidad lineal, velocidad angular, clase de estado del motor y clase de estado de giro, así como también una marca de tiempo en UTC.

Se puede modificar, pero por defecto se tiene un registro de los últimos 60 cuadros. Si se piensa que el juego mantiene unos 60 cuadros por segundo, se tendrá información del último segundo, que es lo mismo que decir que se tiene información de los últimos 1000 milisegundos. El registro es una lista doblemente enlazada que elimina de la cola cada vez que se agrega un nuevo movimiento.

Se eligió tener un registro de los últimos 1000 milisegundos para poder manejar casos de latencia alta.

En cada fin de cuadro, **AJet** pide a ***UDeloreanReplicationMachine*** que guarde el movimiento final.

A continuación, se mostrarán los tres pasos que se siguen para replicar un **AJet**.

Primer Paso de Replicación (Cliente)

Supongamos que un usuario en una partida en red acaba de presionar el botón de acelerar en una nave. Esta acción pasará por **AMotorStateManager**. Supongamos también que esta acción no representa el estado actual del administrador, por lo que el administrador cambiará el estado a **UAcceleratingMotorState**. Luego de hacer eso, el administrador comunicará a **AJet** que el motor sufrió un nuevo estado por lo que el **AJet** estará marcado para replicar el movimiento que guarde al final del cuadro.

Al final del cuadro le dirá a ***UDeloreanReplicationMachine*** que le prepare los estados actuales del motor y de giro guardados para enviarlos (los que acaban de ser guardados en este cuadro), por lo que la máquina de replicación tomará la marca de tiempo y estados del cuadro actual y los empaquetará en una estructura que devolverá a **AJet**. También, al movimiento actual lo marcará como movimiento de envío o recepción (los movimientos que no son para envío o recepción se los marca cada vez que se los crea como movimientos de rutina).

AJet recibirá esta estructura y se la enviará al servidor.

Segundo Paso de Replicación (Servidor)

El servidor recibirá el mensaje enviado desde la nave del cliente a la respectiva nave en su juego.

Dentro de su ***UDeloreanReplicationMachine*** buscará el movimiento con menor diferencia temporal con el de los estados recibidos (para ello utiliza sus propias marcas de tiempo guardadas en los movimientos).

Al movimiento encontrado le copia los estados recibidos. Cabe mencionar que, si llegase a recibirse un mensaje desfasado, el movimiento tiene una entrada especial de marca de tiempo remota. Cuando un mensaje nuevo llega, se fija que la marca de tiempo de ese mensaje sea posterior a la marca de tiempo remota guardada en el movimiento para evitar pisar los datos con datos obsoletos. Esto puede llegar a ocurrir ya que el servidor no tiene sincronizados los cuadros con los clientes, por lo que puede ser que dos movimientos de cuadros distintos del cliente terminen en el mismo movimiento guardado en el servidor.

Ahora se tiene el movimiento real. Los estados que deseaba ejecutar el cliente con la ubicación, rotación, velocidad lineal y velocidad angular “reales” (las del servidor). Se denominará este movimiento el Movimiento Real del Servidor ya que es el que será reenviado al cliente que maneja la nave y a todos los clientes que estén en la partida luego de que el servidor termine de actualizar sus datos.

Con el Movimiento Real del Servidor ya logrado, se puede empezar la resincronización de movimientos del lado del servidor. Lo que se hará es lo siguiente:

Desde el Movimiento Real del Servidor en adelante:

1. Se copiarán los estados del actual al siguiente hasta que el siguiente no sea un movimiento marcado como movimiento de rutina (puede ser que se haya recibido un mensaje más viejo que otros).
2. Se calculará el movimiento siguiente con los datos del movimiento actual.
3. Se alinearán los datos del movimiento que le sigue al siguiente si es que en el movimiento actual hubo un estado de giro diferente a **UCenterSteerState**.

Por último, se establecerá el movimiento más actual del registro (que fue recalculado) en **AJet**.

A partir de ese momento, el servidor se encuentra sincronizado. Luego, envía el Movimiento Real del Servidor al cliente y a los otros clientes que estén en la partida.

Tercer Paso de Replicación (Clientes)

Cada cliente recibe el Movimiento Real del Servidor, **UDeloreanReplicationMachine** buscará el movimiento con menor diferencia temporal con el recibido y realizará la sincronización de movimientos hasta su propio momento actual (del mismo modo que hizo el servidor). Así, el cliente estará sincronizado a los movimientos del servidor en ese momento.

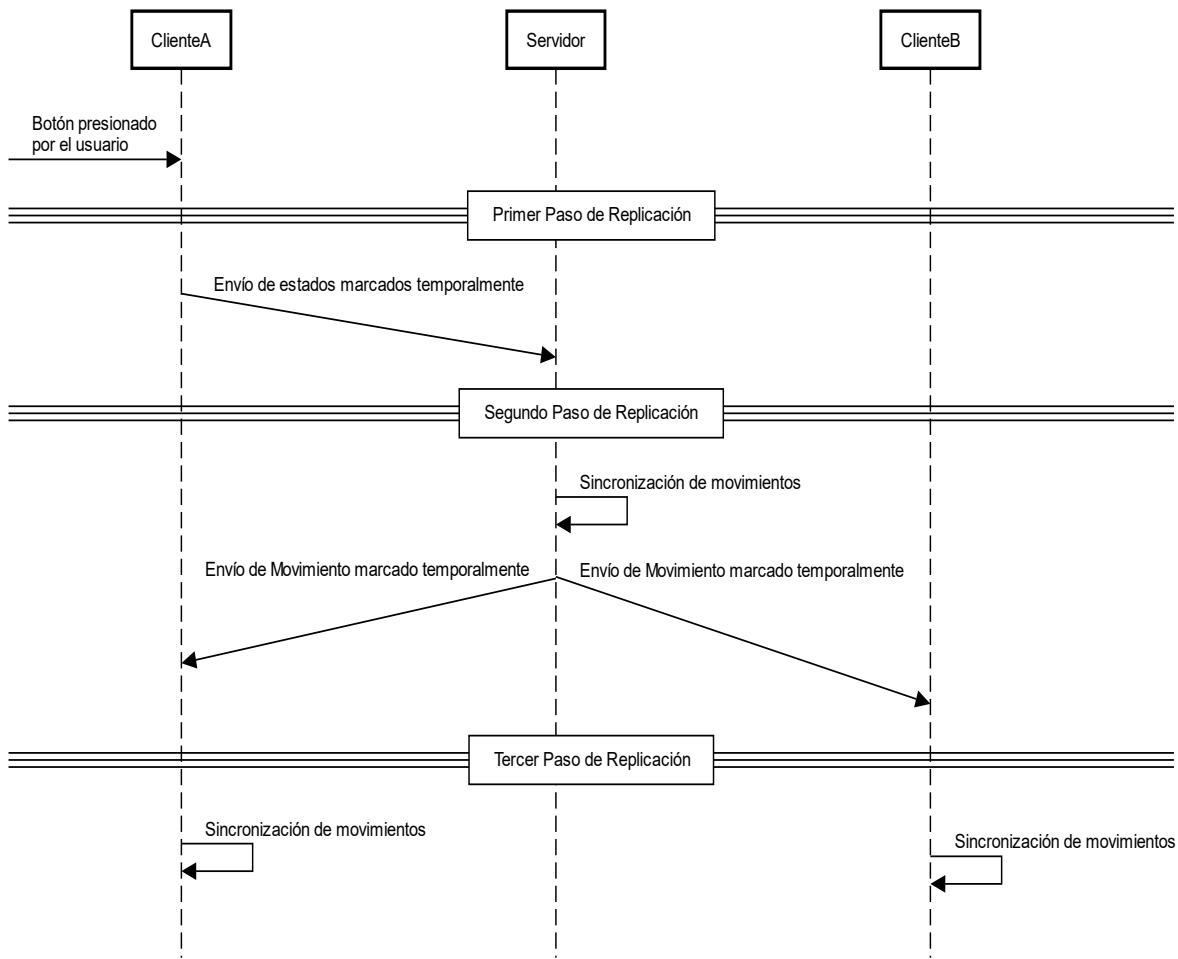


Ilustración 30. Pasos de replicación en **AJet**. ClienteA es el cliente que controla esta instancia de **AJet**.

De esta forma, el servidor y cliente se sincronizan separando las áreas en las que se enfocan. El cliente solo controla los estados que quiere. Solo puede enviar al servidor los estados y cuándo los realizó. Si no fuese así, un usuario podría poner una nave en distintas ubicaciones estratégicas y enviárselas al servidor para hacer trampa y ganar un juego.

El servidor solo controla los movimientos. Ubicación, rotación, velocidad lineal y angular le pertenecen y solo él tiene los datos reales de la partida.

Existe un pequeño asunto que puede ser que se haya pasado por alto mientras se leía estos pasos de replicación. Este es: ¿cómo es que se calcula el movimiento siguiente con los datos del actual?

Para ello, se tiene que simular cada cuadro realizado. Lamentablemente, Unreal Engine no tiene un método para simular un cuadro entero de todo el mundo en el que se esté. Como Unreal Engine 4 utiliza NVIDIA PhysX como motor de físicas, se pudo encontrar un método en sus extensiones que permite calcular el cambio en velocidad lineal y angular por aplicarle a un cuerpo una fuerza y torque por un determinado tiempo. Este método es `computeVelocityDeltaFromImpulse`. Entonces, lo único que debemos hacer es calcular los cambios que se produjeron en aceleraciones en cada cuadro por los componentes de **AJet** junto con los estados de los administradores de estado. De esta forma, cada estado tiene un método que permite saber los cambios que realiza a un **AJet** con los datos de ese momento.

Se debe mencionar que PhysX no es un motor de físicas determinístico, lo que provoca que los resultados de *computeVelocityDeltaFromImpulse* entre el cliente y servidor difieran ligeramente. Para solventar este problema, basta con suavizar los resultados obtenidos con el movimiento actual, por medio de interpolación, para disminuir cambios bruscos, si es que los hubiera.⁸² Esta aplicación de interpolación se deja como tema a abordar en el futuro. En ProjectR no se realiza esta suavización para mostrar la diferencia en simulación con los valores locales cuando se sincroniza.

Se bautizó la clase como ***UDeloreanReplicationMachine*** por ser similar su efecto al que provoca la máquina de tiempo en las películas de “Volver al Futuro”.

Problemas y Soluciones por Pruebas (y problemas de pruebas)

En esta sección se mostrarán algunos problemas que se tuvieron que resolver durante el Desarrollo y cómo se llegó a implementar la solución mediante pruebas, así como también qué problemas surgieron al tratar de resolver por pruebas ya que el sistema de automatización junto con la documentación de Unreal Engine son ambiguos.

Ticks, Mundos y Fuerzas

Supongamos que queremos acelerar nuestra nave (**AJet**). Para ello deberíamos realizar una prueba, dentro de ella crear una instancia de **AJet**, hacer que acelere y verificar que verdaderamente aceleró nuestra nave. Suponemos en este caso que ya se hicieron las pruebas pertinentes para agregar un componente de **AJet** que pueda ser afectado por físicas. Supongamos también que se quiere probar la aceleración a lo largo del eje X y que queremos una aceleración positiva para simplificar las cosas.

La prueba se vería de esta forma:

```
bool FAJetMovesForwardWhenAcceleratedTest::RunTest(const FString& Parameters)
{
    AJet* testJet = NewObject<AJet>();
    float initialLocation = testJet->GetActorLocation().X;
    testJet->accelerate();
    float finalLocation = testJet->GetActorLocation().X;
    TestTrue(TEXT("Jet X location should increase when accelerated."), finalLocation > initialLocation);
    return true;
}
```

`accelerate` está compuesta por:

```
AJet::accelerate()
{
    physicsMeshComponet->AddForce(FVector(5000, 0, 0));
}
```

Parecería que funcionaría, pero uno se estaría olvidando de una cosa: se está creando un objeto en vez de hacerlo aparecer en un mundo, por lo que por más que lo aceleremos, no estaría moviéndose en un espacio. Entonces, debemos primero cargar en el editor un mapa y luego hacer que aparezca una instancia de **AJet** en él:

```
bool FAJetMovesForwardWhenAcceleratedTest::RunTest(const FString& Parameters)
{
    ADD_LATENT_AUTOMATION_COMMAND(FEditorLoadMap(FString("/Game/Tests/TestMaps/VoidWorld")));
    AJet* testJet = Cast<AJet, AActor>( GEditor->GetPIEWorldContext()->World()
    ->SpawnActor(AJet::StaticClass()) );
    float initialLocation = testJet->GetActorLocation().X;
    testJet->accelerate();
    float finalLocation = testJet->GetActorLocation().X;
    TestTrue(TEXT("Jet X location should increase when accelerated."), finalLocation > initialLocation);
    return true;
}
```

FEditorLoadMap es un comando latente proporcionado por Unreal Engine 4.

Mejor, pero existen otros dos problemas: cargar un mapa en el editor y hacer aparecer un actor en un mapa son acciones asincrónicas (ocurren en cuadros/ticks distintos). Por eso, hay que separar el código en pequeños comandos latentes:

```
bool FAJetMovesForwardWhenAcceleratedTest::RunTest(const FString& Parameters)
{
    ADD_LATENT_AUTOMATION_COMMAND(FEditorLoadMap(FString("/Game/Tests/TestMaps/VoidWorld")));
    ADD_LATENT_AUTOMATION_COMMAND(FSpawnJet);
    ADD_LATENT_AUTOMATION_COMMAND(FRetrieveAccelerateAndCheckJet(this));
    return true;
}
```

```
DEFINE_LATENT_AUTOMATION_COMMAND(FSpawnJet);
bool FSpawnJet::Update()
{
    //crea una nave en la posición (X = 0, Y = 0, Z = 0).
    GEditor->GetPIEWorldContext()->World()->SpawnActor(AJet::StaticClass())
    return true;
}
```

```

DEFINE_LATENT_AUTOMATION_COMMAND_ONE_PARAMETER(FRetrieveAccelerateAndCheckJet,
FAutomationTestBase*, test);

bool FRetrieveAccelerateAndCheckJet::Update()
{
    UWorld* testWorld = GEditor->GetPIEWorldContext()->World();

    AJet* testJet = Cast<anActorDerivedClass, AActor>(UGameplayStatics::GetActorOfClass(testWorld,
AJet::StaticClass()));

    if(testJet)
    {
        float initialLocation = testJet->GetActorLocation().X;

        testJet->accelerate();

        float finalLocation = testJet->GetActorLocation().X;

        test->TestTrue(TEXT("Jet X location should increase when accelerated."), finalLocation >
initialLocation);

        return true;
    }
    return false;
}

```

Se agrega un parámetro al comando latente para ser capaz de llamar a la instancia de la prueba en la verificación.

Pero existe aún otro error y ese es que cuando se carga un mapa a un editor, solo se carga estáticamente, nada se mueve dentro. Solo se tiene objetos guardados, como si estuviesen congelados en el mapa.

Lo que se debe hacer es pedir al editor que cree una sesión de juego, hacer la prueba y luego cerrar la sesión de juego:

```

bool FAJetMovesForwardWhenAcceleratedTest::RunTest(const FString& Parameters)
{
    ADD_LATENT_AUTOMATION_COMMAND(FEditorLoadMap(FString("/Game/Tests/TestMaps/VoidWorld")));
    ADD_LATENT_AUTOMATION_COMMAND(FStartPIECommand(true));
    ADD_LATENT_AUTOMATION_COMMAND(FSpawnJet);
    ADD_LATENT_AUTOMATION_COMMAND(FRetrieveAccelerateAndCheckJet(this));
    ADD_LATENT_AUTOMATION_COMMAND(FEndPlayMapCommand);
    return true;
}

```

FStartPIECommand y FEndPlayMapCommand son comandos latentes provistos por Unreal Engine 4.

También se deben actualizar los comandos latentes para esperar a que el editor cree la sesión de juego:

```

bool FSpawnJet::Update()
{
    if (GEditor->IsPlayingSessionInEditor())
    {
        //crea una nave en la posición (X = 0, Y = 0, Z = 0).
        GEditor->GetPIEWorldContext()->World()->SpawnActor(AJet::StaticClass())
        return true;
    }
    return false;
}

bool FRetrieveAccelerateAndCheckJet::Update()
{
    if (GEditor->IsPlayingSessionInEditor())
    {
        UWorld* testWorld = GEditor->GetPIEWorldContext()->World();
        AJet* testJet = Cast<anActorDerivedClass,
        AActor>(UGameplayStatics::GetActorOfClass(testWorld, AJet::StaticClass()));
        if(testJet)
        {
            float initialLocation = testJet->GetActorLocation().X;
            testJet->accelerate();
            float finalLocation = testJet->GetActorLocation().X;
            test->TestTrue(TEXT("Jet X location should increase when accelerated."), finalLocation > initialLocation);
            return true;
        }
    }
    return false;
}

```

Ahora ya se pensaría que todo funciona como se esperaría, pero existe un último error: Cuando uno acelera, le pide al motor de físicas que agregue una fuerza y una fuerza se ve aplicada recién en el cuadro siguiente al que se pide, por lo que debería rearmanse FRetrieveAccelerateAndCheckJet.

También, la verificación tiene que cambiarse, ya que se hará entre cuadros. Para eso, conviene tener un comando latente con otro parámetro más que sea un número, donde en cada cuadro se verificará que la posición de la nave sea mayor a la que se encuentra y también se guarde el valor nuevo luego de verificar:

```
DEFINE_LATENT_AUTOMATION_COMMAND_TWO_PARAMETER(FRetrieveAccelerateAndCheckJet, float,
previousXValue, FAutomationTestBase*, test);

bool FRetrieveAccelerateAndCheckJet::Update()
{
    if (GEditor->IsPlayingSessionInEditor())
    {
        UWorld* testWorld = GEditor->GetPIEWorldContext()->World();
        AJet* testJet = Cast<AJet, AActor>(UGameplayStatics::GetActorOfClass(testWorld, AJet::StaticClass()));
        if(testJet)
        {
            testJet->accelerate();
            float initialLocation = previousXValue;
            float finalLocation = testJet->GetActorLocation().X;
            bool hasMovedForward = finalLocation > initialLocation;
            if(hasMovedForward)
            {
                test->TestTrue(TEXT("Jet X location should increase when accelerated."), hasMovedForward);
                return true;
            }
            previousXValue = finalLocation;
        }
    }
    return false;
}
```

Para llamarlo en la prueba, el parámetro del comando latente que representa el valor de la posición en X se debería enviar con valor cero. Esto es porque cuando se aparece la instancia de **AJet** en el mundo de prueba, se lo ubica en $(X, Y, Z) = (0, 0, 0)$:

```
bool FAJetMovesForwardWhenAcceleratedTest::RunTest(const FString& Parameters)
{
    ADD_LATENT_AUTOMATION_COMMAND(FEditorLoadMap(FString("/Game/Tests/TestMaps/VoidWorld")));
    ADD_LATENT_AUTOMATION_COMMAND(FStartPIECommand(true));
    ADD_LATENT_AUTOMATION_COMMAND(FSpawnJet);
    ADD_LATENT_AUTOMATION_COMMAND(FRetrieveAccelerateAndCheckJet(0, this));
    ADD_LATENT_AUTOMATION_COMMAND(FEndPlayMapCommand);
    return true;
}
```

Esta implementación acelera en cada cuadro. Se podría separar en más comandos latentes para solamente acelerar una vez en toda la prueba.

En este caso, se podría intentar la prueba en cada cuadro infinitamente ya que no hay ningún mecanismo que pare la prueba si es que nunca acelera la nave. Se podría agregar al comando latente de verificación otro parámetro que cuente los cuadros que ya pasaron (similar al ejemplo dado para comandos latentes unas secciones más arriba, en la página 29).

De esta forma, se tiene una prueba que verifica que llamar al método `accelerate` de **AJet** hace acelerar a una nave en dirección X.

Como se ve, una prueba en Unreal Engine requiere una gran preparación. Se pueden crear clases derivadas de pruebas, que faciliten y parametrizan esta preparación.

Se debe tener especial atención a métodos que sean asincrónicos ya que su presencia hace tener que modificar pruebas que deberían ser simples.

Clics en Editor

Para probar clases de UI, es primordial tener la capacidad de hacer clics en el editor, lo que permite verificar el correcto funcionamiento de elementos con los que un usuario puede interactuar.

El problema es que Unreal Engine no expone de manera ordenada su interfaz, por lo que uno está constantemente buscando entre archivos qué método usar para lograr lo que desea.

Ahora, un clic es una acción llevada a cabo por el hardware y entre el clic procesado por el sistema operativo y la llegada a la UI de un programa tiene que haber una interfaz que procese todos estos eventos. Para Unreal Engine, esta interfaz es `FSlateApplication`, que se encarga de administrar eventos de UI así como también todas las ventanas del editor.⁸³

Como un clic es un evento, debe haber un método que lo procese. El método que lo hace es: `ProcessMouseButtonDoubleClickEvent`. Sus parámetros son:

```
bool ProcessMouseButtonDoubleClickEvent( const TSharedPtr< FGenericWindow >& PlatformWindow,
                                         const FPointerEvent& InMouseEvent );
```

Por lo que necesita una ventana en donde se produce el clic y el evento de clic que se realizó.

La ventana se consigue creando una ventana genérica, FGenericWindow que se comunicará con el sistema operativo para obtener el espacio en el que puede extenderse.

FPointerEvent se instancia con:

```
FPointerEvent(
    uint32 InPointerIndex,
    const FVector2D& InScreenSpacePosition,
    const FVector2D& InLastScreenSpacePosition,
    const TSet<FKey>& InPressedButtons,
    FKey InEffectingButton,
    float InWheelDelta,
    const FModifierKeysState& InModifierKeys
);
```

Generalmente, se puede instanciar de esta forma:

```
FSlateApplication& slateApplication = FSlateApplication::Get();
const TSet<FKey> pressedButtons = TSet<FKey>({ EKeys::LeftMouseButton });
FPointerEvent mouseMoveAndClickEvent(
    0,
    slateApplication.CursorPointerIndex,
    atCoordinates,
    FVector2D(0, 0),
    pressedButtons,
    EKeys::LeftMouseButton,
    0,
    slateApplication.GetPlatformApplication()->GetModifierKeys()
);
```

atCoordinates van a ser las coordenadas en (X, Y) donde se quiere hacer el clic, por lo que para cada elemento de UI se debe tener un método que retorne sus coordenadas absolutas (ya que FSlateApplication controla toda la ventana general del programa).

Se creó un método general para poder controlar clics en pruebas:

```

void processEditorClick(FVector2D atCoordinates)
{
    FSlateApplication& slateApplication = FSlateApplication::Get();
    const TSet<FKey> pressedButtons = TSet<FKey>({ EKeys::LeftMouseButton });
    FPointerEvent mouseMoveAndClickEvent(
        0,
        slateApplication.CursorPointerIndex,
        atCoordinates,
        FVector2D(0, 0),
        pressedButtons,
        EKeys::LeftMouseButton,
        0,
        slateApplication.GetPlatformApplication()->GetModifierKeys()
    );
    TSharedPtr<FGenericWindow> genericWindow;
    bool mouseClicked = slateApplication.ProcessMouseButtonDoubleClickEvent(genericWindow,
        mouseMoveAndClickEvent);
}

```

Lo complejo fue encontrar el método para procesar los clics de botones y que no se supiera cómo debían configurarse los parámetros para que funcione como se esperaba. Al lograr la configuración correcta, es un método sencillo, pero el autor hubiera preferido que haya indicaciones de cómo manipular estos métodos ocultos.

Pulsación de Botones

Suponga que quiere probar que, al presionar un botón, un **AJet** debiera acelerar.

Uno pensaría que solamente habría que tomar un control, hacerlo poseer una instancia de **AJet** y llamar un método que procese entradas dentro del control.

Pareciera lo más lógico, pero en realidad, faltaría un paso más y ése es que se necesita hacer aparecer un jugador que controle un control y éste a su vez posea una instancia de **AJet**. Para lograrlo, se debe obtener el modo de juego y pedirle `SpawnPlayerFromSimulate`, que hará exactamente los pasos requeridos (solo recibe como parámetro la ubicación donde se quiere hacer aparecer al jugador). El problema es que entonces se debe configurar el mundo en el que hace la prueba para que instancie el tipo de control y el objeto que vaya a poseer (que será un **AJet**).

Luego de eso, basta con conseguir la instancia del control creado, verificar que posee una instancia de **AJet** y pedirle `InputKey`:

```
virtual bool InputKey(FKey Key, EInputEvent EventType, float AmountDepressed, bool bGamepad);
```

`EInputEvent` especifica qué tipo de acción se quiere realizar con esa “tecla”, presionar, mantener apretado, soltar o que simule un clic (por si se controla una UI por un control).

Pruebas de Replicación

Requerimientos

Para probar la replicación, primero hay que tener en cuenta qué es lo que se necesita para ello.

Para replicación intervienen por lo menos dos entidades: un servidor y un cliente, así como también la conexión entre ellos. Un servidor puede ser cliente al mismo tiempo, pero se perdería el sentido en probar de esa forma, por lo que tienen que estar separados uno de otro.

Entonces lo más normal sería ejecutar dos instancias del editor (una para ser servidor y otra para ser cliente), conectarlas y probar la replicación entre ellas.

El problema es que, para automatizar, se debería esperar al otro programa que se inicie y conecte a nuestro programa servidor.

Por suerte, el editor de Unreal Engine tiene una opción para iniciar una sesión con varios jugadores conectados. El problema es que no está documentada la forma para crear una sesión personalizada.

Búsqueda del Código Necesario

El autor trató varias formas de encontrar el código que se debía ejecutar para lograrlo. La forma en que pudo encontrar el código es la siguiente:

Muchos elementos del editor tienen carteles que aparecen si uno se sitúa sobre un elemento y espera un momento. Esos carteles son un objeto que se le agrega una cadena de caracteres. Esos caracteres tienen que ser escritos en algún archivo para que sean tomados por el objeto del cartel.

Entonces, como el botón para crear sesiones se encuentra en una barra de herramientas para jugar la sesión, se buscó un nombre de archivo similar. Se encontró el archivo “LevelEditorToolBar.cpp”. Dentro de él se encontró una referencia a otro archivo llamado “DebuggerCommands.h”. Desde ahí, se pudo encontrar un método para poder crear una sesión personalizada. Ese método es:

```
void RequestPlaySession(const FRequestPlaySessionParams& InParams);
```

Que está declarado y pertenece a **UEditorEngine** y es el motor del editor.⁸⁴

Al método se accede gracias a **UUnrealEdEngine**⁸⁵, que es una clase derivada de **UEditorEngine**. Se accede a ella comúnmente utilizando GUnrealEd.

FRequestPlaySessionParams⁸⁶ es una estructura que posee los parámetros para crear una sesión de juego. Entre ellos, ULevelEditorPlaySettings⁸⁷ es la clase que permite especificar el número de instancias que se desean y el tipo de sesión (offline, servidor dedicado o servidor como cliente).

Sesiones Personalizadas en Pruebas

Ahora, con estos métodos se puede crear una sesión de juego con varios clientes dentro de una misma instancia del programa (los clientes y servidores se comunicarán por la red, enviando y recibiendo desde localhost).

Existe un pequeño inconveniente: ¿Cómo se hace para distinguir entre servidor y clientes?

Hay que recordar que solo el servidor tiene todos los controles de los clientes, mientras que cada cliente solo tiene su control. De esta forma se puede saber qué es un cliente y qué es un servidor, pero ¿cómo es que se logra obtenerlos?

Cuando el editor crea una sesión de juego, registra la sesión por medio del contexto del mundo en el que se va a jugar. Solo basta con pedirle al editor los contextos:

```
GEeditor->GetWorldContexts();
```

Teniendo cada contexto, ahora se puede saber la cantidad de controles pidiendo el mundo del contexto y a este pedirle la cantidad de controles presentes.

Para saber si el mundo encontrado es el del servidor, solo basta con verificar que esa cantidad sea la misma que la cantidad de jugadores especificada para crear la sesión de juego personalizada.

De esta forma tenemos también acceso directo a cada mundo y se puede pedir a actores replicados que realicen alguna acción en algún cliente y verificar que se replica en el servidor y/o los demás clientes.

Como puede verse, realizar pruebas en Unreal Engine requiere de antemano una gran configuración y conocimiento del funcionamiento del motor. Esto se podría reducir si por defecto existiesen más comandos latentes de parte del motor para personalizar las pruebas, además de mejorar la documentación que se ofrece.

Utilización del Entorno

En esta sección, se hablará de los tiempos de cuadro del juego durante una carrera, a qué característica está limitado (CPU o GPU) y en qué áreas se podría mejorar. También se mostrará el uso de la red por parte del videojuego, enfocándose en los métodos que se envían por ella y que son usados para replicar **AJet**.

Unreal Engine 4 posee un detallado sistema de análisis de rendimiento de software (profiling) dentro de su editor. Permite obtener datos en tiempo real de distintas secciones del motor, así como también grabar datos en un lapso.⁸⁸ Para habilitar estos análisis, se abre la consola de comandos y se especifica qué se quiere analizar.⁸⁹

Se crearon dos mediciones, una para una carrera offline y otra utilizando la red. Las mediciones se tomaron creando una sesión del juego, eligiendo el modo de juego “Singleplayer” y mapa “FlatPlayground”.

Limitación del cuadro de Imagen

Dentro de la carrera se accedió a la consola del editor para utilizar el comando `stat UNIT`, que muestra los tiempos necesarios para cada unidad que compone el cuadro de imagen (para CPU son threads, memoria principal, etc.; para el GPU son la memoria dedicada, procesamiento de triángulos/vértices/píxeles, etc.). El cuadro está limitado por una unidad y es en la que se debe enfocar para mejorar el rendimiento. Luego de mejorarla, se debe correr el análisis para develar otros cuellos de botella en el programa.

`stat UNIT` separa el cuadro en tres unidades: Game (thread principal del juego en CPU), Draw (thread de renderizado del CPU) y GPU.



Ilustración 31. Uso de `stat UNIT` dentro de una carrera. Frame (cuadro): 16,67ms; Game: 5,52ms; Draw: 4,49ms; GPU 16, 62ms

Para tener una mejor idea, se utilizó `startFPSChart` y `stopFPSChart` para obtener estas unidades en un lapso.

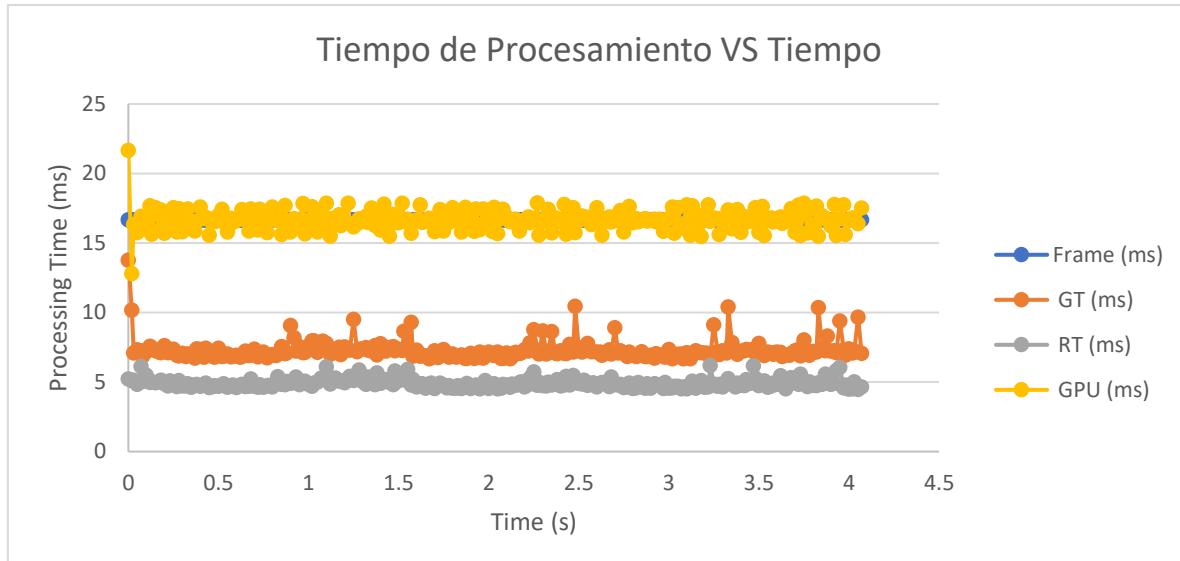


Ilustración 32. Uso de `startFPSChart` y `stopFPSChart` para tener un gráfico de FPS. GT: Game Thread. RT: Render Thread (Draw).

Como se puede ver en la ilustración, lo que más limita al cuadro de imagen es el uso de GPU, que es casi el mismo que el del cuadro mientras que los tiempos de CPU son mucho menores.

Entonces, lo que se debe mejorar es la unidad de GPU en el juego.

Análisis de GPU

Para realizar el análisis en tiempo real, se usa el comando `stat GPU`, que mostrará las distintas etapas del GPU y cuánto tarda cada etapa. Esto nos dice en qué etapas se debería optimizar el juego.



Ilustración 33. Stat GPU en acción.

Guiándonos por la imagen y los datos promedio y máximos, las etapas que requieren mayor tiempo de GPU son: *Shadow Depths*, *VisibilityCommands*, *ScreenSpace AO*, *ScreenSpace Reflections* y *SlateUI*.

VisibilityCommands pareciera ser que es lo que más tiempo consume. En realidad, es lo que el GPU espera a otras etapas de GPU.

Limitando los cuadros a 10 por segundo (usando `t.MaxFPS 10` en la consola) se puede reducir `VisibilityCommands` para exponer las etapas que consumen mayor tiempo en GPU.

GPU STAT GROUP_GPU	Counters	Average	Max	Min
[TOTAL]		8.40	12.56	6.58
Slate UI		0.81	0.93	0.76
Postprocessing		0.80	3.25	0.66
VisibilityCommands		2.97	4.31	2.04
ScreenSpace AO		0.47	0.50	0.46
Basepass		0.32	0.41	0.30
Shadow Projection		0.37	3.38	0.19
TAA		0.34	1.52	0.31
Atmosphere		0.20	1.79	0.27
Translucent Lighting		0.28	3.32	0.23
Lights		0.23	0.24	0.23
Shadow Depths		0.83	3.72	0.40
Reflection Environment		0.12	0.15	0.12
ScreenSpace Reflections		0.08	0.09	0.08
H2B		0.08	0.18	0.08
RenderDeferredLighting		0.05	0.05	0.05
ScreenSpace AO Setup		0.03	0.03	0.03
Prepass		0.03	0.03	0.03
Unaccounted		0.04	1.92	0.02
SortLights		0.02	0.03	0.02
FrameRenderFinish		0.14	2.48	0.00
Translucency		0.03	1.37	0.01
GPUSceneUpdate		0.01	0.01	0.01
Composition PreLighting		0.01	0.04	0.01
Render Velocities		0.00	0.00	0.00
[6 more stats. Use the stats.MaxPerGroup CVar to increase the limit]				

Ilustración 34. Stat GPU con `t.MaxFPS` establecido en 10.

Con el Cuadro corregido, se puede ver que ahora aparecen dos etapas que consumen más tiempo: `PostProcessing` y `Basepass`.

Utilizando `stat startfile` y `stat stopfile` se creó un historial de estadísticas del juego mientras se corría una carrera.

Gracias a la herramienta Unreal Frontend⁹⁰ se puede ver todas las estadísticas guardadas y filtrar según se necesite.

Ahora que se tiene las etapas que requieren un análisis, se utiliza el analizador de Unreal Frontend para filtrarlas.

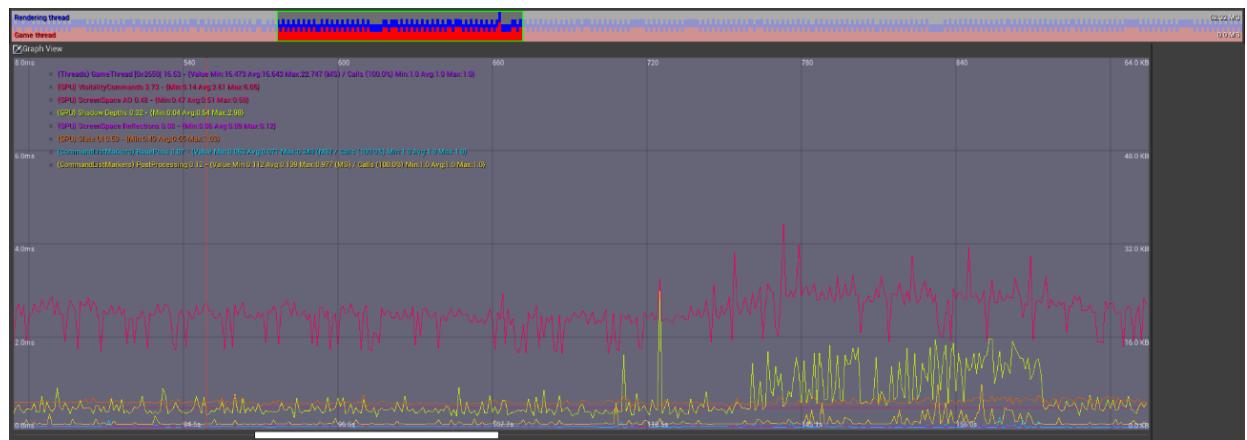


Ilustración 35. Session Frontend en la ventana Profiler.

Si se dejan las estadísticas más importantes:

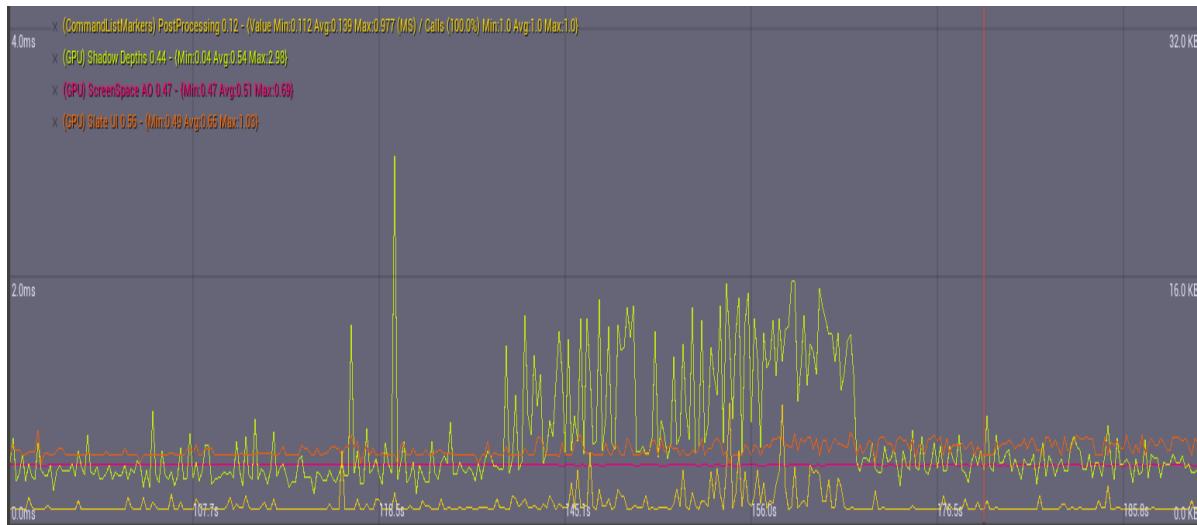


Ilustración 36. Gráfica Tiempo de procesamiento VS Tiempo. Amarillo: ShadowDepths. Mostaza: PostProcessing. Naranja: Slate UI. Fucsia: ScreenSpace AO.

Se puede ver claramente que lo que más tiempo consume son las etapas de ShadowDepths y PostProcessing. PostProcessing se encarga de aplicar efectos visuales luego de procesar las otras etapas (como agregar movimiento borroso, filtros de colores, etc.). Es fácil de modificar ya que solo se tiene que habilitar/deshabilitar características.⁹¹

ShadowDepths es la etapa que se encarga de proyectar sombras dinámicas que se aplican en cuerpos móviles.⁹²

Una advertencia que se tuvo cuando se creaba el ejecutable del videojuego fue que **ATrackGenerator** era un actor muy grande y estaba establecido como móvil. El editor sugería deshabilitar el uso de sombras dinámicas en él (reemplazarlas por estáticas). Como el generador de pistas no se mueve una vez que se inicia una carrera, lo mejor es establecerlo como un objeto estático en el editor. Las fases de vuelta también sufren el mismo problema que el generador de pistas, por lo que se debería también establecerlas como estáticas.

Otro problema relacionado con esto puede ser que los mapas UV de las texturas (matriz en coordenadas (u, v)) se solapen entre ellos, generando un procesamiento innecesario de texturas que no se verán. Ése ya es un problema con texturas en programas de modelado, como Blender.^{93 94}

Otra mejora también puede ser generar niveles de detalle (LOD en inglés) para cada objeto. Los niveles de detalle son distintos modelos de un objeto que se van cargando a medida que uno se aleja de un objeto, teniendo cada modelo cargado menor cantidad de vértices para representarlo (lo que reduce su procesamiento).⁹⁵

Análisis de CPU

Ahora se abordará el análisis de CPU utilizando el mismo historial de estadísticas del juego que se usó en la etapa de análisis de GPU.

Para ello, primero se utilizó el comando `stat GAME` en la consola del editor mientras se corría la carrera.⁹⁶

	CallCount	InclusiveAvg	InclusiveMax	ExclusiveAvg	ExclusiveMax
World Tick Time	2	1.06 ms	1.42 ms	0.09 ms	0.28 ms
Actor BeginPlay	6	0.79 ms		0.01 ms	0.03 ms
GT Tickable Time	2	0.08 ms	0.26 ms	0.00 ms	0.03 ms
PlayerController Tick	1	0.08 ms	0.16 ms	0.00 ms	0.00 ms
MoveComponent(Primitive) Time	1	0.07 ms	0.21 ms	0.00 ms	0.01 ms
Blueprint Time	1	0.03 ms	0.07 ms	0.02 ms	0.06 ms
TickableGameObjects Time	3	0.05 ms	0.24 ms	0.04 ms	0.23 ms
Update Camera Time	2	0.04 ms	0.11 ms	0.03 ms	0.08 ms
UpdateOverlaps Time	1	0.04 ms	0.11 ms	0.00 ms	0.00 ms
Queue Ticks	2	0.04 ms	0.08 ms	0.04 ms	0.08 ms
PerformOverlapQuery Time	2	0.04 ms	0.10 ms	0.01 ms	0.01 ms
Post Tick Component Update	4	0.03 ms	0.07 ms	0.01 ms	0.03 ms
Nav Tick Time	2	0.01 ms	0.03 ms	0.01 ms	0.02 ms
Transform or Render Data	7	0.01 ms	0.03 ms	0.00 ms	0.02 ms
Camera Process Viewfrustum	1	0.01 ms	0.01 ms	0.01 ms	0.01 ms
Blueprint Latent Actions	5	0.00 ms	0.01 ms	0.00 ms	0.01 ms
SpawnActor	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
GC Sweep Time	1	0.00 ms	0.02 ms	0.00 ms	0.00 ms
Teleport Time	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
World Tick Time	2	0.00 ms	0.00 ms	0.00 ms	0.00 ms
Reset Async Trace Time	1	0.00 ms	0.02 ms	0.00 ms	0.02 ms
MoveComponent(SceneComp) Time	1	0.00 ms	0.02 ms	0.00 ms	0.02 ms
Finish Async Trace Time	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
Net Broadcast Tick Time	2	0.00 ms	0.00 ms	0.00 ms	0.00 ms
[7 more stats. Use the stats.MaxPerGroup CVar to increase the limit]					
Counters		Average	Max	Min	
Ticks Queued		28.00	28.00	28.00	
TimerManager Heap Size		4.00	4.00	4.00	

Ilustración 37. Cuadro generado por `stat GAME`.

Se puede ver en el cuadro que lo que más consume en el CPU es el tick, lo que se hace en cada cuadro en el código. Esto es lógico ya que en todo momento se está recibiendo entradas del usuario y maniobrando el **AJet**.

Utilizando el Session Frontend y filtrando las estadísticas que más consumen en hilo del juego, se llegó a que lo que más consume son las proyecciones y físicas.

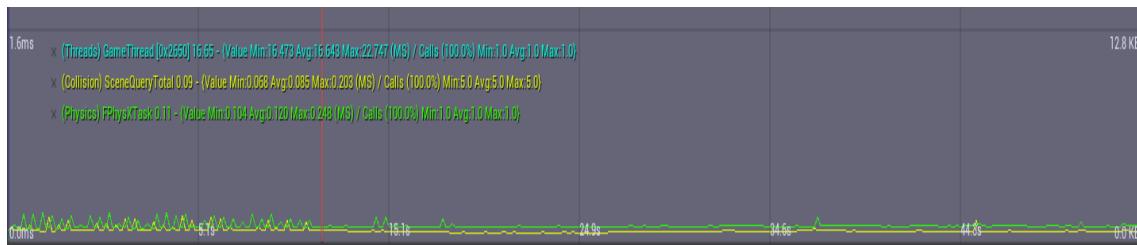


Ilustración 38. Gráfica Tiempo de Procesamiento (ms) VS Tiempo (s) en un juego local. Amarillo: Proyecciones. Verde: Procesamiento de Físicas. Se puede ver el poco tiempo necesario para procesar físicas y proyecciones en contraste con los tiempos de GPU vistos anteriormente.

Esto es normal ya que en cada cuadro se realizan por lo menos 5 proyecciones (antigravedad y normal al piso de **AJet**) y se agregan fuerzas (acelerar, girar, alinear, etc.).

Realizando el mismo proceso para una sesión en red se ve que ocurren más de estas proyecciones y uso de físicas cuando el **UDeloreanReplicationMachine** hace la sincronización de movimientos.

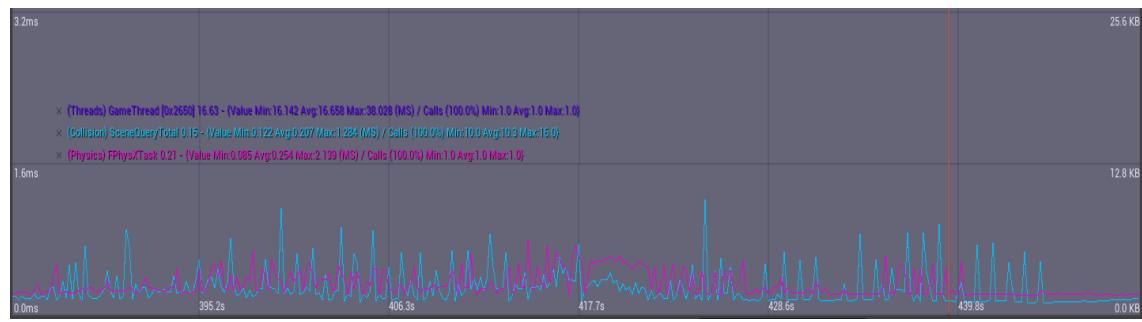


Ilustración 39. Gráfica Tiempo de Procesamiento (ms) VS Tiempo (s) en un juego en red. Celeste: Proyecciones. Violeta: Procesamiento de Físicas. Se puede notar un mayor uso de proyecciones y procesamiento de físicas que un juego local, aunque ninguno de los dos excede los 1.6ms.

Análisis de la Red

Para realizar el análisis de la red, en la misma sesión de juego en red del análisis del CPU, se utilizó el comando `netprofile` (enable y disable) para registrar estadísticas de la red.^{97 98}

Cabe mencionar que la captura muestra los datos del cliente y el servidor conectados. Además, se utilizó el programa `Clumsy`⁹⁹ para generar un retraso de 100ms en el envío/recepción de mensajes.

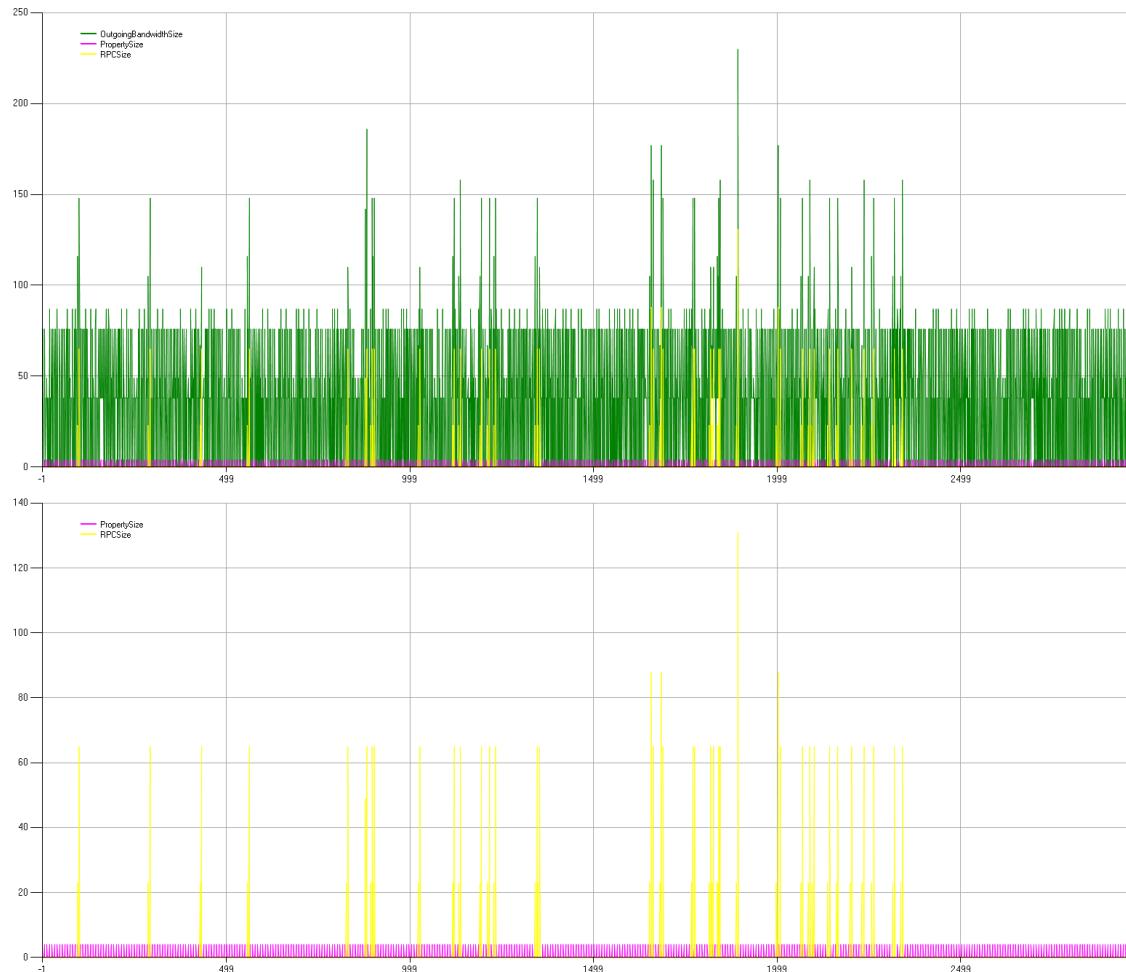


Ilustración 39. Gráfico Bytes enviados VS cuadros (un cuadro cada 16, 67ms). Arriba: en verde: ancho de banda utilizado. Debajo: en amarillo: bytes de métodos remotos y en violeta: bytes de variables replicadas.

Como se puede ver en el gráfico cada cierto cuadro se envía actualizaciones de propiedades y más esporádicamente métodos.

Una vista más cercana permite saber en qué cuadros se envió propiedades o métodos:

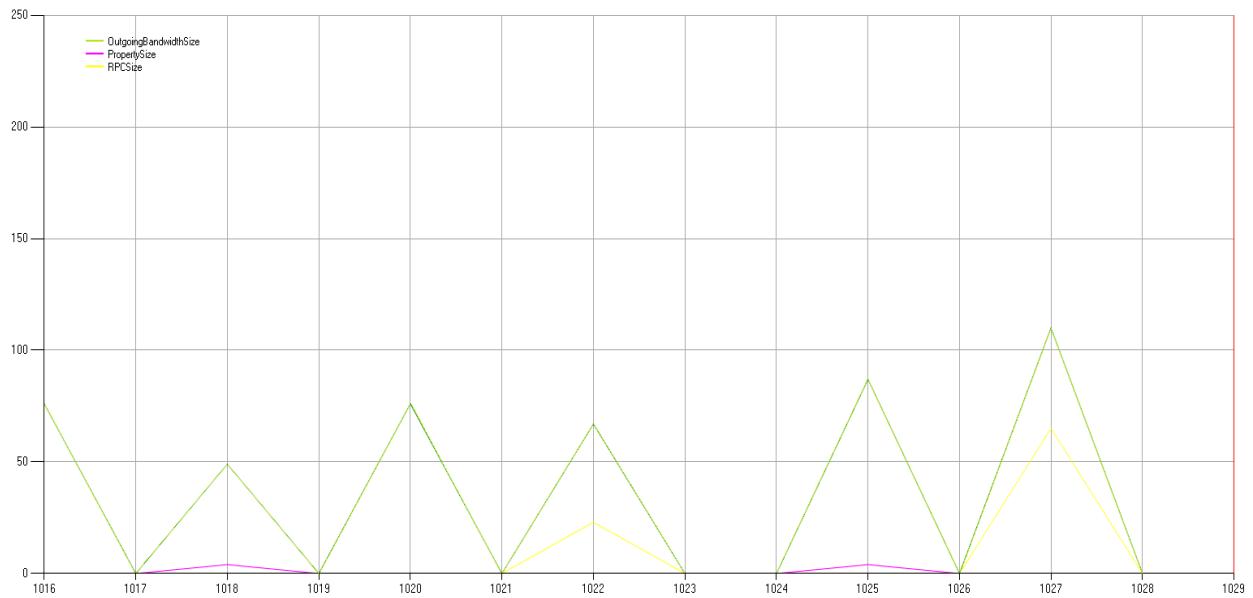


Ilustración 40. Gráfico Bytes enviados VS cuadros (un cuadro cada 16, 67ms). En verde: ancho de banda utilizado. En amarillo: bytes de métodos remotos y en violeta: bytes de variables replicadas.

En el cuadro 1018 se envía una actualización de variable. En los cuadros 1022 y 1027 se envían métodos. La única variable que se replica es ReplicatedWorldTimeSeconds, que se encuentra en **AGameStateBase**, y es responsable de enviar a clientes el tiempo actual en el mundo del servidor.

Los únicos métodos que se replican son serverUpdateMovementWith y multicastSynchronizeMovementWith. serverUpdateMovementWith siempre es enviado por clientes al servidor y lleva como parámetro los estados de giro y motor, así como también la marca de tiempo. Es usado en el primer paso de replicación de **AJet**. multicastSynchronizeMovementWith siempre es enviado por el servidor al resto de los clientes y lleva consigo el movimiento más cercano a la marca de tiempo enviada por el cliente en serverUpdateMovementWith. Se utiliza en el segundo paso de replicación de **AJet**.

Los datos de la variable replicada y los métodos son estos:

Total Size (KBytes)	Count	Average Size (Bytes)	Average Size (Bits)	Time (ms)	Average Time (ms)	Property
1.6	422	4.0	32.0	0.00	0.0000	ReplicatedWorldTimeSeconds

Total Size (KBytes)	Count	Average Size (Bytes)	Average Size (Bits)	Time (ms)	Average Time (ms)	RPC
0.9	40	23.7	189.2	0.00	0.0000	serverUpdateMovementWith
2.6	40	65.5	524.0	0.00	0.0000	multicastSynchronizeMovementWith

Ilustración 41. Datos de las variables y métodos replicados.

ReplicatedWorldTimeSeconds es una variable de tipo float, lo que es normal que su tamaño sea de 4 bytes. Se podrían usar métodos de compresión para reducir su tamaño.

serverUpdateMovementWith es un método con 24 bytes, donde 2 provienen de la marca de tiempo y el resto por las clases de los dos estados que se envían (tipo de clase). Se podría reducir su tamaño reemplazando los estados por enums y comprimiendo el tamaño del int64 usado para la marca de tiempo.

multicastSynchronizeMovementWith tiene un tamaño mayor, de 524 bits ya que además de lo mencionado para serverUpdateMovementWith, se tiene que enviar la ubicación, rotación, velocidad lineal y velocidad angular, que son todos tipos de vectores que poseen floats.

Se debe tener en cuenta que las clases que se habilitan para replicar se agregan internamente a un registro en el motor cuando se instancian. Según su tasa de actualización, el motor chequea la clase buscando si es que el estado de las variables replicadas cambió o debe enviar un método. Si la instancia no tiene que replicar, entonces el motor estuvo perdiendo tiempo en esa clase. A eso se le llama CPU waste. A continuación, se muestra una tabla con el porcentaje de CPU waste de las instancias de clases replicadas en el juego:

Actor	MS	KB/s	Bytes	Count	Update HZ	Rep HZ	Waste
GameplayDebuggerCategoryReplicator	20.47	0.00	0	1833	37.26	0.00	100.00
BP_Jet	13.41	0.00	0	1475	29.98	0.00	100.00
BP_PlayerController	11.71	0.00	0	1859	37.79	0.00	100.00
BP_ProjectRGameState	10.49	0.03	1688	422	8.58	8.58	0.00
FlatPlayground	4.60	0.00	0	1844	37.49	0.00	100.00
WorldSettings	1.55	0.00	0	422	8.58	0.00	100.00
BP_RacePlayerState	0.63	0.00	0	49	1.00	0.00	100.00

Ilustración 42. Tabla con porcentajes de pérdida de tiempo del CPU.

Para explicitar al CPU que no debe verificar una instancia de clase, la instancia debe declarar NetDormancy = ENetDormancy::DORM_Initial en su constructor. De esta forma, solo replicará al cliente cuando se la “despierte”. Luego, se “duerme” por lo que el motor no verificará si necesita replicar por la red. Previamente al momento en que la instancia desee replicar, solamente tiene que llamar a FlushNetDormancy() si es solo en ese cuadro quiere replicar y volver a “dormir” o ForceNetUpdate() si se quiere replicar por ese cuadro y los siguientes. Cuando se use ForceNetUpdate() se debe usar SetNetDormancy(ENetDormancy NewDormancy) para volver a “dormir” la instancia.

Los posibles estados de ENetDormancy¹⁰⁰ son:

- DORM_Never: el actor nunca puede ser dormido por código.
- DORM_Awake: el actor puede ser dormido mediante código.
- DORM_DormantAll: El actor se duerme para todas las conexiones existentes, no replica.
- DORM_DormantPartial: El actor se duerme para algunas conexiones. Se puede llamar a GetNetDormancy() para saber para cuáles está dormido.
- DORM_Initial: El actor está inicialmente dormido cuando se hace aparecer en un mapa.

Arquitectura

1 - Contexto del Proyecto

1.1 - Unreal Engine 4

Es la base en la que se realiza todo el proyecto, por lo que la facilidad de manejo en este motor es una de las principales restricciones en el desarrollo de la arquitectura.

Se pospuso la redacción de la arquitectura al final del desarrollo ya que en el principio se tenía un conocimiento muy básico de sus componentes y funcionamiento.

Se supone que la sección de desarrollo ya fue leída, por lo que el enfoque del motor en esta sección se basará en esa lectura y será más técnico que en el desarrollo.

Se debe mencionar dos características esenciales del motor que van a ayudar a comprenderlo en profundidad: su arquitectura basada en eventos y su sistema de propiedades.

1.1.1 - Arquitectura basada en Eventos

Unreal Engine 4 tiene una arquitectura basada en eventos, por lo que su forma de comunicar componentes es mediante la reacción a eventos, que es posible crearlos, suscribirse a ellos o ejecutarlos.¹⁰¹

En el motor, los eventos son una especialización de “Delegates”, que son macros capaces de llamar métodos suscritos en C++ cuando se los ejecute. Los Delegates pueden ser Simples, Multicast (para referencias débiles a objetos y es de donde se desprenden los Eventos), Dinámicos (que poseen serialización, pero más lentos).^{102 103}
^{104 105}

A estos Delegates se les especifica los parámetros y/o valores de retorno de los métodos que se pueden suscribir a ellos. Esto significa que los métodos que se suscriban tienen que respetar esa forma.

Por ejemplo, si se utilizara:

```
DECLARE_DELEGATE_OneParam(ADelegateName, float)
```

El método que se quisiera suscribir debería tener la forma:

```
void methodToSubscribe(float)
```

Se puede también especificar el tipo de retorno del método a suscribir:

```
DECLARE_DELEGATE_RetVal(RetValType, ADelegateName)
```

Para suscribir un método, solo hace falta tener una instancia del Delegate y utilizar Bind() aportando la referencia al método que se suscriba como parámetro, junto con el objeto que lo debe llamar.

Para ejecutar un Delegate, solo basta llamar Execute() desde la instancia del Delegate. Este método recorrerá una lista con elementos del tipo <referencia a método, objeto que llama> y realizará: (objeto que llama)->(referencia a método) con cada uno.

Los otros tipos de Delegates (se mostró el caso simple) siguen el mismo patrón para declararlos.

En el proyecto, se utilizaron Dynamic Multicast Delegates y se mostrará su uso en los patrones arquitectónicos relevantes.

1.1.2 - Sistema de Propiedades

Unreal Engine 4 utiliza dos herramientas para conseguir la reflexión en el motor: Unreal Header Tool (UHT) y Unreal Build Tool (UBT), que trabajan en conjunto para reconocer e inyectar código necesario para la reflexión cuando se compila el programa.

UBT recorre los archivos .h y registra los módulos que necesitan reflexión. Si algún módulo cambia, UHT analiza estos archivos, crea un conjunto de metadatos y luego genera el código en C++ que los contiene, agregando métodos a cada archivo generado y sumándolo al módulo respectivo de metadatos.

Que se haga este procedimiento cada vez que un módulo se modifica asegura que el código generado esté en sincronía con el archivo binario.

Los métodos proveen funcionalidades como saber la clase a la que pertenece cada objeto, o la capacidad de exponer porciones de código a Blueprints, así como también permitir la replicación de métodos y variables en la red.¹⁰⁶

Por ejemplo, si se quisiera crear una clase, AnotherActor, que derive de AActor, se debería crear dentro del editor y el archivo generado tendría esta forma:

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "AnotherActor.generated.h"

UCLASS()
class PROJECTR_API AAnotherActor : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AAnotherActor();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    UPROPERTY()
    float aNumber;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    UFUNCTION()
    void aMethod();
};
```

A simple vista se puede ver una diferencia con la creación normal de una clase en C++ puro.

Los primeros dos include (CoreMinimal.h y GameFramework/Actor.h) son necesarios para poder comunicarse con el motor y tener la referencia a la clase base. Estos dos includes es normal incluirlos, por lo que no cambiaría su implementación en C++.

El `#include` que se debe agregar para que el archivo sea considerado por el sistema de propiedades (Unreal Property System) es `#include "AnotherActor.generated.h"`. Allí es donde se combinará la clase que se declaró junto con el relleno creado por el sistema de propiedades (definición de las macros usadas).

La macro `UCLASS()` sirve para poder generar el CDO que es la sigla para el “Objeto por defecto de la Clase” en inglés. Este objeto se creará por única vez al iniciar el motor por medio del constructor y será el objeto que se devuelva cuando se llame a `GetClass()`. El CDO tiene una serie de propiedades y métodos que definen a una clase, y son nativos de Unreal Engine, por lo que no es posible modificarlos sin modificar todo el motor.¹⁰⁷

`PROJECTR_API` sirve para indicar a qué módulo pertenece los metadatos a crear.

Se puede notar una ‘A’ extra en `AnotherActor`. Esto es así porque internamente el editor agrega inevitablemente un prefijo a cada clase o estructura que se crea para indicar de qué clase deriva. Se utiliza ‘A’ para actores, ‘U’ para UObjects y F para estructuras.

`GENERATED_BODY()` es el lugar donde se copiará todo el código generado por Unreal Header Tool para esa clase.

`UPROPERTY()` hace que la variable que siga se exponga al sistema de reflexión. Esto es útil para cuando se quiere exponer la variable a Blueprints o al sistema de replicación. Para eso, se debe agregar especificadores dentro de `UPROPERTY` como `VisibleAnywhere` o `Replicated`.^{108 109}

`UFUNCTION()` es similar a `UPROPERTY`, pero enfocado en métodos. `BlueprintCallable` habilita al método a ser llamado por Blueprints. `Server`, `NetMulticast` y `Client` sirven para especificar al sistema de replicación si el método se envía al servidor, a todos los que posean esa instancia en la red o solo al cliente respectivamente.^{110 111}

Unreal Property System es un sistema robusto para lograr la reflexión en tiempo de compilación. El problema es que altera la forma de programar, ofusca el código (no se sabe qué hacen internamente estas macros sin recorrer archivos del motor) y hace que se generen nuevos problemas, como tener que hacer que una clase tenga que utilizar el sistema de reflexión si es que está compuesta por otra clase que lo utiliza, o tener que declarar métodos como `UFUNCTION` porque cierta parte del código no lo reconoce si no está declarado como tal. Otro problema es que no todos los tipos del motor son soportados por el sistema de propiedades, lo que restringe el uso pleno de todas las funcionalidades disponibles. Resuelve muchos problemas y genera otros.

1.2 - Diseño

Se debe tener en cuenta que no solo los programadores van a utilizar las clases, personas que no sabe nada de programación las utiliza (solo las expuestas por el sistema de reflexión, que son la mayoría). Por lo que se debe tener conciencia de qué exponer y qué no.

2 - Requerimientos de Arquitectura

2.1 - Descripción breve de los Objetivos Clave

El objetivo principal es que los usuarios puedan ejecutar el programa en una PC y correr carreras solos o con otras personas (localmente y en red). Para esto se necesita:

- Crear distintos tipos de juego para seleccionar si se quiere correr solo o contra otras personas.
- Tener una interfaz básica que permita la selección de estos tipos de juego y brinde información relacionada a la carrera en curso (posiciones, vueltas, etc.).
- La interfaz de red debe permitir la creación de sesiones, su publicación, búsqueda automática y la unión de otros usuarios de manera simple.
- Proporcionar vehículos con un manejo adecuado. Estos vehículos deben tener cualidades futurísticas (levitación, forma asemejada a aviones jet, etc.).
- Tener un sistema de creación de pistas interno (para el desarrollo) para poder facilitar su creación.

El objetivo secundario es que el proyecto sirva como base para extenderlo y volverlo un producto completo, así como también poder crear otros proyectos en Unreal Engine 4 con lo aprendido durante su desarrollo. Para lograrlo, es necesario:

- Soportar una integración continua de características nuevas, mejoras y arreglos mediante pruebas.
- Extender las herramientas que proporciona Unreal Engine 4 en vez de crear desde cero las propias.
- Exponer solamente lo necesario para que artistas o diseñadores sean capaces de configurar a su gusto los objetos creados por programadores.

2.2 - Casos de Uso de Arquitectura

Usuario:

- Correr una carrera solo.
- Correr una carrera en multijugador local y elegir la cantidad de jugadores.
- Correr una carrera en red. Crearla o unirse a las creadas por otros usuarios.
- Elegir el mapa de la pista a jugar en cualquiera de los tres puntos anteriores.
- Maniobrar el vehículo durante la carrera.
- Salir de una carrera para iniciar otra.

Artistas y diseñadores:

- Configurar vehículos.
- Crear distintas pistas y mapas.
- Configurar carreras (vueltas, ubicación de vehículos al comienzo de la carrera, etc.).

2.3 - Requerimientos de Arquitectura de los Actores

Usuarios (terceros)

- Tener instalado el sistema operativo Windows 10 (64 bits) en su equipo.
- Tener un equipo con tarjeta gráfica que permita la creación de objetos 3D complejos.

Desarrolladores (aplicación)

- Interfaz simple y básica para usuarios.
- Utilización y extensión de las herramientas proporcionadas por el motor.
- Código conciso, simple y adaptable.

Equipo de desarrollo interno (desarrollo)

- Programación mediante pruebas para mantener un enfoque en requerimientos y lograr una base cuyo funcionamiento esté verificado.
- Entorno de integración continua para la automatización de los cambios y la verificación de pruebas para reducir los tiempos de desarrollo por causa de tareas repetitivas y manuales.
- Simplificación periódica de código para su fácil lectura, mantenibilidad y extensión en el futuro.

Unreal Engine 4 (Proveedores)

- Facilidad en el uso del programa.
- Documentación clara y completa.
- Presencia de sistema de automatización o incorporación de otros sistemas de automatización conocidos.

2.4 - Restricciones

- Programación en el lenguaje C++.
- Utilización del esquema de programación de Unreal Engine 4.
- Uso de Git como control de versiones y GitHub para el repositorio en la nube.
- Automatización de tareas por medio de Jenkins.
- Visual Studio 19 como entorno de desarrollo, junto con Resharper para sugerencias y relleno de código, y Unreal Engine 4 para crear configuraciones, mapas, etc.
- Windows 10 como plataforma de desarrollo y pruebas.

2.5 - Requerimientos No Funcionales

- **Rendimiento:** Se quiere lograr un sistema que soporte 4 jugadores en modo multijugador a pantalla partida y soporte en red con 8 jugadores simultáneos. Preferentemente, se quiere que la aplicación funcione a 60 cuadros por segundo en todo momento.
- **Confiabilidad:** se enfocará en remover errores tempranamente por medio de uso de Test-Driven Development y Continuous Integration. Solo se continua la implementación una vez que los errores detectados son removidos.
- **Simplicidad:** se quiere tener la mínima interfaz que proporcione un uso básico de la aplicación por parte de usuarios. Esta interfaz se extenderá más adelante para habilitar más características.

2.6 - Riesgos

Poco conocimiento de la arquitectura de Unreal Engine. Esto provocaría posibles retrasos en implementaciones. Es un riesgo alto ya que, al ser un framework, la forma de implementar características no sea exactamente como se había planeado. Se debe tener un entendimiento claro del sistema para poder tener como mínimo un mapa de secciones de Unreal Engine 4 que son necesarias para cumplir el desarrollo.

Documentación disponible de Unreal Engine 4 escasa o insuficiente. Ocasionaría pequeños retrasos en soluciones de problemas con el motor o el uso de sus características. Es un riesgo medio porque es posible realizar búsquedas de tutoriales o preguntas hechas por otros desarrolladores que resuelvan estos problemas.

Implementaciones no oficiales de soluciones a problemas. Se refiere al uso de soluciones propuestas por otros desarrolladores, que no tienen sustento oficial. Puede llevar a errores que no se detecten de forma temprana si es que no se analiza en profundidad el código propuesto. Es un riesgo bajo.

Tiempos extensos en realizar implementaciones. Junto con los dos primeros riesgos y las suposiciones de los desarrolladores a cómo implementar ciertas características en el motor (se debe recordar que no tienen mucha experiencia con Unreal Engine 4) hacen que se retrasen las implementaciones. Es un riesgo alto al estar vinculado a los primeros riesgos.

Incapacidad de realizar pruebas automatizadas en Unreal Engine 4. Es un riesgo bajo si es que se toman medidas al comienzo del trabajo para investigar la capacidad de automatización del motor. En caso de no existir esa capacidad, el riesgo se vuelve alto y debería replantearse el desarrollo del proyecto.

Errores de código que no son detectados en pruebas automatizadas. Es un riesgo inherente a Test-Driven Development y se convierte en bajo una vez que se tienen varias pruebas que verifiquen de distintas maneras la implementación. El aumento de pruebas refuerza la detección y evita enfocarse en áreas en donde el error no ocurre (las áreas donde las pruebas son exitosas).

Feature Creep o Feature Bloat. Es la adición de características a la aplicación que no son fundamentales al proyecto. Este riesgo existe ya que este proyecto se toma como base para convertir la aplicación en un producto comercial, por lo que características que deberían ir en la aplicación final se pueden filtrar al proyecto. Es un riesgo medio y se puede mitigar teniendo un plan de implementaciones que haya sido analizado para descartar o posponer aquellas características que no sean esenciales al proyecto.

3 - Solución

3.1 - Patrones Arquitectónicos Relevantes

3.1.1 - Patrones de diseño

State

Este patrón facilita que un objeto cambie su comportamiento cuando su estado interno cambia.

Se tiene una clase llamada Context que no implementa directamente el comportamiento del estado interno. En vez de eso, Context lo delega en una interfaz llamada State que posee un método llamado handle, el cual representa el acceso al comportamiento. Por medio de herencia se logra que Context sea independiente del comportamiento de los State concretos que se creen. De esta manera se tiene encapsulado el comportamiento para cada estado.

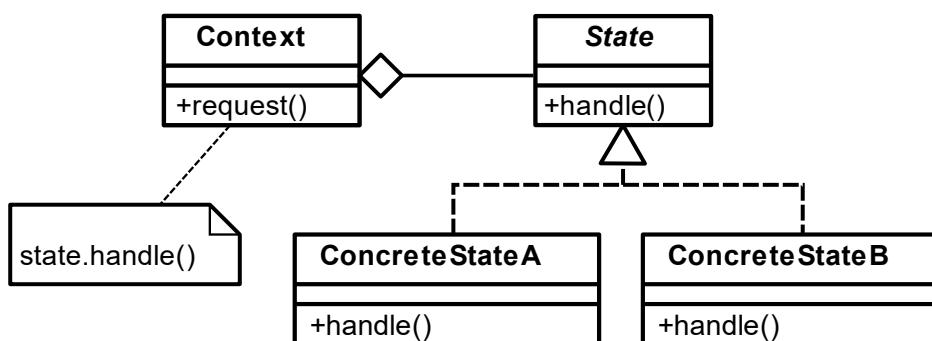


Ilustración 1. Representación en UML del patrón de diseño State.

Se utiliza el patrón de diseño State para administrar los estados del motor de un AJet y para los estados de la dirección. También se lo utiliza en el modo de juego de la carrera para separar las distintas fases de ella en estados. Para administrar las vueltas de una carrera también se utilizó este patrón. De esta forma se tiene la representación de una vuelta en distintas fases que la componen (inicial, intermedia y final).

Se eligió este patrón para separar los distintos comportamientos posibles por medio de una única interfaz. Así, se delega en las clases concretas el comportamiento específico de cada estado en vez de implementarlo directamente.

En el caso del motor del AJet:

- El administrador de estados AMotorStateManager es el contexto.
- UMotorState representa a State.
- Los estados concretos son UNeutralMotorState, UAcceleratingMotorState y UReversingMotorState.
- El handle es el método void activate(UMotorDriveComponent*).

En el caso de la dirección del Ajet:

- El administrador de estados ASteerStateManager es el contexto.
- USteerState representa a State.
- Los estados concretos son UCenterSteerState, ULeftSteerState y URightSteerState.
- El handle es el método void activate(USteeringComponent*).

En el caso de la carrera:

- ARaceGameMode es el contexto.
- URaceStage representa a State.
- Los estados concretos son ARacePreparationStage, ARaceBeginningStage, ARaceRunningStage y ARaceEndedStage.
- El handle es el método void start().

En el caso del administrador de vueltas:

- ALapManager es el contexto.
- ALapPhase representa a State.
- Los estados concretos son AInitialLapPhase, AIntermediateLapPhase y AFinalLapPhase.
- Los métodos booleanos usados como handle son: nextPhasels(ALapPhase*), comesFromInitialLapPhase(), comesFromIntermediateLapPhase() y comesFromFinalLapPhase().

Composition Over Inheritance

Se separan responsabilidades e implementaciones en objetos que se encargan de ellas. Da mayor flexibilidad. Se prefiere el uso de “tener algo” a “ser algo” ya que con herencia (“ser algo”), la introducción de pequeños cambios a largo plazo a las clases derivadas puede ser que rompa la unión entre ellas.

Utilizando este patrón, Ajet delega responsabilidades de la clase en subclases de las que está compuesta. Se compone de UMotorDriveComponent (que se encarga de la aceleración y frenado de la nave), USteerDriveComponent (que realiza la dirección hacia los lados), UAntiGravityComponent (que se implementa la levitación) y UDeloreanReplicationMachine (que se ocupa de sincronizar la nave con los datos que se reciben de la red).

Junto con estos componentes se tiene a los dos administradores de estado: AMotorStateManager y ASteerStateManager.

Se lo utilizó también en ARaceGameMode para contener las distintas etapas de la carrera y al administrador de vueltas.

3.1.2 - Patrones arquitectónicos

Eventos

En el contexto del proyecto se habló de cómo Unreal Engine 4 tiene una arquitectura basada en eventos. Se hace uso de eventos cuando se notifica que un jugador realiza una vuelta o cambia de fase (de la vuelta), para “magnetizar” naves a la pista (mantenerlas si es que la pista se encuentra invertida) y para notificar a la carrera que una etapa se terminó y se puede iniciar la siguiente.

Servidor-Cliente

Unreal Engine utiliza el patrón arquitectónico de red de Servidor-Cliente para comunicar distintos usuarios en una sesión en red. El servidor tiene la versión oficial de una partida y la comunica a los clientes que corresponda cuando se modifica. Los clientes envían peticiones al servidor para realizar cambios sobre ella. De esta forma, el servidor es el único punto de contacto entre ellos.

El mayor uso que se hace es cuando una nave cambia sus estados, así como también la actualización recibida por clientes que se desencadena por realizar ese cambio y enviarlo hacia el servidor. También, el servidor informa datos de la carrera, como la posición y vueltas hechas cuando ocurre algún cambio en ellos.

3.2 - Breve Descripción de la Arquitectura

Al ser el proyecto relativamente grande (se cuenta con 49 clases de objetos), se centrará en ciertas clases que adoptan los patrones mencionados en el punto anterior.

Se debe aclarar también que los diagramas presentados no representan todas las relaciones que tienen los objetos con otros. Solo las relaciones relevantes en cada contexto son mostradas.

AMotorStateManager y UMotorStates (State)

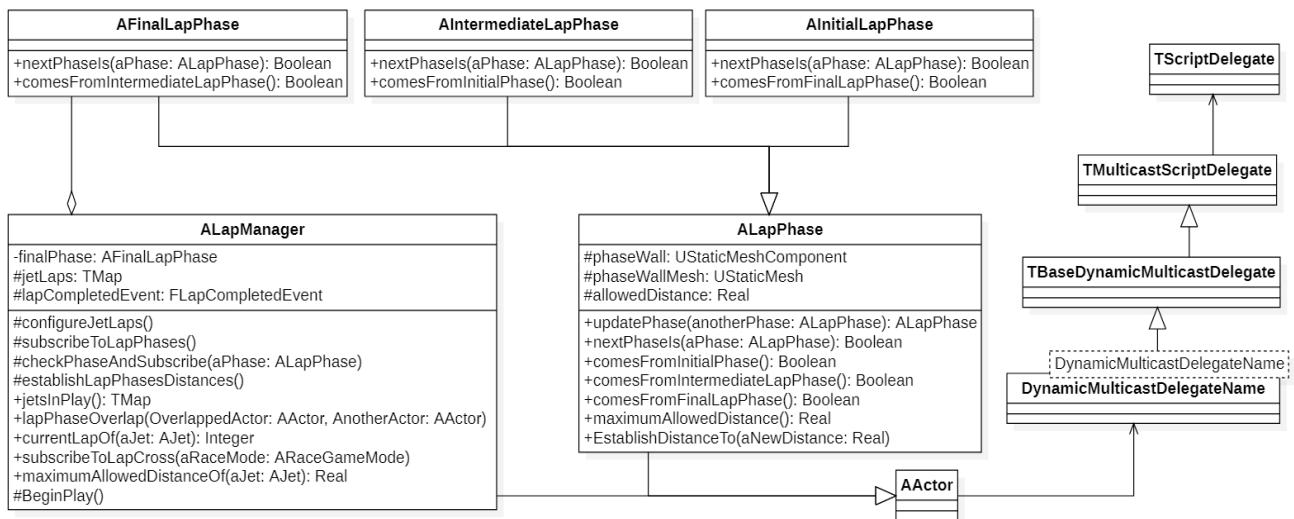


Ilustración 2. Diagrama UML entre AMotorStateManager y UMotorState. Se omite AJet intencionalmente.

Como se puede ver en el diagrama, AMotorStateManager cumple el rol del contexto con el request llamado activar. Internamente cambia su motorState cuando se le pide acelerar, frenar, neutralizar o mezclar sus estados. Se muestra también el handle de UMotorState (también llamado activar) sobreescrito en cada tipo de estado. En este caso se utilizó composición en vez de agregación porque la clase del estado del motor no tiene sentido fuera de un administrador de estados del motor. UMotorState y AMotorStateManager son clases derivadas de UObject y AActor respectivamente, que son parte de Unreal Engine 4. Se omiten sus atributos y operaciones por ser clases extensas, y para evitar violaciones del contrato del programa.

AJet y Componentes (Composition Over Inheritance)

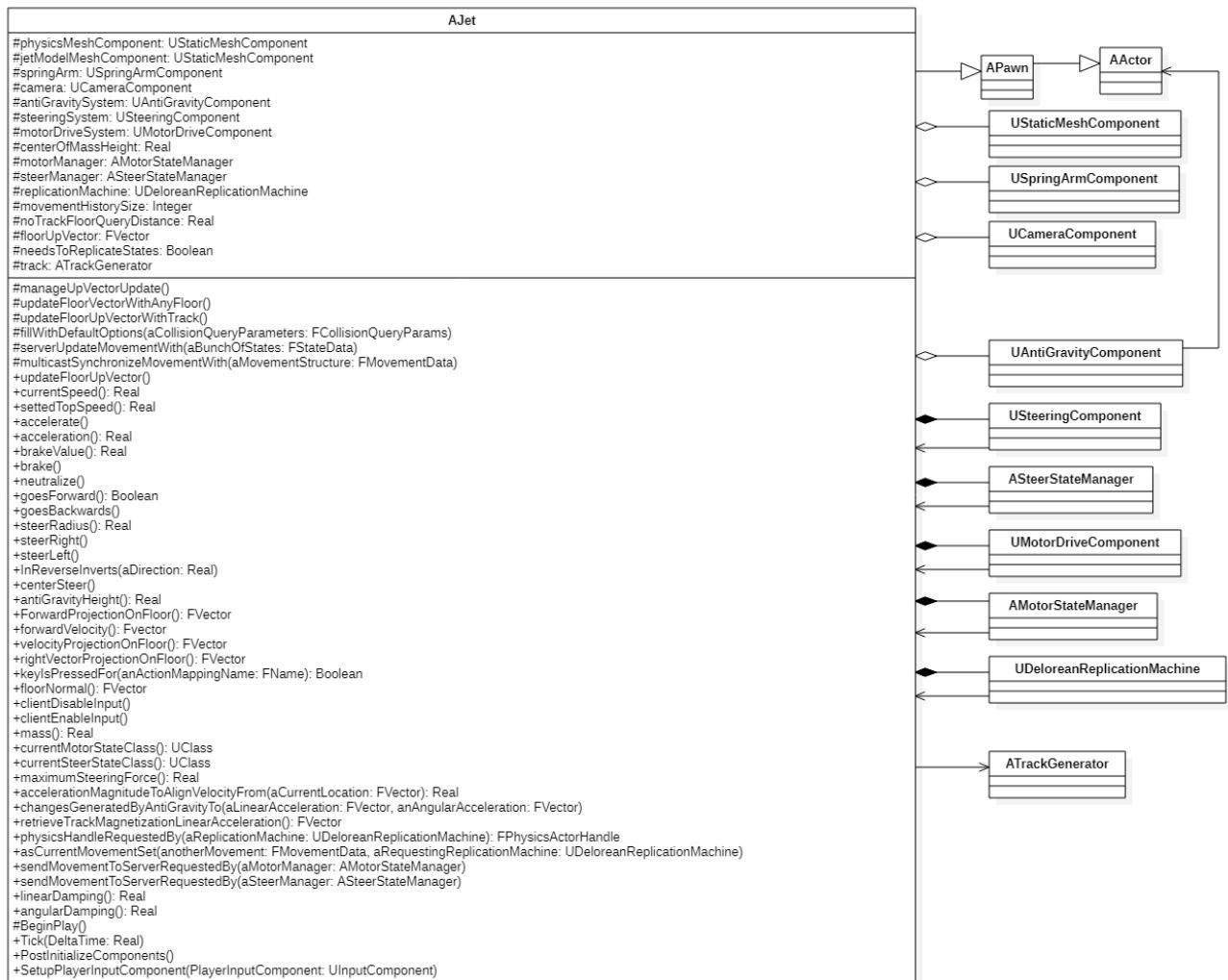


Ilustración 3. Diagrama UML de AJet junto con sus distintos componentes.

Se puede ver claramente el tamaño de la clase AJet. La mayoría de sus métodos son métodos de delegación a sus componentes, que se muestran a la derecha del diagrama (excepto ATrackGenerator y la relación de generalización mostrada para esclarecer su conexión con el motor del juego). UStaticMeshComponent, USpringArmComponent y UCameraComponent son componentes que provee el motor por defecto, por lo que su relación es de agregación ya que pueden ser utilizados en otros APawn o AActor.

Los componentes creados para el proyecto pueden verse en detalle en el próximo cuadro:

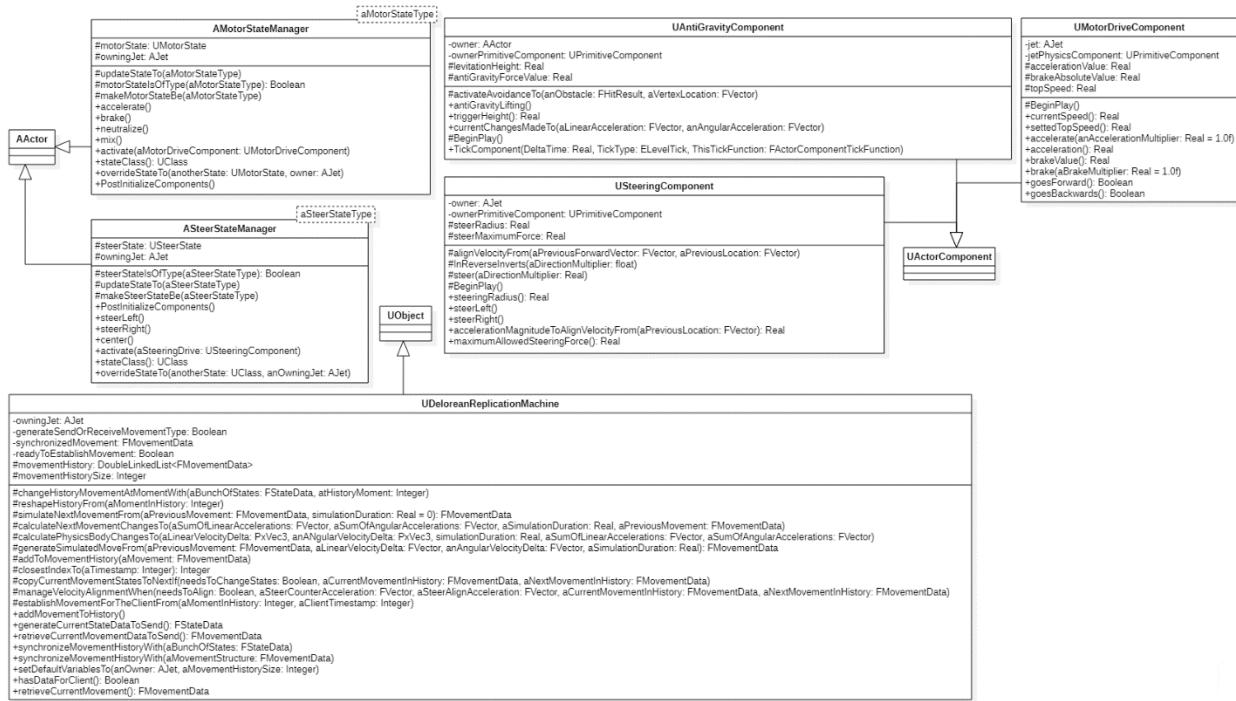


Ilustración 4. Diagrama UML de los componentes que conforman AJet (expandido).

Puede verse la diferencia que existe en derivación de los distintos componentes. Esto es así ya que durante el desarrollo hubo una transición en el uso de clases derivadas de UActorComponent hacia UObject (pasando por AActor). Esta transición fue producto de una mayor familiarización con el entorno de Unreal Engine 4, así como también la posibilidad de probar de mejor manera las clases (las derivadas de UActorComponent fueron solamente producto de refactorización y simplificación de código, mientras que el resto son clases probadas separadamente de AJet).

En el diagrama se logra ver que los componentes que conforman AJet no son muy grandes (a excepción de UDeloreanReplicationMachine), por medio de la cantidad de atributos y métodos que poseen. Aunque cabe aclarar que la mayoría de los métodos de AJet son métodos de delegación¹¹² y/o creados a partir de la extracción de código de métodos más grandes, lo que hace aumentar la cantidad de llamadas, pero proporciona un mejor seguimiento del flujo de código.

También, se debe comentar que ciertas características de C++ no pudieron ser implementadas ya que ocasionan problemas con el sistema de Unreal Engine. Una de estas características podría ser el uso de friend para declarar clases “amigas” a otra. De esa forma, la clase “amiga” puede acceder a métodos y atributos que no son públicos de la otra clase, lo que reduciría la cantidad de métodos utilizados solamente para exponer características de AJet a sus componentes.¹¹³ AMotorStateManager y ASteerStateManager podrían hacer mayor uso de template programming si se pudiera crear una clase base a ellas que sea parametrizable para proveerle el tipo de estado que se quiere (UMotorState o USteerState) y el tipo de componente necesario en la activación (UMotorDriveComponent y USteeringComponent). ¹¹⁴

ALapManager y ALapPhases (Eventos)

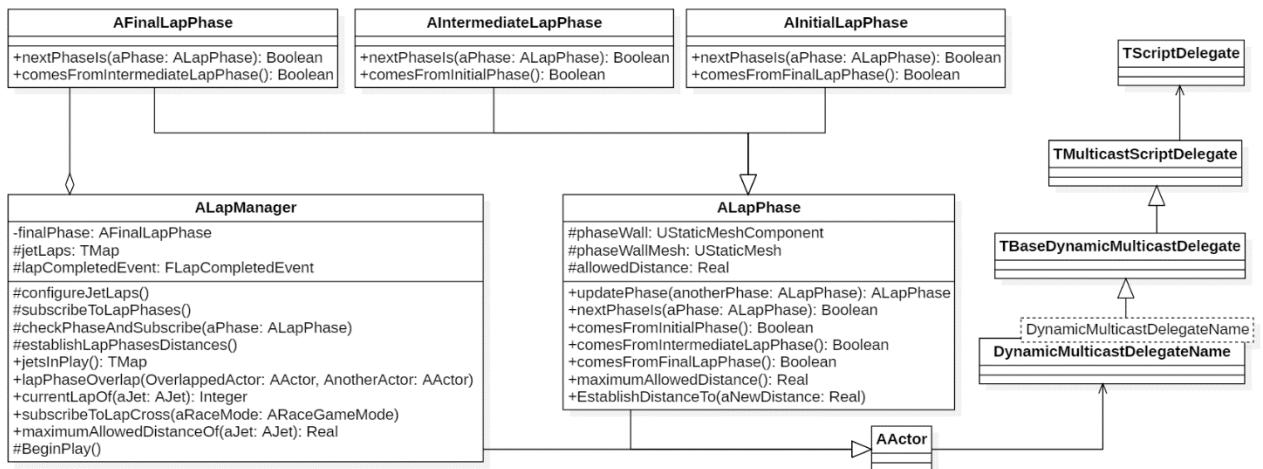


Ilustración 4. Diagrama UML entre ALapManager y los distintos tipos de ALapPhase.

ALapPhase utiliza eventos para comunicar cuando un AActor la cruza (cruza su representación en el espacio). ALapManager se suscribe a estos eventos al llamar a `subscribeToLapPhases()`. A primera vista, no se ve una clase eventos que ALapPhase o ALapManager posean. Las dos clases derivan de AActor que sí tiene eventos registrados. Estos eventos se declaran con macros donde se indica el nombre que tendrá la clase de eventos, los parámetros que deben tener los métodos que se subscríban al evento y el tipo de retorno de esos métodos. El nombre de la clase se utilizará internamente para crear una clase parametrizada en base a `DynamicMulticastDelegateName`, la cual deriva de `TBaseDynamicMulticastDelegate`, la cual finalmente deriva de `TMulticastScriptDelegate` que es la clase encargada de ser un evento. A su vez, `TMulticastScriptDelegate` utiliza a `TScriptDelegate` para tener una lista de métodos a invocar cuando se comunica la realización del evento.^{115 116 117}

De esta forma, ALapManager se suscribe al evento de solapado del AActor (por medio de ALapPhase) para ser notificado de ello.

ARacePlayerState y AProjectRGameState (Servidor-Cliente)

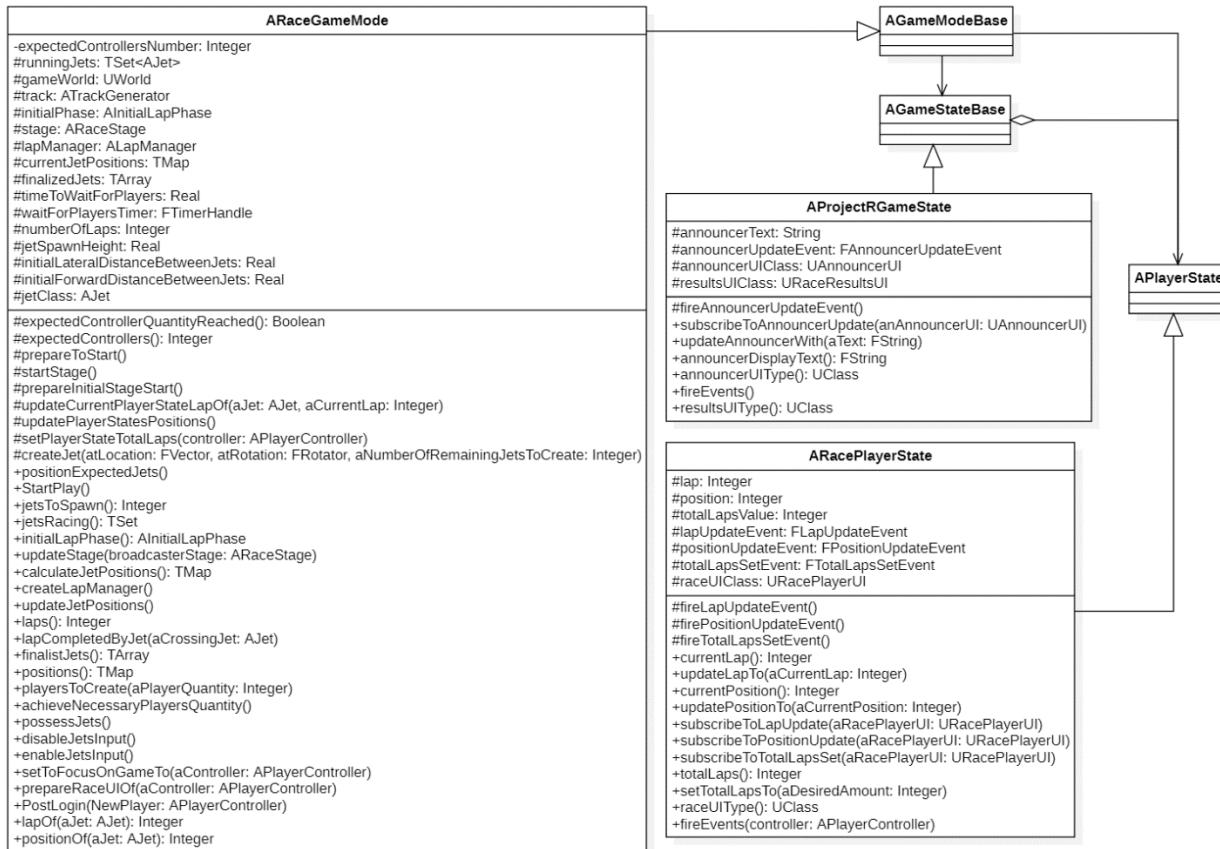


Ilustración 5. Diagrama UML de ARacePlayerState y AProjectRGameState junto con ARaceGameMode.

A grandes rasgos, no se ve comunicación con un cliente o un servidor por los métodos. Como se explicó en la sección de desarrollo, AGameModeBase es creada solamente en el servidor, mientras que AGameStateBase y APlayerState son replicadas (esto es, el cliente posee una copia de ellas, que el servidor actualiza mediante la red). Entonces, ARaceGameMode se comunica con AProjectRGameState y ARacePlayerState de forma normal, solo invoca métodos (accede a ellas mediante su conocimiento de AGameStateBase y APlayerState).

Lo que uno hace es especificar qué variables se replican (copian) al cliente cuando se modifican. Para ello, se utiliza UPROPERTY(Replicated), del sistema de propiedades, antes de declarar la variable en la interfaz de la clase. En realidad, se utiliza UPROPERTY(ReplicatedUsing=) para especificar un método que se dispare en el cliente cuando la actualización de la variable llegue por la red. Por ejemplo, en ARacePlayerState, el método usado para comunicar la actualización de la variable lap es fireLapUpdateEvent() que, como dice su nombre, dispara el evento de cambio de vuelta. Este evento es al que se subscribe URacePlayerUI para mostrar en pantalla el cambio de vuelta de la nave cuando se corre una carrera.

Solo en partes específicas del código se hizo uso de replicación. Estas partes son en el ProjectRPlayerController, AJet, AProjectRGameState y ARacePlayerState. AJet y ProjectRPlayerController utilizan replicación mediante métodos. Se explicó el uso de la replicación por métodos por AJet, pero no se dijo qué métodos se utilizan para ello. Son solamente dos métodos: `serverUpdateMovementWith(in aBunchOfStates:FStateData)` para cuando se quiere comunicar al servidor un cambio en los estados y `multicastSynchronizeMovementWith(in aMovementStructure:FMovementData)` para replicar los movimientos oficiales a clientes que posean esa instancia de AJet.

3.3 - Vistas Estructurales

Según el modelo arquitectónico 4+1¹¹⁸, se mostrará la vista lógica (Logical View) y la vista de despliegue (Deployment View):

3.3.1 - Vista Lógica

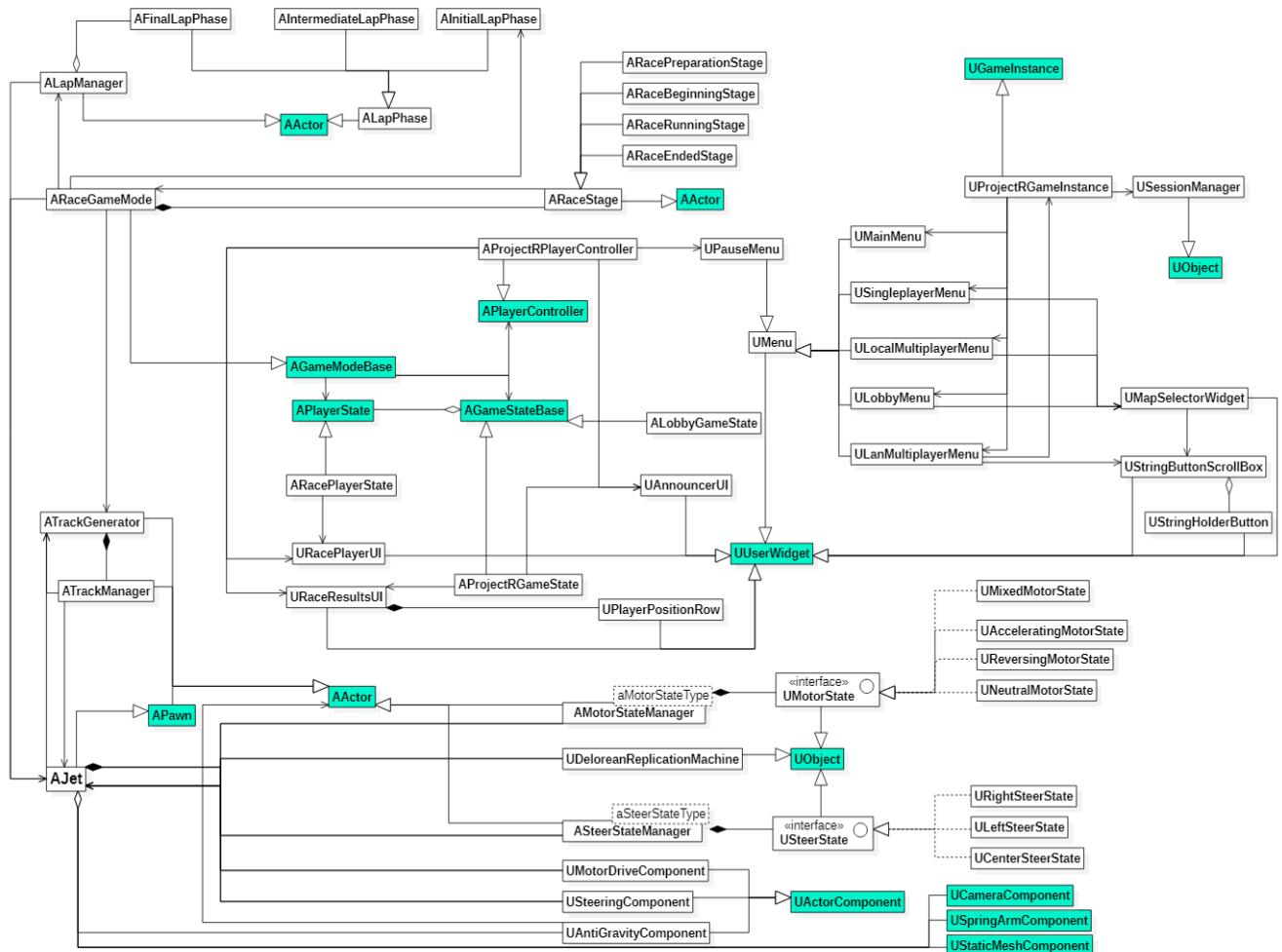


Ilustración 6. Diagrama UML de la vista lógica (Logical View) del programa. Las clases coloreadas en turquesa provienen del motor.

El diagrama muestra la vista completa de las 49 clases de objetos presentes en el proyecto. Se pueden ver tres grandes áreas: las relaciones con ARaceGameMode, AJet y UProjectRGameInstance.

ARaceGameMode junto con ALapManager, ATrackGenerator, ATrackManager, ARacePlayerState, AProjectRPlayerController, AProjectRGameState y otras clases para UI conforman el conjunto de clases necesarias para una carrera (reglas, pistas, control, información de la carrera global e individual).

AJet y sus distintos componentes es el vehículo que se necesita al correr una carrera.

UProjectRGameInstance y su relación con varias clases de UI son la forma de acceder a un juego y seleccionar distintas carreras.

En general, se podría reducir aún más el acoplamiento entre AJet y sus componentes, eliminando la relación de asociación dirigida entre los componentes hacia AJet, pasando la instancia de AJet al componente cada vez que se requiera en vez de establecer el AJet que controla ese componente en su creación. En este momento, la asociación no es perjudicial, pero podría volverse en un futuro si es que se necesitan más componentes para AJet.

Otra cosa que se podría modificar es la derivación de los componentes de AJet. Algunos derivan de UObject y otros de AActor o UActorComponent. En un futuro, se podría migrar las derivaciones hacia UObject o AActor, ya que a UActorComponent no es posible realizarle pruebas si no está unido a un actor. Se puede posicionar fácilmente a AActor en un mundo para realizar pruebas, pero muchas veces, estos componentes no necesitan una ubicación en el mundo, además de otras características que tiene AActor y no se utilizarían. UObject es el objeto base de todo el resto, por lo que tiene funcionalidades básicas y más que necesarias para lo que ofrecen los componentes de AJet. Tiene ciertas desventajas como que se tienen que declarar con UPROPERTY() en AJet para evitar que el recolector de basura los elimine. También, en las pruebas, se debería crear el UObject dentro de un AActor, pero ese obstáculo ya se solucionó en el desarrollo.

ALapPhase y ARaceStage podrían volverse interfaces y transformar sus pruebas a pruebas sobre un mock que implemente esas interfaces. La principal ventaja de volver esas clases en interfaces es que ningún otro programador podría instanciarlas y utilizarlas directamente ya que no tiene sentido usar una fase de vuelta si no representa ninguna fase en especial de una vuelta. Lo mismo ocurre con ARaceStage.

3.3.2 - Vista de Despliegue

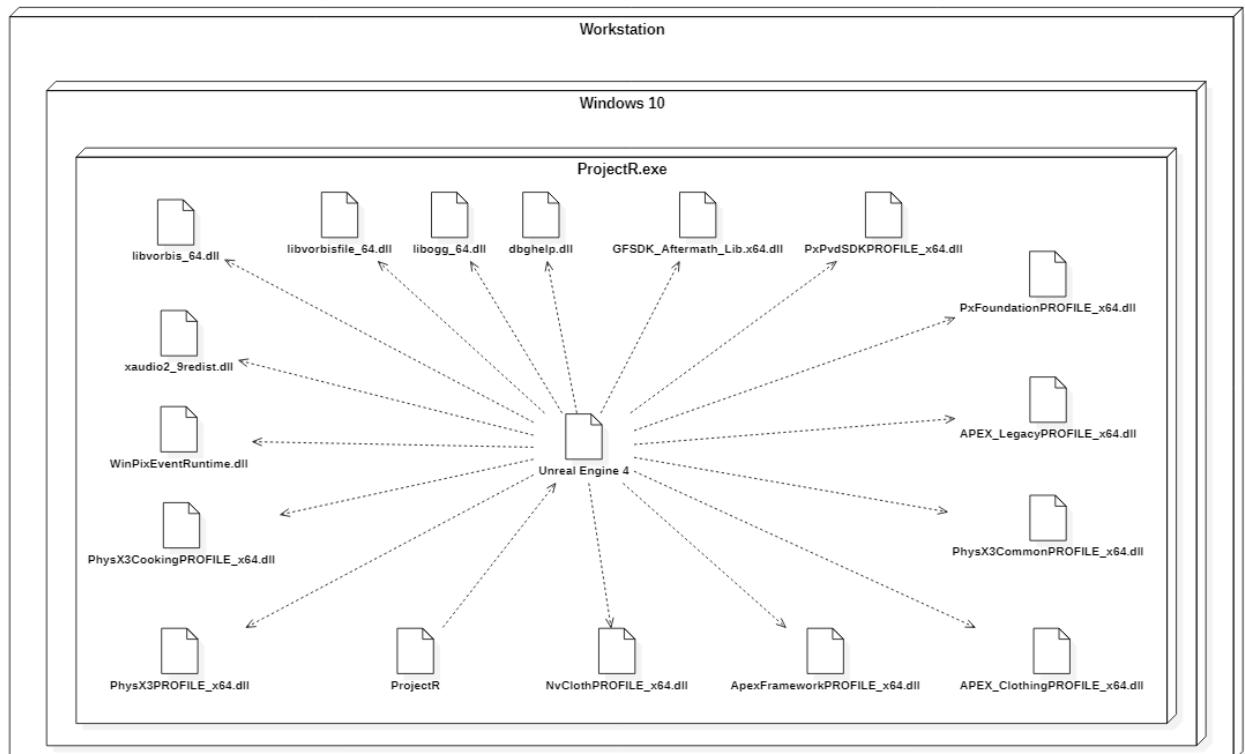


Ilustración 7. Vista de despliegue de ProjectR.

Al compilar el programa y empaquetarlo para la plataforma Windows 10 (64 bits) se obtiene un archivo ejecutable (ProjectR.exe) que se conecta con varios dll (Dynamic-link library), que son archivos con bibliotecas de código para comunicarse modularmente con otros programas en Windows. Estos archivos incrementan la reutilización de código, despliegue y mantenibilidad de programas.¹¹⁹ La diferencia con un archivo .exe es que un .dll no tiene un punto de inicio, por lo que se necesita de un .exe para poder ejecutarlo.¹²⁰

Estos DLL son utilizados por el motor (unido al código de ProjectR, que lo utiliza). Son creados por defecto cuando se compila el programa.

Los dll que se sabe que ProjectR utiliza son:

- GFSDK_Aftermath_Lib.x64.dll (para generar reportes de fallas mediante NVIDIA Aftermath).¹²¹
- APEX_Legacy_x64.dll (para utilizar módulos menos recientes de NVIDIA APEX).¹²²
- PhysX3_x64.dll, PhysX3Common_x64.dll, PhysX3Cooking_x64.dll, PxFoundation_x64.dll y PxPvdSDK_x64.dll (para acceder a todo el framework de NVIDIA PhysX, el sistema de físicas utilizado por Unreal Engine 4).¹²³

Algunos no son utilizados por ProjectR, como:

- libbogg_64.dll (para el manejo de audio en formato ogg).¹²⁴
- libvorbis_64.dll, libvorbisfile_64.dll (para compresión de audio).¹²⁵
- xaudio2_9redist.dll (para acceder a la API de audio de Windows).¹²⁶
- openvr_api.dll (para realidad virtual, VR).¹²⁷
- APEX_Clothing_x64.dll (para generar personajes con vestimenta que se comporta realísticamente).¹²⁸
- NvCloth_x64.dll (para vestimenta que se comporta realísticamente y más apropiado para videojuegos).¹²⁹

3.4 - Vistas de Comportamiento

Se describirán comportamientos de distintas clases de objetos en el proyecto mediante el empleo de diagramas UML de secuencia. ¹³⁰

3.4.1 - Secuencia de Aceleración en AJet

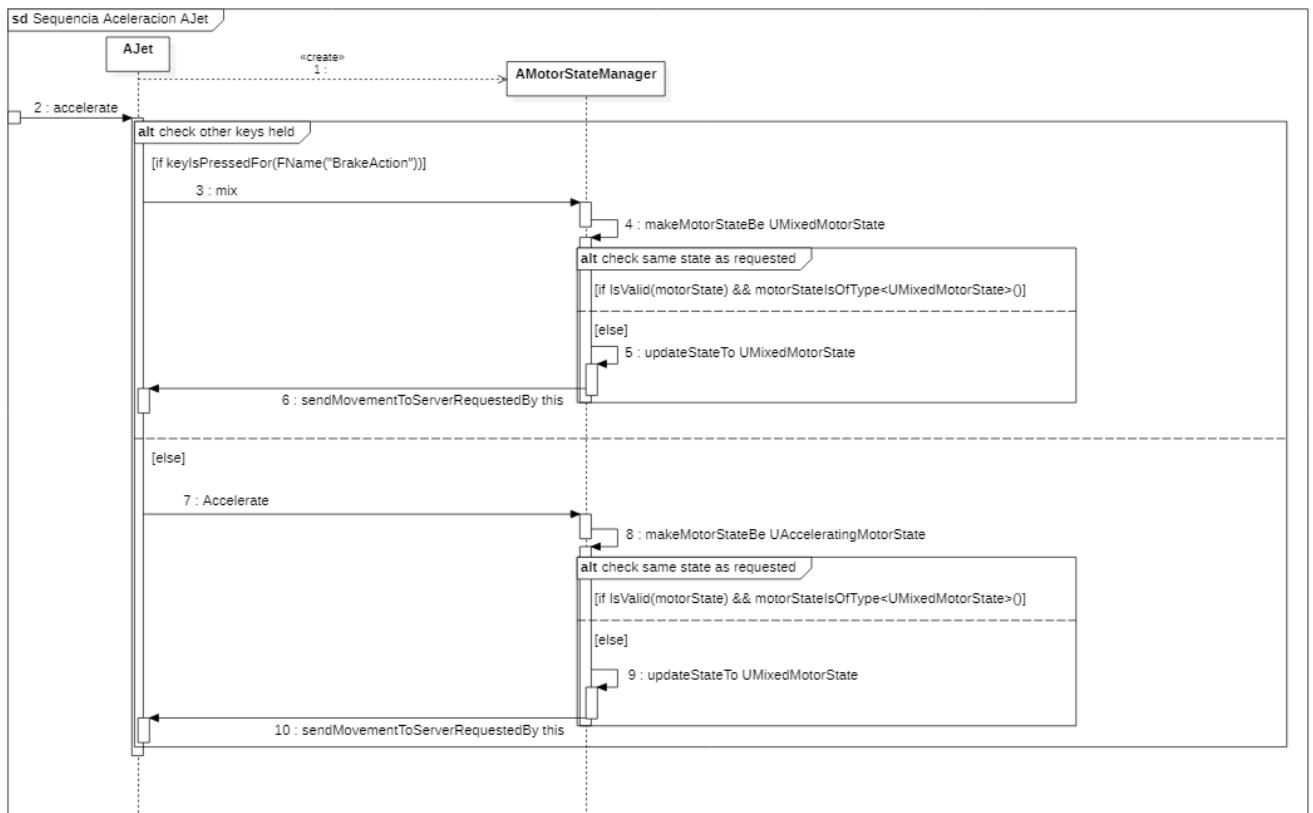


Ilustración 8. Diagrama UML de Secuencia de aceleración en AJet.

Luego de que se invoque accelerate() la instancia de AJet se pregunta si la tecla asignada a la acción de frenar (“BrakeAction”) está presionada. Se divide la lógica en dos casos: que esté presionada o no.

Si lo está, quiere decir que no se va a acelerar solamente, sino que se establecerá un estado mixto (donde se substraen la fuerza de frenado a la fuerza de aceleración). Entonces se le dice al administrador de motor propio (instancia de AMotorStateManager) que realice mix().

El administrador del motor a su vez se invocará el método makeMotorStateBe<UMixedMotorState>() (nótese que este método es uno parametrizable, que recibe el tipo de clase de UMotorState que se quiere establecer).

Dentro de este método se verificará que el estado que se quiere no es el que ya se posee (se verá su ventaja en un momento). Si ya se posee ese tipo de estado, se retorna. Si no, se actualiza el estado con updateStateTo<UMixedMotorState>(), el cual creará una instancia de esa clase de UMotorState y la establecerá como el estado del administrador del motor.

Luego, `makeMotorStateBe`, le dirá a la instancia de `AJet` que lo contiene `sendMovementToServerRequestedBy(this)` que verificará que el `AMotorStateManager` que se envía como parámetro es el propio y establecerá una variable booleana para que en el momento que se actualice el cuadro de `AJet`, envíe la solicitud al servidor de que hubo un cambio en los estados locales. Si no se verificase si el estado pedido es el que se tiene, en todo momento se estaría notificando al servidor el estado final, en vez de notificar solo cuando cambia.

Si la tecla de frenado no está presionada, se acelera normalmente, llamando `makeMotorStateBe` igual que en el caso anterior, pero utilizando `UAcceleratingMotorState` como tipo parametrizado.

Uno pensaría que `accelerate()` generaría la acción de acelerar. En cambio, solamente establece el estado del motor que se desea. Lo que genera la acción de acelerar es cuando, en la actualización del cuadro de `AJet`, `AJet` le dice a su `AMotorStateManager` `activate(physcisMeshComponent)`, que al `UStaticMeshComponent` pasado como parámetro le aplicará la fuerza necesaria para acelerarlo.

Ahora se sabe cómo acelera el `AJet`, invocando solamente `accelerate`, pero ¿cómo es que se transfiere la pulsación de una tecla en un teclado (o control) hacia el objeto `AJet`?

Para eso, primero se enlaza la acción de acelerar “`AccelerateAction`” con el método `accelerate()`, dentro de `SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)`, que es un método de `APawn` y es sobreescrito por `AJet`.

Para enlazar `accelerate()` con la acción, dentro de `SetupPlayerInputComponent` se hace:

```
PlayerInputComponent->BindAction("AccelerateAction", EInputEvent::IE_Pressed &AJet::accelerate);
```

Cuando un usuario presiona la tecla asociada a la acción de acelerar, se le pasa la tecla al `AProjectRPlayerController` (que posee un `AJet`) de esta forma:

```
InputKey(FKey Key, EInputEvent EventType, float AmountDepressed, bool bGamepad);
```

Que guardará la tecla presionada.

Luego, cuando se requiere procesar las teclas o botones presionados, se llama en el control a:

```
ProcessPlayerInput(const float DeltaTime, const bool bGamePaused)
```

Que generará una pila de entradas realizadas y le pedirá a su `UPlayerInput` que las procese. `UPlayerInput` tiene todos los enlaces a las acciones y con la pila de entrada invoca los métodos asignados a la acción que representa cada tecla. De esta forma finalmente se llama al método `accelerate()` de la instancia `AJet` que controla el control.

3.4.2 - Secuencia de Actualización de Posiciones

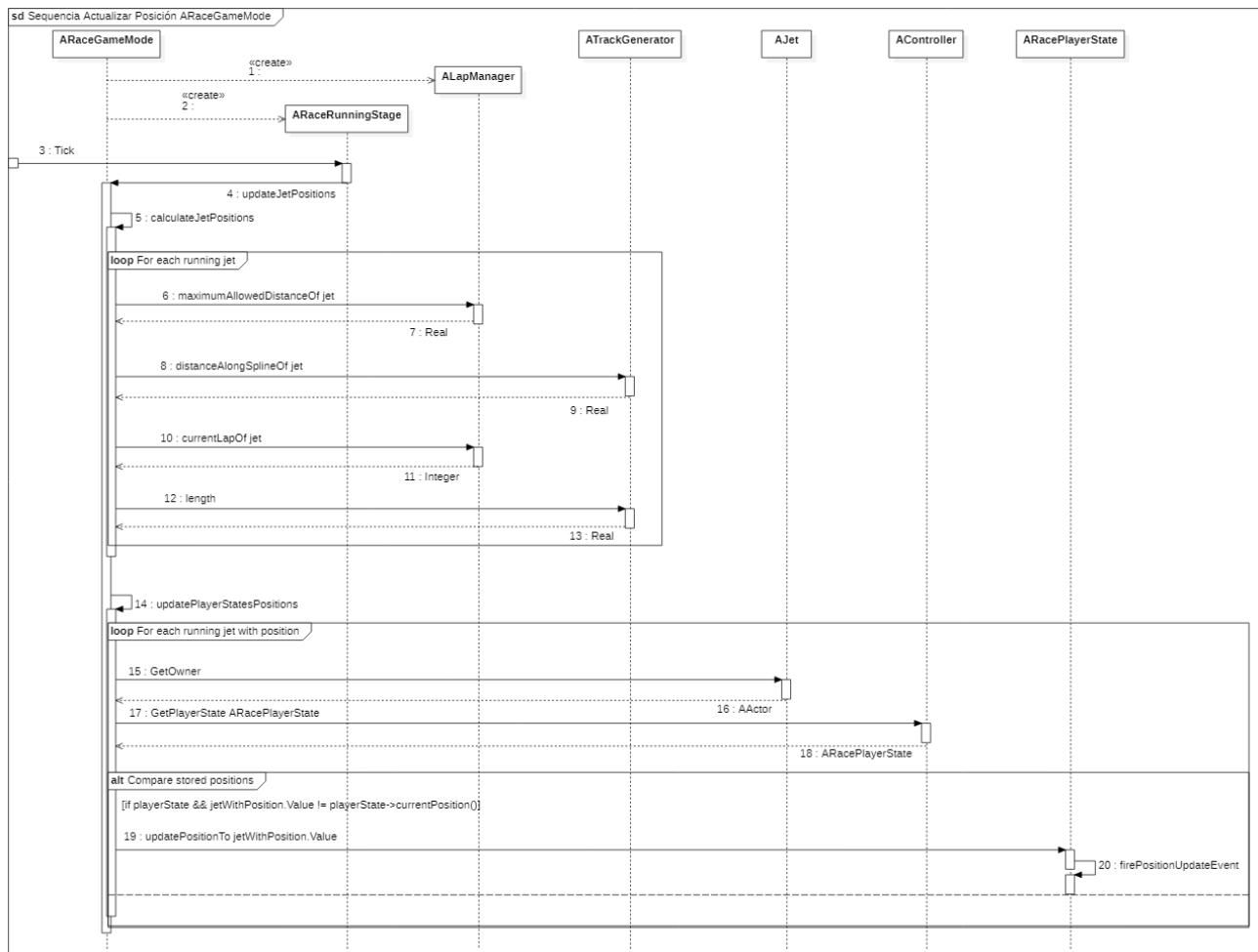


Ilustración 9. Diagrama UML de Secuencia actualización de posición en ARaceGameMode.

Una vez que ARaceGameMode se crea y se llega a la etapa ARaceRunningStage (etapa donde ya se está corriendo la carrera), cuando esta etapa tiene que actualizar su cuadro, por medio de Tick (como el resto de las clases), le dice al modo de juego updateJetPositions(), que se separa en dos métodos compuestos por ciclos: calculateJetPositions() y updatePlayerStatePositions().

calculateJetPositions(), por cada instancia de AJet que esté corriendo la carrera calculará la distancia total que recorrió (teniendo en cuenta ciertos casos para evitar trampas, como recorrer la carrera en sentido reverso). Para ello combinará los siguientes métodos:

- maximumAllowedDistanceOf(AJet* aJet), perteneciente a ALapManager. Según la fase de vuelta en la que una nave esté, no se puede exceder la distancia entre la fase siguiente y el inicio de la pista.
- distanceAlongSplineOf(AActor* anActor), perteneciente a ATrackGenerator. Dice la distancia total (siguiendo la curva que representa la pista) entre el inicio de la pista hasta el actor pasado como parámetro.
- currentLapOf(AJet* aJet), perteneciente a ALapManager. Retorna el número de vuelta en el que se encuentra la nave especificada.
- length(), perteneciente a ATrackGenerator. Devuelve la longitud de la pista.

Se utilizará el menor valor entre `maximumAllowedDistanceOf` y `distanceAlongSplineOf` para tener la distancia permitida de la nave.

Para saber el largo total que se recorrió por esa nave en este cuadro, se suma la distancia permitida de la nave a la multiplicación entre el número de vueltas recorridas por la nave y la longitud de la pista.

Al finalizar el ciclo, se tiene la distancia total que recorrió cada nave hasta ese momento.

Luego, se ordena cada distancia (junto a la nave a la que pertenece) de mayor a menor. Con ese mapa junto a la lista de finalistas se crea otra lista con los finalistas primero y los que siguen corriendo la carrera, después.

De esta forma se consigue la posición actual de cada nave.

`updatePlayerStatePositions()` es más simple ya que recorre todos los `ARacePlayerState` de cada control y si la posición es distinta a la que tiene guardada, se actualiza. Como esa variable es una replicada que dispara un método al copiarla a un cliente, `ARacePlayerState` invocará `firePositionUpdateEvent()` para informar a sus subscriptores (`URacePlayerUI`) que hubo un cambio en la variable (la interfaz de usuario actualizará su valor sin tener que estar verificando en todo momento).

Luego de terminar este ciclo, `ARaceGameMode` habrá actualizado las posiciones de cada `AJet` en juego e informará a cada `ARacePlayerState` que su posición se modificó, si es que realmente hubo una modificación.

3.4.3 Secuencia de actualización de vueltas

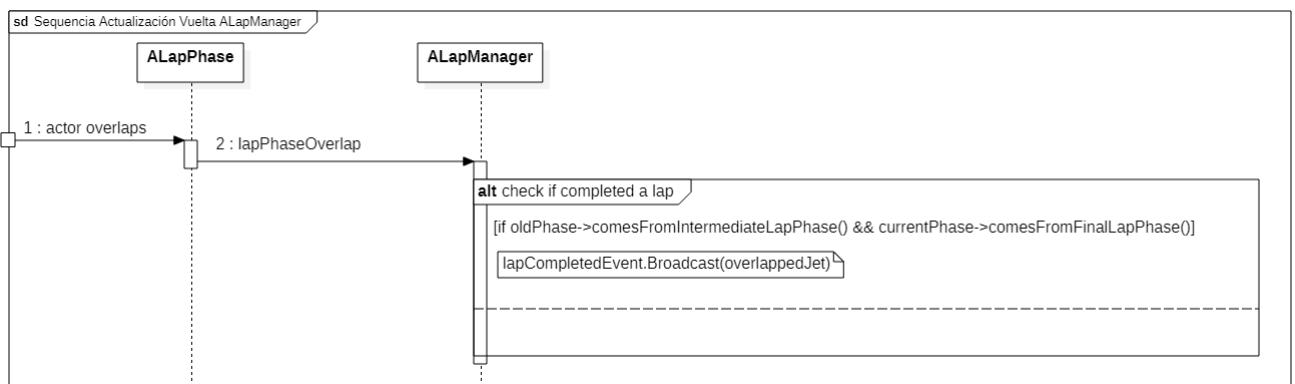


Ilustración 10. Diagrama UML de Secuencia de actualización de vueltas (porción de `ALapManager`).

Una vez que un `AActor` atraviesa una fase de vuelta suscrita en `ALapManager`, esta fase disparará el evento `OnActorBeginOverlap` que llamará a `lapPhaseOverlap` dentro de la instancia de `ALapManager`.

Esta instancia verificará si es que ese actor es un `AJet` y, si es que completó una vuelta, disparará su evento `lapCompletedEvent`.

Es una lógica simple, pero se estaría describiendo la mitad del proceso si solo se quedase en disparar el evento.

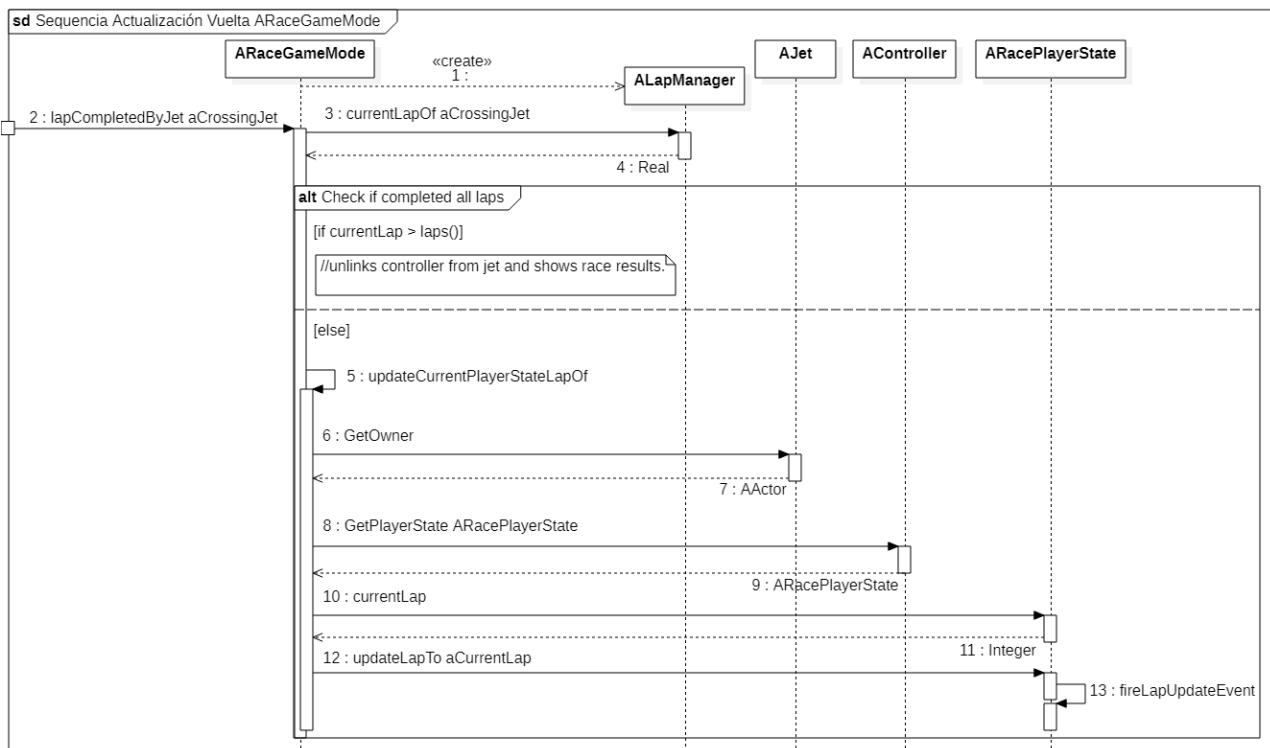


Ilustración 10. Diagrama UML de Secuencia de actualización de vueltas (porción de ARaceGameMode).

ARaceGameMode crea la única instancia de ALapManager en una carrera, cuando lo hace, se subscribe al evento lapCompletedEvent mediante su método `lapCompletedByJet(AJet* aCrossingJet)`.

`lapCompletedByJet` primero le pedirá a la instancia de ALapManager las vueltas recorridas por la nave pasada como parámetro. Se divide el método en dos casos: que las vueltas que realizó la nave son más que las totales o no.

Si la nave recorrió más que las totales (terminó la última vuelta o pasó otra vuelta luego de terminar la carrera), se desprenderá de su control (para inhabilitar que el jugador siga corriendo la carrera y pueda alterar el desempeño de otros jugadores), removerá la nave del conjunto de naves corriendo la carrera y la agregará a la colección de naves finalistas (una cola, para mantener el orden).

Si a la nave aún le quedan vueltas por recorrer, se llamará al método `updateCurrentPlayerStateLapOf(AJet* ajet, int aCurrentLap)` con la nave hablada como parámetro, junto con las vueltas que recorrió.

`updateCurrentPlayerStateLapOf` primero le pide a esa instancia de AJet su “dueño”, que es su control devuelto como AActor y transformado en AController.

A ese AController se le pide `GetPlayerState<ARacePlayerState>()`. Nótese que se utiliza una parametrización de tipo de clase para que el tipo de retorno sea del de esa parametrización.

Ai ARacePlayerState devuelto se le pedirá la vuelta actual. Si esa vuelta no coincide con la dada como parámetro, se le dirá `updateLapTo` al ARacePlayerState para que cambie su valor de vuelta y así dispare su `fireLapUpdateEvent()` cuando cada cliente reciba la actualización del valor (y la instancia de URacePlayerUI pueda actualizar el valor de su vuelta, al estar suscrita a ese evento).

De esta forma, se tiene una vista completa de cómo se actualiza una vuelta en

3.5 - Temas de Implementación

- **Configuración de pruebas.** Establecer el entorno CI resultó fácil utilizando Jenkins. Del lado de Unreal Engine hubo una mayor dificultad por la escasa documentación que se tiene. La configuración de pruebas tomó el primer mes de desarrollo del proyecto.
- **Creación de pruebas.** Encontrar la forma de simular una instancia de juego, entradas de usuarios y replicación fue costoso ya que no hay documentación fácilmente accesible de pruebas similares hechas por otros desarrolladores.
- **Documentación del motor ambigua y escasa.** En muchas ocasiones se tuvo que recorrer código del proyecto para entender la forma de funcionamiento de algunos métodos provistos. La documentación oficial provee ejemplos muy cortos y poco claros. Existe un foro oficial de consultas, pero para temas específicos hay muy poca información.
- **El sistema no permite usar todas las características de C++.** El compilador propio junto con el sistema de reflexión imposibilita utilizar características del lenguaje no abarcadas por ellos, lo que hace que ciertas soluciones se tengan que repensar para adaptarlas al motor.
- **Clases del motor muy grandes.** Esto provocó que muchas veces se tenga que releer el código para entender la secuencia lógica de ciertos métodos.
- **Pequeños detalles arbitrarios para implementar código del juego retrasaban el desarrollo.** Por ejemplo, variables que se quieran exponer al sistema de propiedades deben tener visibilidad al menos Protected porque de lo contrario, pueden ocurrir errores extraños cuando se ejecute el código. Otro ejemplo es que algunos métodos que se ejecuten dentro del método Tick del motor, tienen que estar declarados como UFUNCTION(). Se pudo darse cuenta de esta información solo por comentarios hechos en los foros por otros desarrolladores.

4 - Análisis de la Arquitectura

4.1 - Análisis de Escenarios

- **Implementar Continuous Delivery.** Como uno de los objetivos del proyecto es prepararlo para ser un producto comercial, será necesario extender el entorno de CI creado para aprovechar las ventajas que provee Continuous Delivery en producción.
- **Implementar sonido.** Para la presentación del trabajo final se omitió esta característica ya que su adición no era fundamental. Para volverlo un producto, va a ser indispensable implementar sonido. No es una característica difícil de implementar, pero es sumamente necesaria para el juego.
- **Agregar más características para volverlo un juego completo.** Inteligencia artificial, más modos de juego, torneos, turbo en naves, efectos especiales, agregar sistema de desafíos y recompensas. Todo esto y más aumentará el valor del juego, así como también su reusabilidad. Se podría analizar el uso de automatización de algunos procesos para generar contenido del mismo tipo dentro del juego.
- **Adoptar servicios de juego online.** Se necesitará utilizar un servicio externo que provea características de conexión online, rankings, lobbies de juego, Matchmaking (emparejamiento de jugadores según su desempeño y/o estabilidad de conexión), etc.
- **Mejorar y pulir interfaces y clases.** Especial atención debe darse a la clase AJet, ya que tiene varios componentes que pueden simplificarse aún más, como se habló en otras secciones.
- **Profundizar el “Gameplay”.** Se debe hacer un análisis exhaustivo de la comodidad y respuesta de los controles por parte de usuarios para garantizar una experiencia de juego elevada.
- **Considerar la migración a otro motor en el futuro.** Se debe pensar seriamente si se desea utilizar Unreal Engine 4 en otros proyectos ya que posee algunas inconveniencias. Se tendría que analizar si vale la pena utilizar otro motor o crear uno propio, para compensar estas inconveniencias, o si es que son pequeños detalles que se vuelven mínimos frente a la amplia gama de herramientas ya existentes en el motor.

Conclusiones y Aportes al Tema

Como puede verse, realizar pruebas en Unreal Engine requiere de antemano una gran configuración y conocimiento del funcionamiento del motor. Esto se podría reducir si por defecto existiesen más comandos latentes de parte del motor para personalizar las pruebas, además de mejorar la documentación que se ofrece oficialmente.

Test Driven Development resultó ser una muy buena herramienta para el desarrollo del trabajo. El desarrollo dirigido por pruebas da seguridad sobre lo que se hace y enfoca al programador en la tarea encomendada, que es que el resultado de la prueba sea correcto.

Esto no quiere decir que sea la única forma de pruebas, ya que otros errores se encontraron mientras se jugaba el juego. Por otro lado, que haya pruebas realizadas y sean satisfactorias reduce el conjunto posible de espacios en los que el error se manifiesta.

Tener un entorno de CI (aunque haya sido pequeño) resultó muy conveniente a la hora de correr pruebas automáticamente cuando se realizaban cambios. Se llegó a tener tiempos de compilación de dos minutos y tiempos de testeo de 15 minutos. Si se hubiera hecho lo mismo de forma manual, se hubiera perdido muchísimo tiempo en esperar el resultado de las pruebas en el editor del motor.

El autor considera indispensable tener un entorno de CI para el desarrollo (y de CD para producción) en un videojuego. Intervienen muchos elementos y la realización de pruebas automáticas genera un diagnóstico temprano de cambios implementados, así como también una reducción grande de tiempo en tareas que se hubiesen hecho manualmente.

Unreal Engine 4 tiene un grave problema y es la falta de documentación unido a que uno tiene que confiar en soluciones de otros usuarios con problemas similares y la falta de apoyo oficial a esas soluciones.

Ciertas decisiones de diseño del motor no le parecen acertadas al autor, como tener un uso excesivo de macros, como en el caso del Unreal Property System (sistema de reflexión). También esto fuerza a programar de una manera en la que se viola el principio de encapsulación por medio de macros para exponer variables y métodos al sistema de reflexión.

Además, el hecho de que se cree una instancia por defecto de cada clase para el sistema de reflexión hace que ciertas configuraciones o llamadas a métodos en constructores generen advertencias en el momento de ejecución del programa, sin previo aviso.

Otras cosas que generan ruido son “convenciones ocultas” como tener que declarar a un puntero a un UObject como UPROPERTY() para evitar que el recolector de basura elimine el objeto.

Por otro lado, el autor cree que la interacción con el motor es escasa como para saber si en realidad son decisiones desacertadas, fueron decisiones necesarias en ese momento por falta de otras mejores o verdaderamente son las decisiones que se deben tomar en el desarrollo de un framework tan grande y que posee ya más de 20 años.

Aun así, la duda de crear un motor propio y moderno o seguir con Unreal Engine 4 solo se puede esclarecer teniendo un mayor contacto con el motor (según el autor).

Por suerte, el sistema de logs del motor es suficientemente potente como para encontrar fallas y sus causas.

Lo que se pudo aprender luego de 8 meses en el desarrollo del videojuego, 49 clases de objetos, más de 2470 commits en el repositorio de cambios y más de 340 pruebas realizadas en TDD es no solo útil para la aplicación en Unreal Engine 4 sino que sirvió para tener una mejor idea del desarrollo general en un videojuego.

Todavía existen muchos temas que no se abordaron y que quedan pendientes para convertir este proyecto en un producto comercial, pero lo aprendido servirá como base para futuros proyectos.

Temas Abiertos

Como se habló en la conclusión existen bastantes temas no abordados en el proyecto para convertirlo en un producto comercial.

Debería hacerse un análisis de GPU y CPU más exhaustivo en distintos entornos.

En ningún momento se implementó el sonido en el proyecto, lo que es primordial en un producto comercial. No es un tema complicado, pero se debe aclarar que no se trató en el desarrollo.

Un mejor entendimiento en modelos, texturas e iluminación debería ser necesario, o contratar personas que se especialicen en esos temas.

No se habló de Continuous Delivery ya que pertenece al área de producción más que desarrollo, aunque se va a tener que solucionar más adelante.

Tampoco se tomó ventaja de paralelización y es un tema que al autor le gustaría abordar en el futuro.

El uso de Behaviour Driven Development se descartó, pero puede tenerse en cuenta para otros proyectos.

Cómo agregar nuevas características para el producto es algo de lo que tampoco se planificó, pero debería hacerse para evitar tener un exceso de ellas (feature bloating o feature creep¹³¹).

Para poder tener sesiones online, se debería asociarse a algún servicio que provea Online Services, como GameTech (Amazon)¹³² Playfab (Microsoft)¹³³ o Steamworks (Valve)¹³⁴.

Se debería también registrarse en los portales de desarrollo para consolas de videojuegos de Nintendo¹³⁵, Sony¹³⁶ y/o Microsoft¹³⁷ para tener acceso a sus kits de desarrollo de software (SDKs).

Referencias Bibliográficas

-
- ¹ *15 Great Games That Use The Unreal 4 Game Engine* [en línea]. <<https://www.thegamer.com/great-games-use-unreal-4-game-engine/>> [Consulta: 22 de septiembre de 2020].
- ² *List of Unreal Engine games* [en línea]. <https://en.wikipedia.org/wiki/List_of_Unreal_Engine_games#Unreal_Engine_4> [Consulta: 22 de septiembre de 2020].
- ³ *Unreal Engine 4* [en línea]. <<https://www.unrealengine.com/en-US/>> [Consulta: 24 de septiembre de 2020].
- ⁴ *A world of possibilities* [en línea]. <<https://www.unrealengine.com/en-US/industry/games>> [Consulta: 24 de septiembre de 2020].
- ⁵ *Unreal Engine 4 Features* [en línea]. <<https://www.unrealengine.com/en-US/features>> [Consulta: 24 de septiembre de 2020].
- ⁶ *Downloading Unreal Engine Source Code* [en línea]. <<https://docs.unrealengine.com/en-US/GettingStarted/DownloadingUnrealEngine/index.html>> [Consulta: 24 de septiembre de 2020].
- ⁷ *Unreal Property System (Reflection)* [en línea]. <<https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection>> [Consulta: 24 de septiembre de 2020].
- ⁸ *Blueprints Visual Scripting* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html>> [Consulta: 24 de septiembre de 2020].
- ⁹ *Introduction to Blueprints* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html>> [Consulta: 24 de septiembre de 2020].
- ¹⁰ *UObject* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/CoreUObject/UObject/UObject/index.html>> [Consulta: 24 de septiembre de 2020].
- ¹¹ *Programming Guide* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/index.html>> [Consulta: 24 de septiembre de 2020].
- ¹² *Actors and Geometry* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Actors/index.html>> [Consulta: 24 de septiembre de 2020].
- ¹³ *AActor* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AActor/index.html>> [Consulta: 24 de septiembre de 2020].
- ¹⁴ *Components* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Actors/Components/index.html>> [Consulta: 24 de septiembre de 2020].
- ¹⁵ *Components Window* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Components/index.html>> [Consulta: 24 de septiembre de 2020].

¹⁶ *Components* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Components/index.html>> [Consulta: 24 de septiembre de 2020].

¹⁷ *UActorComponent* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Components/UActorComponent/index.html>> [Consulta: 24 de septiembre de 2020].

¹⁸ *Pawn* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn/index.html>> [Consulta: 24 de septiembre de 2020].

¹⁹ *Possessing Pawns* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/HowTo/PossessPawns/index.html>> [Consulta: 24 de septiembre de 2020].

²⁰ *APawn* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/APawn/index.html>> [Consulta: 24 de septiembre de 2020].

²¹ *Character* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn/Character/index.html>> [Consulta: 24 de septiembre de 2020].

²² *Setting Up a Character* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Animation/CharacterSetupOverview/index.html>> [Consulta: 24 de septiembre de 2020].

²³ *Setting Up Character Movement in Blueprints* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/HowTo/CharacterMovement/Blueprints/index.html>> [Consulta: 24 de septiembre de 2020].

²⁴ *ACharacter* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/ACharacter/index.html>> [Consulta: 24 de septiembre de 2020].

²⁵ *PlayerController* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/PlayerController/index.html>> [Consulta: 24 de septiembre de 2020].

²⁶ *APlayerController* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/APlayerController/index.html>> [Consulta: 24 de septiembre de 2020].

²⁷ *AIController* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/AIController/index.html>> [Consulta: 24 de septiembre de 2020].

²⁸ *AAIController* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/AIModule/AAIController/index.html>> [Consulta: 24 de septiembre de 2020].

²⁹ *Geometry Brush Actors* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Actors/Brushes/index.html>> [Consulta: 24 de septiembre de 2020].

³⁰ *Geometry Editing Content Examples* [en línea]. <<https://docs.unrealengine.com/en-US/Resources/ContentExamples/Brushes/index.html>> [Consulta: 24 de septiembre de 2020].

³¹ *ABrush* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/ABrush/index.html>> [Consulta: 24 de septiembre de 2020].

³² *FSlateBrush* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/SlateCore/Styling/FSlateBrush/index.html>> [Consulta: 24 de septiembre de 2020].

³³ *UBrushComponent* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Components/UBrushComponent/index.html>> [Consulta: 24 de septiembre de 2020].

³⁴ *Levels* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/Levels/index.html>> [Consulta: 24 de septiembre de 2020].

³⁵ *Level Editor* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/UI/LevelEditor/index.html>> [Consulta: 24 de septiembre de 2020].

³⁶ *Level Design Content Examples* [en línea]. <<https://docs.unrealengine.com/en-US/Resources/ContentExamples/LevelDesign/index.html>> [Consulta: 24 de septiembre de 2020].

³⁷ *World Composition User Guide* [en línea]. <<https://docs.unrealengine.com/en-US/Engine/LevelStreaming/WorldBrowser/index.html>> [Consulta: 24 de septiembre de 2020].

³⁸ *UWorld* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/UWorld/index.html>> [Consulta: 24 de septiembre de 2020].

³⁹ *UGameInstance* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/UGameInstance/index.html>> [Consulta: 20 de abril de 2021].

⁴⁰ *Game Mode and Game State* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/GameMode/index.html>> [Consulta: 24 de septiembre de 2020].

⁴¹ *AGameModeBase* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AGameModeBase/index.html>> [Consulta: 24 de septiembre de 2020].

⁴² *AGameMode* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AGameMode/index.html>> [Consulta: 24 de septiembre de 2020].

⁴³ *Game Mode and Game State* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/GameMode/index.html#gamestate>> [Consulta: 24 de septiembre de 2020].

⁴⁴ *AGameState* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AGameState/index.html>> [Consulta: 24 de septiembre de 2020].

⁴⁵ *GamePlay Framework Quick Reference* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/Framework/QuickReference/index.html>> [Consulta: 24 de septiembre de 2020].

⁴⁶ *APlayerState* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/APlayerState/index.html>> [Consulta: 24 de septiembre de 2020].

⁴⁷ *Unreal Architecture* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/index.html>> [Consulta: 24 de septiembre de 2020].

⁴⁸ *Gameplay Guide* [en línea]. <<https://docs.unrealengine.com/en-US/Gameplay/index.html>> [Consulta: 24 de septiembre de 2020].

⁴⁹ *Programming Guide* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/index.html>> [Consulta: 24 de septiembre de 2020].

⁵⁰ *Actor LifeCycle* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Actors/ActorLifecycle/index.html>> [Consulta: 25 de septiembre de 2020].

⁵¹ *TestDrivenDevelopment* [en línea]. <<https://martinfowler.com/bliki/TestDrivenDevelopment.html>> [Consulta: 22 de septiembre de 2020].

⁵² *Continuous Integration* [en línea]. <<https://martinfowler.com/articles/continuousIntegration.html>> [Consulta: 22 de septiembre de 2020].

⁵³ *Jenkins* [en línea]. <<https://www.jenkins.io/>> [Consulta: 22 de septiembre de 2020].

⁵⁴ *ngrok* [en línea]. <<https://ngrok.com/>> [Consulta: 22 de septiembre de 2020].

⁵⁵ *OpenCppCoverage* [en línea]. <<https://github.com/OpenCppCoverage/OpenCppCoverage>> [Consulta: 22 de septiembre de 2020].

⁵⁶ *Automation System Overview* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Automation/index.html>> [Consulta: 22 de septiembre de 2020].

⁵⁷ *Functional Testing* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Automation/FunctionalTesting/index.html>> [Consulta: 22 de septiembre de 2020].

⁵⁸ *Screenshot Comparison Tool* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Automation/ScreenShotComparison/index.html>> [Consulta: 22 de septiembre de 2020].

⁵⁹ *FBX Test Builder* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Automation/FBXTestBuilder/index.html>> [Consulta: 22 de septiembre de 2020].

⁶⁰ *FAutomationTestBase* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Core/Misc/FAutomationTestBase/index.html>> [Consulta: 22 de septiembre de 2020].

⁶¹ *Automation Technical Guide* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Automation/TechnicalGuide/index.html>> [Consulta: 22 de septiembre de 2020].

⁶² *Automation Spec* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Automation/AutomationSpec/index.html>> [Consulta: 22 de septiembre de 2020].

⁶³ *Coding Standard* [en línea]. <<https://docs.unrealengine.com/en-US/Programming/Development/CodingStandard/index.html>> [Consulta: 22 de septiembre de 2020].

⁶⁴ *Creating an Editor Module In Unreal Engine 4* [en línea]. <<http://cairansteverink.nl/cairansteverink/blog/creating-an-editor-module-in-unreal-engine-4/>> [Consulta: 16 de abril de 2021].

⁶⁵ *Creating C++ module* [en línea]. <<https://www.ue4community.wiki/creating-cpp-module-oshdsg2t>> [Consulta: 16 de abril de 2021].

⁶⁶ *Public and Private dependency modules* [en línea]. <<https://answers.unrealengine.com/questions/225405/public-and-private-dependency-modules.html>> [Consulta: 16 de abril de 2021].

⁶⁷ *EHostType::Type* [en línea]. <https://docs.unrealengine.com/en-US/API/Runtime/Projects/EHostType_Type/index.html> [Consulta: 16 de abril de 2021].

⁶⁸ *Modules* [en línea]. <<https://docs.unrealengine.com/en-US/ProductionPipelines/BuildTools/UnrealBuildTool/ModuleFiles/index.html>> [Consulta: 16 de abril de 2021].

⁶⁹ *ELoadingPhase::Type* [en línea]. <https://docs.unrealengine.com/en-US/API/Runtime/Projects/ELoadingPhase_Type/index.html> [Consulta: 16 de abril de 2021].

⁷⁰ *How to Include files from another module* [en línea]. <<https://answers.unrealengine.com/questions/54681/how-to-include-files-from-another-module.html>> [Consulta: 16 de abril de 2021].

⁷¹ *UUserWidget* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/UMG/Blueprint/UUserWidget/index.html>> [Consulta: 21 de abril de 2021].

⁷² *Static Mesh Components* [en línea]. <<https://docs.unrealengine.com/en-US/Basics/Components/StaticMesh/index.html>> [Consulta: 21 de abril de 2021].

⁷³ *UStaticMeshComponent* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Components/UStaticMeshComponent/index.html>> [Consulta: 21 de abril de 2021].

⁷⁴ *UCameraComponent* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/UCameraComponent/index.html>> [Consulta: 21 de abril de 2021].

⁷⁵ *Using Cameras* [en línea]. <<https://docs.unrealengine.com/en-US/InteractiveExperiences/UsingCameras/index.html>> [Consulta: 21 de abril de 2021].

⁷⁶ *Trabajo* [en línea]. <[https://es.wikipedia.org/wiki/Trabajo_\(f%C3%ADsica\)](https://es.wikipedia.org/wiki/Trabajo_(f%C3%ADsica))> [Consulta: 22 de abril de 2021].

⁷⁷ *Energía Cinética* [en línea] <https://es.wikipedia.org/wiki/Energ%C3%ADa_cin%C3%A9tica> [Consulta: 22 de abril de 2021].

⁷⁸ *Fast-Paced Multiplayer (Part I): Client-Server Game Architecture* [en línea]. <<https://www.gabrielgambetta.com/client-server-game-architecture.html>> [Consulta: 26 de abril de 2021].

⁷⁹ *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization* [en línea]. <<https://web.cs.wpi.edu/~claypool/courses/4513-B03/papers/games/bernier.pdf>> [Consulta: 26 de abril de 2021].

⁸⁰ *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization* [en línea]. <https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization> [Consulta: 26 de abril de 2021].

⁸¹ *Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation* [en línea]. <<https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html>> [Consulta: 26 de abril de 2021].

⁸² *Movement Prediction* [en línea].

<https://www.gamasutra.com/blogs/BartlomiejWaszak/20181104/329703/Movement_Prediction.php> [Consulta: 27 de abril de 2021].

⁸³ *FSlateApplication* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Slate/Framework/Application/FSlateApplication/index.html>> [Consulta: 28 de abril de 2021].

⁸⁴ *UEditorEngine* [en línea]. <<https://docs.unrealengine.com/en-US/API/Editor/UnrealEd/Editor/UEditorEngine/index.html>> [Consulta: 28 de abril de 2021].

⁸⁵ *UUnrealEdEngine* [en línea]. <<https://docs.unrealengine.com/en-US/API/Editor/UnrealEd/Editor/UUnrealEdEngine/index.html>> [Consulta: 28 de abril de 2021].

⁸⁶ *FRequestPlaySessionParams* [en línea]. <<https://docs.unrealengine.com/en-US/API/Editor/UnrealEd/FRequestPlaySessionParams/index.html>> [Consulta: 28 de abril de 2021].

⁸⁷ *ULevelEditorPlaySettings* [en línea]. <<https://docs.unrealengine.com/en-US/API/Editor/UnrealEd/Settings/ULevelEditorPlaySettings/index.html>> [Consulta: 28 de abril de 2021].

⁸⁸ *Performance and Profiling Overview* [en línea]. <<https://docs.unrealengine.com/en-US/TestingAndOptimization/PerformanceAndProfiling/Overview/index.html>> [Consulta: 30 de abril de 2021].

⁸⁹ *Stat Commands* [en línea]. <<https://docs.unrealengine.com/en-US/TestingAndOptimization/PerformanceAndProfiling/StatCommands/index.html#startfile>> [Consulta: 30 de abril de 2021].

⁹⁰ *Unreal Frontend* [en línea]. <<https://docs.unrealengine.com/en-US/SharingAndReleasing/Deployment/UnrealFrontend/index.html>> [Consulta: 30 de abril de 2021].

⁹¹ *GPU Profiling* [en línea]. <<https://docs.unrealengine.com/en-US/TestingAndOptimization/PerformanceAndProfiling/GPU/index.html>> [Consulta: 30 de abril de 2021].

⁹² *ShadowDepths in GPU visualizar hurting performance* [en línea]. <<https://forums.unrealengine.com/t/shadowdepths-in-gpu-visualizer-hurting-performance/73584>> [Consulta: 30 de abril de 2021].

⁹³ *UV Editor – Blender* [en línea]. <<https://docs.blender.org/manual/en/latest/editors/uv/introduction.html#uvs-explained>> [Consulta: 30 de abril de 2021].

⁹⁴ *My UV maps Overlap* [en línea]. <<https://blender.stackexchange.com/questions/73243/my-uv-maps-overlap>> [Consulta: 30 de abril de 2021].

⁹⁵ *Level Of Detail* [en línea]. <[https://en.wikipedia.org/wiki/Level_of_detail_\(computer_graphics\)](https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics))> [Consulta: 30 de abril de 2021].

⁹⁶ *CPU Profiling* [en línea]. <<https://docs.unrealengine.com/en-US/TestingAndOptimization/PerformanceAndProfiling/CPU/index.html>> [Consulta: 30 de abril de 2021].

⁹⁷ *Network Profiler* [en línea]. <<https://docs.unrealengine.com/en-US/InteractiveExperiences/Networking/NetworkProfiler/index.html>> [Consulta: 30 de abril de 2021].

⁹⁸ *Networking in UE4: Server Optimizations* [en línea]. <<https://www.youtube.com/watch?v=mT8VUVuk-CY>> [Consulta: 30 de abril de 2021].

⁹⁹ *Clumsy 0.2* [en línea]. <<https://jagt.github.io/clumsy/>> [Consulta: 30 de abril de 2021].

¹⁰⁰ *ENetDormancy* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/ENetDormancy/index.html>> [Consulta: 30 de abril de 2021].

¹⁰¹ *Event-Driven Architecture* [en línea]. <https://en.wikipedia.org/wiki/Event-driven_architecture> [Consulta: 4 de mayo de 2021].

¹⁰² *Delegates* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/index.html#declaringdelegates>> [Consulta: 4 de mayo de 2021].

¹⁰³ *Multi-Cast Delegates* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/Multicast/index.html>> [Consulta: 4 de mayo de 2021].

¹⁰⁴ *Events* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/Events/index.html>> [Consulta: 4 de mayo de 2021].

¹⁰⁵ *Dynamic Delegates* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/Dynamic/index.html>> [Consulta: 4 de mayo de 2021].

¹⁰⁶ *Unreal Property System* [en línea]. <<https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection>> [Consulta: 4 de mayo de 2021].

¹⁰⁷ *Class Specifiers* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/GameplayArchitecture/Classes/Specifiers/index.html>> [Consulta: 4 de mayo de 2021].

¹⁰⁸ *Properties* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/GameplayArchitecture/Properties/index.html>> [Consulta: 4 de mayo de 2021].

¹⁰⁹ *Property Replication* [en línea]. <<https://docs.unrealengine.com/en-US/InteractiveExperiences/Networking/Actors/Properties/index.html>> [Consulta: 4 de mayo de 2021].

¹¹⁰ *UFunctions* [en línea]. <<https://docs.unrealengine.com/en-US/ProgrammingAndScripting/GameplayArchitecture/Functions/index.html>> [Consulta: 4 de mayo de 2021].

¹¹¹ *Remote Procedure Calls* [en línea]. <<https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection>> [Consulta: 4 de mayo de 2021].

¹¹² *Delegation Pattern* [en línea]. <https://en.wikipedia.org/wiki/Delegation_pattern> [Consulta: 15 de mayo de 2021].

¹¹³ *Friend UClasses not working?* [en línea].

<<https://answers.unrealengine.com/questions/82544/friend-uclasses-not-working.html>> [Consulta: 15 de mayo de 2021].

¹¹⁴ *Template UClass or an other way ?* [en línea].

<<https://forums.unrealengine.com/t/template-uclass-or-an-other-way/142160>> [Consulta: 15 de mayo de 2021].

¹¹⁵ *TBaseDynamicDelegate* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Core/Delegates/TBaseDynamicDelegate/index.html>> [Consulta: 15 de mayo de 2021].

¹¹⁶ *TMulticastScriptDelegate* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Core/UObject/TMulticastScriptDelegate/index.html>> [Consulta: 15 de mayo de 2021].

¹¹⁷ *TScriptDelegate* [en línea]. <<https://docs.unrealengine.com/en-US/API/Runtime/Core/UObject/TScriptDelegate/index.html>> [Consulta: 15 de mayo de 2021].

¹¹⁸ *4+1 architectural view model* [en línea].

<https://en.wikipedia.org/wiki/4%2B1_architectural_view_model> [Consulta: 18 de mayo de 2021].

¹¹⁹ *What is a DLL* [en línea]. <<https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library>> [Consulta: 20 de mayo de 2021].

¹²⁰ *Dynamic-Link Library* [en línea]. <https://en.wikipedia.org/wiki/Dynamic-link_library> [Consulta: 20 de mayo de 2021].

¹²¹ *NVIDIA Nsight Aftermath SDK* [en línea]. <<https://developer.nvidia.com/nsight-aftermath>> [Consulta: 20 de mayo de 2021].

¹²² *APEX: Deployment: The legacy module* [en línea].

<https://gameworksdocs.nvidia.com/APEX/1.4/docs/APEX_Framework/APEX_PG_Deployment.html#the-legacy-module> [Consulta: 20 de mayo de 2021].

¹²³ *PhysX SDK* [en línea]. <<https://developer.nvidia.com/physx-sdk>> [Consulta: 20 de mayo de 2021].

¹²⁴ *The Ogg container format* [en línea]. <<https://www.xiph.org/ogg/>> [Consulta: 20 de mayo de 2021].

¹²⁵ *Vorbis audio compression* [en línea]. <<https://xiph.org/vorbis/>> [Consulta: 20 de mayo de 2021].

¹²⁶ *XAudio2 Introduction* [en línea]. <<https://docs.microsoft.com/en-us/windows/win32/xaudio2/xaudio2-introduction>> [Consulta: 20 de mayo de 2021].

¹²⁷ *OpenVR SDK* [en línea]. <<https://github.com/ValveSoftware/openvr>> [Consulta: 20 de mayo de 2021].

¹²⁸ *NVIDIA APEX Clothing* [en línea]. <<https://www.nvidia.com/en-us/drivers/apex-clothing/>> [Consulta: 20 de mayo de 2021].

¹²⁹ *NV Cloth* [en línea].

<<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/nvCloth/index.html>> [Consulta: 20 de mayo de 2021].

¹³⁰ *Sequence Diagram* [en línea]. <https://en.wikipedia.org/wiki/Sequence_diagram>

[Consulta: 18 de mayo de 2021].

¹³¹ *Feature Creep* [en línea]. <https://en.wikipedia.org/wiki/Feature_creep> [Consulta: 30 de abril de 2021].

¹³² *AWS Game Tech* [en línea]. <<https://aws.amazon.com/gametech/>> [Consulta: 30 de abril de 2021].

¹³³ *Microsoft Azure PlayFab* [en línea]. <<https://playfab.com/>> [Consulta: 30 de abril de 2021].

¹³⁴ *Steamworks* [en línea]. <<https://partner.steamgames.com/>> [Consulta: 30 de abril de 2021].

¹³⁵ *Nintendo Developer Portal* [en línea]. <<https://developer.nintendo.com/>> [Consulta: 30 de abril de 2021].

¹³⁶ *PlayStation Partners* [en línea]. <<https://partners.playstation.net/>> [Consulta: 30 de abril de 2021].

¹³⁷ *Developing Games |Xbox and Windows 10* [en línea]. <<https://www.xbox.com/en-US/developers>> [Consulta: 30 de abril de 2021].