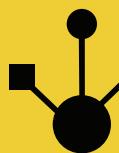




Escuela  
Politécnica  
Superior

# UnrealHome: Reproducción de comportamientos en entornos virtuales para la generación de datasets sintéticos



Grado en Ingeniería Multimedia

## Trabajo Fin de Grado

Autor:

César Moltó Morilla

Tutor/es:

José García Rodríguez

Pablo Martínez González

Enero 2022



Universitat d'Alacant  
Universidad de Alicante



# UnrealHome: Reproducción de comportamientos en entornos virtuales para la generación de datasets sintéticos

---

## Autor

César Moltó Morilla

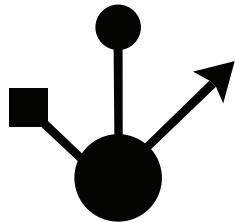
## Tutor/es

José García Rodríguez

*Departamento de Tecnología Informática y Computación*

Pablo Martínez González

*Departamento de Tecnología Informática y Computación*



Grado en Ingeniería Multimedia



Escuela  
Politécnica  
Superior



Universitat d'Alacant  
Universidad de Alicante



# Resumen

En este proyecto, proponemos la creación de UnrealHome<sup>1</sup>, una plataforma de simulación diseñada para llevar a cabo la generación de datos sintéticos visuales fotorrealistas de interacciones humanas con objetos interactivos en entornos domésticos virtuales.

UnrealHome ha sido desarrollado en el motor de juegos Unreal Engine 4 (UE), y está inspirado en otros proyectos similares como UnrealRox [17], VirtualHome [23] y ElderSim [11]. Este software, está compuesto por dos tipos de actores (avatares y objetos), que intervienen directamente en la ejecución de las simulaciones, y un total de tres componentes o subsistemas responsables de realizar todos los subprocesos necesarios para asegurar el correcto funcionamiento de su lógica interna. Entre estos subprocesos destacan la reproducción e interpolación de animaciones realizadas por el sistema de animación, el desplazamiento de los avatares por la escena llevada a cabo por el sistema de navegación con *pathfinding* y los agarres de objetos por parte de los avatares gestionados por el sistema de agarres.

Por último, en este proyecto también se ha perseguido conseguir que la generación de datos sintéticos visuales sea más interactiva e intuitiva. Por ello, UnrealHome posibilita el uso de mecánicas *Point and Click* en tiempo real durante las simulaciones con el objetivo de enviar instrucciones a los avatares de la escena y que estos realicen nuevas interacciones definidas por dichas órdenes.

---

<sup>1</sup><https://github.com/3dperceptionlab/unrealhome>



# **Agradecimientos**

Este proyecto no habría sido posible sin mi tutor, José García, quién me propuso la temática del trabajo y me ha ayudado en todo lo posible durante este año. También quiero dar las gracias, por el apoyo y los consejos, a los miembros del Departamento de Tecnología Informática y Computación, en particular a Pablo Martínez , John Castro y Sergiu Ovidiu. Por último, quiero agradecer a mis amigos y familia por su ánimo e interés incondicionales durante todo este proceso. Es a todos ellos a quien dedico este trabajo.



*Un hombre que se atreve a perder una hora  
no ha descubierto el valor de la vida*

Charles Darwin.



# Índice general

## **Lista de Acrónimos y Abreviaturas**

xix

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Objetivos . . . . .	1
1.3. Planificación . . . . .	2
1.4. Estructura . . . . .	2
<b>2. Marco teórico</b>	<b>3</b>
2.1. Datos sintéticos . . . . .	3
2.2. Beneficios y limitaciones de los datos sintéticos . . . . .	3
2.3. Tipos de datos sintéticos . . . . .	4
2.3.1. Datos completamente sintéticos . . . . .	5
2.3.2. Datos parcialmente sintéticos . . . . .	5
2.3.3. Texto sintético . . . . .	6
2.3.4. Archivos multimedia sintéticos . . . . .	6
2.3.5. Datos tabulares sintéticos . . . . .	7
2.4. Técnicas para la generación de datos sintéticos . . . . .	8
2.4.1. Redes generativas antagónicas . . . . .	8
2.4.2. Autoencoders variacionales . . . . .	9
2.5. Datos sintéticos simulados . . . . .	10
2.6. Trabajos similares . . . . .	11
2.6.1. VirtualHome . . . . .	11
2.6.1.1. Agentes . . . . .	12
2.6.1.2. Entornos . . . . .	13
2.6.1.2.1. Objetos . . . . .	13
2.6.1.2.2. EnvironmentGraph . . . . .	13
2.6.1.3. Programas . . . . .	15
2.6.1.4. UnityCommunication . . . . .	15
2.6.2. ElderSim . . . . .	15
2.6.2.1. Escenarios . . . . .	16
2.6.2.2. Personajes . . . . .	17
2.6.2.3. Movimiento . . . . .	17
2.6.2.4. Perspectiva de las cámaras . . . . .	18
2.6.2.5. Iluminación . . . . .	18
2.6.2.6. Objetos . . . . .	19
2.6.2.7. Interfaz de usuario . . . . .	19
2.6.3. UnrealROX . . . . .	20
2.6.3.1. Agentes robóticos . . . . .	20

xi

2.6.3.2. Subsistema controlador . . . . .	20
2.6.3.3. Subsistema de HUD . . . . .	21
2.6.3.4. Subsistema de agarre . . . . .	21
2.6.3.5. Subsistema multi-cámara . . . . .	22
2.6.3.6. Subsistema de grabado . . . . .	22
2.6.3.7. Subsistema de reproducción . . . . .	24
2.7. Aplicaciones de los datos sintéticos . . . . .	25
2.7.1. Machine Learning . . . . .	25
2.7.2. Desarrollo ágil de software . . . . .	26
2.7.3. Ensayos clínicos y científicos . . . . .	26
2.7.4. Seguridad . . . . .	26
2.7.5. Marketing . . . . .	27
2.7.6. Monetización de datos . . . . .	27
<b>3. Materiales</b>	<b>29</b>
3.1. Software . . . . .	29
3.1.1. Unreal Engine 4 . . . . .	29
3.1.2. Visual Studio . . . . .	30
3.1.3. GitHub . . . . .	31
3.1.4. Axis neuron . . . . .	32
3.2. Assets . . . . .	32
3.3. Hardware . . . . .	33
<b>4. Desarrollo</b>	<b>35</b>
4.1. Avatares . . . . .	35
4.2. Objetos interactivos . . . . .	37
4.3. CharacterMovementComponent . . . . .	37
4.4. MyAnimationComponent . . . . .	39
4.4.1. Reproducción de animaciones . . . . .	40
4.4.2. Interpolación entre animaciones . . . . .	41
4.4.2.1. Interpolaciones desplazamiento-desplazamiento . . . . .	42
4.4.2.2. Interpolaciones desplazamiento-interacción . . . . .	43
4.5. MyGripComponent . . . . .	44
4.5.1. Agarre de objetos . . . . .	44
4.5.2. Liberación de objetos . . . . .	45
4.6. Mecánicas <i>Point and Click</i> . . . . .	46
4.7. Interfaz de usuario (GUI) . . . . .	48
<b>5. Limitaciones</b>	<b>51</b>
5.1. Limitaciones del sistema de navegación . . . . .	51
5.2. Limitaciones del sistema de agarres . . . . .	52
<b>6. Conclusiones y trabajo futuro</b>	<b>57</b>
6.1. Conclusiones . . . . .	57
6.2. Trabajo futuro . . . . .	57

---

<b>Bibliografía</b>	<b>59</b>
<b>A. Anexo I: Extensión del código de la clase <i>UBlendSpace1D</i></b>	<b>61</b>



# Índice de figuras

1.1.	Diagrama Gantt del proyecto . . . . .	2
2.1.	Datos completamente sintéticos VS Datos parcialmente sintéticos. Fuente: Statice	5
2.2.	Texto sintético generado a partir de texto original de Shakespeare por el modelo GPT-3. Fuente: GPT-3 Creative Fiction . . . . .	6
2.3.	Imágenes de personas no reales generadas por artificialmente. Fuente: <a href="https://thispersondoesnotexist.com">https://thispersondoesnotexist.com</a> . . . . .	7
2.4.	Proceso para generar datos sintéticos con Red Generativa Antagónica (RGA)s. Fuente: Statice . . . . .	9
2.5.	Proceso para generar datos sintéticos con Autoencoder Variacional (AEV)s. Fuente: Statice . . . . .	10
2.6.	Proceso de creación de datos sintéticos en VirtualHome. Fuente: <a href="https://arxiv.org/pdf/1806.07011.pdf">https://arxiv.org/pdf/1806.07011.pdf</a> . . . . .	12
2.7.	Representación física de los agentes de VirtualHome. Fuente: <a href="https://arxiv.org/pdf/1806.07011.pdf">https://arxiv.org/pdf/1806.07011.pdf</a> . . . . .	13
2.8.	Entornos disponibles en VirtualHome para la generación de datos sintéticos. Fuente: <a href="https://arxiv.org/pdf/1806.07011.pdf">https://arxiv.org/pdf/1806.07011.pdf</a> . . . . .	14
2.9.	Vista superior de cuatro de los escenarios implementados en ElderSim. Fuente: <a href="https://arxiv.org/pdf/2010.14742.pdf">https://arxiv.org/pdf/2010.14742.pdf</a> . . . . .	16
2.10.	Representación física de los personajes de ElderSim. Fuente: <a href="https://arxiv.org/pdf/2010.14742.pdf">https://arxiv.org/pdf/2010.14742.pdf</a> . . . . .	17
2.11.	Ejemplo de configuraciones de cámaras virtuales basadas en los puntos de vista disponibles en ElderSim. Fuente: <a href="https://arxiv.org/pdf/2010.14742.pdf">https://arxiv.org/pdf/2010.14742.pdf</a> . . . . .	18
2.12.	Imagen de la Interfaz de Usuario Gráfica (GUI) de ElderSim para generar datos de acción. Fuente: <a href="https://arxiv.org/pdf/2010.14742.pdf">https://arxiv.org/pdf/2010.14742.pdf</a> . . . . .	19
2.13.	Pepper y Maniquí de UE integrados en UnrealRox con <i>colliders</i> . Fuente: <a href="https://arxiv.org/pdf/1810.06936.pdf">https://arxiv.org/pdf/1810.06936.pdf</a> . . . . .	21
2.14.	Información y mensajes de error mostrados en el Head-Up Display (HUD) de UnrealRox. Fuente: <a href="https://arxiv.org/pdf/1810.06936.pdf">https://arxiv.org/pdf/1810.06936.pdf</a> . . . . .	22
2.15.	Secuencia de 6 fotogramas mostrando la acción de agarre en UnrealRox. Fuente: <a href="https://arxiv.org/pdf/1810.06936.pdf">https://arxiv.org/pdf/1810.06936.pdf</a> . . . . .	23
2.16.	Cámaras unidas a los brazos del actor <i>Pawn</i> de UnrealRox. Fuente: <a href="https://arxiv.org/pdf/1810.06936.pdf">https://arxiv.org/pdf/1810.06936.pdf</a> . . . . .	24
2.17.	Modos de renderizado del subsistema de reproducción de UnrealRox. Fuente: <a href="https://arxiv.org/pdf/1810.06936.pdf">https://arxiv.org/pdf/1810.06936.pdf</a> . . . . .	25
3.1.	Logotipo del motor de juegos UE. . . . .	30
3.2.	Logotipo del Entorno de Desarrollo Integrado (IDE) Visual Studio. . . . .	31
3.3.	Logotipo de la plataforma Github. . . . .	31

3.4.	Ventana principal del software axis neuron. Muestra si los sensores están funcionando correctamente. . . . .	32
3.5.	Ejemplos de algunos de los modelos 3D utilizados para representar a los objetos interactivos. . . . .	33
4.1.	Lista de componentes y representación física de los avatares. . . . .	36
4.2.	Entidad de la clase <i>ANavMeshBoundsVolume</i> situada en la escena. El color verde representa las zonas navegables por los avatares. . . . .	38
4.3.	Configuración de la clase <i>ANavMeshBoundsVolume</i> para conseguir una mejor adaptación a la distribución de la escena virtual. . . . .	38
4.4.	Código visual utilizado para reproducir animaciones desde el <i>AnimationInstance</i> . . . . .	40
4.5.	Código visual utilizado para calcular la velocidad a la que se desplazan los avatares por la escena. . . . .	43
4.6.	<i>Sockets</i> situados en las manos de los avatares de UnrealHome. . . . .	45
4.7.	Declaración del evento <i>GetActorUnderMouse</i> en la configuración del proyecto. . . . .	46
4.8.	Declaración del <i>Trace Channel</i> específico para avatares y objetos en la configuración del proyecto. . . . .	46
4.9.	GUI del sistema mostrando la lista de actividades disponibles a realizar con el objeto interactivo seleccionado. . . . .	48
4.10.	Código presente en <i>MyPlayerController</i> para actualizar la información mostrada en la GUI. . . . .	48
5.1.	Avatar incapaz de alcanzar el objeto interactivo seleccionado debido a la falta de precisión del <i>NavMeshBoundVolume</i> para adaptarse al escenario. . . . .	51
5.2.	Avatar bloqueado por otro avatar debido que el sistema de <i>pathfinding</i> no reconoce a los avatares como obstáculos. . . . .	52
5.3.	Escena virtual utilizada para comprobar el funcionamiento de los subsistemas de UnrealHome. . . . .	53
5.4.	Objetos con forma compacta y tamaño adecuados para las manos de los avatares. . . . .	53
5.5.	Agarres realizados sobre objetos con tamaño adecuado para las manos de los avatares y forma compacta. . . . .	54
5.6.	Objetos con tamaños y formas muy heterogéneos. . . . .	55
5.7.	Agarres realizados sobre objetos con tamaño y forma muy heterogéneos. . . . .	55

# Índice de Códigos

4.1.	Código básico para el funcionamiento del sistema de navegación con <i>pathfinding</i>	39
4.2.	Código para la gestión de eventos de superposición presente en el <i>MyAvatarController</i>	40
4.3.	Código para la asignación de nuevas animaciones a una instancia de la clase <i>UBlendSpace1D</i> desde el <i>MyAnimationComponent</i>	42
4.4.	Código para realizar la transición desde una animación de desplazamiento hacia una animación de interacción.	43
4.5.	Función utilizada para ejecutar la acción de agarre de objetos interactivos.	44
4.6.	Función utilizada para ejecutar la acción de soltado de objetos interactivos.	45
4.7.	Código para enlazar el evento <i>GetActorUnderMouse</i> con la función 4.8	47
4.8.	Código para seleccionar el avatar u objeto bajo el ratón cuando el evento <i>GetActorUnderMouse</i> es disparado.	47
A.1.	Código añadido al archivo <i>BlendSpace1D.h</i> para la asignación dinámica de animaciones desde C++	61
A.2.	Código añadido al archivo <i>BlendSpace1D.cpp</i> para la asignación dinámica de animaciones desde C++	62



# **Lista de Acrónimos y Abreviaturas**

<b>3DPL</b>	3D Perception Lab.
<b>AEV</b>	Autoencoder Variacional.
<b>API</b>	Aplication Programming Interface.
<b>DL</b>	Deep Learning.
<b>DTIC</b>	Departamento de Tecnología Informática y Computación.
<b>FBX</b>	Filmbox.
<b>FoV</b>	Field of View.
<b>GUI</b>	Interfaz de Usuario Gráfica.
<b>HUD</b>	Head-Up Display.
<b>IA</b>	Inteligencia Artificial.
<b>IDE</b>	Entorno de Desarrollo Integrado.
<b>ML</b>	Machine Learning.
<b>RGA</b>	Red Generativa Antagónica.
<b>RGPD</b>	Reglamento General de Protección de Datos.
<b>RV</b>	Realidad Virtual.
<b>TFG</b>	Trabajo Final de Grado.
<b>UA</b>	Universidad de Alicante.
<b>UE</b>	Unreal Engine 4.



# 1. Introducción

En este proyecto exploramos la generación de datos visuales sintéticos relacionados con el comportamiento humano, su uso en el campo del aprendizaje profundo, en inglés Deep Learning (DL), y la posibilidad de incluir la interacción con el usuario para generar dichos datos. En consecuencia, se ha desarrollado un sistema, llamado UnrealHome, capaz de interpretar instrucciones generadas a partir de mecánicas *Point and Click* y transformarlas en una lista de tareas a realizar por una serie de actores en una escena virtual en UE.

## 1.1. Motivación

El proyecto nace de una colaboración con el grupo 3D Perception Lab (3DPL) perteneciente al Departamento de Tecnología Informática y Computación (DTIC) en la Universidad de Alicante (UA). La línea de investigación de este grupo se centra principalmente en técnicas de aprendizaje automático, visión por computador y computación paralela sobre procesadores gráficos. Dicha colaboración persigue, como objetivo, facilitar la generación de datos visuales sintéticos para alimentar algoritmos de DL e implementar la interacción del usuario como parte sustancial de este proceso.

A nivel personal, siempre he tenido un gran interés por el desarrollo de videojuegos, las simulaciones y el software interactivo. Por ello pensé que formar parte de esta colaboración me daría la oportunidad de obtener resultados relevantes, al combinar la creación de una plataforma de simulación de estas características e incluir la interacción del usuario como parte esencial del proceso de simulación.

## 1.2. Objetivos

El objetivo principal del proyecto es el desarrollo de UnrealHome, una plataforma de simulación que permite generar datos sintéticos a partir de una serie de personajes ubicados en una escena virtual tridimensional que realizan una lista de tareas básicas definidas por instrucciones generadas a partir de la interacción del usuario con los elementos presentes en dicha escena. Estos personajes deben llevar a cabo las tareas indicadas generando secuencias completas de acciones encadenando las animaciones necesarias de forma automática. Así mismo, también se persigue que estas secuencias de animaciones presenten una alta variabilidad y que su reproducción sea lo más realista posible. Respecto al primer objetivo, para incrementar la variabilidad se pretende disponer de diferentes animaciones para cada una de las actividades básicas a realizar por los personajes (reposo o *idle*, andar, comer, sujetar, etc.) y de diversos modelos 3D para representar los múltiples elementos distribuidos en la escena con los que los personajes podrán interactuar.

Por otro lado, como objetivo secundario, se propone el diseño y la implementación de mecánicas *Point and Click* mediante las cuales el usuario interactuará con los actores y

objetos de la escena generando las instrucciones anteriormente mencionadas.

### 1.3. Planificación

La planificación del proyecto se divide en tres bloques de actividades a realizar a lo largo de sus ocho meses de su duración, desde mediados de octubre de 2020 hasta finales de mayo de 2021. Podemos ver la distribución de estos tres bloques de forma visual en la figura 1.1.

El primer bloque, se corresponde con la introducción al proyecto, donde se investiga sobre aplicaciones similares ya existentes, y se desarrollan las primeras pruebas con animaciones en UE. Este bloque se extenderá desde mediados de octubre de 2020 hasta mediados de noviembre de 2021.

Seguidamente, en el segundo bloque, se abarca el desarrollo de las principales características de UnrealHome: Interpolación y reproducción de animaciones, sistema de navegación con *pathfinding*, sistema de agarre de objetos, mecánicas *Point and Click* y GUI. Estas tareas son desarrolladas desde mediados de noviembre de 2020 hasta finales de abril de 2021.

Finalmente, el último bloque corresponde a la redacción de este documento y sus diferentes secciones. Esta parte empieza a mediados de enero y abarca hasta finales de mayo de 2021.

Tareas a realizar	Octubre '20	Noviembre '20	Diciembre '20	Enero '21	Febrero '21	Marzo '21	Abril '21	Mayo '21
<b>Introducción al proyecto</b>								
Estudiar aplicaciones similares								
Realizar pruebas con animaciones en UE4								
<b>Desarrollo del proyecto</b>								
Interpolación y reproducción de animaciones								
Sistema de navegación con <i>pathfinding</i>								
Sistema de agarre de objetos								
Mecánicas <i>Point and Click</i>								
Interfaz gráfica (GUI)								
<b>Redacción de memoria del proyecto</b>								
Capítulo de introducción								
Capítulo de marco teórico								
Capítulo de materiales								
Capítulo de desarrollo								
Capítulo de limitaciones								
Capítulo de conclusiones y trabajos futuros								

Figura 1.1: Diagrama Gantt del proyecto.

### 1.4. Estructura

La estructura de este documento se organiza del siguiente modo. En el Capítulo 1, introducimos el proyecto, su motivación y los objetivos que se pretenden alcanzar. A continuación, en el Capítulo 2 se expone el marco teórico, donde se dan a conocer proyectos similares y técnicas para la generación de datos sintéticos. En el Capítulo 3, se enumera y describe el software y hardware utilizado en el desarrollo del proyecto. Seguidamente, en el Capítulo 4 se describe el sistema desarrollado y su funcionamiento, así como la forma de interactuar con él a través de las mecánicas *Point and Click*. En el Capítulo 5, se exponen las limitaciones que presenta UnrealHome y que pueden afectar negativamente al proceso de generación de datos sintéticos. Finalmente, en el Capítulo 6 se presentan las conclusiones del trabajo realizado, y se proponen mejoras y correcciones para el mismo.

## 2. Marco teórico

En este capítulo exponemos el estado actual de la generación de datos sintéticos. En primer lugar, definimos qué son los datos sintéticos, y destacamos sus beneficios y limitaciones más relevantes. Seguidamente, damos a conocer los tipos de datos sintéticos tradicionales y exponemos algunas de las técnicas más usadas para su creación. Más adelante, describimos que son los datos sintéticos simulados, y listamos los proyectos que se han tomado como referencia en el desarrollo de este Trabajo Final de Grado (TFG) y sus características más destacables. Finalmente, para concluir, citamos algunas de las aplicaciones más interesantes del uso de datos sintéticos en diversos campos de estudio y trabajo.

### 2.1. Datos sintéticos

Los datos sintéticos son datos generados artificialmente con ayuda de algoritmos de DL o simulaciones, y son usados en una gran variedad de industrias y sectores [8].

En el artículo [8], se indica que el uso de datos sintéticos empezó en la década de los 90, pero es partir de 2010 cuando gracias al incremento del poder computacional y la capacidad de almacenamiento digital, los datos sintéticos comenzaron a ser mucho más relevantes, especialmente en sectores como el de la Inteligencia Artificial (IA).

Para muchos expertos los datos sintéticos son la clave para acelerar el desarrollo y aprendizaje de la IA [10]. Esto se debe a que el uso de este tipo de datos permite solucionar muchos de los problemas más importantes que aparecen al trabajar con datos reales, y por ello en la actualidad se considera a la generación de datos sintéticos como un recurso extremadamente útil en numerosas industrias y campos de investigación. [10].

### 2.2. Beneficios y limitaciones de los datos sintéticos

En el apartado anterior se expone que los datos sintéticos presentan ciertos beneficios frente a los datos reales. Sin embargo, debemos tener en cuenta que el uso de este tipo de datos también conlleva consigo unas limitaciones no presentes al usar datos reales. En esta sección vamos a listar tanto las ventajas como las limitaciones del uso de datos sintéticos y a explicar con detalle las más importantes.

En primer lugar, en cuanto a las ventajas del uso de datos sintéticos podemos destacar las siguientes:

- Evitar la carencia de datos: Un problema muy destacado que puede originarse al trabajar con datos reales es que los datos necesarios no existan o sean muy escasos. Esto no ocurre en el caso de los datos sintéticos, ya que estos pueden ser generados específicamente para satisfacer necesidades o condiciones no existentes o muy difíciles de hallar en el mundo real [3, 26].

- Aumentar la seguridad y el anonimato: Como señala [26], la seguridad y el anonimato de los datos son dos aspectos fundamentales a tener en cuenta cuando se trabaja con datos de carácter sensible como pueden ser nombres, correos electrónicos, direcciones postales o imágenes personales. Por ello cuando se trabaja con este tipo de datos reales se suelen utilizar técnicas de anonimización y encriptación para garantizar su seguridad y privacidad [3]. Sin embargo, [3] advierte que este tipo de técnicas también pueden ser contraproducentes ya que cuanto más se aumente la privacidad de un dato menor será su utilidad. Es por esto que el uso de datos sintéticos es una solución perfecta para este problema, dado que no es necesaria su anonimización al proceder de fuentes completamente artificiales, lo que permite aprovechar al máximo su utilidad.
- Abaratar costes: Los datos sintéticos, a diferencia de los datos reales, son más baratos de generar tanto en términos económicos como temporales. Esta es otra característica de los datos sintéticos por la cuál están tan cotizados en la actualidad, ya que en industrias como la del automóvil generar datos reales para, por ejemplo, construir coches autónomos puede suponer un coste sumamente alto [8].

En segundo lugar, encontramos las siguientes limitaciones del uso de datos sintéticos:

- La utilidad depende de la calidad: Según [3], algo que se debe recordar a la hora de generar datos sintéticos, es que su utilidad depende en gran medida o completamente de su calidad, es decir, de su parecido con la realidad. Producir datos sintéticos de alta calidad supone todo un reto, y es por ello una de las grandes limitaciones del uso de datos sintéticos. Cuanta menos calidad tenga nuestro conjunto de datos, los resultados obtenidos a partir de él estarán más alejados de los resultados que obtendríamos con datos generados a partir de eventos reales.
- La ausencia de valores atípicos: Los datos sintéticos solo pueden imitar a los datos obtenidos del mundo real, y por tanto no son una réplica exacta de estos [8]. Por esta razón, los conjuntos de datos sintéticos pueden carecer de valores atípicos que si están presentes en los datos reales, concluyendo en la obtención de resultados con menor precisión [8].
- El control de los resultados es necesario: [8] señala que en casos en los que los conjuntos de datos sintéticos utilizados son muy complejos, se debe asegurar que los resultados obtenidos son precisos. La mejor forma de llevar a cabo esta comprobación es comparando los datos sintéticos con conjuntos de datos reales, ya que al generar dichos datos sintéticos se han podido producir inconsistencias fruto de intentar replicar complejidades presentes en la realidad [8].

### **2.3. Tipos de datos sintéticos**

Los datos sintéticos pueden clasificarse de dos formas diferentes dependiendo de la información que utilicemos para su clasificación. En este apartado listamos y explicamos las características más destacables de estos dos grupos, exponemos los tipos de datos sintéticos que podemos encontrar dentro de cada uno de ellos y mostramos algunos ejemplos concretos de su aplicación en el mundo real.

---

En el primer grupo, clasificamos a los datos sintéticos dependiendo de su origen, lo que da como resultado dos tipos diferentes de datos sintéticos, los datos completamente sintéticos y los datos parcialmente sintéticos [5].

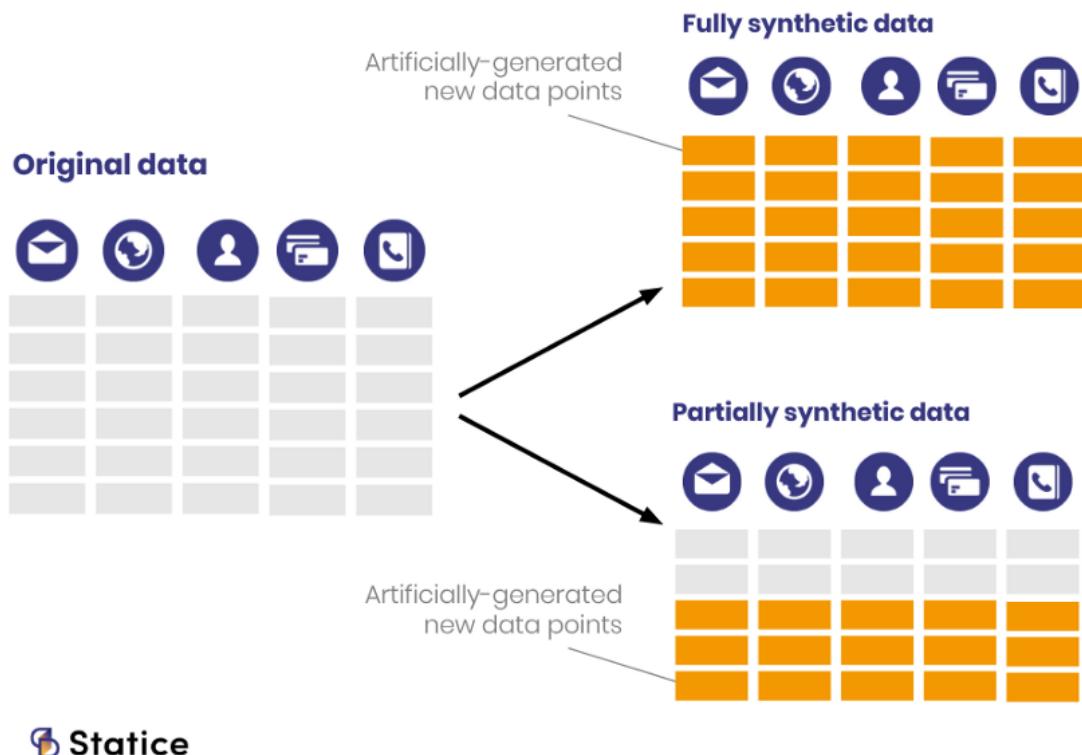
En el segundo grupo, las clasificación se realiza en función de las propiedades que los datos sintéticos presentan y diferenciamos entre texto sintético, archivos multimedia sintéticos y datos tabulares sintéticos [5].

### 2.3.1. Datos completamente sintéticos

Los datos completamente sintéticos son aquellos que no contienen ningún dato original. Esto quiere decir que la re-identificación de datos originales es imposible, sin embargo todas las variables presentes en los datos están todavía plenamente disponibles para su utilización. En general, estos datos sintéticos son a menudo usados cuando las restricciones de privacidad no permiten el uso de datos reales [8, 5].

### 2.3.2. Datos parcialmente sintéticos

Los datos parcialmente sintéticos son aquellos en los que solo los datos reales sensibles han sido sustituidos por datos generados artificialmente. Según indican [8, 5], estos datos son especialmente útiles a la hora de expandir conjuntos de datos ya existentes o generar muestras estadísticas específicas ausentes o con poca representación.



**Figura 2.1:** Datos completamente sintéticos VS Datos parcialmente sintéticos. Fuente: Statice

### 2.3.3. Texto sintético

Siguiendo la definición proporcionada por [6, 5], el texto sintético es texto generado artificialmente mediante el uso de modelos de IA entrenados para crear texto. Debido a la complejidad que presentan los lenguajes, generar texto sintético realista es todo un reto. Sin embargo, con la aparición de nuevos y más eficientes modelos de Machine Learning (ML) es posible crear sistemas de generación de lenguaje natural que permiten generar texto sintético mucho más similar al texto real escrito por humanos [6, 5].

En 2020, el equipo OpenAI presentó GPT-3, un modelo de IA capaz de generar texto similar al humano. Actualmente, se pueden encontrar multitud de ejemplos de textos escritos por el modelo GPT-3, generados a partir de muestras de texto específicas de autores célebres históricos [5].

**William Shakespeare** 

**Transformer AI poetry: Poetry classics as reimagined and rewritten by an artificial intelligence.**  
**“Sonnet 18”**  
**William Shakespeare**  
 Shall I compare thee to a summer's day?  
 Thou art more lovely and more temperate:  
 Rough winds do shake the darling buds of May,  
 And summer's lease hath all too short a date;  
 A winter's day, when beams of sun are few,  
 And skies are grey with clouds intemperate,  
 Is to thy beauty somehow sordid grown.  
 Her lovely-lovely eye is quite shut up,  
 Like windows pale before a sunny shower.  
 The earth that's cold and moist, like to my mistress,  
 When once that she unbraided, her lovely jewel,  
 Grows as both ugly and uncouthly.  
 Poor soul, the center of my motley host,  
 See, now they request thy sovereign presence;  
 The din of merry hearts hath brought me thus  
 To greet thee, thou unsoild and saint-like beauty.  
 A Translation by an Artificial Neural Network that draws inspiration from François Connor's translation.  
 Soften thou, for showers often stain the daylight,  
 And longer they torment my Mary's day,

**Figura 2.2:** Texto sintético generado a partir de texto original de Shakespeare por el modelo GPT-3.  
 Fuente: GPT-3 Creative Fiction

### 2.3.4. Archivos multimedia sintéticos

Los datos sintéticos también pueden ser archivos multimedia como vídeos, imágenes o sonidos. Estos tipos de datos presentan propiedades muy similares a los datos reales. Esta similitud permite que estos datos sintéticos se puedan usar como reemplazo de datos reales [6, 5].

Este tipo de archivos multimedia generados artificialmente son especialmente útiles para expandir bases de datos utilizadas para entrenar algoritmos de ML, ya que solventan problemas de privacidad y de falta o ausencia de muestras de datos específicas [6, 5]. Asimismo, otra ventaja de estos datos multimedia es que la etiquetación se realiza de forma automática y con

precisión de píxel perfecto, evitando tener que invertir un esfuerzo humano desproporcionado para llevar a cabo esta tarea.



**Figura 2.3:** Imágenes de personas no reales generadas por artificialmente. Fuente: <https://thispersondoesnotexist.com>

Por otro lado, estos datos presentan varias desventajas como la dependencia directa de su calidad al software con el que han sido generados. Si el software utilizado no es capaz de simular la realidad con suficiente detalle los datos generados pueden carecer de ruido que si estaría presente en imágenes reales y contener patrones o geometrías no perceptibles por el ojo humano pero que pueden influenciar negativamente los resultados obtenidos con modelos de ML.

Un ejemplo concreto del uso de este tipo de datos lo podemos encontrar en la compañía Waymo. Esta compañía subsidiaria de Alphabet, utiliza vídeos e imágenes sintéticas para entrenar sistemas de conducción autónoma para vehículos terrestres [6].

### 2.3.5. Datos tabulares sintéticos

Continuando con las definiciones formuladas en [6, 5], los datos tabulares sintéticos son datos generados artificialmente que imitan a los datos reales y que están almacenados en tablas. Estos datos se estructuran en filas y columnas, y estas pueden contener todo tipo de información desde datos de usuarios o pacientes hasta información analítica o registros financieros [5]. Este tipo de datos es el más utilizado y importante, ya que todas las organizaciones

---

que trabajan con grandes cantidades de datos disponen de ellos en forma de tablas.

[6] menciona que en el sector de los seguros, la compañía Suiza La Mobilière utilizó datos sintéticos para entrenar modelos de predicción de abandono. El equipo a cargo de este proceso creó datos sintéticos a partir de datos de clientes reales que eran demasiado sensibles para ser usados y entrenó sus modelos de aprendizaje automático con ellos [6].

## 2.4. Técnicas para la generación de datos sintéticos

La generación de datos sintéticos se fundamenta en aprender la distribución de probabilidad que presenta un conjunto de datos reales para generar, a partir de él, un nuevo conjunto de datos artificiales con la misma distribución [6].

En principio, con un conjunto de datos simple es posible mapear la distribución de probabilidad con relativa facilidad, y de este modo generar datos sintéticos de forma rápida y sencilla. Sin embargo, cuanto más complejo sea nuestro conjunto de datos original más difícil será mapear esta distribución de probabilidad correctamente.

Cuantas más muestras de datos formen parte de nuestro conjunto, más combinaciones posibles aparecerán. En modelos de generación de datos sintéticos simples esto puede llevar a la ausencia de puntos de información clave para aprender adecuadamente la distribución de probabilidad del conjunto de datos original. Es por esto que se necesitan modelos robustos para abordar la generación de datos sintéticos a partir de conjuntos de datos reales complejos [6].

Los avances en el área del ML en los últimos años ha desembocado en el desarrollo de modelos capaces de generar un amplio abanico de tipos de datos sintéticos.

Mediante predicción y corrección, como apunta [6], las redes neuronales aprenden a reproducir datos y generar representaciones que podrían originarse a partir de ellos. Esta es una cualidad que convierte a las redes neuronales en sistemas bien adaptados a la generación de datos sintéticos.

Algunos ejemplos de modelos de generación de datos sintéticos basados en redes neuronales son los siguientes:

- RGA
- AEV

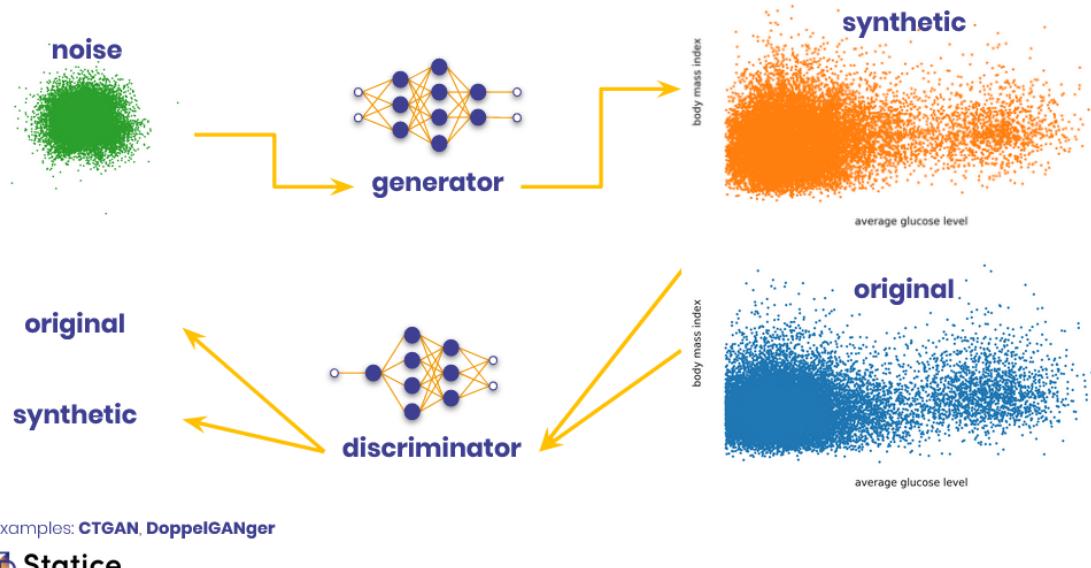
### 2.4.1. Redes generativas antagónicas

Las RGA son una clase de algoritmos de inteligencia artificial que se utilizan en el aprendizaje no supervisado, implementadas por un sistema de dos redes neuronales que compiten mutuamente en una especie de juego de suma cero [25, 30]. En términos de arquitectura, las RGA entran simultáneamente dos redes neuronales de forma opuesta: Una red generativa y una red discriminatoria, las cuales intentan superarse entre sí.

La red generativa aprende a asignar elementos de un espacio latente a una distribución de datos determinada, mientras la red discriminatoria diferencia entre elementos de la distribución de datos originales y los candidatos producidos por el generador[30]. El objetivo del aprendizaje de la red generativa es aumentar el índice de error de la red discriminatoria, es decir, engañar a la red discriminatoria produciendo nuevos elementos sintéticos que parecen

---

provenir de la distribución de datos auténticos [30]. Ambas redes están conectadas de forma que la red generativa puede acceder a la toma de decisiones de la red discriminatoria [6].



**Figura 2.4:** Proceso para generar datos sintéticos con RGAs. Fuente: Statice

Cuando ambas redes se entrena n juntas, la red discriminatoria necesita aprender si los patrones de datos recibidos parecen lo suficientemente realistas, mientras que la red generativa intenta ser más astuta y producir salidas más realistas a partir de las entradas aleatorias, para así burlar a la red discriminatoria.

Según [6], la ventaja que presentan las RGA es que la red discriminatoria no necesita formular un error de reconstrucción, ya que aprende directamente las características que componen a los datos reales. Esta aproximación es particularmente interesante para la generación de imágenes sintéticas, ya que es difícil traducir en funciones las características de la realidad. En general, las RGA ofrecen mejores resultados cuando trabajan con datos no estructurados.

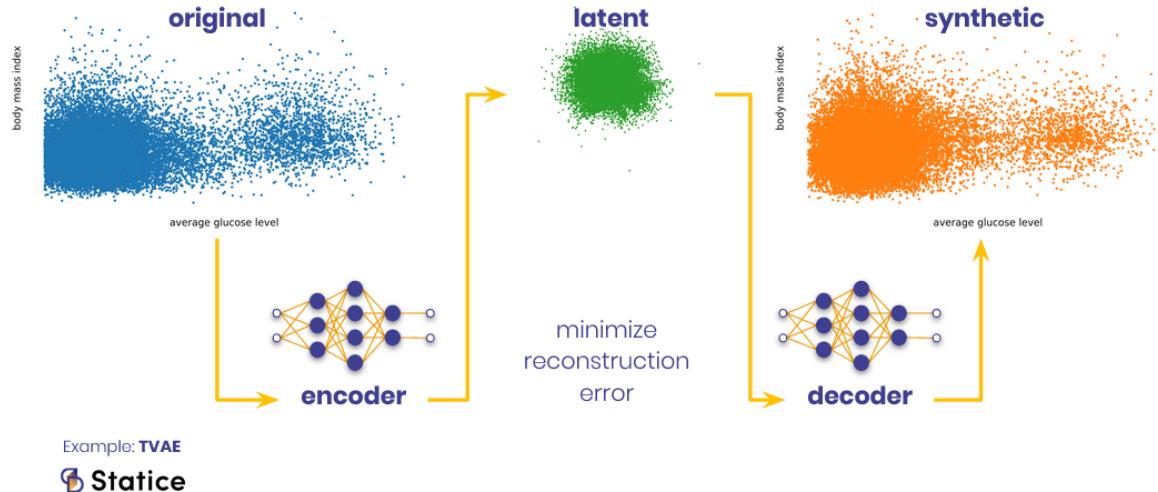
Por otro lado, las RGA son más difíciles de entrenar que los AEV y se requiere más experiencia para ello. Las RGA también son propensas a caer en el llamado modo colapso, donde el generador comienza a producir solo un pequeño subconjunto de los datos originales en lugar de la distribución completa.

#### 2.4.2. Autoencoders variacionales

Al igual que las RGA, los AEV provienen del campo del aprendizaje no supervisado. Como modelos generativos, están diseñados para aprender la distribución subyacente en los conjuntos de datos originales y son muy eficientes generando conjuntos de datos sintéticos complejos [7].

Como se dice en [6], los AEV funcionan en dos pasos. En primer lugar, una red de codificadores transforma una distribución compleja original en una distribución latente. Luego, una red de decodificadores transforma la distribución de nuevo al espacio original. Esta doble

transformación [13], codificación-decodificación, parece poco eficiente a primera vista, pero es necesaria para formular un error de reconstrucción cuantificable. Para minimizar este error se entrena a los AEV al mismo tiempo que un sistema adicional regulariza y controla a la distribución latente.



**Figura 2.5:** Proceso para generar datos sintéticos con AEVs. Fuente: Statice

Los AEV [6] son una aproximación sencilla para resolver el problema de la generación de datos artificiales, ya que son relativamente fáciles de implementar y entrenar. Sin embargo, su punto débil reside en su proceso de entrenamiento. A medida que tu conjunto de datos originales se vuelve más heterogéneo, es más difícil para los AEV formular un error de reconstrucción que funcione bien para todos los componentes de datos. Si, por ejemplo, el error de reconstrucción pone demasiado énfasis en obtener las partes continuas de los datos correctamente la calidad de las partes categóricas puede verse afectada.

## 2.5. Datos sintéticos simulados

Como se señala en [4], existen una gran variedad de métodos utilizados para generar datos sintéticos, desde las sofisticadas RGA hasta estrategias más sencillas como los AEV. Sin embargo, la mayoría de estas técnicas presenta serias limitaciones. Debido a que son fundamentalmente estáticos, los conjuntos de datos utilizados en estas técnicas deben ajustarse y regenerarse constantemente para reflejar nuevos parámetros, distribuciones y objetivos de entrenamiento. En definitiva, presentan poca flexibilidad, especialmente cuando los datos utilizados son visuales. Esto se debe a que generalmente los datos están diseñados para realizar una única tarea, lo que no permite que en los entrenamientos los modelos de IA aprendan como lo hacen los humanos, de forma receptiva y en tiempo real. Esto supone una limitación, ya que cada día los modelos desarrollados se vuelven más robustos, con la capacidad de

aprender de forma más intuitiva. Sin embargo, esta capacidad se ve limitada dado el aspecto estético inherente a los datos sintéticos tradicionales.

Los datos sintéticos simulados evitan muchas de las limitaciones que los datos sintéticos tradicionales presentan. Son datos sintéticos capturados por “cámaras” virtuales que operan en simulaciones fotorrealistas teniendo en cuenta las leyes físicas que actúan en la realidad. Estas simulaciones tienen dimensiones espaciales y una gama completa de detalles físicos y visuales que permiten simular la realidad con alta precisión. Por otro lado, los datos simulados son creados a partir de datos reales en 3D y fusionan el hiperrealismo con un nivel único de flexibilidad y personalización. Al hacer uso de datos simulados, podemos fotografiar o grabar “virtualmente” un entorno 3D hiperrealista desde diferentes ángulos, con diferentes lentes o incluso en diferentes momentos del día. Además, en las simulaciones donde se generan los datos simulados se nos permiten modificar multitud de parámetros, como los elementos situados en el entorno 3D y sus características, la iluminación o incluso propiedades físicas como podría ser la gravedad, permitiendo así simular escenarios o situaciones muy difíciles o casi imposibles de capturar en la realidad. Debido a que todas estas modificaciones y variaciones se aplican sobre datos 3D escaneados en alta calidad, se conserva el realismo y la plausibilidad en la simulación. Esto nos permite evitar los costes de la recopilación manual a gran escala de datos, las preocupaciones de privacidad asociadas con los datos personales y el sesgo inherente a los conjuntos de datos recopilados manualmente.

Es verdad que para algunas aplicaciones, los datos sintéticos tradicionales son más que suficientes. Sin embargo, debemos tener en cuenta que estos datos sintéticos tienen sus limitaciones y superarlas podría suponer un salto generacional en campos de estudio como la visión por computador o el entrenamiento de modelos de IA. Por esta razón, los datos sintéticos simulados representan el objetivo final de simular por completo el mundo que nos rodea incluyendo todo tipo de detalles sintéticos representados de forma hiperrealista.

## 2.6. Trabajos similares

A continuación, se enumeran los trabajos similares ya existentes que se han tomado como referencia para la realización de este TFG, y se mencionan y describen sus características más destacables.

### 2.6.1. VirtualHome

VirtualHome [22, 23] es un software de simulación que hace uso de múltiples agentes para simular actividades cotidianas del hogar y ha sido la mayor referencia para la realización de este proyecto de final de grado. Los agentes que encontramos en VirtualHome se representan como humanoides que pueden interactuar con su entorno a partir de instrucciones de alto nivel dadas por el usuario.

El entorno en el que se desarrollan las simulaciones de VirtualHome permite modificar el número de agentes presentes en él, lo que posibilita interacciones complejas con los objetos situados en dicho entorno. Entre estas interacciones encontramos coger o inspeccionar objetos, o encender y apagar dispositivos. Este sistema de simulación está desarrollado en Unity y se controla mediante llamadas realizadas a través de una simple Application Programming Interface (API) de Python.

---

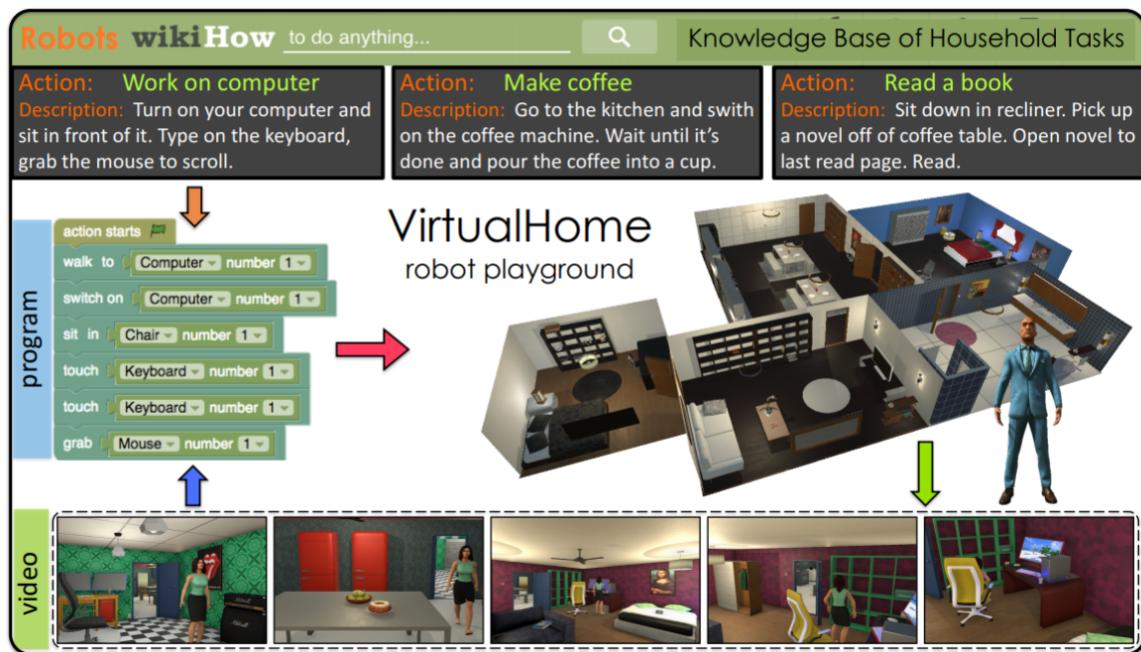


Figura 2.6: Proceso de creación de datos sintéticos en VirtualHome. Fuente: <https://arxiv.org/pdf/1806.07011.pdf>

En última instancia, VirtualHome puede ser utilizado para tareas como grabar de vídeos de actividades humanas o entrenar agentes para que realicen interacciones complejas con objetos. Por otro lado, para facilitar su uso, VirtualHome también incluye una lista de instrucciones base que permiten ejecutar una gran variedad de actividades.

Las simulaciones en VirtualHome están formadas por tres componentes principales: los agentes, representados como avatares humanos que llevan a cabo acciones; los entornos, que representan diferentes viviendas con objetos con los cuales los agentes pueden interactuar; y los programas, que se encargan de definir de qué forma los agentes deben interactuar con su entorno.

### 2.6.1.1. Agentes

Los agentes son avatares humanoides que pueden interactuar con su entorno y realizar acciones. En VirtualHome, los agentes tienen un componente llamado NavMeshAgent, que les permite desplazarse en el entorno utilizando las rutas más cortas, evitando obstáculos y haciendo giros suaves. Además, estos agentes hacen uso de RootMotion FinalIK, un sistema de cinemática inversa que añaderealismo a las animaciones cuando los agentes interactúan con objetos. VirtualHome permite añadir múltiples agentes a la simulación y hacer que todos ellos interactúen con el entorno al mismo tiempo.



**Figura 2.7:** Representación física de los agentes de VirtualHome. Fuente: <https://arxiv.org/pdf/1806.07011.pdf>

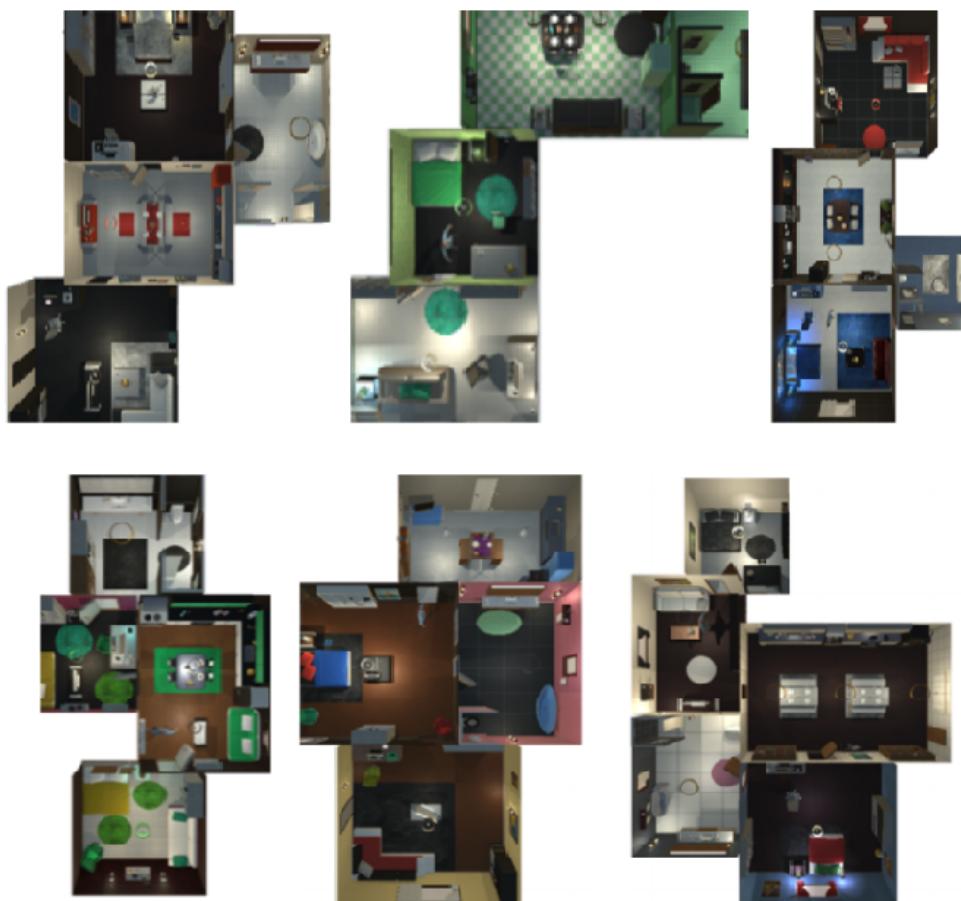
### 2.6.1.2. Entornos

Existen siete entornos diferentes en VirtualHome (Ver figura 2.8) donde los agentes pueden interactuar. Cada uno de estos entornos representa el interior de una vivienda con diferentes habitaciones y repleta de objetos con los que los agentes pueden interactuar. A pesar de que los siete entornos disponibles son inalterables, VirtualHome permite añadir, quitar y modificar objetos en los entornos, lo que permite crear diferentes escenas para generar diversos vídeos o entrenar a los agentes.

**2.6.1.2.1. Objetos** Tal y como se ha indicado anteriormente, los entornos disponibles en VirtualHome están repletos de objetos 3D para que los agentes puedan interactuar con ellos. Estos objetos se clasifican en tres tipos:

- **Estáticos:** Los objetos estáticos son aquellos hacia los que los agentes pueden desplazarse pero no pueden ni modificar ni interactuar con ellos de ningún modo.
- **Interactivos:** Los objetos interactivos son aquellos con los cuales los agentes pueden interactuar y cambiar su estado (abrir un armario, encender una estufa).
- **Agarrables:** Los objetos agarrables son aquellos que los agentes pueden agarrar y colocar en diferentes lugares del entorno.

**2.6.1.2.2. EnvironmentGraph** Los entornos en VirtualHome están representados por un EnvironmentGraph. Un grafo donde cada nodo representa a un objeto presente en el entorno y las aristas representan relaciones espaciales.



**Figura 2.8:** Entornos disponibles en VirtualHome para la generación de datos sintéticos. Fuente: <https://arxiv.org/pdf/1806.07011.pdf>

---

Los nodos del EnvironmentGraph contienen el nombre del objeto, su identificador numérico para interactuar con él, sus coordenadas, sus relaciones con otros objetos y su estado actual. Además, el EnvironmentGraph puede ser utilizado para consultar el estado del entorno y modificarlo antes de ejecutar alguna simulación.

### 2.6.1.3. Programas

En VirtualHome, las tareas a realizar por los agentes son representadas por programas, secuencias de instrucciones con la siguiente forma:

```
1 <char_id> [action] <object> (object_id)
```

Donde **char\_id** corresponde al identificador del agente que realiza la acción y **object\_id** identifica el objeto con el que interactuar, en caso de que haya más objetos del mismo tipo. VirtualHome guarda su estado después de ejecutar un programa, por lo que si ejecutamos un programa después de otro, el estado del entorno en la segunda llamada dependerá de lo que los agentes hicieron en el primer programa.

Un ejemplo de programa podría ser el siguiente:

```
1 program = ['<char0> [walk] <chair> (1)', '<char0> [sit] <chair> (1)']
```

Por otro lado, los agentes también pueden ejecutar múltiples instrucciones al mismo tiempo, para ello solo se necesita incluir en el programa todas las acciones a llevar a cabo por el agente separadas por el carácter '|':

```
1 program = ['<char0> [walk] <chair> (1) | <char1> [walk] <kitchen> (1)←
    ↪ ', '<char0> [sit] <chair> (1)']
```

### 2.6.1.4. UnityCommunication

A parte de los tres elementos citados anteriormente, VirtualHome presenta un componente extra llamado UnityCommunication object. El UnityCommunication object es el responsable de manejar todas las interacciones del usuario con VirtualHome, y para ello este objeto se une a cada una de las instancias del simulador. Esto capacita a los usuarios para utilizar múltiples instancias de VirtualHome, lo que permite aumentar el numero de simulaciones ejecutándose simultáneamente y acelerar el aprendizaje de los agentes.

## 2.6.2. ElderSim

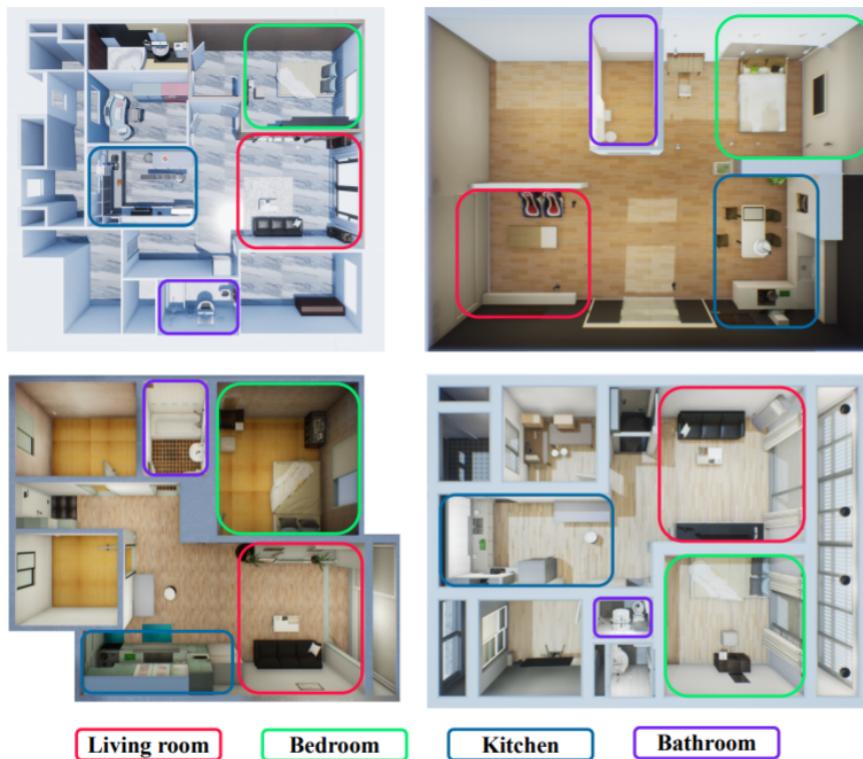
ElderSim [11] es otro de los proyectos que se ha tomado como referencia para realizar este TFG. Se trata de una plataforma de simulación diseñada para llevar a cabo simulaciones de actividades realizadas por ancianos. Este simulador fue desarrollado haciendo uso de UE y el software de modelado y animación 3D Autodesk Maya. El uso de estos dos programas permite construir escenarios virtuales que representan de forma fotorrealista hogares donde los ancianos de la simulación llevan a cabo sus actividades. Además, en el desarrollo de ElderSim también se utilizó la tecnología *Motion Capture* con el objetivo de que las animaciones reproducidas por los ancianos en la simulación presentaran un alto grado de realismo.

Por otro lado, ElderSim cuenta con 55 clases diferentes de actividades que los personajes simulados pueden llevar a cabo durante las simulaciones, permite personalizar el punto de vista de las cámaras y ajustar las condiciones de iluminación.

En las siguientes secciones se explicará con más detalle algunas de las características por las que se ha tomado ElderSim como referencia en este TFG y que lo convierten en uno de los proyectos más interesantes dentro del campo de la generación de datos sintéticos simulados.

### 2.6.2.1. Escenarios

En cuanto a los escenarios de ElderSim, la plataforma cuenta con cuatro modelos de serie (Figura 2.9), todos ellos modelados a partir de medidas y fotografías reales de interiores residenciales con el objetivo de conseguir el máximo realismo. Además, en caso de ser necesario, la plataforma también permite añadir modelos propios de escenarios para ser usados en las simulaciones. Para garantizar el máximo realismo visual, ElderSim hace uso de materiales basados en físicas aplicando Post-Process Volumes, unos *assets* propios de UE que equivaldrían a los *shaders*. Cada uno de los escenarios disponibles por defecto en ElderSim contiene cuatro áreas diferentes: sala de estar, dormitorio, cocina y baño, y en cada una de ellas solo se pueden simular actividades que se puedan llevar a cabo en las mismas. Por ejemplo, lavarse la cara únicamente puede ser simulado en el baño, mientras que jugar con el teléfono móvil se puede simular en cualquiera de las áreas.



**Figura 2.9:** Vista superior de cuatro de los escenarios implementados en ElderSim. Fuente: <https://arxiv.org/pdf/2010.14742.pdf>

### 2.6.2.2. Personajes

Por lo que concierne a los personajes, ElderSim ofrece trece modelos diferentes de personas ancianas, siete mujeres y seis hombres, y dos modelos de personas jóvenes, una mujer y un hombre (Figura 2.10). Estos modelos fueron creados tomando como referencia las formas capturadas con sensores de profundidad de Kinect, y la caras fueron generadas de forma aleatoria para evitar problemas de privacidad. Asimismo, cada uno de los modelos viste diferentes ropas ajustadas a su edad, lo que junto a todo lo listado anteriormente permite una gran variedad de apariencia en los personajes.



**Figura 2.10:** Representación física de los personajes de ElderSim. Fuente: <https://arxiv.org/pdf/2010.14742.pdf>

### 2.6.2.3. Movimiento

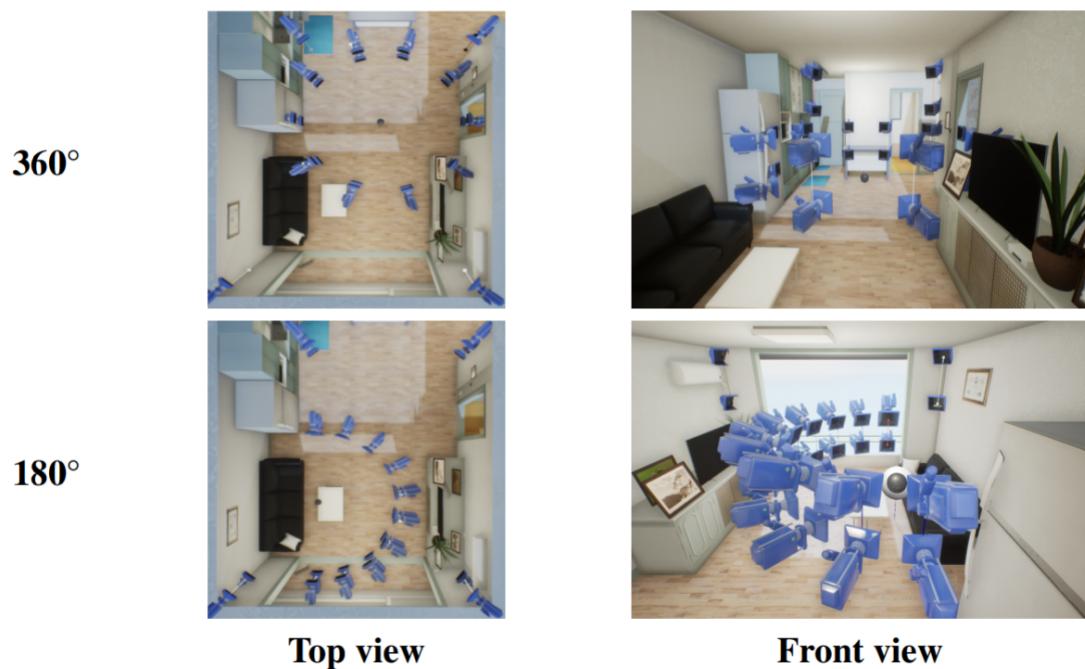
Como se ha citado en la introducción de esta sección, ElderSim provee 55 clases de actividades diferentes, cuyas animaciones fueron obtenidas con tecnología *Motion Capture*. Concretamente se utilizaron dieciséis cámaras que capturaron los movimientos de los sujetos usando cuarenta marcadores colocados alrededor de sus cuerpos. Además, en el momento de grabar dichas animaciones, no se dieron instrucciones concretas a los sujetos con el objetivo de aumentar el realismo y la diversidad de las grabaciones.

---

#### 2.6.2.4. Perspectiva de las cámaras

Respecto a la perspectiva de las cámaras, ElderSim permite elegir entre cámaras con dos puntos de vista distintos: puntos de vista robot y puntos de vista de vigilancia.

Los puntos de vista robot adquiere vídeo simulando la perspectiva de robots diseñado para el cuidado de personas dependientes, y están situadas en circulo rodeando al personaje objetivo de las grabaciones.



**Figura 2.11:** Ejemplo de configuraciones de cámaras virtuales basadas en los puntos de vista disponibles en ElderSim. Fuente: <https://arxiv.org/pdf/2010.14742.pdf>

Por otro lado, las cámaras con puntos de vista de vigilancia simulan la perspectiva cámaras de vigilancia. Estas, a diferencia de las cámaras con punto de vista robot, se encuentran situadas a alturas de entre 1.5m y 2.2m en cuatro de las esquinas de cada área de los escenarios, simulando de esta forma una instalación de cámaras de vigilancia real.

#### 2.6.2.5. Iluminación

Las condiciones de iluminación en ElderSim se ven influenciadas por la luz natural del sol y la fuentes de iluminación internas modeladas en UE. Para simular el efecto de la luz solar a lo largo del tiempo, ElderSim utilizar el componente SkySphere Blueprint de UE que incluye un parámetro temporal que permite ajustar la luz solar a cien valores diferentes. En cuanto a las luces interiores, están situadas de acuerdo a distribuciones de iluminación en hogares reales, se apagan por el día y se encienden al anochecer.

### 2.6.2.6. Objetos

Entre las cincuenta y cinco clases de actividades disponibles en ElderSim, treinta y cinco de ellas contienen interacciones con objetos. Estos objetos con los que los ancianos simulados pueden interactuar presentan diferentes características como rigidez (v.gr. taza) o deformabilidad (v.gr. ropa), y cada uno de ellos está modelado de tres formas diferentes para agregar diversidad. Cuando los personajes interactúan con estos objetos durante las simulaciones, estos se unen a las partes del cuerpo con las que entran en contacto y se mueven junto a ellas para que la interacción se vea natural.

### 2.6.2.7. Interfaz de usuario

Como último punto de esta sección vamos a hablar de la interfaz de ElderSim. ElderSim presenta una GUI que permite seleccionar al usuario las opciones con las que deseé que se ejecuten las simulaciones. Entre las opciones disponibles se encuentran las siguientes: elegir el subconjunto de actividades que los personajes podrán llevar a cabo, elegir el escenario donde se ejecutará la simulación, ajustar la iluminación tanto interior como exterior, escoger los modelos que tendrán los objetos en la escena y modificar la perspectiva de las cámaras. Además, para las actividades que contienen movimientos repetitivos, la GUI proporciona tres valores diferentes para la duración de los movimientos; una iteración, múltiples iteraciones y movimientos secuenciales. Asimismo, ElderSim proporciona resolución de vídeo y frame rate ajustable hasta 1920x1080 píxeles y 60 frames por segundo, respectivamente. Finalmente, el software también permite elegir entre tres modalidades de output para los datos generados; RGB vídeo, 2D, y 3D Skeleton.



**Figura 2.12:** Imagen de la GUI de ElderSim para generar datos de acción. Fuente: <https://arxiv.org/pdf/2010.14742.pdf>

### 2.6.3. UnrealROX

UnrealROX [17, 16] es un entorno de simulación en primera persona que incorpora tecnología Realidad Virtual (RV) para la generación de datos sintéticos visuales fotorrealistas. Este entorno está desarrollado en UE, y tiene como objetivo reducir la brecha entre realidad y simulación haciendo uso de escenarios interiores hiperrealistas que son explorados por agentes robóticos que interactúan con objetos también de una manera visualmente realista. Los escenarios fotorrealistas y los agentes robóticos son renderizados por UE en el dispositivo de RV que captura el campo de visión de un agente robótico, para que este pueda ser controlado por un operador humano. Este entorno de realidad virtual permite a los investigadores de visión robótica generar datos realistas y visualmente plausibles para ser usados en una amplia variedad de problemas, como segmentación semántica de clase e instancia, detección de objetos, estimación de profundidad, captación visual y navegación.

#### 2.6.3.1. Agentes robóticos

Una de las partes más importantes de UnrealROX es la representación de los agentes robóticos en el entorno virtual. Estos están representados por una clase base que contiene un modelo 3D que determina su aspecto visual y toda su lógica de comportamiento. Además de la malla 3D que define la geometría de los robots, estos incorporan un esqueleto que define cómo se deformará esa geometría según la posición relativa y la rotación de sus huesos. En total, UnrealRox introduce dos tipos de robots: el Maniquí de UE y el Pepper de Aldebaran (Ver figura 2.13).

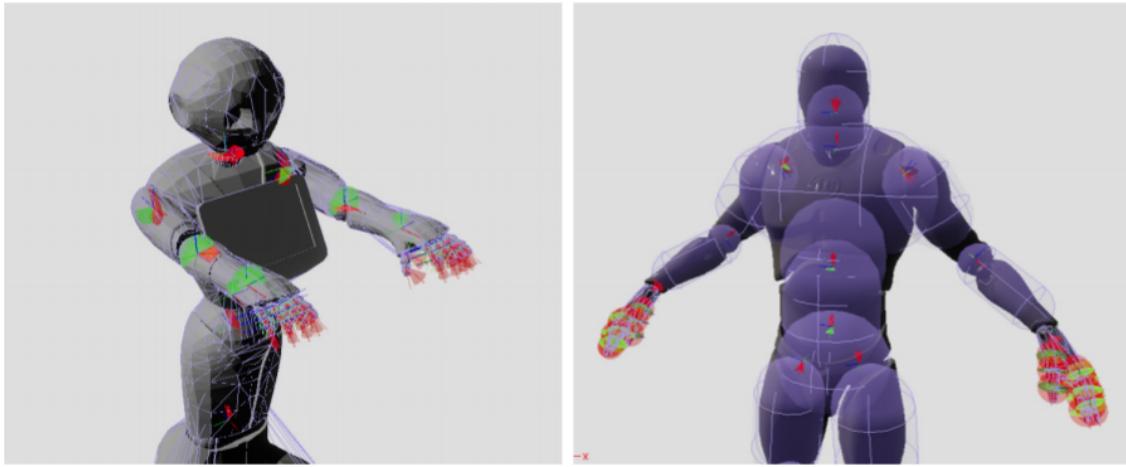
Por lo que concierne al input recibido, los agentes robóticos reaccionan a dos tipos diferentes de input; aquellos que vienen dados por pulsar botones o modificar el valor de ejes, y aquellos que se generan al mover los controladores de movimiento RV. Estos últimos son gestionados por un componente de UE que debe ser agregado a la clase robot y que se encarga de modificar su posición de acuerdo al movimiento realizado por los controladores RV en el mundo real.

Finalmente, la animaciones en UnrealROX son manejadas por una clase específica, la clase animación. Esta clase es la responsable de que los robots del sistema ejecuten animaciones como movimientos de brazos mediante el uso de técnicas de cinemática inversa, la animación de cierre de la mano para el sistema de agarre, y, en el caso de los robots con piernas, controlar la velocidad de desplazamiento para ejecutar la animación de caminar a diferentes velocidades.

#### 2.6.3.2. Subsistema controlador

El sistema controlador de UnrealROX extiende la clase básica *PlayerController* de UE y está desarrollado con la intención de ser compatible con un amplia gama de setups de RV, por lo que es posible usar diferentes dispositivos (como Oculus Rift y HTC Vive Pro) sin excesivo esfuerzo. Esencialmente, este subsistema se encarga de manejar todos los inputs generados por el usuario tanto desde teclado como desde los controladores de RV. Asimismo, el sistema controlador es el responsable de interpretar dicho input y realizar llamadas al sistema de movimiento y de agarre, y a funciones globales del sistema como iniciar y parar el subsistema de grabación, reiniciar la escena, restablecer la posición del dispositivo de visión de RV o controlar el subsistema de HUD para mostrar información de depuración.

---



**Figura 2.13:** Pepper y Maniquí de UE integrados en UnrealRox con *colliders*. Fuente: <https://arxiv.org/pdf/1810.06936.pdf>

#### 2.6.3.3. Subsistema de HUD

UnrealROX ofrece una interfaz que muestra información de depuración al usuario si este la solicita (Figura 2.14). Esta información es presentada en un HUD que se puede habilitar o deshabilitar según lo que decida el usuario. Este subsistema de HUD puede incluso ser completamente desacoplado del sistema para obtener el máximo rendimiento. Los principales tipos de información que el HUD proporciona son los siguientes:

- Grabación: Una línea de texto con el estado de grabación es siempre mostrada al usuario para que este pueda saber si sus movimientos en la escena están siendo grabados.
- Estado: Notifica al usuario con un mensaje en la pantalla que indica los botones relevantes presionados, las articulaciones en contacto con un objeto, etc.
- Error: Imprime un mensaje rojo que indica un error que se muestra en la pantalla durante 30 segundos (o hasta que otro error ocurre).
- Captura de escena: Permite al usuario ver al agente robótico desde un punto de vista diferente al de la cámara en primera persona.

#### 2.6.3.4. Subsistema de agarre

El subsistema de agarre [20] es uno de los componentes centrales de UnrealROX. La acción de agarre está totalmente controlada por el usuario mediante los controles de RV, los cuales se encuentran limitados a los grados de libertad del cuerpo humano. La idea principal de este subsistema consiste en que el usuario pueda manipular e interactuar con diferentes objetos, independientemente de su geometría y pose. De esta forma, el usuario puede decidir libremente con qué objetos interactuar sin restricciones. Un ejemplo del proceso de agarre puede observarse en la figura 2.15.



**Figura 2.14:** Información y mensajes de error mostrados en el HUD de UnrealRox. Fuente: <https://arxiv.org/pdf/1810.06936.pdf>

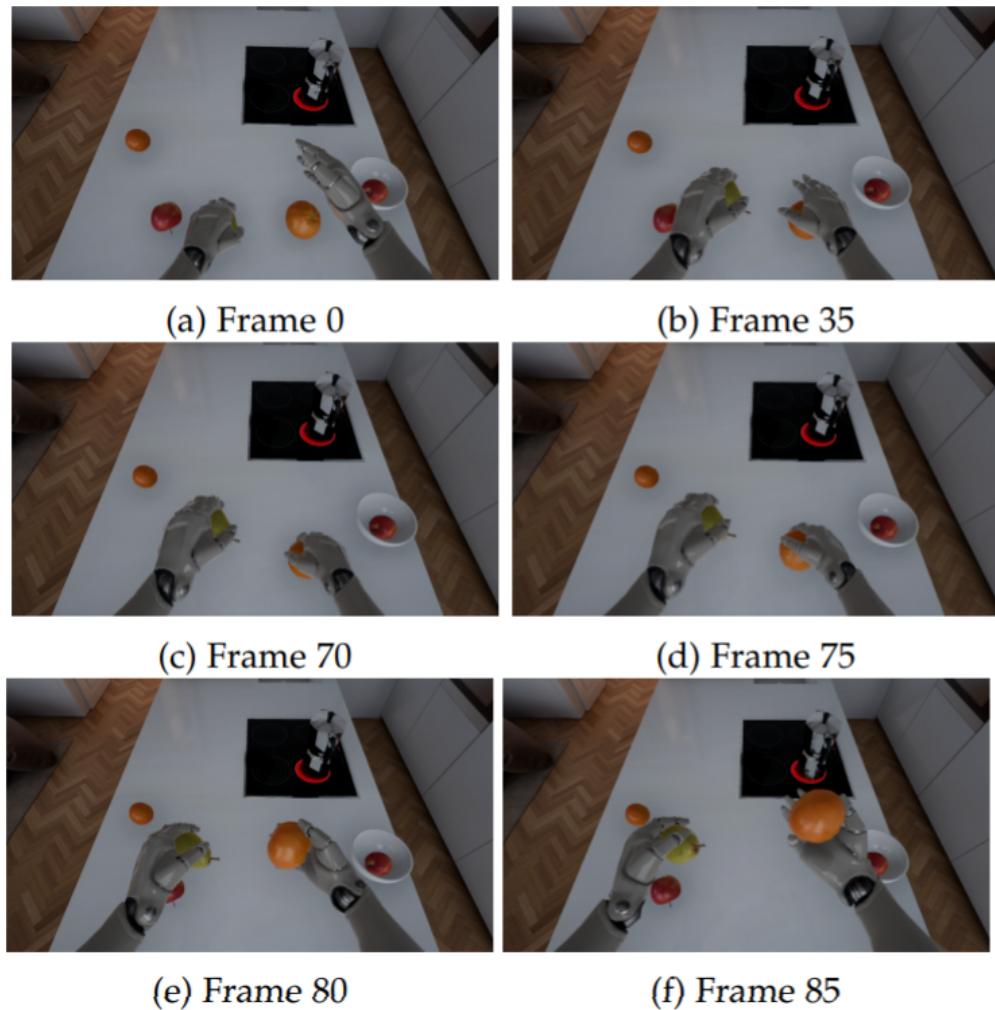
Los agentes robóticos pueden manipular un objeto con cada mano, mover un objeto de una mano a otra, manipular dos objetos al mismo tiempo, dejarlos caer libremente o tirarlos alrededor de la escena. Así mismo, los dedos de los robots cambian su posición suavemente con el objetivo de replicar el movimiento natural de las manos humanas y además evitar que estas atraviesen los objetos.

#### 2.6.3.5. Subsistema multi-cámara

La mayoría de los robots disponibles en el mercado integran múltiples cámaras en diferentes partes de sus cuerpos. Sin embargo, estas pueden ser complementadas con cámaras externas al sistema para proporcionar datos obtenidos desde diferentes puntos de vista. Por esta razón, UnrealROX permite al usuario añadir cámaras estáticas en la escena y colocar cámaras en el cuerpo de los robots (Figura 2.16). De este modo el usuario puede generar datos desde cualquier punto de vista en la escena. Por otra parte, además de ofrecer la posibilidad de añadir tanto cámaras estáticas como unidas al cuerpo de los robots, UnrealROX permite modificar la configuración de las cámaras mediante su interfaz, lo que posibilita al usuario ajustar propiedades como el Field of View (FoV), la gradación del color, el mapeo de tonos y varios efectos de renderizado.

#### 2.6.3.6. Subsistema de grabado

UnrealROX desacopla los procesos de grabación y la generación de datos con el objetivo de lograr altas tasas de fotogramas mientras se recopilan datos en RV sin disminuir rendimiento



**Figura 2.15:** Secuencia de 6 fotogramas mostrando la acción de agarre en UnrealRox. Fuente: <https://arxiv.org/pdf/1810.06936.pdf>

---



**Figura 2.16:** Cámaras unidas a los brazos del actor *Pawn* de UnrealRox. Fuente: <https://arxiv.org/pdf/1810.06936.pdf>

debido a tareas de procesamiento adicionales como cambiar modos de renderizado, cámaras y realizar la escritura de imágenes en disco.

Debido a esta forma de trabajar, el subsistema de grabación recopila en cada fotograma toda la información que el subsistema de reproducción necesitará para reconstruir la secuencia completa y generar todos los datos solicitados.

Para implementar este comportamiento desacoplado entre recopilación de información y generación de datos, UnrealROX presenta un actor llamado ROXTracker. En cada fotograma, este actor itera por todas las cámaras y agentes robóticos de la escena y escribe toda la información necesaria en un archivo de texto de forma asíncrona. Después de que la secuencia sea completamente grabada, el archivo de texto es procesado y convertido en un archivo JSON más estructurado que puede ser fácilmente interpretado por el subsistema de reproducción.

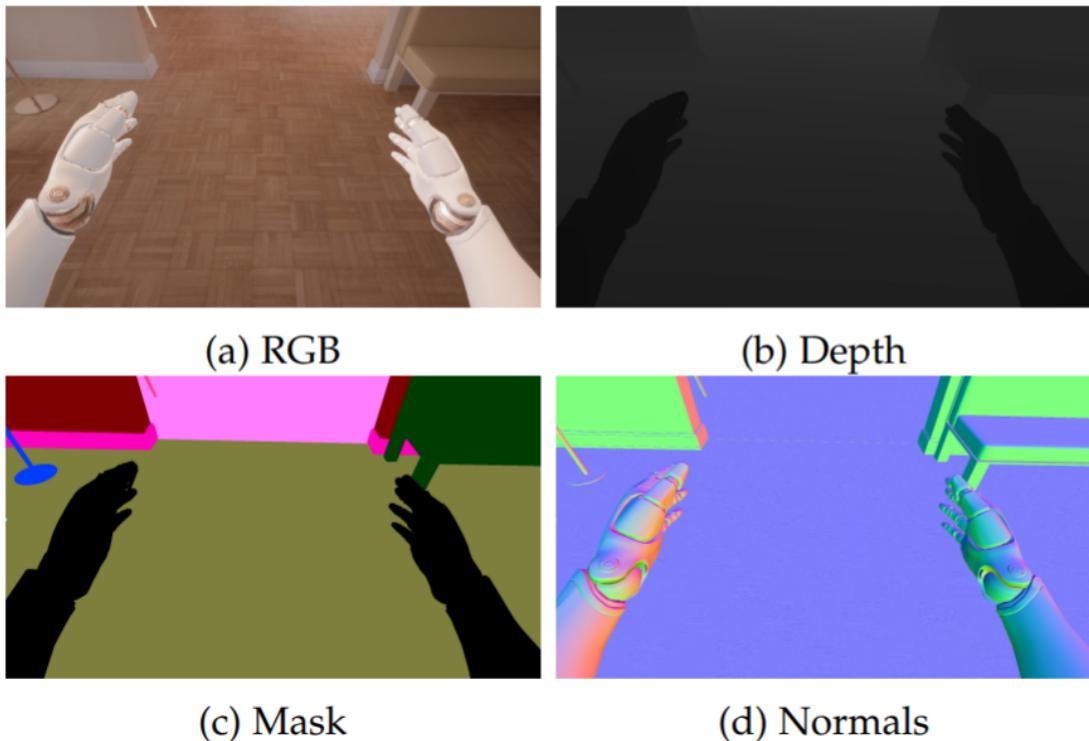
#### 2.6.3.7. Subsistema de reproducción

Este subsistema es el responsable de interpretar el archivo JSON generado por el subsistema de grabación y a partir de este generar los datos solicitados por el usuario. Para ello, deshabilita cualquier simulación de físicas (ya que las poses de los objetos y los esqueletos estarán determinadas por la grabación de la secuencia en el archivo JSON) y luego genera todos los datos solicitados: imágenes RGB, mapas de profundidad, máscaras de segmentación de instancias y normales.

Para generar los datos, cada fotograma, el subsistema de grabación mueve todos los objetos y todas las articulaciones de los agentes robóticos a la posición y rotación previamente grabadas. Una vez todos los elementos están posicionados y rotados, el subsistema itera por

---

cada una de las cámaras generando las imágenes solicitadas utilizando los previamente mencionados modos de renderizado (Ver figura 2.17).



**Figura 2.17:** Modos de renderizado del subsistema de reproducción de UnrealRox. Fuente: <https://arxiv.org/pdf/1810.06936.pdf>

## 2.7. Aplicaciones de los datos sintéticos

Como se ha visto, el uso de datos sintéticos presenta beneficios muy interesantes frente al uso de datos reales. Por esta razón, los datos sintéticos presentan multitud de aplicaciones en diferentes áreas, como la seguridad, el desarrollo de IA, la investigación o el marketing. En esta sección vamos a listar y explicar algunas de las más relevantes.

### 2.7.1. Machine Learning

El rendimiento de los modelos de ML puede ser sustancialmente mejorado gracias al entrenamiento con combinaciones de datos sintéticos y datos reales. Estas combinaciones de ambos tipos de datos permiten proporcionar significativamente más muestras que usando únicamente datos reales y prevenir la infrarepresentación de clases de datos minoritarias. En definitiva, el uso de datos sintéticos supone una gran ayuda que permite a los modelos de ML aprender y entender patrones de forma mucha más precisa [19, 21]. Un ejemplo de esto se puede observar en el desarrollo de coches autónomos, una área de estudio pionera en el uso

de datos sintéticos.

### **2.7.2. Desarrollo ágil de software**

En cuando desarrollo de software se refiere, los conjuntos de datos generados artificialmente son a menudo una mejor opción ya que elimina la necesidad de esperar a que los datos reales se generen fruto de la interacción de los usuarios con el software desarrollado. En última estancia, esto se traduce en una reducción del tiempo de pruebas y un aumento de la flexibilidad durante el desarrollo [24].

Por otro lado, debido a las limitaciones para la obtención de datos personales después de la introducción del Reglamento General de Protección de Datos (RGPD) en Europa, los datos sintéticos han ganado mucho protagonismo en el desarrollo software, y han demostrado jugar un papel fundamental evitando las restricciones de protección que sí acompaña al uso de datos reales [24].

### **2.7.3. Ensayos clínicos y científicos**

En el área de los ensayos clínicos y científicos [1] los datos sintéticos pueden ser utilizados como referencia base para estudios y pruebas futuras cuando los datos reales aún no existen. A pesar de que el uso de datos sintéticos en este campo está aún emergiendo, podemos ver un ejemplo concreto de esta práctica en los ensayos clínicos con enfermedades raras. Estos ensayos se caracterizan por la falta de muestras estadísticas lo que da lugar al uso de datos sintéticos para crear grupos de control artificiales con la intención de obtener resultados más representativos.

### **2.7.4. Seguridad**

Los datos sintéticos pueden ser usados para proteger organizaciones y propiedades tanto de forma virtual como física. Entre las aplicaciones del uso de datos sintéticos en el área de la seguridad podemos destacar los siguientes casos:

- Videovigilancia: En la actualidad el reconocimiento de imágenes juega un papel muy relevante en el campo de la videovigilancia. Sin embargo, para poder aprovechar los beneficios del reconocimiento de imágenes las organizaciones necesitan crear y entrenar sus propios modelos de redes neuronales, lo que presenta dos grandes limitaciones. En primer lugar, el gran volumen de datos necesarios para entrenar una red neuronal. Y en segundo lugar, el gran coste temporal y económico que supone clasificar y etiquetar esta gran cantidad de datos manualmente [8]. En última instancia, estas dos limitaciones se pueden resolver fácilmente gracias al uso de datos generados artificialmente, ya que estos pueden ayudar a reducir el coste de adquirir y clasificar la información necesaria para entrenar una red neuronal gracias a sus beneficiosas características [14].
- Reconocimiento facial: Una de las técnicas para el desarrollo y la comprobación del funcionamiento de sistemas de reconocimiento facial es el uso de *Deep Fakes* [8], un método de inteligencia artificial que permite editar vídeos falsos de personas que aparentemente son reales, utilizando para ello algoritmos de aprendizaje no supervisados (como las ya citadas RGA) y vídeos o imágenes ya existentes [28].

### 2.7.5. Marketing

En marketing [8] los datos sintéticos ayudan a las organizaciones a crear y ejecutar simulaciones muy detalladas a nivel individual para medir y mejorar el impacto de sus campañas en diferentes tipos de usuarios clasificados por edad, sexo, lugar de residencia, etc. y de esta forma ajustar y sacar más rendimiento a sus inversiones en publicidad. Algo a tener en cuenta es que dichas simulaciones no estarían permitidas con uso de datos reales sin el consentimiento de los usuarios debido a la RGPD. Sin embargo, ya que los datos sintéticos presentan las propiedades de los datos reales pero no están restringidos por las regulaciones de privacidad, pueden ser utilizados de forma fiable en estas simulaciones.

### 2.7.6. Monetización de datos

Actualmente, muchos modelos de negocio se basan completamente en monetizar los datos que obtienen de sus usuarios. Las compañías pueden guardar datos, llevar a cabo análisis y vender cualquier información a negocios externos que tengan interés en obtenerla. Algunas organizaciones venden los datos sin procesar para que las compañías externas pueda realizar sus propios análisis, sin embargo esto conlleva muchos problemas normativos, y a menudo los datos se consideran demasiados sensibles para esto [15]. Con el uso de datos generados artificialmente las normativas y regulaciones sobre los datos no son ningún problema. En consecuencia, el valor de esos datos y la velocidad a la que se puede generar valor a partir de ellos aumentan drásticamente. Las empresas pueden incluso generar fuentes de ingresos completamente nuevas. Después de todo, el valor de la mayoría de los datos no es la información personal, sino los conocimientos adquiridos a partir de ellos. Además, los datos sintéticos son más flexibles que los datos reales, ya que pueden automatizarse y reproducirse infinitamente, lo que abre aún más oportunidades de monetización.



## 3. Materiales

Este capítulo está dirigido a enumerar y describir las características de los materiales utilizados para la realización de este TFG. Primeramente, en la sección de software hablamos sobre UE, Visual Studio 2019, GitHub y Axis neuron, las principales herramientas usadas para llevar a cabo el desarrollo de UnrealHome. Más adelante, en el apartado de *assets* citamos aquellos conjuntos de modelos 3D y animaciones han sido utilizados. Finalmente, en la sección de hardware exponemos una tabla con las características de mi ordenador personal, que ha sido el empleado durante todo el proceso de desarrollo.

### 3.1. Software

Como se ha dicho en la introducción, en esta sección vamos a describir las distintas tecnologías usadas para el desarrollo de UnrealHome. En primer lugar, UE ha sido el motor gráfico elegido debido a que este ofrece la posibilidad de crear entornos hiperrealistas para nuestro simulador y ofrece una gran cantidad de herramientas que han facilitado el desarrollo de algunas de las características del proyecto como el sistema de *pathfinding* o la interpolación entre animaciones.

Seguidamente, para todo el proceso de programación se empezó utilizando Visual Studio Code en virtud de su pequeño tamaño comparado con la versión tradicional de Visual Studio. Sin embargo, finalmente se acabó haciendo uso del IDE Visual Studio 2019 dado que presentaba una mayor compatibilidad con UE que facilitó en gran medida la creación de código.

A continuación, GitHub fue el software elegido para realizar el control de versiones del código fuente. La principal razón para el uso de GitHub fue que los supervisores de este proyecto ya habían utilizado esta plataforma para alojar y revisar el código de los proyectos de alumnos anteriores.

Finalmente, Axis neuron, junto a su traje, fue el programa de *motion capture* utilizado por los miembros del grupo 3DPL para grabar algunas de las animaciones que se reproducen en las simulaciones de UnrealHome.

#### 3.1.1. Unreal Engine 4

Unreal Engine<sup>1</sup> [31, 9] (Figura 3.1) es un motor gráfico desarrollado en C++, creado por la compañía Epic Games y mostrado inicialmente en el videojuego de disparos (*shooter*) en primera persona *Unreal* en 1998. Aunque inicialmente fue desarrollado para *shooters* en primera persona, el uso de UE se ha popularizado en la creación de videojuegos pertenecientes a otros géneros y multitud de proyectos no relacionados con los videojuegos. Esta popularización se debe principalmente a que desde sus inicios UE permitió a los desarrolladores trabajar con

---

<sup>1</sup>[www.unrealengine.com](http://www.unrealengine.com)



**Figura 3.1:** Logotipo del motor de juegos UE.

gráficos de más calidad, y texturas e iluminación más realistas que otros software disponibles en el mercado.

Actualmente, UE es considerado uno de los motores de juegos más flexibles ya que es utilizado en multitud de industrias: videojuegos, cine, investigación, exploración espacial, creación de automóviles, etc. Asimismo, UE también permite exportar los proyectos desarrollados a una gran variedad de plataformas entre las que se encuentran PC, Xbox, PlayStation o Nintendo Switch.

Otra de las grandes características que hacen a UE un excelente motor es su gran comunidad. Debido a que se trata de un software de código abierto desde su cuarta versión, actualmente UE cuenta con una gran cantidad de usuarios en sus foros oficiales, lo cuales pueden ser de gran ayuda a la hora de resolver dudas sobre el uso del motor.

Finalmente, la gran cantidad de usuarios que hacen uso de UE también se debe a una de sus herramientas más populares, los Blueprints. Los Blueprints son una herramienta que permite a los usuarios programar sus proyectos de forma visual mediante el uso de nodos, cosa que agiliza en gran medida el proceso de desarrollo, y lo facilita a aquellos usuarios que no tienen conocimientos avanzados de programación.

A pesar de que existen versiones más actuales del motor, en este TFG estamos utilizando la versión no binaria de UE 4.24.3. Las razones que apoyan esta decisión son la estabilidad que esta versión mostraba a la hora de implementar las características de nuestro proyecto, la necesidad de modificar el código fuente del propio motor, y la experiencia previa que se tenía al haber trabajado anteriormente con esta versión.

### 3.1.2. Visual Studio

Visual Studio<sup>2</sup> [32] (Figura 3.2) es un IDE desarrollado por Microsoft. Se utiliza para desarrollar programas informáticos, así como páginas y aplicaciones web, o servicios y aplicaciones móviles. Es compatible con múltiples lenguajes de programación, tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby y PHP, al igual que entornos de desarrollo web, como ASP.NET MVC o Django entre otros. Asimismo, Visual Studio incluye un editor de código que admite IntelliSense, un componente que es de gran ayuda gracias a funciones como el auto completado y la refactorización del código.

A parte de todas estas características que hemos citado, la razón principal por la que se ha escogido Visual Studio como el IDE para el desarrollo del código, en concreto la versión

---

<sup>2</sup><https://visualstudio.microsoft.com>



**Figura 3.2:** Logotipo del IDE Visual Studio.

2019, es por su alta compatibilidad con UE. Esta compatibilidad ofrece ventajas específicas para código de UE y tiempos de compilación más rápidos.

### 3.1.3. GitHub



**Figura 3.3:** Logotipo de la plataforma Github.

GitHub<sup>3</sup> [29] (Figura 3.3) es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git y es utilizado principalmente para la creación de código fuente de programas de ordenador. Ofrece el control distribuido de versiones y la funcionalidad de administración de código fuente (SCM) de Git, además de sus propias características como control de acceso y funciones de colaboración, seguimiento de errores, solicitudes de funciones, gestión de tareas, integración continua y wikis para cada proyecto.

El punto central para la elección de este software, es que ofrece sus servicios de forma gratuita, se trata de una de las plataformas de control de código fuente más extendidas, y es compatible con Visual Studio, lo que permite actualizar los cambios realizados en nuestro código fuente desde el propio IDE.

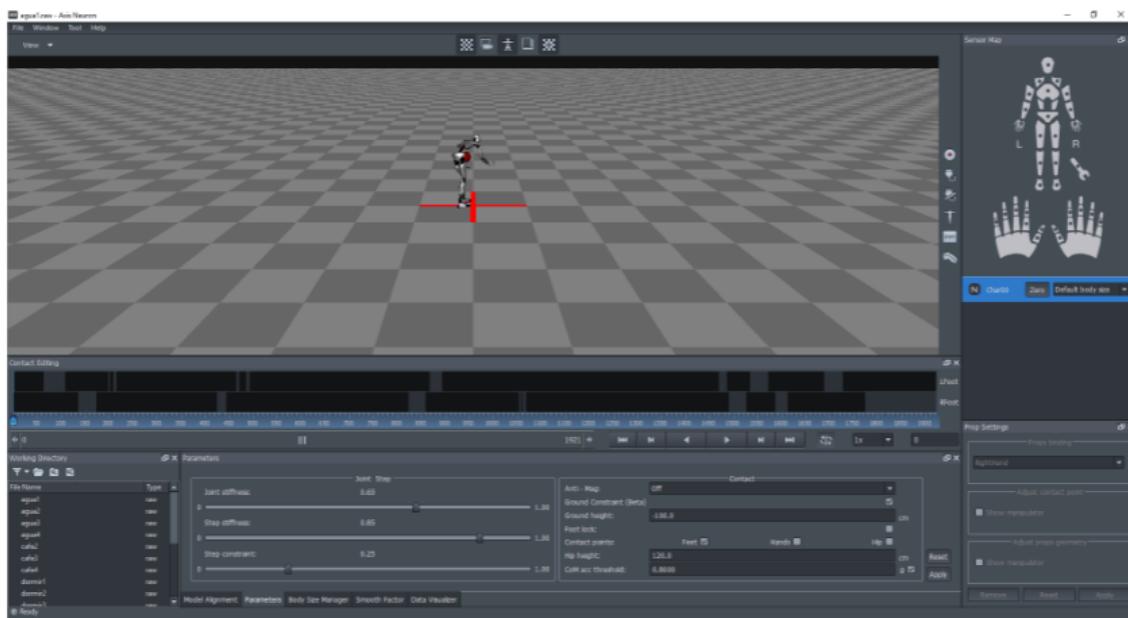
---

<sup>3</sup><https://github.com>

---

### 3.1.4. Axis neuron

Para la obtención de algunas de las animaciones utilizadas en UnrealHome, se ha utilizado Axis neuron<sup>4</sup> junto al traje de *motion capture* asociado a este software. Este programa, permite una gestión sencilla del sistema de *motion capture* y una calibración rápida y sencilla del traje. Los datos se pueden transmitir fácilmente de un entorno a otro con la posibilidad de utilizar la herramienta como servidor. El movimiento del traje se puede transmitir en tiempo real. Además, la grabación es fácil y rápida de realizar con muchas opciones configurables que brindan un alto nivel de flexibilidad (Ver figura 3.4).



**Figura 3.4:** Ventana principal del software axis neuron. Muestra si los sensores están funcionando correctamente.

## 3.2. Assets

Los modelos 3D utilizados en UnrealHome para representar a los objetos interactivos provienen del dataset YCB [2], un conjunto de digitalizaciones de objetos reales en alta resolución. En la figura 3.5 se pueden observar algunos ejemplos modelos que se pueden encontrar en dicho dataset.

Por otro parte, a pesar de que algunas de la animaciones presentes en UnrealHome fueron grabadas con el software Axis neuron, la gran mayoría de ellas provienen de un conjunto de animaciones convertidas a Filmbox (FBX)<sup>5</sup> que tiene su origen en la biblioteca de animaciones de la Universidad Carnegie-Mellon [27].

<sup>4</sup><https://neuronmocap.com>

<sup>5</sup><https://forum.unity.com/threads/huge-free-fbx-mocap-library-released.261401/>



**Figura 3.5:** Ejemplos de algunos de los modelos 3D utilizados para representar a los objetos interactivos.

### 3.3. Hardware

Como se ha dicho en la introducción de este capítulo el hardware utilizado para llevar a cabo los procesos de desarrollo, programación e investigación ha sido mi ordenador portátil personal, el cual presenta las siguientes especificaciones técnicas:

Modelo	Asus ROG Strix GL553DV-DM470
Procesador	Intel® Core™ i5-7300HQ 4 Núcleos, 6M Cache, 2.5GHz hasta 3.5GHz
Memoria RAM	32GB DDR4 2400MHz
Disco duro	1TB 7200rpm SATA
Disco duro SSD	WD Blue SN550 SSD 500GB
Controlador gráfico	NVIDIA® GeForce® GTX1050 4GB GDDR5 VRAM
Batería	48WHrs, 4 Celdas, Ion de Litio
Sistema operativo	Windows® 10 Pro

**Tabla 3.1:** Especificaciones técnicas del ordenador utilizado para desarrollar el proyecto.



## 4. Desarrollo

El proceso de desarrollo de UnrealHome ha involucrado la creación de diferentes actores y componentes, cada uno de ellos responsable de realizar unas funciones determinadas. En este capítulo, vamos explicar con más detalle cómo funcionan a nivel de código estos componentes y actores, y describir de forma más elaborada la funciones que estos llevan a cabo durante las simulaciones.

En primer lugar, hablaremos de los actores presentes en el sistema, a los que llamaremos avatares. Son los personajes responsables de interactuar con su entorno llevando a cabo las animaciones necesarias. Tras ellos, describiremos los objetos interactivos, que serán aquellos objetos situados en los escenarios con los que los avatares tienen la posibilidad de interactuar. Seguidamente, expondremos los componentes y controladores que se encargan de ejecutar o intervienen de algún modo en toda la lógica necesaria para el correcto funcionamiento de las interacciones entre avatares y objetos. En estas secciones, explicaremos como funciona el sistema de navegación con *pathfinding*, la reproducción de animaciones, el sistema de agarre y las mecánicas *Point and Click*. Finalmente, terminaremos este capítulo explicando el proceso de creación y de implementación de la GUI.

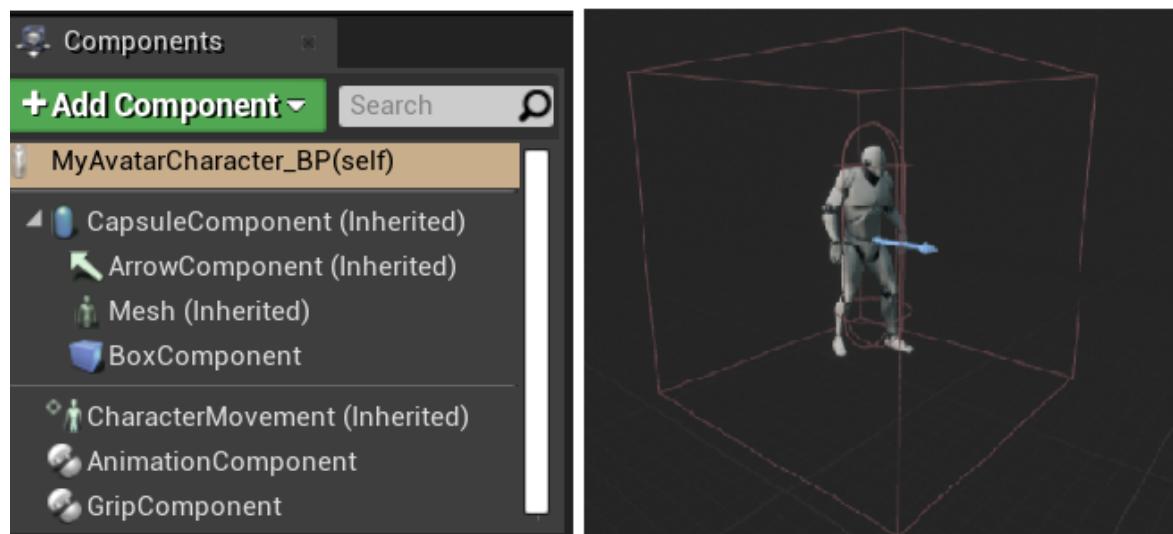
### 4.1. Avatares

Los avatares de UnrealHome (Figura 4.1) son el equivalente a los agentes de VirtualHome [22, 23], los personajes de ElderSim [11] o los agentes robóticos de UnrealROX [17]. Estos avatares tienen como función simular el comportamiento humano de forma realista mediante la realización de actividades cotidianas básicas como pueden ser desplazarse caminando, mirar o sujetar un objeto, comer, etc. Para llevar a cabo estas actividades de la forma más realista posible, los avatares están formados por una serie de componentes y controladores que son los encargados de gestionar los múltiples subprocesos que involucran la ejecución de dichas actividades de forma acertada. Entre estos podemos destacar los siguientes:

- **SkeletalMeshComponent:** Este componente es la representación física de nuestro avatar en la escena virtual. Consiste de un modelo 3D y esqueleto al que se le pueden asignar animaciones para que este las reproduzca.
- **CapsuleCollision:** Componente básico de UE con forma de cápsula que representa el área de colisión de los avatares y es el encargado de generar los eventos de colisión cuando entra en contacto con diferentes elementos de la escena como suelo, paredes, objetos, etc.
- **BoxCollision:** Componente muy similar al *CapsuleCollision*. Sin embargo, tiene forma de cubo, presenta un tamaño variable y es el responsable de generar eventos de superposición, no de colisión, cuando entra en contacto con un objeto interactivo. Estos

eventos de superposición indican que el avatar se encuentra a una distancia óptima para interactuar con el objeto en contacto con la *BoxCollision*.

- CharacterMovementComponent: Componente encargado de gestionar parte del desplazamiento de los avatares alrededor de la escena y ejecutar parte de la lógica del sistema de navegación, procurando que estos eviten obstáculos y encuentren las rutas más cortas para llegar a sus objetivos mediante la implementación de *pathfinding*. Este componente, hereda de la clase *UPawnMovementComponent* que es la clase más básica de UE que permite aplicar funciones de navegación a los actores que la implementan.
- MyAnimationComponent: Componente creado desde cero a partir de la clase *UActorComponent*, para asegurar la correcta reproducción de las animaciones y llevar a cabo la interpolación entre estas. El *MyAnimationComponent* se encuentra en comunicación constante con el *CharacterMovementComponent*, el *MyGripComponent* y el *BoxCollision*, a través del *MyAvatarController*, para saber cuándo reproducir las animaciones de desplazamiento o las de interacción.
- MyGripComponent: Al igual que el *MyAnimationComponent* se trata de un componente creado desde cero y que es el encargado de ejecutar las acciones de agarre de objetos mediante el uso de funciones de unión entre actores y *Sockets* situados en las manos de los avatares.
- MyAvatarController: Controlador responsable de gestionar los eventos de superposición generados por el *BoxCollision*, llevar a cabo las comunicaciones entre el *CharacterMovementComponent*, el *MyAnimationComponent*, el *MyGripComponent* y entre los diferentes elementos necesarios para el correcto funcionamiento del sistema de navegación con *pathfinding*.



**Figura 4.1:** Lista de componentes y representación física de los avatares.

## 4.2. Objetos interactivos

Como ya se mencionó en la introducción, los objetos interactivos son aquellos objetos situados en las diferentes escenas con los que los avatares tienen la posibilidad de interactuar. Estos presentan una serie de características que los diferencian del resto de objetos en la escena.

Los objetos interactivos son objetos creados a partir de la clase *MyObject*, una clase propia que añade tres nuevas variables a la clase *AStaticMeshActor* de UE. Estas tres nuevas variables son las siguientes:

- Interaction Sequence: La secuencia de animación seleccionada de entre todas las disponibles que el avatar reproducirá al interactuar con el objeto seleccionado. Esta secuencia no tiene un valor fijo y puede ser modificada en tiempo de ejecución mediante el uso de la GUI.
- Available Sequences: Se trata de la lista de secuencias de animaciones disponibles que los avatares pueden reproducir al interactuar con un objeto interactivo concreto. Los elementos de esta lista se deben establecer manualmente antes de empezar la ejecución de las simulaciones ya que por defecto estará vacía. En el caso de tratarse de un objeto interactivo que representara comida, esta lista podría contener secuencias de animaciones como comer, sujetar, observar, etc.
- Interaction radius: Como su nombre indica, esta variable determina el rango óptimo al que los avatares deben encontrarse para poder interactuar con un objeto determinado. Cuando un avatar es seleccionado para interactuar con un objeto interactivo, el valor de *Interaction Radius* del objeto determinará el tamaño del componente *BoxCollision* del avatar interactuará con el mismo.
- Initial location: Variable de tipo Vector 3D que contiene la inicial del objeto al empezar las simulaciones. Se usa para ubicar a los objetos en su posición original después de que los avatares hayan interactuado con ellos.

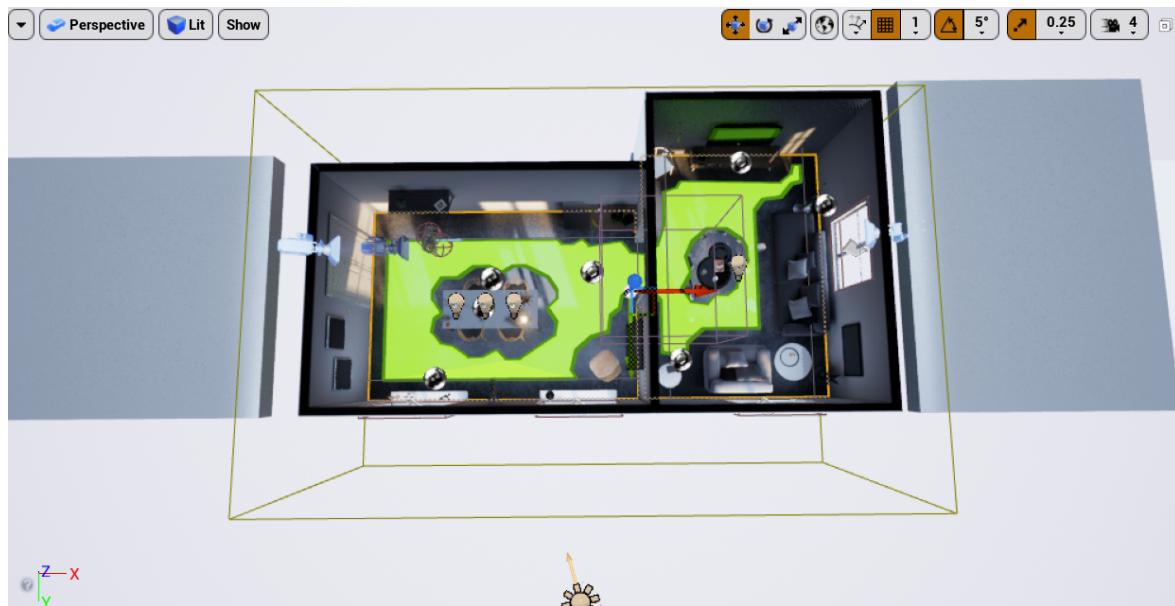
Así mismo, los objetos interactivos también tienen un componente visual llamado *StaticMeshComponent* que les proporciona una representación física en las escenas.

## 4.3. CharacterMovementComponent

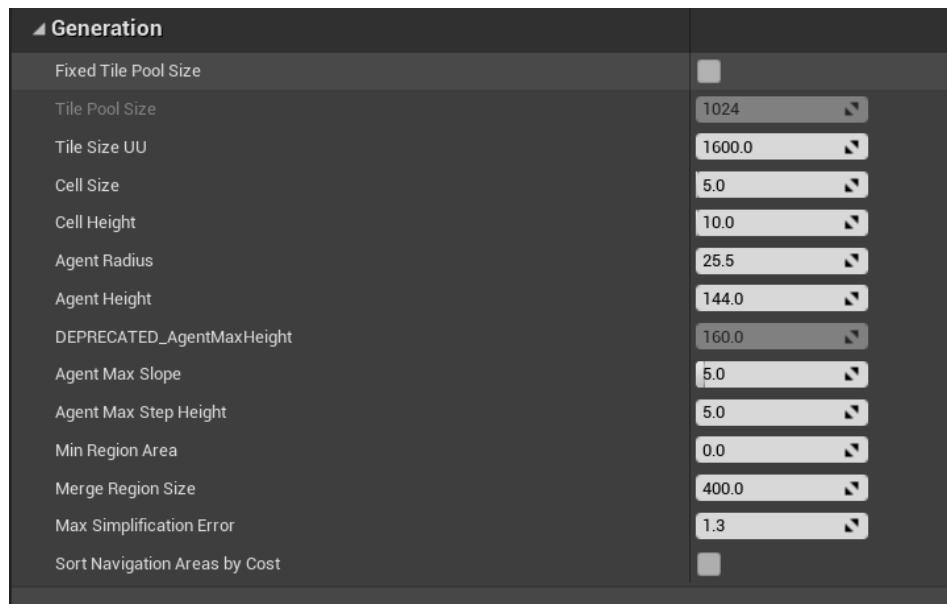
El componente *CharacterMovementComponent*, gestiona parte del funcionamiento del sistema de desplazamiento. Sin embargo, este componente por sí solo no es suficiente para llevar a cabo esta tarea. Para ejecutar el desplazamiento de los avatares hacen falta dos elementos más: el *MyAvatarController*, un controlador que hereda directamente de la clase *AAIController* de UE, y el *ANavMeshBoundsVolume*, un elemento que debe ser colocado en nuestra escena y que calcula qué zonas son navegables por nuestros avatares en función de unos parámetros determinados. En las figuras 4.2 y 4.3 podemos ver el *ANavMeshBoundsVolume* colocado en la escena y los parámetros utilizados en su configuración, respectivamente.

Respecto al *MyAvatarController*, este controlador es responsable gestionar las comunicaciones entre el *CharacterMovementComponent* y el *ANavMeshBoundVolume*. A nivel de

---



**Figura 4.2:** Entidad de la clase *ANavMeshBoundsVolume* situada en la escena. El color verde representa las zonas navegables por los avatares.



**Figura 4.3:** Configuración de la clase *ANavMeshBoundsVolume* para conseguir una mejor adaptación a la distribución de la escena virtual.

código, esta gestión de las comunicaciones entre ambos componentes no es visible para el desarrollador y se hace de forma automática, por lo que la implementación de un sistema de navegación de estas características resulta muy simple. En el bloque 4.1 podemos ver el código necesario en *MyAvatarController* para que el sistema de navegación funcione de forma correcta.

Código 4.1: Código básico para el funcionamiento del sistema de navegación con *pathfinding*

```

1 void AMyAvatarController::Tick(float DeltaTime)
2 {
3     Super::Tick(DeltaTime);
4
5     // Check if this controller instance has all the needed components
6     if (!MovementComponent || !AnimationComponent || !GripComponent)
7         return;
8
9     // Check if this controller instance has a target
10    if (!Target) { return; }
11
12    if (ControllerMode == EControllerMode::Interact)
13    {
14        // Check if interaction animation is already done
15        if (!AnimationComponent->IsInteractAnimationPlaying())
16        {
17            // Set controller mode to move mode
18            ControllerMode = EControllerMode::Move;
19
20            // Drop grabbed object
21            GripComponent->DropObject(Target);
22        }
23    }
24
25    if (ControllerMode == EControllerMode::Move)
26    {
27        // Move actor towards target
28        MoveToActor(Target, Target->GetInteractionRadius() * GetPawn()->GetActorScale().X);
29    }
30 }
```

Como se puede observar, si nuestros avatares contienen el componente *CharacterMovementComponent* y nuestras escenas presentan un *ANavMeshBoundVolume* con la configuración adecuada la implementación del sistema de navegación con *pathfinding* es completamente automática y muy simple. Únicamente necesitamos llamar, dentro de la función *Tick()* del controlador, al método *MoveToActor()* que recibe por parámetro el actor hacia el que queremos que nuestro avatar se dirija y la distancia a la que queremos que se sitúe de él, y UE se encarga de toda la lógica interna por nosotros.

## 4.4. MyAnimationComponent

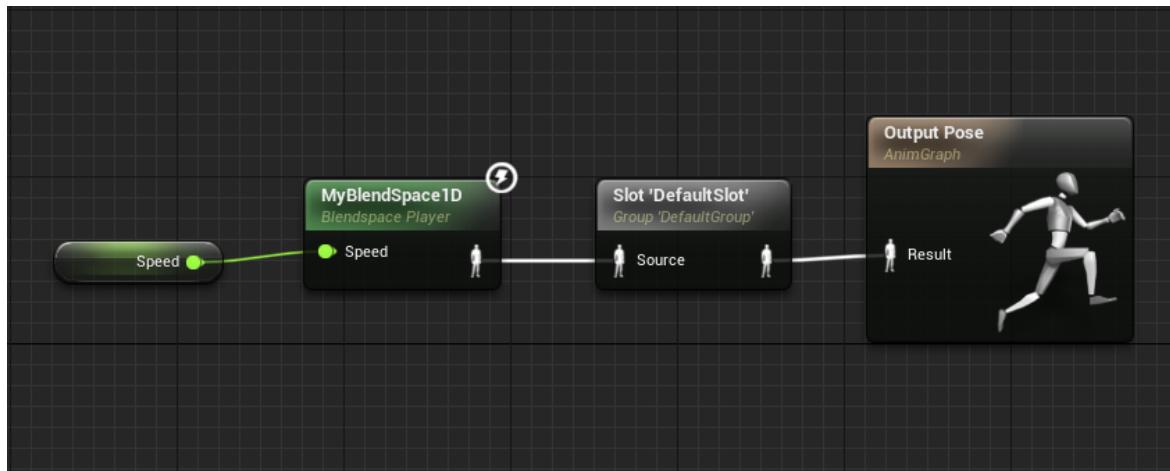
Este componente es el responsable de realizar la reproducción de animaciones y de gestionar la interpolación entre estas. Sin embargo, para el correcto funcionamiento de la interpolación entre animaciones en UnrealHome ha sido necesaria la utilización de diferentes soluciones dependiendo del tipo de interpolación que se quiera realizar. El tipo de interpolación viene dado por el tipo de animaciones que intervienen en ella. Concretamente, en este TFG hemos determinado dos tipos diferentes de animaciones en función del uso que se hace de estas

durante las simulaciones:

- Animaciones de desplazamiento o navegación: Todas aquellas animaciones que estén involucradas en el proceso de navegación (*idle* y caminar).
  - Animaciones de interacción: Aquellas animaciones que representen actividades que los avatares pueden realizar con los objetos interactivos (sujetar, inspeccionar, observar, etc.).

#### **4.4.1. Reproducción de animaciones**

En lo que concierne a la reproducción de las animaciones, esta se realiza desde el *AnimationInstance* de nuestros avatares. Concretamente haciendo uso de su *Blueprint* y de la clase *UBlendSpace1D* de UE.



**Figura 4.4:** Código visual utilizado para reproducir animaciones desde el *AnimationInstance*.

Como se puede apreciar en la figura 4.4, el código presente en el *Blueprint* del *AnimationInstance* se compone de un total de tres nodos. En primer lugar, un nodo llamado *MyBlendSpace1D* que es una referencia a un objeto de la clase *UBlendSpace1D* la cual hemos citado anteriormente. Este nodo recibe como parámetro de entrada una variable tipo *float* (cuya función explicaremos más adelante en la sección de 4.4.2), y es el encargado de generar la animación de desplazamiento que reproducirá nuestro avatar al navegar por la escena.

Seguidamente, apreciamos otro nodo llamado *Slot 'DefaultSlot'*. Su función es reproducir la animación de interacción cuando el *MyAvatarController* lo requiera, interrumpiendo de forma temporal la animación de desplazamiento generada por *MyBlendSpace1D*. Esto ocurrirá cuando el *BoxCollision* generé un evento de superposición al entrar en contacto con el objeto interactivo objetivo de nuestro avatar, momento en el cuál se ejecutará el bloque de código 4.2.

Código 4.2: Código para la gestión de eventos de superposición presente en el *MyAvatarController*

```
1 void AMyAvatarController::OnOverlapBegin(UPrimitiveComponent* HitComponent, AActor* OtherActor, ↵
    ↪ UPrimitiveComponent* OtherComponent, int32 OtherComponentIndex, bool bFromSweep, const ↵
    ↪ FHitResult& Hit)
```

```

2 {
3     if (Target == OtherActor)
4     {
5         // Set controller mode to interact mode
6         ControllerMode = EControllerMode::Interact;
7
8         // Set and play interact animation in the animation component
9         if (AnimationComponent && GripComponent && BoxComponent && SkeletalMesh)
10        {
11            // Grab selected object
12            GripComponent->GrabObject(Target, SkeletalMesh);
13
14            // Play selected interaction animation
15            AnimationComponent->SetInteractAnimation(Target->GetInteractionSequence());
16            AnimationComponent->PlayInteractAnimation();
17
18            // Set BoxComponent size to 0 to avoid accidental overlapping events
19            BoxComponent->SetBoxExtent(FVector::ZeroVector);
20        }
21    }
22}

```

Para que esta interrupción se vea natural, es necesario realizar una transición gradual desde las animaciones de desplazamiento hacia las animaciones de interacción. En la sección 4.4.2 explicaremos con más detalle como se ha llevado a cabo todo este proceso.

Finalmente, podemos ver un último nodo con el nombre *Output Pose* el cual recibe como parámetro de entrada la animación generada por *MyBlendSpace1D* o *Slot 'DefaultSlot'*, y la reproduce en el *SkeletalMeshComponent* de nuestro avatar.

#### 4.4.2. Interpolación entre animaciones

Respecto a la interpolación entre animaciones, este proceso es de gran importancia si se quiere incrementar el nivel de realismo de las simulaciones, ya que una interrupción abrupta durante la reproducción de una animación es visualmente artificial y puede afectar directamente de forma negativa a la calidad de los datos sintéticos generados con nuestro software. Como se ha expresado unos párrafos más atrás, se han desarrollado dos soluciones distintas para la implementación de esta funcionalidad, una solución por cada uno de los tipos de interpolación posibles en el sistema. Los tipos de interpolaciones están determinados por el tipo de las animaciones que intervienen en dicho proceso, de este modo existen dos tipos posibles de interpolaciones:

- Interpolación desplazamiento-desplazamiento: Aquellas que se realizan desde una animación de desplazamiento hacia otra animación de desplazamiento. Por ejemplo, cuando un avatar empieza a navegar por la escena se debe realizar una transición desde la animación de *idle* hacia la animación de caminar.
- Interpolación desplazamiento-interacción: Aquellas interpolaciones que se producen desde una animación de desplazamiento hacia una animación de interacción. Por ejemplo, cuando un avatar se ha desplazado lo suficiente como para interactuar con un objeto interactivo de la escena se debe realizar una transición desde la animación de caminar hacia la animación de interacción correspondiente.

#### 4.4.2.1. Interpolaciones desplazamiento-desplazamiento

Por lo que respecta a este tipo de interpolaciones, cabe destacar que estas se llevan a cabo mediante el uso de la clase *UBlendSpace1D* de UE. Esta clase permite realizar interpolaciones entre dos o más animaciones en bucle, mediante el uso de un eje unidimensional cuyo valor indicará el peso que tendrán cada una de las animaciones en el fundido. Sin embargo, debido a las propias limitaciones que presenta la versión binaria de UE no es posible, desde C++, asignar a nuestras instancias de la clase *UBlendSpace1D* las animaciones entre las que queremos interpolar. Esto supone una gran limitación para este proyecto ya que durante las simulaciones es necesario realizar interpolaciones entre distintas parejas de animaciones, lo que con la versión binaria de UE supondría tener que crear una instancia de la clase *UBlendSpace1D* de forma manual por cada pareja de animaciones entre las que fuera posible realizar una interpolación. Para solucionar esta limitación se optó por descargar el código fuente de UE y extender la clase *UBlendSpace1D* con código de los archivos *SAnimationBlendSpace1D.cpp* y *SAnimationBlendSpace1D.h* para que esta permitiera la asignación de nuevas animaciones en sus instancias de forma dinámica desde C++. Esta ampliación del código puede consultarse en el anexo A.

Una vez realizada la extensión de la clase *UBlendSpace1D*, el *MyAnimationComponent* ya tiene la capacidad de cambiar las animaciones asignadas a las instancias de dicha clase mediante la ejecución del siguiente código (4.3):

Código 4.3: Código para la asignación de nuevas animaciones a una instancia de la clase *UBlendSpace1D* desde el *MyAnimationComponent*

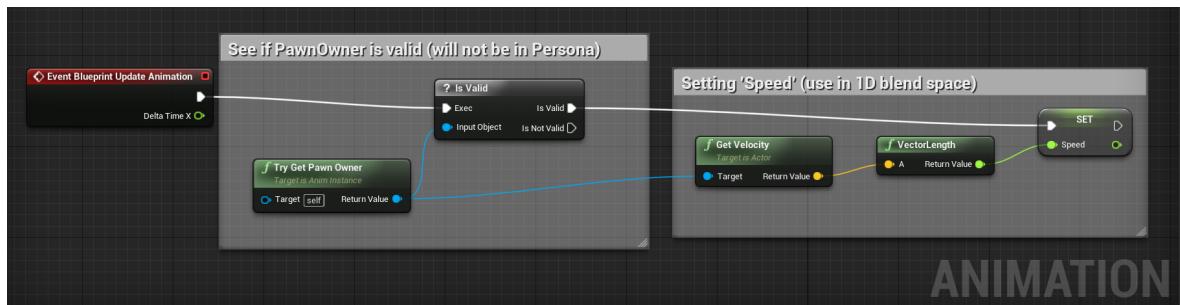
```

1 void UMyAnimationComponent::PlayMoveAnimation()
2 {
3     UE_LOG(LogTemp, Warning, TEXT("Play move animation!"));
4
5     if (BlendSpace)
6     {
7         BlendSpace->ClearData();
8
9         if (!IdleAnimation || !WalkAnimation)
10            return;
11
12        BlendSpace->AddSample(IdleAnimation, FVector(0, 0, 0));
13        BlendSpace->AddSample(WalkAnimation, FVector(100, 0, 0));
14
15        BlendSpace->ResampleData();
16    }
17 }
```

Observando el código 4.3, vemos que el *MyAnimationComponent* accede a una instancia de la clase *UBlendSpace1D* para, en primer lugar, limpiar las animaciones previamente asignadas a dicha instancia y, posteriormente, asignarle dos nuevas animaciones (*idle* y *walk*). Así mismo, a la hora de asignar la animaciones, el método *AddSample()* recibe un segundo parámetro de entrada, un *FVector*, que es esencial para llevar a cabo la interpolación entre las dos animaciones recientemente asignadas. El eje X de este *FVector*, indica la posición que ocupará la animación asignada en el eje unidimensional de la clase *UBlendSpace1D*. Cuando todas las animaciones han sido asignadas y situadas en sus correspondientes posiciones del eje, la interpolación entre las mismas se realizará en función del valor del eje unidimensional. Para explicar este comportamiento, tomamos el caso concreto del código 4.3. En este caso se asignan dos animaciones a la instancia de la clase *UBlendSpace1D*, la animación de *idle*

en la posición 0 y la animación *walk* en la posición 100 del eje unidimensional. En estas condiciones, si el valor del eje fuera 0 o 100 únicamente se reproduciría una de las dos animaciones asignadas, *idle* o *walk* respectivamente, ya que esta representaría el 100% del peso de la interpolación. Sin embargo, si el valor del eje fuera cualquiera número entre 0 y 100 lo que se reproduciría sería un fundido entre ambas animaciones, donde el peso de las animaciones en dicho fundido vendría determinado por la diferencia entre su posición en el eje y el valor del mismo, a más diferencia menor peso.

Hasta ahora, hemos explicado que nuestro proyecto realiza las interpolaciones entre animaciones de desplazamiento haciendo uso de la clase *UBlendSpace1D* y el valor del eje unidimensional que contiene dicha clase. Sin embargo, aún queda aclarar como, desde nuestro software, el valor de dicho eje es modificado durante la ejecución de las simulaciones. Aquí es donde entra en acción la variable tipo *float* llamada *Speed* que podemos ver en la figura 4.4, y que contiene la velocidad a la que los avatares se están desplazando por la escena. Como se aprecia en la figura 4.4, el valor de dicha variable es asignado como el valor de eje unidimensional de la clase *UBlendSpace1D*. De este forma, cuando la velocidad de nuestro avatar sea 0 reproducirá la animación de *idle*, cuando su velocidad sea máxima reproducirá la animación *walk* y cuando el avatar se encuentre acelerando o frenando se reproducirá un fundido entre ambas animaciones. En la figura 4.5, se puede ver como es calculada la velocidad de los avatares.



**Figura 4.5:** Código visual utilizado para calcular la velocidad a la que se desplazan los avatares por la escena.

#### 4.4.2.2. Interpolaciones desplazamiento-interacción

Las interpolaciones desplazamiento-interacción se realizan de una forma mucho más simple que las interpolaciones desplazamiento-desplazamiento, ya que UE ofrece recursos muy útiles como la función *PlaySlotAnimationAsDynamicMontage()* presente en la clase *AnimationInstance*. Este método ofrece la posibilidad de reproducir animaciones con transición de entrada y salida a través de un nodo de tipo *Slot* en el *AnimationInstance* de nuestros avatares. Así pues, para llevar a cabo este tipo de interpolaciones ejecutamos el siguiente código (4.4) cuando el componente *BoxCollision* de nuestro avatar genera un evento de superposición con el objeto interactivo seleccionado:

Código 4.4: Código para realizar la transición desde una animación de desplazamiento hacia una animación de interacción.

```
1 void UMyAnimationComponent::PlayInteractAnimation()
```

```

2{
3    if (AnimInstance && InteractAnimation)
4    {
5        UE_LOG(LogTemp, Warning, TEXT("Play interact animation"));
6
7        AnimInstance->PlaySlotAnimationAsDynamicMontage(InteractAnimation, TEXT("DefaultSlot"), 1.←
8            ↩ f, 1.f);
9}

```

Como podemos apreciar en el bloque de código 4.4, la función *PlaySlotAnimationAsDynamicMontage()* recibe cuatro parámetros de entrada. En primer lugar, la animación que queremos reproducir. En segundo lugar, una variable tipo *FName* que indicará el *Slot* a través del cual queremos reproducir la animación, “*Default Slot*” en este caso. Finalmente, el método recibe dos parámetros más de tipo *float* que indican el tiempo de fundido de entrada y de salida de la interpolación. Respecto al segundo parámetro, si recordamos la figura 4.4 veremos que existe un nodo llamado “*Default Slot*”. Este nodo será el encargado de recibir la animación de interacción enviada por el método *PlaySlotAnimationAsDynamicMontage()*, junto a todos sus parámetros, y redirigirla al nodo *Output Pose*, lo que interrumpirá temporalmente la reproducción de las animaciones de desplazamiento enviadas por la instancia de la clase *UBlendSpace1D*, presente también en la figura 4.4, hasta que la animación de interacción se haya completado.

## 4.5. MyGripComponent

El sistema de agarre de UnrealHome, como se ha expresado anteriormente, es el encargado de ejecutar las acciones de agarre de objetos interactivos por parte de los avatares de la escena. Para llevar a cabo estas acciones de agarre el *MyGripComponent* se apoya en el uso de *Sockets*, una clase de objeto de UE que nos permite unir unos actores a otros en localizaciones concretas, y funciones específicas para unir y separar los objetos seleccionados a dichos *Sockets*. En la figura 4.6 podemos observar los *Sockets* de los que el sistema de agarre de UnrealHome hace uso.

Respecto a las funciones utilizadas para unir y separar los objetos interactivos de los *Sockets* debemos destacar especialmente dos de ellas: *GrabObject()* y *DropObject()*, encargadas de hacer que los avatares agarren y suelten objetos respectivamente.

### 4.5.1. Agarre de objetos

Respecto a la función *GrabObject()*, en el código 4.2 podemos observar que esta es ejecutada cuando el avatar se encuentra a una distancia óptima para interactuar con el objeto seleccionado.

Código 4.5: Función utilizada para ejecutar la acción de agarre de objetos interactivos.

```

1 void UMyGripComponent::GrabObject(AMyObject* __Object, USkeletalMeshComponent* __SkeletalMesh)
2{
3    if (!__Object || !__SkeletalMesh) { return; }
4
5    // Attach object to right hand socket in skeletal
6    __Object->AttachToComponent(__SkeletalMesh, FAttachmentTransformRules(EAttachmentRule::←
7        ↩ SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepWorld, true), TEXT("←
8        ↩ hand_rSocket"));

```



**Figura 4.6:** Sockets situados en las manos de los avatares de UnrealHome.

```

7   // Disable object physics simulation
8   UStaticMeshComponent* ObjectMesh = _Object->GetStaticMesh();
9   if (ObjectMesh)
10     ObjectMesh->SetSimulatePhysics(false);
11 }
12 }
```

En el código 4.5 apreciamos que el proceso de agarre de objetos se compone principalmente de dos pasos. En primer lugar, unir el objeto interactivo, recibido como parámetro, con la instancia de la clase *USkeletalMeshComponent*, también recibida como parámetro, mediante el uso de la función *AttachToComponent()*. En segundo lugar, desactivar la simulación de físicas por parte del objeto para evitar que este caiga al suelo por acción de la gravedad mientras los avatares interactúan con él.

#### 4.5.2. Liberación de objetos

Por otro lado, en cuanto a la función *DropObject()*, observamos que esta es ejecutada en el código 4.1 cuando el avatar ha terminado de reproducir la animación de interacción con el objeto seleccionado.

Código 4.6: Función utilizada para ejecutar la acción de soltado de objetos interactivos.

```

1  if (!__Object) { return; }
2
3  // Detach object from attached actor/component
4  __Object->DetachFromActor(FDetachmentTransformRules::KeepWorldTransform);
5
6  // Enable object physics simulation
7  UStaticMeshComponent* ObjectMesh = __Object->GetStaticMesh();
8  if (ObjectMesh)
9    ObjectMesh->SetSimulatePhysics(true);
10 }
```

```

11     // Reset object location (This has to be done after enabling the physics simulation NOT before)
12     _Object->ResetLocation();

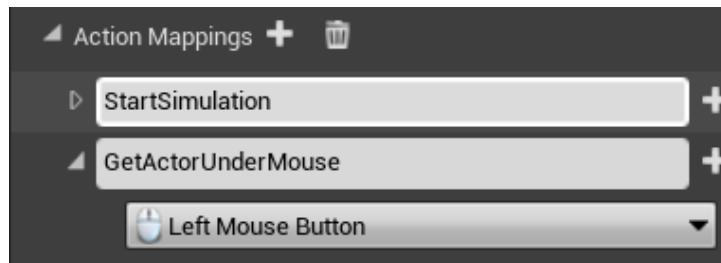
```

En el código 4.6 vemos que la función de soltar objetos es ciertamente similar a la función *GrabObject()*, sin embargo, esta consiste de tres pasos. En primer lugar, separar el objeto interactivo del *Socket* al que se encuentra unido a través de la función *DetachFromActor()*. Seguidamente, reactivar la simulación de físicas por parte del objeto. Y finalmente, devolver el objeto a la posición inicial en la que se encontraba antes de que algún avatar interactuará con él.

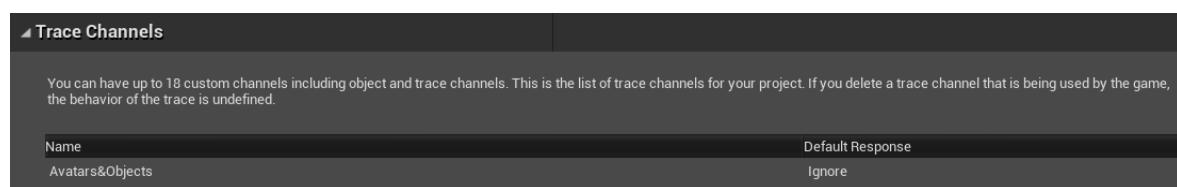
## 4.6. Mecánicas *Point and Click*

Las mecánicas *Point and Click* del proyecto tienen la finalidad de hacer que la ejecución de las simulaciones sea más interactiva. Esta funcionalidad permite a los usuarios del sistema seleccionar avatares y objetos específicos para que estos intervengan en las actividades a realizar durante las simulaciones.

Toda la gestión de estas mecánicas se lleva a cabo en la clase *MyPlayerController*, clase encargada de comunicar las acciones del usuario al sistema, mediante el uso de un evento de ratón llamado *GetActorUnderMouse* (Figura 4.7), un *Trace Channel* creado específicamente para los avatares y los objetos interactivos (Figura 4.8), y la ejecución del código en 4.7 y 4.8.



**Figura 4.7:** Declaración del evento *GetActorUnderMouse* en la configuración del proyecto.



**Figura 4.8:** Declaración del *Trace Channel* específico para avatares y objetos en la configuración del proyecto.

En el código 4.7, se observa como se realiza el enlace entre la función *GetActorUnderMouse()*, función encargada de manejar la selección de avatares y objetos interactivos, con la activación del evento *GetActorUnderMouse*, evento que es activado cuando el botón izquierdo del ratón es pulsado y cuya declaración en la configuración del proyecto podemos ver en la

figura 4.7.

Código 4.7: Código para enlazar el evento *GetActorUnderMouse* con la función 4.8

```

1 void AMyPlayerController::BeginPlay()
2 {
3     // Init input handling
4     EnableInput(this);
5     check(InputComponent);
6     InputComponent->BindAction("GetActorUnderMouse", IE_Pressed, this, &AMyPlayerController::
7         ↪ GetActorUnderMouse);
8
9     // Show cursor on screen
10    bShowMouseCursor = true;
11 }
```

Una vez, que la función *GetActorUnderMouse()* y el evento generado al pulsar el botón izquierdo del ratón están enlazados, esta función se ejecutará cada vez el evento sea disparado. En el código 4.8 vemos todos los pasos involucrados en la selección de avatares y objetos interactivos en dicha función. En primer lugar, se declara un objeto de tipo *FHitResult* que creará una línea unidimensional de colisión con longitud infinita, dirección igual a la de la cámara y origen en la posición del ratón. Cuando este objeto ha sido declarado, se le pasa como parámetro de entrada a la función *GetHitResultUnderCursor()* que filtrará los actores que hayan colisionado con la linea de colisión en función de si estos ignoran o no las colisiones con el *Trace Channel* enviado también como parámetro de entrada. En este caso concreto el *Trace Channel* enviado es el declarado en la figura 4.8, cuyo comportamiento por defecto es *ignore* pero que ha sido configurado para colisionar con avatares y objetos interactivos. Seguidamente, cuando uno de los actores supera el filtro de colisión, el código diferencia si se trata de un avatar o un objeto interactivo para asignarlo a la variable de selección correspondiente. Por último, cuando el actor seleccionado es un objeto interactivo se actualiza la GUI, mediante el evento *AddAvailableActionToUI()*, para que esta muestre las acciones que los avatares tienen disponibles para realizar con dicho objeto.

Código 4.8: Código para seleccionar el avatar u objeto bajo el ratón cuando el evento *GetActorUnderMouse* es disparado.

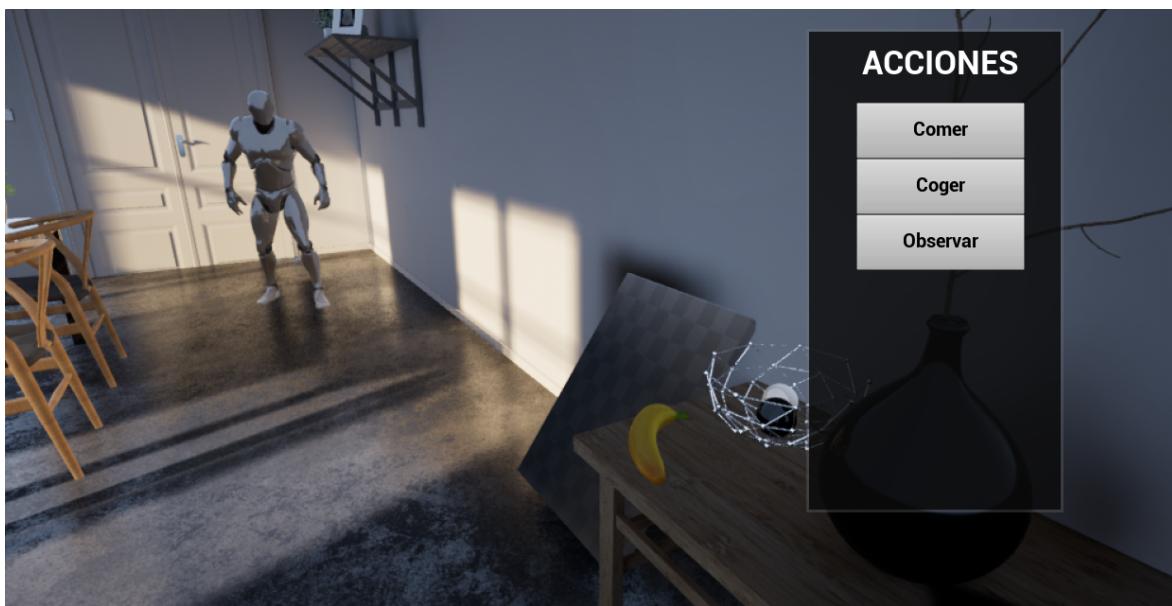
```

1 void AMyPlayerController::GetActorUnderMouse()
2 {
3     FHitResult hitResult;
4     if (GetHitResultUnderCursor(ECollisionChannel::ECC_GameTraceChannel1, true, hitResult))
5     {
6         UE_LOG(LogTemp, Warning, TEXT("Click Mouse %s"), *hitResult.GetActor()->GetName());
7
8         SelectedAvatar = Cast(hitResult.GetActor());
9         if (SelectedAvatar)
10             SelectedAvatarCtrl = Cast(SelectedAvatar->GetController());
11     }
12     else
13     {
14         SelectedObject = Cast(hitResult.GetActor());
15         if (!SelectedObject)
16             return;
17
18         AddAvailableActionsToUI();
19     }
20 }
```

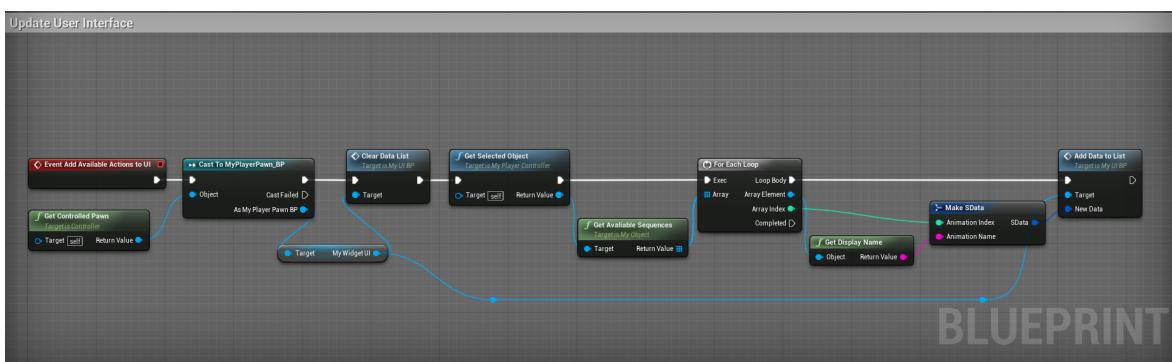
21}

## 4.7. Interfaz de usuario (GUI)

La GUI de UnrealHome permite al usuario seleccionar las acciones que los avatares deben realizar al interactuar con los objetos interactivos distribuidos por la escena. Algo que cabe destacar, es que la GUI es actualizada cada vez que un objeto interactivo diferente es seleccionado, de este modo es capaz de mostrar información actualizada sobre las acciones que se pueden realizar con dicho objeto. En la figura 4.9 podemos ver como la GUI muestra una lista con todas la actividades disponibles a realizar con un plátano.



**Figura 4.9:** GUI del sistema mostrando la lista de actividades disponibles a realizar con el objeto interactivo seleccionado.



**Figura 4.10:** Código presente en *MyPlayerController* para actualizar la información mostrada en la GUI.

La implementación de esta parte del TFG se ha realizado mediante el uso de la clase de UE *Widget Blueprint*. Esta clase permite diseñar GUIs de forma muy intuitiva gracias al uso de un editor visual proporcionado por el propio motor UE. Además, al tratarse de una GUI dinámica que no siempre mostrará la misma información, ha sido necesario programar el comportamiento para actualizar dicha información cuando el objeto seleccionado es cambiado, haciendo uso para ello de *Blueprints*. Concretamente, este código visual se ha implementado en la clase *MyPlayerController* ya que, al igual que las mecánicas *Point and Click*, el uso de la interfaz se considera como una forma de comunicación entre el usuario y el sistema. Esta implementación se puede ver en la figura 4.10, donde observamos que la información mostrada por la GUI es actualizada cada vez que el evento *AddAvailableActionsToUI()* es disparado desde el código 4.8. Una vez este evento es disparado, lo que hace el código es acceder a la lista de secuencias disponibles del objeto seleccionado para nuestro avatar y añadir la información de estas secuencias a nuestra instancia de la clase *Widget Blueprint* a través del nodo *AddDataToList*. Esto actualizará la información de nuestra GUI dándole un formato de lista como el que observamos en la figura 4.9.

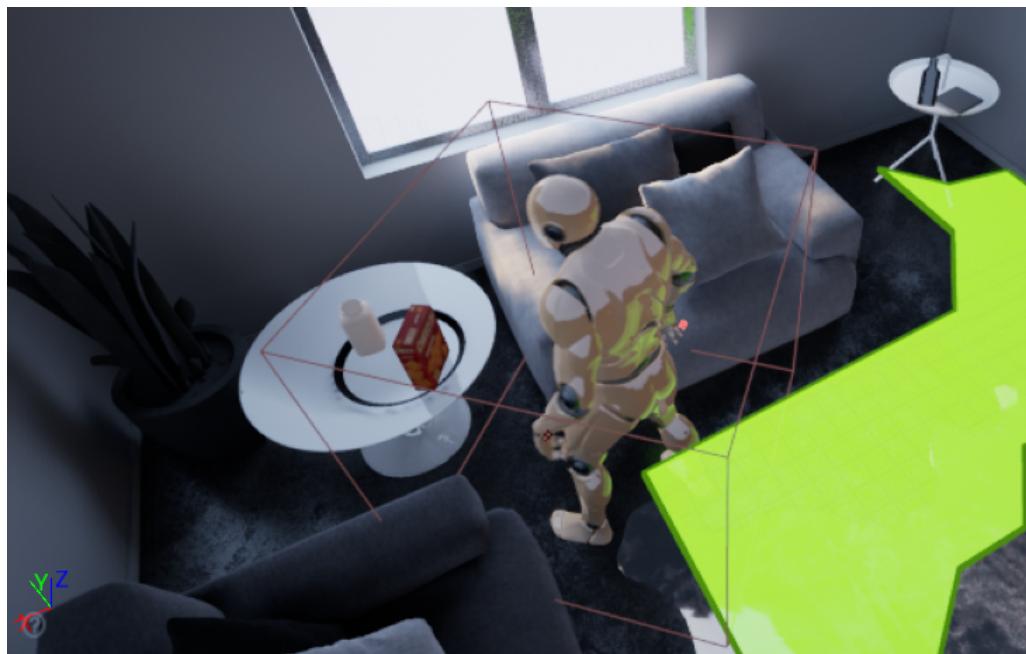


## 5. Limitaciones

UnrealHome no es un sistema perfecto, ya que algunos de sus componentes principales presentan problemas y limitaciones que no debemos ignorar si queremos usar este software para generar datos sintéticos simulados. Este capítulo está dedicado a estudiar en más profundidad dichas limitaciones y exponer el origen de sus causas. Todo esto con el objetivo de ayudar a entender las mejoras para el sistema se proponen en el capítulo 6. Para que el contenido de esta sección sea conciso y ordenado expondremos las limitaciones encontradas clasificándolas según al componente de UnrealHome al que pertenezcan: Limitaciones del sistema de navegación y limitaciones del sistema de agarre.

### 5.1. Limitaciones del sistema de navegación

A pesar de que la navegación con *pathfindig* de UnrealHome ofrece buenos resultados en la mayoría de situaciones, hay ocasiones en las que los avatares son incapaces de alcanzar los objetos interactivos seleccionados por el usuario debido a varias limitaciones inherentes a este sistema que podemos observar en las figuras 5.1 y 5.2.



**Figura 5.1:** Avatar incapaz de alcanzar el objeto interactivo seleccionado debido a la falta de precisión del *NavMeshBoundVolume* para adaptarse al escenario.



**Figura 5.2:** Avatar bloqueado por otro avatar debido que el sistema de *pathfinding* no reconoce a los avatares como obstáculos.

Concretamente, en la figura 5.1 vemos que el componente *NavMeshBoundsVolume* no está correctamente adaptado a la distribución de la escena virtual, por lo que algunas zonas que a primera vista podrían parecer navegables por los avatares no lo son. Esta limitación podría atenuarse o incluso eliminarse completamente ampliando el radio de interacción de los objetos interactivos que se encuentren en dicha zona no navegable. Sin embargo, esto afectaría negativamente al realismo del sistema de agarres, por lo que una solución más adecuada podría ser utilizar una escena menos compleja y con menos obstáculos que la empleada en este caso (Figura 5.3).

Por otra parte, en la figura 5.2 observamos que a un primer avatar bloqueando el paso a un segundo, evitando que este pueda situarse a una distancia adecuada para interactuar con el objeto interactivo seleccionado. Esto ocurre porque el sistema de *pathfinding* no interpreta a los avatares de la escena como obstáculos, produciendo así este tipo de situaciones. Debemos tener en cuenta que cuantos más avatares haya presentes en nuestra escena, más probable será que surjan este tipo de bloqueos.

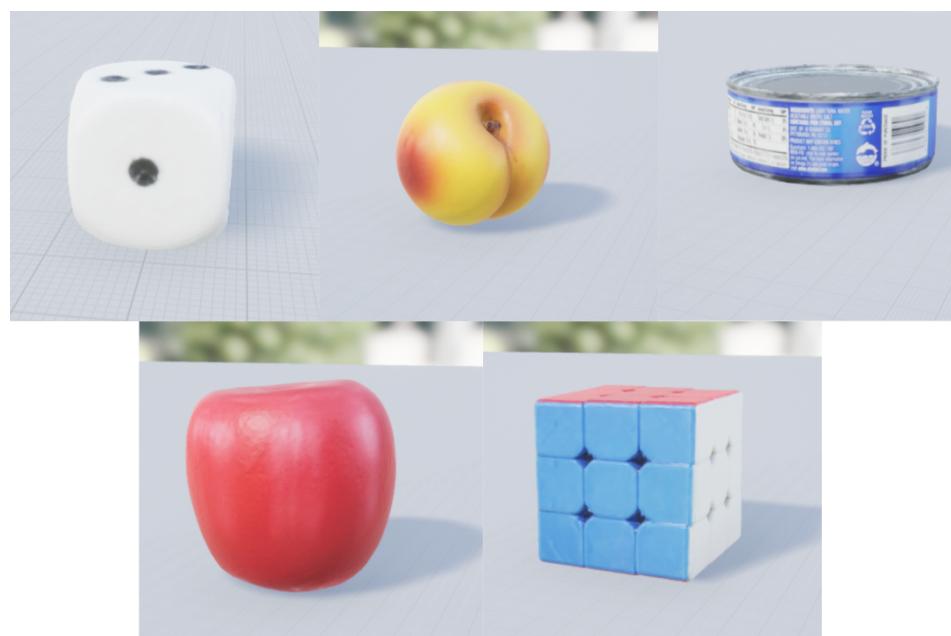
## 5.2. Limitaciones del sistema de agarres

En cuanto al sistema de agarres, vale la pena mencionar que la aparición de algunas de las limitaciones y los problemas que citamos a continuación está fuertemente relacionada con la forma y el tamaño de los objetos sobre los que se efectúen los agarres. En el caso de que los objetos utilizados presenten formas compactas y tamaños similares a los de las manos de los avatares, como los que podemos observar en la figura 5.4, los agarres conseguidos tendrán un alto grado de naturalidad y realismo. Esto se puede apreciar en la figura 5.5.

En cambio, si los objetos involucrados en la interacción presentan formas y tamaños muy



**Figura 5.3:** Escena virtual utilizada para comprobar el funcionamientos de los subsistemas de UnrealHome.



**Figura 5.4:** Objetos con forma compacta y tamaño adecuados para las manos de los avatares.



**Figura 5.5:** Agarres realizados sobre objetos con tamaño adecuado para las manos de los avatares y forma compacta.

heterogéneos, como los que se pueden ver en la figura 5.6, los agarres ejecutados por los avatares se alejaran de la naturalidad y el realismo anteriormente mencionados, y parecerán mucho más artificiales. Esto se puede ver muy claramente en los dos primeros casos de la figura 5.7, donde la forma y tamaño menos uniforme de los objetos ha llevado a que se produzcan solapamientos muy notables entre estos y las manos de los avatares.

Por otro lado, el sistema de agarres también genera varios problemas para nada relacionados con el tamaño y la forma de lo objetos. En primer lugar, cuando se agarra un objeto, este es desplazado automáticamente a la posición del *Socket* localizado en la mano del avatar. Si bien este comportamiento es inherente al modo en el que el sistema de agarres ha sido desarrollado, no se puede ignorar que este desplazamiento automático de los objetos reduce en gran medida el realismo de los agarres ejecutados durante las simulaciones. En segundo lugar, como se ha explicado anteriormente en 4.5, al terminar de reproducir la animación de interacción el *MyGripComponent* reactiva la simulación de físicas de los objetos interactivos. Esto hace que en algunos casos los objetos con los que se ha interactuado caigan al suelo o se desplacen a zonas inalcanzables para los avatares, evitando así que estos puedan volver a interactuar con ellos si no se reinicia la simulación.



**Figura 5.6:** Objetos con tamaños y formas muy heterogéneos.



**Figura 5.7:** Agarres realizados sobre objetos con tamaño y forma muy heterogéneos.

---



# **6. Conclusiones y trabajo futuro**

Para terminar con la redacción de este documento, a continuación mostramos las conclusiones elaboradas después de estudiar el campo de la generación de datos sintéticos y completar el desarrollo del software UnrealHome. Además, en este capítulo también listamos una serie de mejoras que se podrían incorporar a UnrealHome con el objetivo de atenuar o directamente eliminar algunas de sus limitaciones actuales.

## **6.1. Conclusiones**

Como hemos podido ver a lo largo de este documento, la generación de datos sintéticos es una herramienta cada vez más utilizada en multitud de campos de trabajo y de investigación. Por ello, es bastante probable que las técnicas y procesos usados para llevar a cabo estas tareas ganen más y más relevancia en los próximos años. De igual modo, en este documento también hemos explicado las diferencias entre datos sintéticos tradicionales y simulados, destacando que estos últimos presentan una serie de ventajas que los convierten en un recurso de gran valor, capaz de acelerar y abaratar el coste de desarrollo de multitud proyectos provenientes de industrias como la aeroespacial o la automovilística. Debido a esto, los entornos de simulación similares a UnrealHome, diseñados específicamente para generar datos sintéticos simulados hiperrealistas, podrían ser el inicio de toda una revolución en el campo de la generación de datos sintéticos.

La generación de datos simulados no solo permite obtener datos de una forma más interactiva y visual que las técnicas tradicionales. Sino que además, da la posibilidad de hacer que todo el proceso de generación sea mucho más flexible. Esta flexibilidad permite superar muchas de las limitaciones que presentan los modelos de generación de datos sintéticos tradicionales como las RGA o los AEV, y posibilita la modificación de muchas de las variables involucradas en el proceso de generación sin perjudicar el realismo de los datos generados. Así mismo, más allá de todos estos beneficios y al igual que los datos sintéticos tradicionales, los datos simulados nos permiten sortear los costes de la recopilación y etiquetado manual de datos a gran escala, las preocupaciones de privacidad asociadas a los datos personales y el sesgo inherente a los conjuntos de datos recopilados manualmente. Sin embargo, cabe destacar que los sistemas como UnrealHome son novedosos en el campo de la generación de datos sintéticos por lo que será necesario esperar para poder explotar todo el potencial que esta nueva tecnología nos ofrece.

## **6.2. Trabajo futuro**

Como se ha mencionado en el capítulo 5, UnrealHome es un software que presenta muchas limitaciones que pueden afectar al realismo y la calidad de las simulaciones ejecutadas.

Por ello, en esta sección listamos aquellos cambios o mejoras que se podrían realizar para perfeccionar su rendimiento, y transformarlo en un software mucho más completo y preciso.

1. Modificar los parámetros de la generación del *NavMeshBoundsVolume* (Figura 4.3) para que se ajuste de forma más precisa a los escenarios virtuales y corregir la lógica del sistema de *pathfinding* para que reconozca a los demás avatares de la escena como obstáculos. Todo esto con el objetivo de enmendar los problemas del sistema de navegación que se han explicado en 5.1.
2. Combinar técnicas de cinemática inversa en las articulaciones de los brazos con el algoritmo de agarre de Unreal Grasp [20] para incrementar el realismo de los agarres producidos por el *MyGripComponent*.
3. Añadir objetos interactivos con estados alterables como los utilizados en VirtualHome [23] para aumentar la variabilidad de las acciones realizables por los avatares y la diversidad de los datos generados.
4. Implementar un sistema de grabación y reproducción interno similar al descrito en la sección de UnrealRox [17] (2.6.3) para posibilitar a los usuarios la generación de datos sin necesidad de usar un software externo.
5. Extender las mecánicas *Point and Click* para que los objetos y avatares seleccionados sean destacados, y puedan ser diferenciados fácilmente del resto de elementos de la escena virtual no seleccionados.

# Bibliografía

- [1] Anju. What clinical trial leaders need to know about using synthetic data, 2020.
- [2] Y. Benchmarks. Ycb benchmarks – object and model set.
- [3] J. Cambronero. Datos sintéticos con gans (i): ¿por qué datos sintéticos?, 2020.
- [4] Datagen. Simulated data is synthetic data 2.0, 2020.
- [5] E. Devaux. Types of synthetic data and real-life examples, 2018.
- [6] E. Devaux and D. C. Wehmeyer. An overview of synthetic data types and generation methods, 2021.
- [7] C. Dilmegani. Synthetic data generation: Techniques, best practices and tools, 2021.
- [8] C. Dilmegani. The ultimate guide to synthetic data in 2021, 2021.
- [9] I. Epic Games. Unreal engine.
- [10] A. R. Fernández. Los datos sintéticos, la clave para mejorar la inteligencia artificial, 2018.
- [11] H. Hwang, C. Jang, J. C. Geonwoo Park, and I.-J. Kim. Eldersim: A synthetic data generation platform for human action recognition in eldercare applications, 2020.
- [12] J. Jordan. Variational autoencoders., 2018.
- [13] D. P. Kingma and M. Welling. Auto-encoding variational bayes, 2014.
- [14] R. Koch. How synthetic data combats security and privacy challenges in regulated industries, 2020.
- [15] R. Malde. The many use cases for synthetic data, 2020.
- [16] P. Martinez-Gonzalez, S. Oprea, J. A. Castro-Vargas, A. Garcia-Garcia, S. Orts-Escalano, J. Garcia-Rodriguez, and M. Vincze. Unrealrox+: An improved tool for acquiring synthetic data from virtual 3d environments, 2021.
- [17] P. Martinez-Gonzalez, A. G.-G. Sergiu Oprea, A. Jover-Alvarez, S. Orts-Escalano, and J. Garcia-Rodriguez. Unrealrox: An extremely photorealistic virtual reality environment for robotics simulations and synthetic data generation, 2019.
- [18] N. Mayer, E. Ilg, P. Fischer, C. Hazirbas, D. Cremers, A. Dosovitskiy, and T. Brox. What makes good synthetic training data for learning disparity and optical flow estimation?, 2018.

- [19] S. I. Nikolenko. Synthetic data for deep learning, 2019.
- [20] S. Oprea, P. Martinez-Gonzalez, A. Garcia-Garcia, J. A. Castro-Vargas, S. Orts-Escalano, and J. Garcia-Rodriguez. A visually plausible grasping system for object manipulation and interaction in virtual reality environments, 2019.
- [21] M. Platzer. Boost your machine learning accuracy with synthetic data, 2020.
- [22] X. Puig. Virtualhome documentation.
- [23] X. Puig, K. Ra, M. Boben, J. Li, T. Wang, S. Fidler, and A. Torralba. Virtualhome: Simulating household activities via programs, 2018.
- [24] Syntho. Synthetic data: software test and development environment.
- [25] L. Tan. Generating synthetic tabular data, 2019.
- [26] M. D. Tenorio. ¿qué son los datos sintéticos?, 2020.
- [27] C. M. University. Carneige mellon university - home page.
- [28] Wikipedia. Deepfake.
- [29] Wikipedia. Github.
- [30] Wikipedia. Red generativa antagónica.
- [31] Wikipedia. Unreal engine.
- [32] Wikipedia. Visual studio.

## A. Anexo I: Extensión del código de la clase *UBlendSpace1D*

Código A.1: Código añadido al archivo *BlendSpace1D.h* para la asignación dinámica de animaciones desde C++

```
1 // Beginning of BlendSpace1D extension for TFG2020 Cesar Molto Morilla
2 public:
3     ENGINE_API void ClearData();
4     ENGINE_API void ResampleData();
5
6 private:
7
8     struct FIndexLinePoint
9     {
10         float Position;
11         int32 Index;
12         FIndexLinePoint(const float InPosition, const int32 InIndex) : Position(InPosition), Index(InIndex) ↪
13             ↪ {}
14     };
15
16     struct FLineElement
17     {
18         // Explicit constructor since we need to populate the Range value
19         explicit FLineElement(const FIndexLinePoint& InStart, const FIndexLinePoint& InEnd) : Start(←
20             ← InStart), End(InEnd), bIsFirst(false), bIsLast(false)
21         {
22             Range = End.Position - Start.Position;
23         }
24
25         bool PopulateElement(const float ElementPosition, struct FEditorElement& InOutElement) const;
26         bool IsBlendInputOnLine(const FVector& BlendInput) const
27         {
28             return (BlendInput.X >= Start.Position) && (BlendInput.X <= End.Position);
29         }
30
31         const FIndexLinePoint Start;
32         const FIndexLinePoint End;
33         bool bIsFirst;
34         bool bIsLast;
35         float Range;
36     };
37
38     /** Populates EditorElements based on the Sample points previously supplied to AddSamplePoint */
39     void CalculateEditorElements();
40
41     /**
42      * Data Structure for line generation
43      * SamplePointList is the input data
44      */
45     TArray<float> SamplePointList;
46
47     /** Line elements generated from the given samples */
48     TArray<FLineElement> LineElements;
```

```

47  /** Editor elements generated by CalculateEditorElements */
48  TArray<struct FEditorElement> EditorElements;
49
50
51  /** Defines the range of the editor */
52  float MinGridValue;
53  float MaxGridValue;
54
55  /** Number of points that we have to generate FEditorElements for */
56  // int32 NumGridPoints;
57  int32 NumGridDivisions;
58
59  //~ End of BlendSpace1D extension for TFG2020 Cesar Molto Morilla

```

Código A.2: Código añadido al archivo *BlendSpace1D.cpp* para la asignación dinámica de animaciones desde C++

```

1 // Beginning of BlendSpace1D extension for TFG2020 Cesar Molto Morilla
2
3 void UBlendSpace1D::ClearData()
4 {
5     SampleData.Empty();
6 }
7
8 void UBlendSpace1D::ResampleData()
9 {
10    SamplePointList.Reset();
11
12    const TArray<FBBlendSample>& BlendSamples = GetBlendSamples();
13    if (BlendSamples.Num())
14    {
15        for (const FBBlendSample& Sample : BlendSamples)
16        {
17            // Add X value from sample (this is the only valid value to be set for 1D blend spaces / aim offsets
18            if (Sample.bIsValid)
19            {
20                SamplePointList.Add(Sample.SampleValue.X);
21            }
22        }
23        CalculateEditorElements();
24
25        // Create point to sample index list
26        TArray<int32> PointListToSampleIndices;
27        PointListToSampleIndices.Init(INDEX_NONE, SamplePointList.Num());
28        for (int32 PointIndex = 0; PointIndex < SamplePointList.Num(); ++PointIndex)
29        {
30            const float Point = SamplePointList[PointIndex];
31            for (int32 SampleIndex = 0; SampleIndex < BlendSamples.Num(); ++SampleIndex)
32            {
33                if (BlendSamples[SampleIndex].SampleValue.X == Point)
34                {
35                    PointListToSampleIndices[PointIndex] = SampleIndex;
36                    break;
37                }
38            }
39        }
40        FillupGridElements(PointListToSampleIndices, EditorElements);
41    }
42 }
43
44 bool UBlendSpace1D::FLineElement::PopulateElement(const float ElementPosition, FEditorElement& ↪
45                                                 ↪ InOutElement) const
46 {
47     // If the element is left of the line element

```

```

48 if (ElementPosition < Start.Position)
49 {
50     // Element can only be left of a point if it is the first one (otherwise line is incorrect)
51     if (!bIsFirst)
52     {
53         return false;
54     }
55
56     InOutElement.Indices[0] = Start.Index;
57     InOutElement.Weights[0] = 1.0f;
58     return true;
59 }
60 // If the element is right of the line element
61 else if (ElementPosition > End.Position)
62 {
63     // Element can only be right of a point if it is the last one (otherwise line is incorrect)
64     if (!bIsLast)
65     {
66         return false;
67     }
68     InOutElement.Indices[0] = End.Index;
69     InOutElement.Weights[0] = 1.0f;
70     return true;
71 }
72 else
73 {
74     // If the element is between the start/end point of the line weight according to where it's closest to
75     InOutElement.Indices[0] = End.Index;
76     InOutElement.Weights[0] = (ElementPosition - Start.Position) / Range;
77
78     InOutElement.Indices[1] = Start.Index;
79     InOutElement.Weights[1] = (1.0f - InOutElement.Weights[0]);
80     return true;
81 }
82 }
83
84 void UBlendSpace1D::CalculateEditorElements()
85 {
86     const FBlendParameter& BlendParameter = GetBlendParameter(0);
87
88     MinGridValue = BlendParameter.Min;
89     MaxGridValue = BlendParameter.Max;
90     int32 NumGridPoints = BlendParameter.GridNum + 1;
91     NumGridDivisions = BlendParameter.GridNum;
92
93     // Always clear line elements
94     LineElements.Empty(SamplePointList.Num() > 1 ? SamplePointList.Num() - 1 : 0);
95
96     // Only create lines if we have more than one point to draw between
97     if (SamplePointList.Num() > 1)
98     {
99         // Sort points according to position value
100        SamplePointList.Sort([](const float& PointA, const float& PointB) { return PointA < PointB; });
101
102        // Generate lines between sampling points from start to end (valid since they were sorted)
103        for (int32 PointIndex = 0; PointIndex < SamplePointList.Num() - 1; ++PointIndex)
104        {
105            const int32 EndPointIndex = PointIndex + 1;
106            const FIndexLinePoint StartPoint(SamplePointList[PointIndex], PointIndex);
107            const FIndexLinePoint EndPoint(SamplePointList[EndPointIndex], EndPointIndex);
108            LineElements.Add(FLineElement(StartPoint, EndPoint));
109        }
110
111        // Set first and last sample flags (safe because at this point there always at least one sample)

```

```

112     LineElements[0].bIsFirst = true;
113     LineElements.Last().bIsLast = true;
114 }
115
116 // Number of division between grid edges
117 const float GridRange = MaxGridValue - MinGridValue;
118 const float GridStep = GridRange / NumGridDivisions;
119
120 // Initialize editor elements to required number of points
121 EditorElements.Empty(NumGridPoints);
122 EditorElements.AddDefaulted(NumGridPoints);
123
124 if (LineElements.Num() == 0)
125 {
126     // Since we did not generate any lines all the samples should correspond to the first sample and fully ↪
127     // ↪ weighted to 1
128     for (FEditorElement& Element : EditorElements)
129     {
130         Element.Indices[0] = 0;
131         Element.Weights[0] = 1.0f;
132     }
133 else
134 {
135     for (int32 ElementIndex = 0; ElementIndex < NumGridPoints; ++ElementIndex)
136     {
137         FEditorElement& Element = EditorElements[ElementIndex];
138         const float ElementGridPosition = (GridStep * ElementIndex) + MinGridValue;
139
140         // Try and populate the editor element
141         bool bPopulatedElement = false;
142         for (const FLineElement& LineElement : LineElements)
143         {
144             bPopulatedElement |= LineElement.PopulateElement(ElementGridPosition, Element);
145             if (bPopulatedElement)
146             {
147                 break;
148             }
149         }
150
151         // Ensure that the editor element is populated using the available sample data
152         check(bPopulatedElement);
153     }
154 }
155}
156
157//~ End of BlendSpace1D extension for TFG2020 Cesar Molto Morilla

```