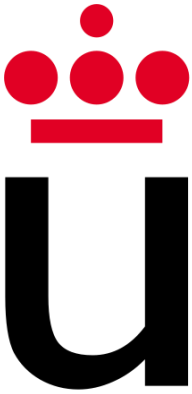

Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Curso Académico 2019/2020

Trabajo Fin de Grado

UTILIZACIÓN DE TECNOLOGÍAS MULTIHILO PARA LA IMPLEMENTACIÓN DE UN SISTEMA DE ANÁLISIS DE LA VISILIDAD DE DIFERENTES OBJETOS EN ESCENA

Autores: Luis Miguel Moreno López
y Ángel Noguero Salgado

Tutor: Daniel Palacios Alonso

“Si no conozco una cosa, la investigaré.”
-Louis Pasteur.

AGRADECIMIENTOS

Agradezco en primer lugar a mis padres. Siempre han estado ahí y este trabajo es tan suyo como mío.
Agradezco al resto de mi familia por ser como son, tan cariñosos y unidos.
Agradezco a mis amigos por estar tanto en las buenas como en las malas.
Agradezco a todos mis profesores que de una forma u otra me han llevado hasta aquí.
En especial quiero dar las gracias a Dani y Ángel porque sin ellos este proyecto no hubiera sido posible.

-Luis Miguel Moreno López

Quiero comenzar dando las gracias a mi padre y a mi madre, Juan Carlos Noguero Sánchez y María Luz Salgado Álvarez. Ellos me han traído a la vida, me han dado sentido y me han mantenido en ella.
Quiero dar las gracias a mi hermano, Alejandro Noguero Salgado, que me ha ayudado siempre a perseguir mis sueños, y que sigue haciéndolo.
Me gustaría agradecer de manera póstuma a mi mascota, *Lassie*, que me acompañó desde que era un niño y que falleció mientras desarrollaba este trabajo. Siempre estuvo allí.
Por último, también debo mi más sincero agradecimiento a mi tutor, Daniel Palacios Alonso y a mi compañero Luis Miguel Moreno López. Ellos han hecho posible este trabajo.
Muchas Gracias.

-Ángel Noguero Salgado.

RESUMEN

En los videojuegos, la experiencia de usuario es vital. En algunos casos, esta puede verse arruinada por diversos factores entre los que destaca un incorrecto sistema de reaparición, también conocido como *“respawn”*. Una de las características fundamentales del *“respawn”* que muchas veces queda relegada a un segundo plano es la visibilidad.

El cálculo de la visibilidad suele verse reducido a su aplicación en computación gráfica. Actualmente, este cálculo se lleva a cabo en tiempo real. Esto provoca tiempos de cómputo y coste elevados; poniendo de relieve, la necesidad de utilización de técnicas que permitan maximizar el rendimiento para un correcto desarrollo.

Nuestra propuesta postula una solución al problema de los sistemas de reaparición. Ofrecemos a la comunidad de desarrolladores la posibilidad de integrar la visibilidad en sus proyectos. En contracorriente a lo anteriormente mencionado, este trabajo ofrece una solución preprocesada que garantiza una incidencia mínima en el rendimiento.

Seguidamente, se expondrá la estructura de la presente memoria. En primer lugar, se explica el estado del arte de las tecnologías consideradas. Además, se listan una serie de objetivos. En la siguiente sección, se detalla la descripción informática llevada a cabo para obtener el producto final y las herramientas utilizadas para ello. También se incluye un análisis de los resultados obtenidos. Finalmente, se listan una serie de conclusiones en relación con el proyecto y se concretan unas líneas futuras del mismo.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

CONTEXTO

Uno de los elementos más importantes en los videojuegos es el sistema de reaparición también conocido como ***"respawn"*** [1]**.** Debido a la variedad en los géneros de este medio, cada juego adapta su propio sistema. **Existiendo elementos comunes** en todas las diferentes implementaciones. Sin embargo, **algunos adolecen de errores** que repercuten de manera negativa en la experiencia de juego.

Una buena implementación de sistema de reaparición es el llevado a cabo por Valve para el videojuego "Left 4 Dead". Conocido como **"The Director"** [2]**,** este sistema es apreciado tanto por consumidores como por competidores. "The Director" es una inteligencia artificial constituida por una serie de algoritmos que trabajan de manera coordinada para dar una buena experiencia de juego. Su función es manejar la reaparición de los agentes valorando distintos aspectos como puede ser el *"game pacing"* [3].

Definir un sistema de reaparición es complejo. Implica considerar conceptos abstractos como el propio usuario, el entorno, el tiempo, y los distintos agentes de inteligencia artificial. Todos estos factores deben incluirse a la hora diseñar el sistema, pero no puede perderse la intención del desarrollador en el camino.

El **análisis de la visibilidad** no es algo nuevo, existen varios estudios [4] sobre la materia, cabe destacar algunos como el *Hierarchical Z-buffer*, *Visibility octree*, *Portals* y *Shadow Culling*. Todos ellos se enfocan en el cálculo gráfico, es decir, pretenden ser una herramienta que aligere la carga del cauce gráfico. Valoramos la posibilidad de dar una nueva utilidad a estos conceptos implementándolos en un sistema de análisis de la visibilidad.

Nuestra propuesta se asemeja a la descrita en el artículo *"The visibility octree: a data structure for 3D navigation"* de C. Saona-Vázquez, I. Navazo y P. Brunet [5]. Nosotros aplicamos los mismos principios, pero con diferencias irreconciliables tanto en la implementación como en el objetivo.

Hoy en día tenemos el deber de realizar **programas eficientes**, más aún cuando se trata de cálculos de visibilidad. Es por esto por lo que decidimos usar **Data-Oriented Technology Stack (DOTS)** [6]**.** Esta tecnología se encuentra enfocada a datos y permite aplicar de manera optimizada técnicas multihilo [7].

DOTS está constituido, principalmente, por los paquetes Entity Component System [8], Job System [9] y Burst [10].

El primero de ellos implementa un patrón de arquitectura software homónimo que surgió en 2007. Este popularizó las siguientes ideas:

"Systems" as a first-class element, "Entities as IDs", "Components as raw Data", and "Code stored in Systems, not in Components or Entities" [11].

Define tres elementos (Entidades, Componentes y Sistemas) con el objetivo final de separar funciones de datos, consiguiendo así mejorar el rendimiento y la claridad del código.

Unity implementa de manera estable por primera vez el sistema de Jobs en su versión 2018.1. La primera versión estable del compilador Burst aparece en la versión de Unity 2019.1. Sin embargo, ECS continua en *preview* (0.14). Esto plantea un **ambiente de desarrollo cambiante** con poca información disponible, lo que entorpece el desarrollo.

Aun así, si algo no se encuentra en la documentación oficial, puede recurrirse a la comunidad de Unity. En ella varios desarrolladores (inclusive miembros del equipo de Unity) ofrecen su conocimiento sobre DOTS.

Como nota final**,** nos es casi de obligado cumplimiento resaltar que, **según los desarrolladores de Unity, el futuro de Unity pasa por DOTS** [12]**.**

PRESENTACIÓN

Considerando el contexto, surge ***Spawntaneous***. Consistente en un programa modular que asiste a los desarrolladores de videojuegos a la hora de implementar su sistema de reaparición. Es precisamente esta faceta modular lo que le dota de una gran escalabilidad funcional.

En este TFG se ha desarrollado el módulo cuya función es calcular la visibilidad de una escena. El concepto de visibilidad puede ser entendido de la siguiente forma: en un espacio euclídeo dado, dos puntos son visibles entre ellos si la línea que los une no se corta con ningún obstáculo [13]. Para realizar dichos cálculos hemos creado una aproximación inspirada en el *Visibility Octree*. Este tiene un carácter preprocesado, lo que implica una gran carga computacional; para solucionar esto, recurrimos a **Data-Oriented Technology Stack (DOTS)** [6]. Durante el desarrollo de este trabajo hemos creado el **segundo *Octree* del mundo** que utiliza DOTS; siendo el nuestro el primero en ser usado en con fines de almacenamiento de cálculos en preprocesado.

OBJETIVOS

1. Aprovechar la tecnología Data-Oriented Technology Stack (DOTS) de Unity para asegurar la eficiencia del software.

Al utilizar **DOTS** se pretende conseguir un **alto rendimiento.** Se busca que el programa sea suficientemente rápido como para que el usuario no se sienta lastrado al utilizarlo.

Usar DOTS supone una labor de **investigación**, ya que no se nos dio noción ninguna acerca de esta tecnología en la carrera. Al emplearla, buscamos convertirnos en **pioneros**. Queremos adquirir conocimiento en un terreno todavía ignoto para la gran mayoría. Buscamos ser punta de lanza de esta innovación, asegurándonos una ventaja en el ámbito profesional.

2. Dar al usuario una herramienta auxiliar para el desarrollo en Unity.

Debe aportar un valor real al usuario, este tiene que verse reflejado en la utilidad de la aplicación y en la usabilidad de esta. El resultado del proyecto debe asistir en la tarea de elaboración de un sistema de reaparición. Ya sea en su gestación o en su comprobación. Es decir, debemos proporcionar un instrumento que los desarrolladores puedan usar de soporte para crear o analizar sus sistemas de desarrollo.

3. Desarrollar una herramienta que permita el cálculo de la visibilidad de un espacio dado.

El programa debe ser capaz de calcular, de manera preprocesada, la visibilidad de todos los puntos posibles de una escena. Es por esto por lo que se estudiará la aproximación desarrollada en *"The visibility octree: a data structure for 3D navigation"* de C. Saona-Vázquez, I. Navazo y P. Brunet [5].

Para poder ordenar el espacio a procesar deberá usarse una estructura de datos apropiada para la división espacial.

DESCRIPCIÓN INFORMÁTICA

ENTORNOS DE DESARROLLO

Se han utilizado los siguientes entornos de desarrollo:

- **Visual Studio Code** es un editor de código de Microsoft gratuito, ampliable mediante complementos (*plugins*). Permite trabajar en varios lenguajes de programación, entre los que se incluye el utilizado para este proyecto, **C#** [14].

- **Unity** es un motor de videojuegos que promulga como uno de sus valores el *"Build once, deploy everywhere"* [15]. Permite el desarrollo multiplataforma de manera simplificada, lo cual explica su popularidad en el desarrollo de los videojuegos. Utiliza **OpenGL** [16] y **Direct3D** [17]. Entre los lenguajes de programación soportados por Unity se encuentra **C#**, que es él que utilizamos. Actualmente trabajamos en la versión **2019.3.12f1**.

Dentro de Unity usamos varios paquetes (por orden alfabético):

- **Burst 1.3.6:** Este compilador está altamente optimizado para el trabajo multihilo. El uso de este compilador junto con Jobs supone un gran aumento del rendimiento en el programa.

- **Collections preview.6 - 0.9.0:** Proporciona una serie de colecciones pensadas para poder ser usadas en operaciones en paralelo.

- **Entities preview.4 - 0.11.1:** Este paquete incorpora la funcionalidad del **Entity Component System**. Se encuentra aún en una fase muy temprana de desarrollo, por lo que sufre cambios de forma constante.

- **Hybrid Renderer preview.4 - 0.5.2:** Incluye soporte de renderizado usando **DOTS**. No es una tubería de renderizado (arquitectura para generar gráficos), sino que funciona como un puente entre la arquitectura de renderizado existente de Unity y DOTS. Existen dos versiones:

  - V1 es más limitada y se introdujo en la versión de Unity **2019.1,** ya no se encuentra en desarrollo actualmente.
  - V2 es la versión experimental introducida en **2020.1** [18]**.** Aún sigue siendo muy limitada, aunque hay planes para ampliarla en un futuro.

En el proyecto se utiliza **Hybrid Renderer v1.**

- **Jobs preview.12 - 0.2.10:** Añade nuevos tipos de ***job***, un *job* es una unidad de trabajo en programación. Empleado en nuestro caso para programación multihilo (un hilo alternativo al principal lleva a cabo la tarea ahorrando así mucho tiempo de trabajo) [19].

- **Mathematics 1.2.1:** Proporciona algunas funciones matemáticas. El compilador Burst presenta una dependencia con este paquete.

- **UIBuilder preview.6 - 1.0.0:** Permite desarrollar interfaces que utilizan la tecnología UIElements [20]. Esta herramienta novedosa de Unity que supone el futuro en el desarrollo de interfaces de usuario. En un primer momento su funcionalidad estaba pensada exclusivamente para **ECS**, en tiempo de ejecución solo podía ser usado con ECS. Sin embargo, su uso en el Editor no presentaba ninguna dependencia [21]**.** Finalmente ampliaron su compatibilidad con MonoBehaviour, que es la forma actual de trabajo en Unity [22].

  Implementa una nueva forma de elaborar interfaces, usa en dúo **UXML** y **USS.** UXML es un formato inspirado en XML, XAML y HTML y contiene la estructura jerárquica de la interfaz [23]. **USS** contiene los estilos de los elementos [24]. Para trabajar con los elementos de un UXML, deben hacerse búsquedas (**UQuery**) de los elementos de la jerarquía. De este modo, se obtiene acceso a ellos y su contenido puede ser modificado [21].

  La herramienta presenta una interfaz gráfica (GUI) que funciona arrastrando los elementos deseados dentro de la jerarquía.

  Este paquete sigue siendo muy inestable.

- **Unity Collaborate 1.2.16:** Permite el trabajo simultáneo de manera remota, funciona de manera semejante a herramientas como **GitHub,** pero está integrada dentro de Unity. Existe simultáneamente una copia en local y una en remoto, evitando así conflictos cuando se trabaja en un mismo archivo.

- **SourceGear DiffMerge 4.2:** Programa utilizado para solucionar conflictos a la hora de subir archivos al repositorio en línea [25]. Compara las versiones local y remota de un archivo, pudiendo así comprobar rápidamente los cambios realizados.

También hemos usado las librerías **Math3d** y **MiConvexHull**:

- Math3d es un conjunto de funciones desarrolladas por **Bit Barrel Media** [26]**.** Permite realizar operaciones matemáticas como intersección línea-plano.
- **MiConvexHull**. Permite generar mallas de modelos convexos a partir de un conjunto de puntos en el espacio [27]**.** Basa su lógica en una serie de algoritmos matemáticos del mismo nombre, **ConvexHull** [28]. Su aplicación se deduce de esta publicación del foro de Unity [29].

REQUISITOS FUNCIONALES Y NO FUNCIONALES

A continuación, procedemos a listar una serie de requisitos funcionales y no funcionales que fueron considerados a la hora de diseñar y desarrollar la aplicación.

REQUISITOS FUNCIONALES	
ID	DESCRIPCIÓN
RF1	Deberá existir en el menú GameObject la pestaña Spawntaneous que recoja todas las funcionalidades del programa.
RF2	Habrá que poder instanciar en la escena todos los GameObjects necesarios para la operatividad del módulo, estando estos debidamente configurados.
RF3	Tendrá que contar con una interfaz que permita interactuar con el módulo.
RF4	Será posible activar y desactivar los instrumentos del módulo.
RF5	Se podrán desplegar y recoger las utilidades de la herramienta.
RF6	Se podrán añadir GameObjects a la lista de *oclusores* de manera manual mediante un instrumento dedicado para ello. No suponen objetos susceptibles a ser añadidos los GameObjects que sirvan para configurar el módulo, y todos aquellos que estén fuera del volumen contendor.
RF7	Se podrán borrar GameObjects de la lista de oclusores de manera manual mediante un instrumento dedicado.
RF8	Se dará la opción de seleccionar automáticamente los oclusores mediante una comprobación matemática.
RF9	Se podrá procesar la visibilidad de la escena dentro del volumen contenedor, y solo se tendrá en cuenta el espacio contenido en este.
RF10	Habrá un modo de navegar a través de la estructura que almacene la división espacial en la escena.
RF11	El usuario podrá seleccionar libremente el objeto que funcionará como volumen contenedor.
RF12	Se podrá seleccionar que GameObject debe actuar como referencia de proyección.
RF13	Se podrá seleccionar que GameObject debe actuar como referencia de oclusión.
RF14	En ningún caso se podrá dejar sin GameObject ninguno de los roles necesarios para el funcionamiento de la aplicación.
RF15	Cuando la navegación llegue a la división espacial que suponga el volumen proyector mostraremos su visibilidad.

RF16	La partición espacial ha de ser visible cuando el usuario lo requiera.

Tabla 1.- Requisitos Funcionales.

REQUISITOS NO FUNCIONALES		
ID	DESCRIPCIÓN	TIPO
RNF1	La interfaz debe estar integrada con el sistema de inspector de Unity.	Usabilidad
RNF2	La interfaz de usuario debe tener un diseño adaptable o *responsive* a fin de garantizar la comodidad del usuario.	Usabilidad
RNF3	El manejo de la aplicación se realizará principalmente a través del cursor del ratón y del clic izquierdo.	Usabilidad
RNF4	La interfaz debe mostrar sugerencias de cómo usarse.	Usabilidad
RNF5	La interfaz debe dejar elegir al usuario el idioma que desee.	Usabilidad
RNF6	El funcionamiento de la interfaz debe acomodarse al funcionamiento de las herramientas nativas de Unity.	Usabilidad
RNF7	El sistema deberá contar con manual de uso.	Usabilidad
RNF8	El sistema deberá mostrar mensajes de error bien formulados e informativos para el usuario final.	Usabilidad
RNF9	Se debe poder añadir o quitar oclusores haciendo uso de tres clics de ratón o menos. Esto se cuenta desde introducir Spawntaneous en la escena.	Usabilidad
RNF10	El pintado de oclusores en la escena debe ser fácilmente reconocible.	Usabilidad
RNF11	El usuario debe saber en todo momento si se encuentra haciendo uso de la herramienta. También debe saber si la ha desactivado.	Usabilidad

RNF12	El usuario debe aprender a manejar el módulo en menos de veinte minutos.	Usabilidad
RNF13	El usuario dispondrá de información de contacto rápida tal como nombre de las personas encargadas del desarrollo.	Usabilidad
RNF14	Debemos limitar los proyectores y oclusores, a fin de reducir la carga computacional.	Eficiencia
RNF15	El sistema rechazará realizar cálculos que supongan tiempos de ejecución inmensamente grandes.	Eficiencia
RNF16	El usuario debe poder ejecutar el procesado de la visibilidad en cualquier momento mientras este no se ya procesando.	Confianza
RNF17	El sistema debe funcionar correctamente mínimo un 90% de las veces.	Confianza
RNF18	El manejo a través de los niveles del *octree* debe ser correcto en un 80% de las veces mínimo.	Confianza
RNF19	El desarrollo del módulo se realiza utilizando metodologías ágiles como Kanban.	Organizacionales
RNF20	El módulo debe utilizar la tecnología DOTS.	Desarrollo
RNF21	Se debe programar lo máximo bajo el patrón de arquitectura ECS.	Desarrollo
RNF22	El código del programa debe estar debidamente comentado y compartimentado.	Desarrollo
RNF23	Todas las librerías y API externas deben tener o estar en disposición de una licencia que permita su uso.	Legislativo

Tabla 2.- Requisitos no funcionales.

DIAGRAMAS

A continuación, se muestran una serie de diagramas que usamos durante el desarrollo del programa. Estos permiten vislumbrar tanto el funcionamiento de la aplicación como los componentes y sistemas de esta.

DIAGRAMA DE CLASES

Tal y como puede apreciarse en la Figura 1, para trazar una línea paralela con ECS, separamos en Componentes, Sistemas y Scripts. Dentro de los componentes se separa por carpeta en función del tipo de entidad que tendrá dicho componente. Ejemplos de ello son los *"Octree Node Components"*, que constituyen el conjunto de componentes que tendrá una entidad de un octante. Cada uno de los sistemas utilizados se haya encapsulado en una carpeta homónima. Finalmente, los Scripts son métodos ajenos a ECS, es por esto por lo que no se encuadran ni como entidad, ni componente ni sistema. Representan las funciones y variables que habrá en cada uno de los archivos de código que tenga la aplicación. El nombre de dicho archivo aparece en un rótulo encima de la lista de sus variables.

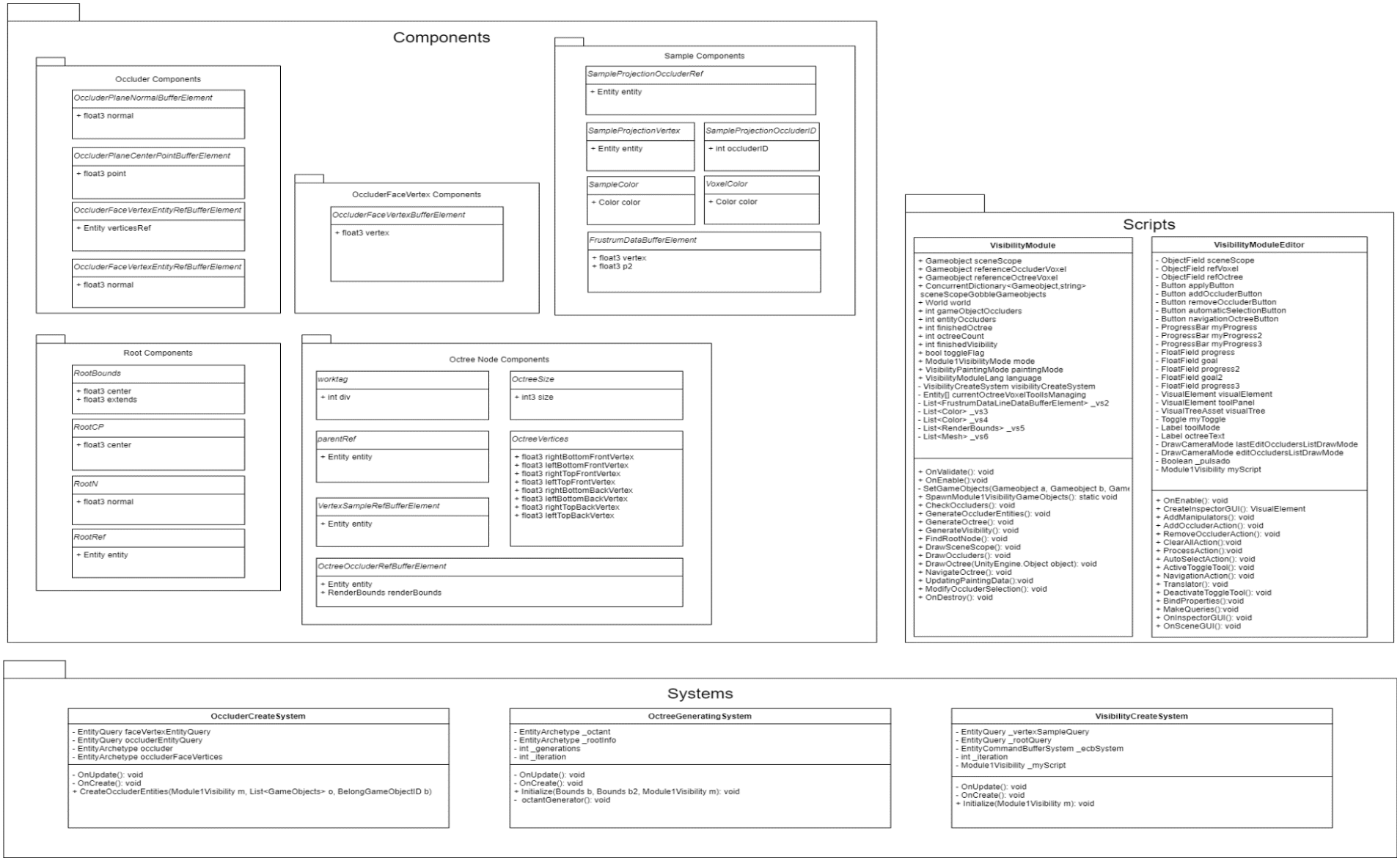

Figura 1.- Diagrama de clases.

DIAGRAMA DE FLUJO

En el siguiente diagrama, Figura 2, se muestra el curso que llevaría el funcionamiento de nuestra aplicación, pasando por cada fase de este y ponderando las consecuencias de cada una de las posibles decisiones.

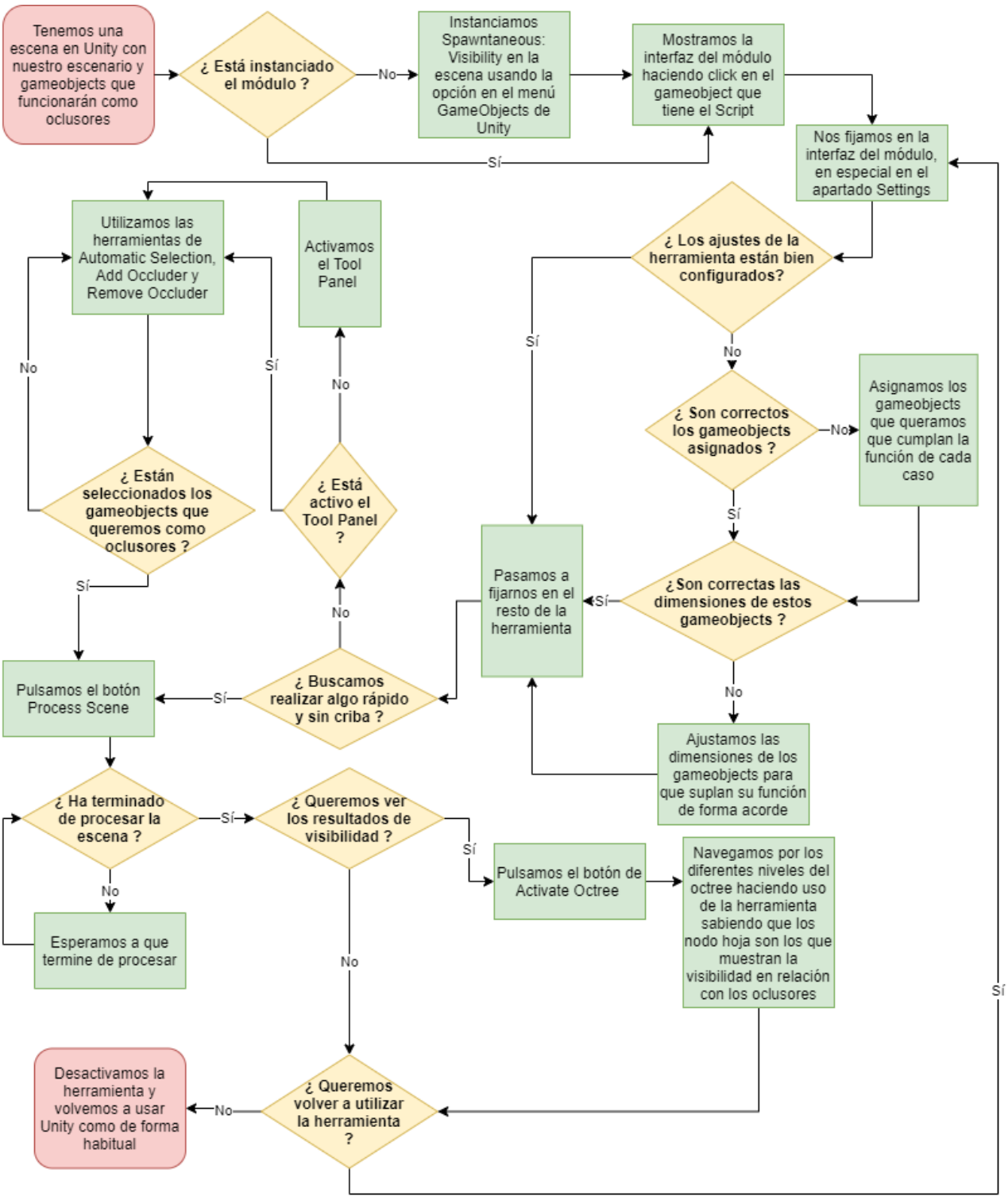

Figura 2.- Diagrama de Flujo.

EXPLICACIÓN DE FUNCIONAMIENTO Y CONFIGURACIÓN

En primer lugar, se debe incluir los archivos de Spawntaneous en el proyecto. Para ello el usuario debe descargar el **Unity Package** [30] y ejecutarlo. El paquete se encarga de colocar los archivos en sus correspondientes rutas.

El uso de nuestra herramienta presenta como prerrequisito una escena con un grupo de GameObjects. Para instanciar el módulo se hace clic en el menú en la pestaña de GameObject y se despliega la opción **"Spawntaneous"**. Dentro de ella ***"Visibility"*** (véase Figura 3).

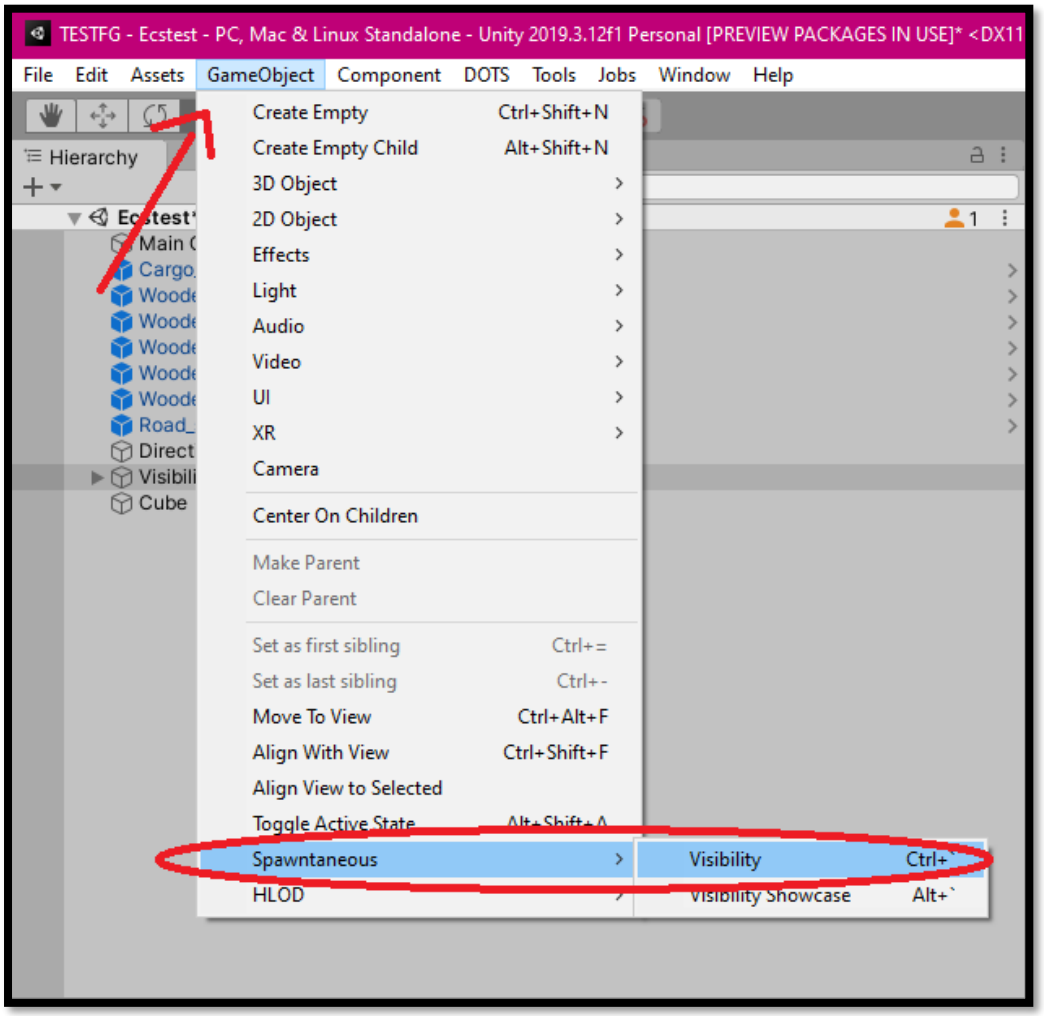

Figura 3.- Menú desplegable GameObject de Unity.

Al hacer clic aparece en la jerarquía un GameObject llamado ***"Visibility Module"***, este guarda en su interior los GameObjects *"Scene Scope"*, *"Octree Reference Voxel"* y *"Occluder Reference Voxel"*.

En la Figura 4 se muestra la interfaz del editor del *"Visibility Module"* en inglés.

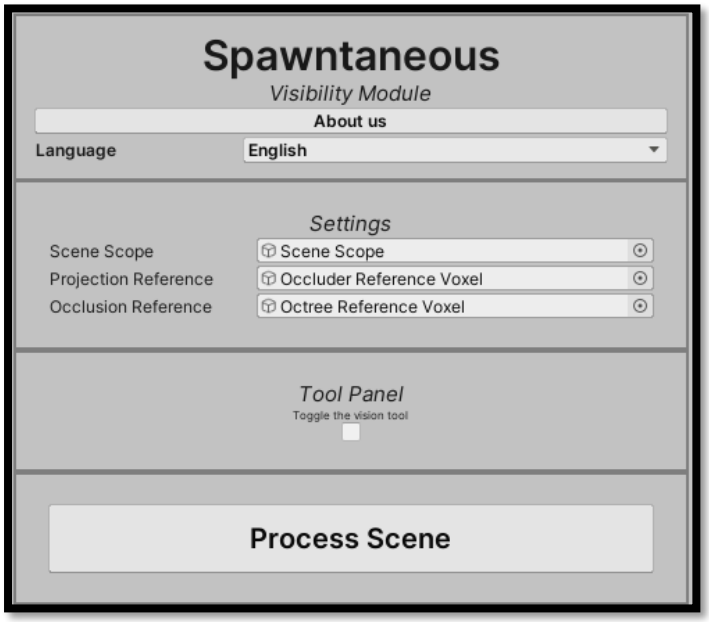

Figura 4.- Interfaz inglesa no desplegada.

Por defecto se utiliza la interfaz inglesa (véase Figura 4), debido a que la mayoría de los usuarios de Unity son angloparlantes. Sin embargo, para esta guía usaremos la versión española de la interfaz (véase Figura 5). Hay que hacer clic en el campo que aparece como ***"Language"*** y seleccionar español.

Figura 5.- Interfaz española no desplegada.

Los tamaños y posiciones de los objetos referenciados en el panel de Ajustes (véase Figura 5) deben ser modificados. Tal modificación debe responder a las necesidades del usuario. Es posible elegir cualquier GameObject para que ejerza estos roles. A continuación, se listará cada uno de los objetos de dicho panel y su función.

- ***Scene Scope:*** Es el volumen contenedor de la escena a procesar. El usuario tiene que ajustar este objeto para que englobe todo el espacio que quiera procesar.

  El diagrama de la Figura 6 sirve para explicar el funcionamiento de este objeto: El *Scene Scope* se ha hecho transparente y subrayado en color azul por razones de claridad. Los GameObjects señalados con una **X** roja (dibujada para esta explicación) **no son tomados en cuenta**. Esto se debe a que todo aquel objeto que quiera ser considerado como oclusor, **debe estar dentro del *Scene Scope* en su totalidad.**

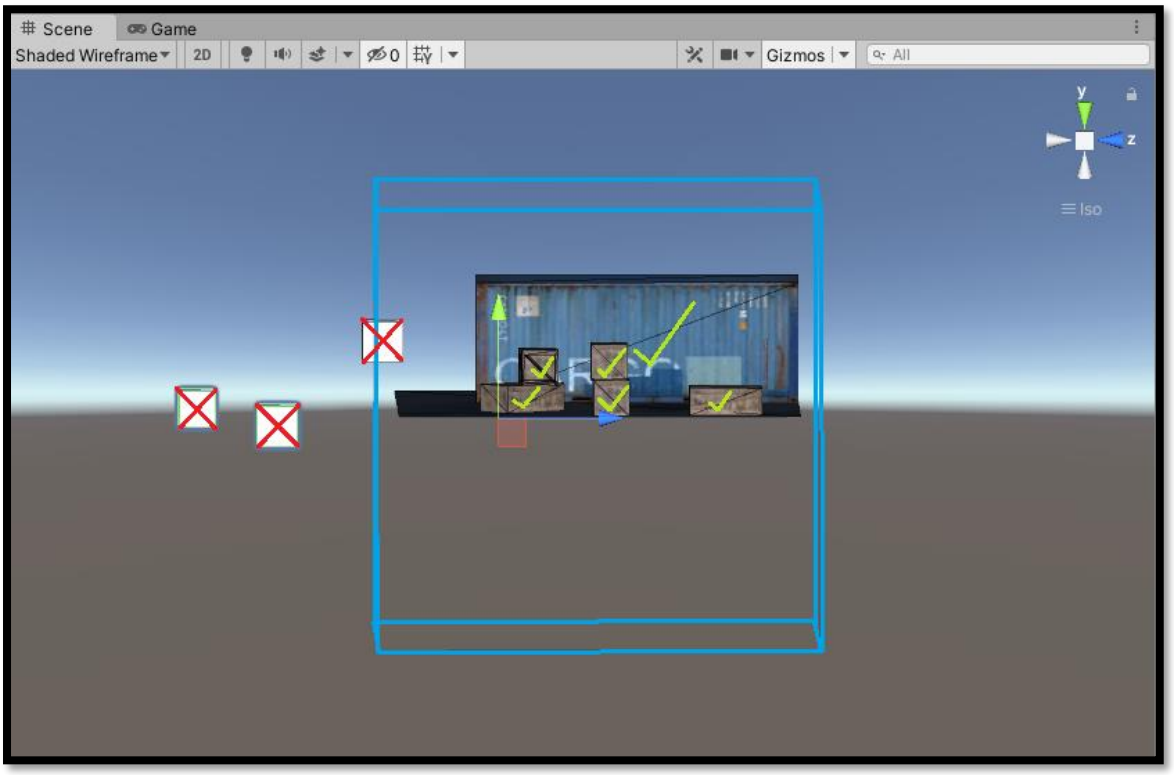

Figura 6.- Diagrama explicativo sobre la colocación de GameObjects oclusores.

- **Referencia de la proyección *(Octree Reference Voxel):*** Se usa para calcular las dimensiones del octante mínimo del octree. Modificar su tamaño también modifica las dimensiones del octante mínimo.

- **Referencia de la oclusión *(Occluder Reference Voxel):*** Se tiene en cuenta para determinar si un GameObject es un oclusor. Solo se emplea para la selección automática.

Una vez se ha ajustado el tamaño y posición de los objetos anteriores, el usuario podría pulsar el botón **"Procesar escena" *(Process Scene)*** y realizar el cálculo de visibilidad de esa escena. Sin embargo, vamos a seguir explorando la interfaz y sus posibilidades.

En la zona derecha, *Tool Panel*, activamos el interruptor del panel (véase Figura 7)

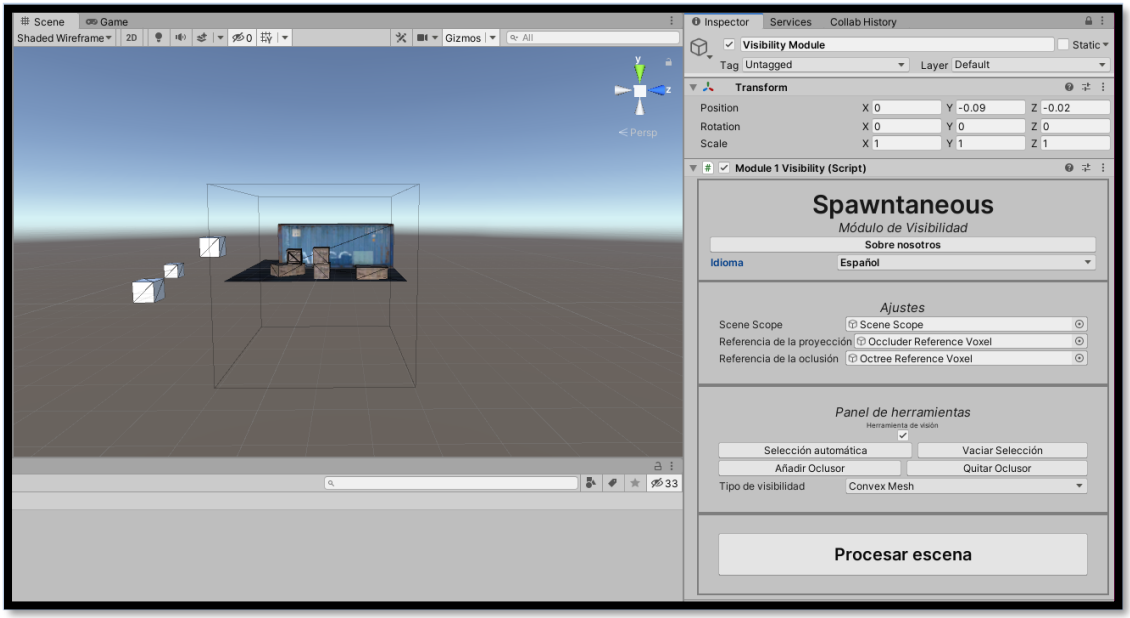

Figura 7.- Interfaz expandida

Se puede ver que la escena ha cambiado su modo de visualización. Ahora se aprecia el *wireframe* de todos los objetos involucrados y en gris se dibuja el *Scene Scope*, que será nuestro volumen inicial del *octree*. Esto también se aprecia en la Figura 7.

En el apartado de **"Panel de Herramientas" *(Tool Panel)***, aparecen una serie de botones anteriormente no visibles:

1. **Selección Automática *(Automatic Selection):*** Este botón realiza una selección de todos los GameObjects de mayor envergadura que el *Occluder Reference Voxel*, pero solo se comprueban aquellos GameObjects situados dentro del *Scene Scope*. Usar este botón descarta la anterior lista de oclusores.

   Los objetos seleccionados lucen encapsulados en una caja roja, tal y como puede verse en la Figura 8.

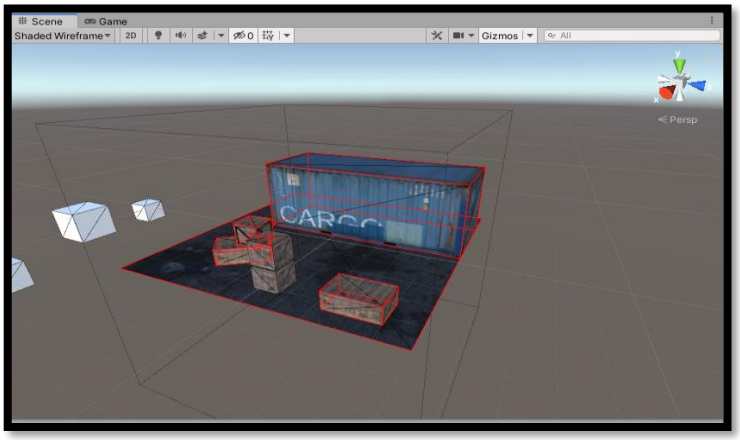

Figura 8.- Señalización de GameObjects seleccionados para el cálculo de la visibilidad.

2. **Vaciar Selección *(Clear Selection):*** La función de este botón es limpiar por completo la lista de los oclusores.

3. **Añadir Oclusor *(Add Occluder):*** Pulsar este botón activa el modo de adición manual de oclusores. Hacer clic en un GameObject, teniendo este modo activo, lo añade a la lista de oclusores.

4. **Quitar Oclusor *(Remove Occluder):*** Pulsar este botón activa el modo de sustracción manual de oclusores. Para eliminar un objeto de la lista de oclusores usando este modo, solo hemos de hacer clic sobre él.

5. **Activar/Desactivar *Octree (Activate/Deactivate Octree):*** Sirve para activar /desactivar la funcionalidad de navegación en el *octree*. Cuando hay un *octree* procesado activo, esta toma un color azul.

Solo es visible cuando se procesa una escena (véase Figura 9).

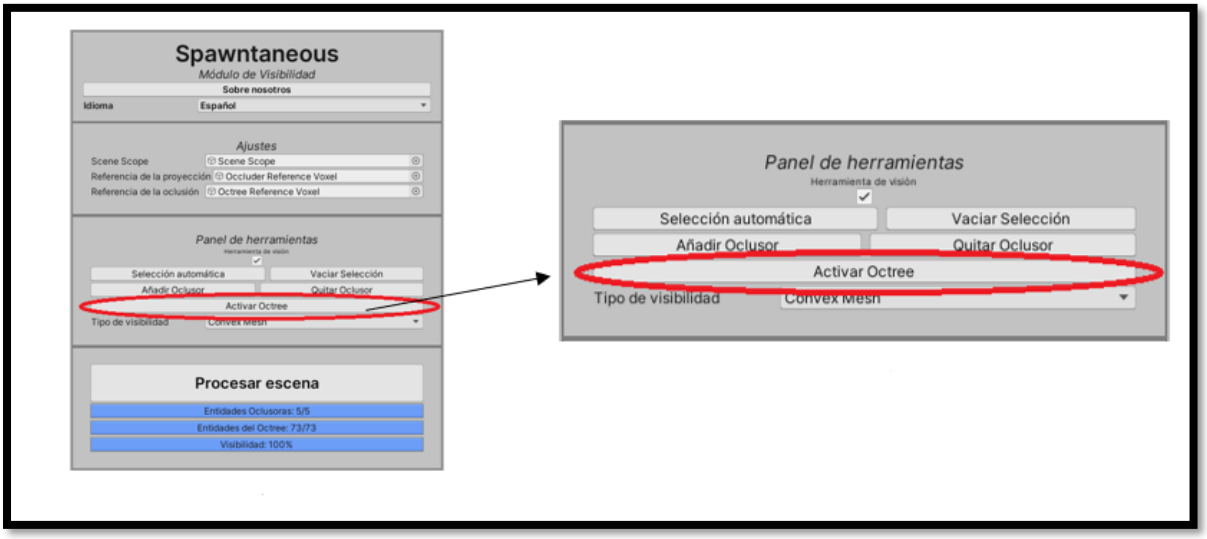

Figura 9.- Botón Activar Octree, vista general y ampliación.

6. **Tipo de visibilidad *(Visibility Type):*** Se puede elegir el tipo de pintado para los resultados del cálculo de la visibilidad:

- ***Lines Sample Color:*** Un color por cada pareja de vértice proyector y oclusor. Sólo se proyectan líneas (véase Figura 10).
- ***Convex Mesh:*** **Experimental.** Se crea una malla por cada volumen dado por cada pareja vértice-oclusor del octante (*Sample*) (véase Figura 11).
- ***Lines Voxel Color:*** Un color por cada oclusor, solo se proyectan líneas (véase Figura 12).

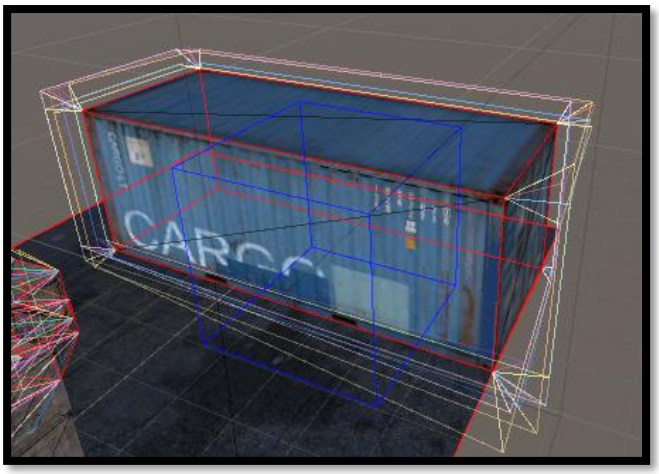

Figura 10.- Representación de visibilidad utilizando "Line Sample Color".

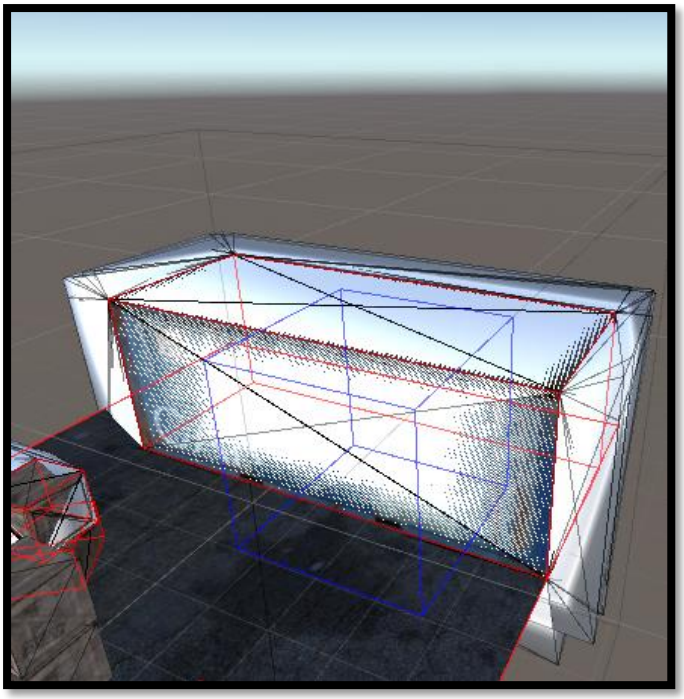

Figura 11.- Representación de visibilidad utilizando "Convex Mesh".

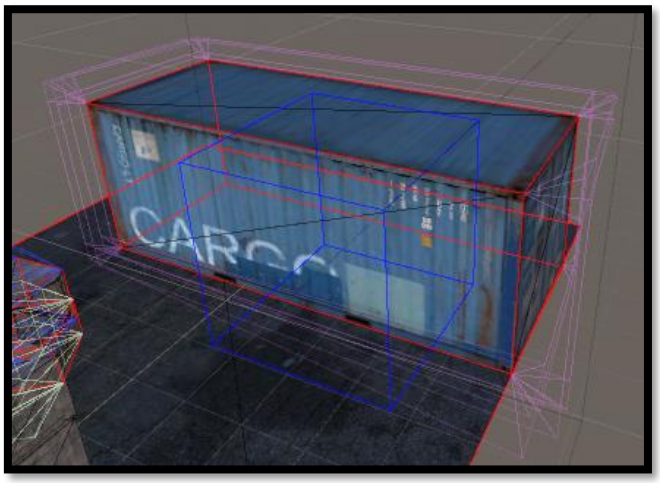

Figura 12.- Representación de visibilidad utilizando "Line Voxel Color".

Una vez se han seleccionado los oclusores deseados, se pulsa el botón “Procesar escena” y se calcula la visibilidad. Si no se ha realizado previamente una selección, esta se realiza de manera automática (véase Figura 13)

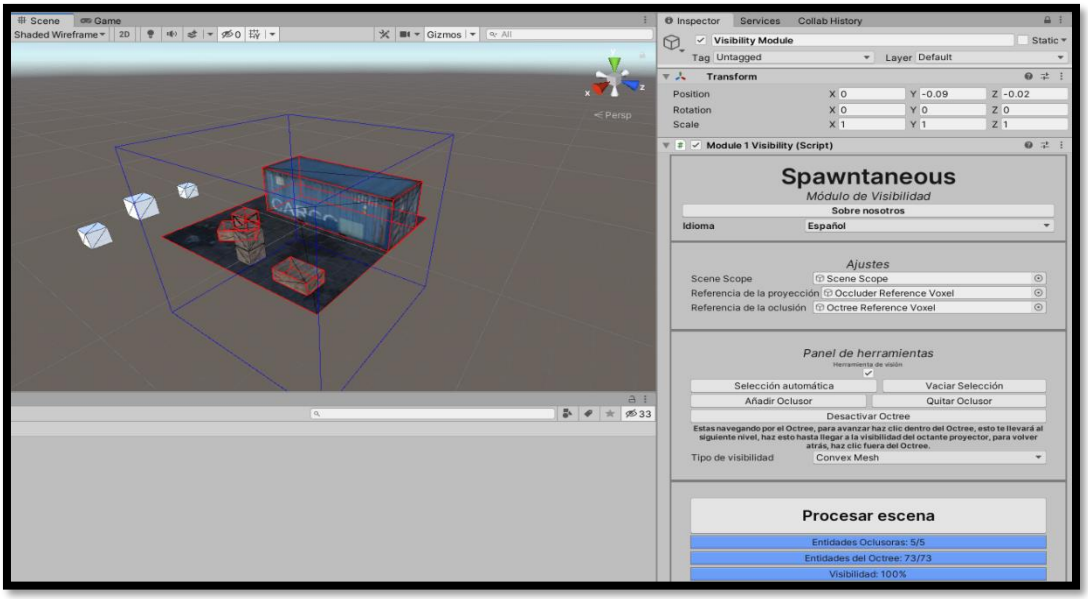

Figura 13.- Oclusores seleccionados de manera automática al procesar la escena.

Tras esto se activa (de no estarlo) el Panel de herramientas y el *octree* toma un color azul, lo que indica que se puede trabajar con él. Como se puede ver en la Figura 13, se ofrecen una serie de instrucciones que enseñan cómo navegar por el *octree*. La forma de navegar por el octree puede resumirse de la siguiente forma:

Para avanzar en profundidad dentro del *octree* procesado hay que hacer clic dentro de los límites azules, véase Figura 14, Figura 15, Figura 16 y Figura 17. Llegados a la máxima profundidad se pintan las proyecciones generadas desde ese octante (véase Figura 18). Estas se corresponden con el terreno no visible desde dentro de ese espacio.

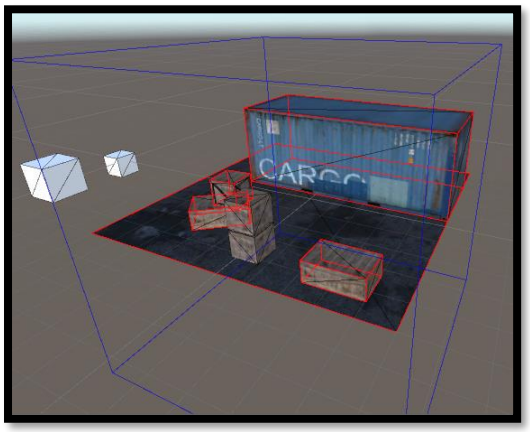

Figura 14.- Representación del Octree. Profundidad 0.

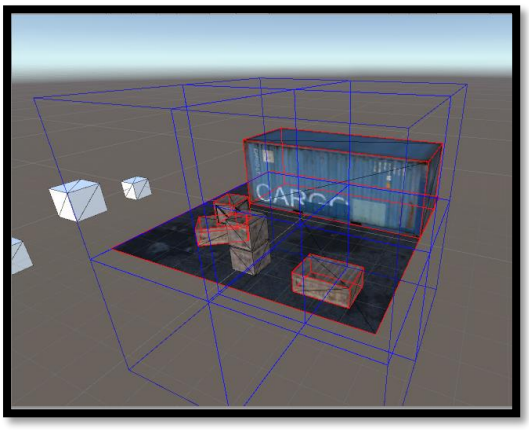

Figura 15.- Representación del Octree. Profundidad 1.

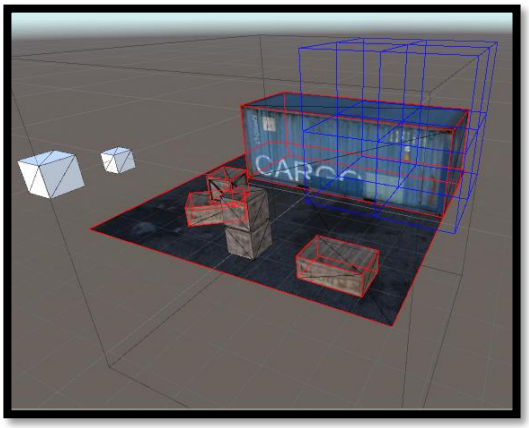

Figura 16.- Representación del Octree. Profundidad 2

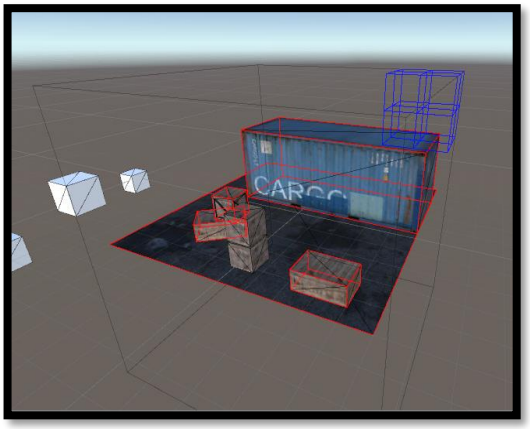

Figura 17.- Representación del Octree. Profundidad 3

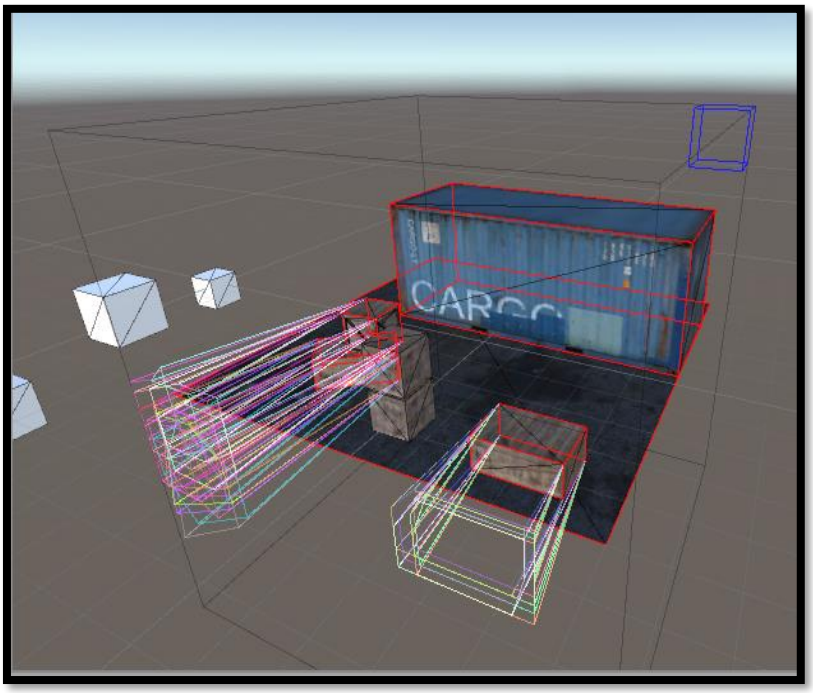

Figura 18.- Representación del Octree. Visibilidad.

Para salir de esta vista y retornar al modo normal hay que apagar el interruptor del panel.

Como nota adicional, el botón “Sobre nosotros” muestra una ventana flotante con un breve mensaje:

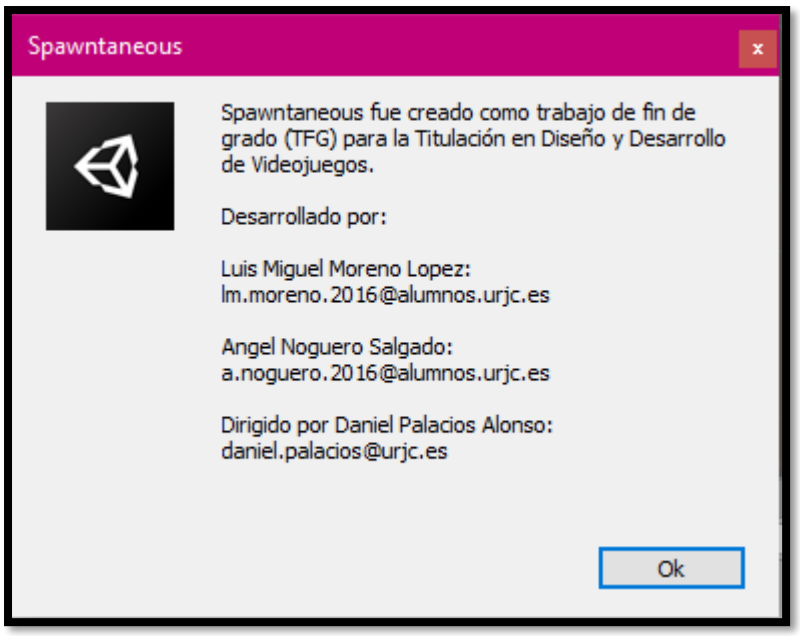

Figura 19.- Sobre Nosotros.

VERSIONES DE DESARROLLO

ANÁLISIS TEÓRICO

Data-Oriented Technology Stack

Elegimos usar esta tecnología por los motivos explicados a continuación:

- Al encontrarse todavía en desarrollo, nos permite investigar sobre una tecnología creciente.
- DOTS, con **ECS**, **Jobs** y **Burst Compiler** da un rendimiento muy superior al convencional MonoBehaviour [31].

DOTS está compuesto por múltiples paquetes, los principales se detallan a continuación:

- **Entity Component System:** separa entre identidad (entidades), datos (componentes) y comportamiento (sistemas) [32].

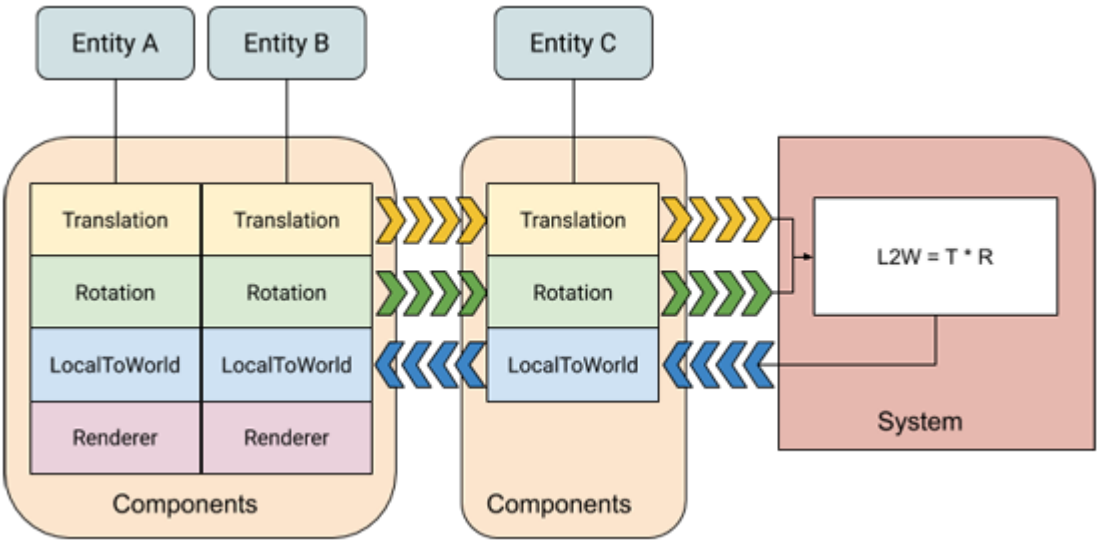

Figura 20.- Esquema del funcionamiento de ECS. Unity Docs [32].

Este diagrama de la documentación de Unity (véase Figura 20) muestra el funcionamiento de ECS con un pequeño ejemplo:

1. Se nos presentan tres entidades A B y C, y un sistema.
2. Cada una de estas entidades tiene una serie de datos, almacenados en los componentes.
3. El sistema modifica la información que contienen los componentes de las entidades.

Las **Entidades** son identificadores que no poseen ni datos ni comportamiento, solo indican qué componentes van juntos [32]. Las entidades de un mundo son gestionadas por el **EntityManager** de este [33].

Los **Componentes** son los datos de la aplicación [34]. Las entidades que poseen los mismos componentes pertenecen al mismo **Arquetipo**, en otras palabras, un arquetipo es tan solo una descripción de los componentes que debe tener una entidad [34]. Cuando las entidades son del mismo arquetipo se colocan en el mismo ***chunk***, que es simplemente un conjunto de memoria [34]. (véase la Figura 21)

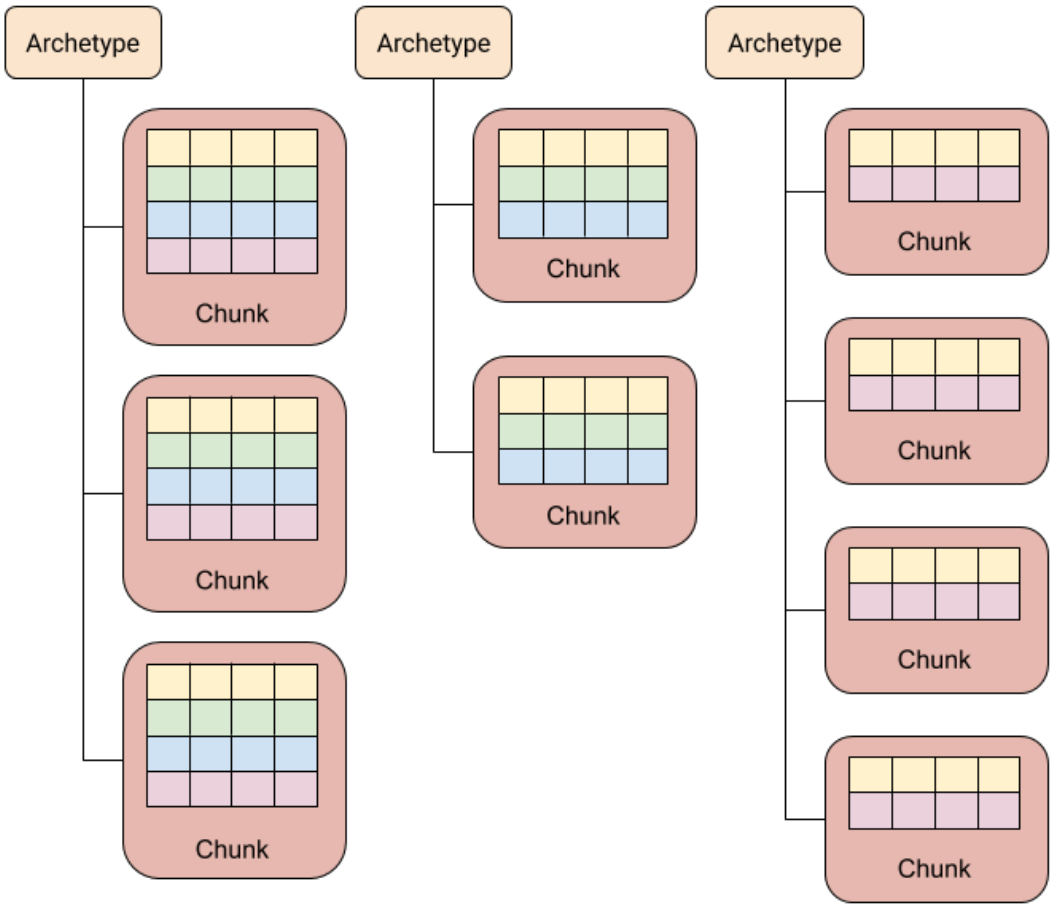

Figura 21.- Esquema de memoria ECS. Unity Docs [34].

Una entidad puede recibir componentes o puede perderlos, cuando esto sucede se produce un cambio estructural. Cambia su arquetipo y cambia su posición en la memoria.

Los **Sistemas** son la base de la lógica de ECS, contienen métodos e instrucciones que modifican los datos (componentes) del programa [35]. (véase Figura 22)

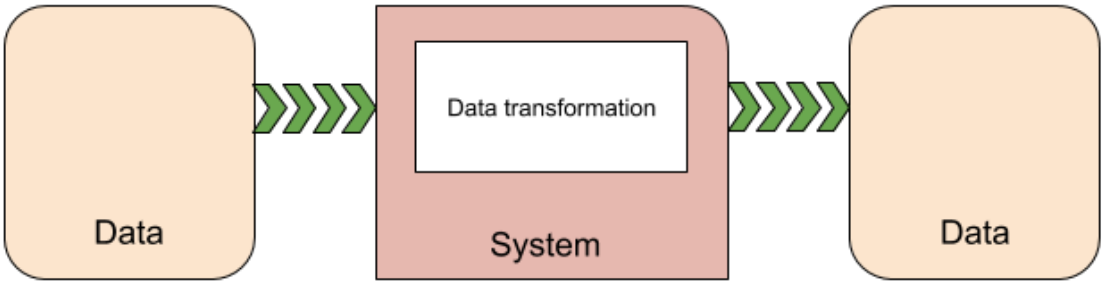

Figura 22.- Esquema de un sistema ECS. Unity Docs [35].

- **Job System:** Este paquete aporta herramientas para el **desarrollo multihilo** (programación concurrente) [36]. El que sea multihilo significa que se ejecutan **varias instrucciones al mismo tiempo**, esto puede ser gestionado de varias formas:
    - En el hilo principal definimos una tarea para ser ejecutada en paralelo. Se crea un hilo adicional que ejecuta la tarea y muere.
    - Emplear un *"pool".* Este permite que varios hilos se queden a que haya una tarea disponible para ser ejecutada. Cuando se da el caso, uno de estos hilos ejecuta tal tarea. Cuando acaba la ejecución, el hilo no es destruido, sino que se devuelve al *pool* donde estará disponible para ejecutar otra tarea [37].

Job System implementa una serie de hilos llamados **workers.** estos se encuentran activos de manera constante ejecutando las tareas (***tasks***) de la **pila de ejecución** del sistema. Cuando un sistema define un *job* lo añade a una pila de tareas, y allí espera a ser ejecutado por un worker [9].

La programación concurrente tiene varios **riesgos**, ya que los hilos comparten **recursos comunes** y **deben ser coordinados**. Existe la posibilidad de que varios hilos deban ejecutar una serie de instrucciones en un orden determinado, pero debido a que se ejecutan a la vez, este orden no está garantizado. Aparecen así errores en la ejecución del programa conocidos como **"condición de carrera"** [38].

También errores como la **posible inconsistencia de datos**, de la cual hay tres posibles vertientes:

- escritura tras lectura
- lectura tras escritura
- escritura tras escritura

En todas ellas, el problema está en que más de un hilo se encuentra realizando operaciones de manera simultánea con la misma información [39].

Finalmente, también existen los llamados **"*deadlock*"** y **"*livelock*"** [40].

Job System implementa una solución automática para todos esos problemas. Incorpora un **sistema de seguridad** que detecta posibles inconsistencias en la ejecución del programa, y cuando esto ocurre se da una copia idéntica del recurso original a cada uno de los hilos involucrados en el posible error. Para que esto funcione, los datos manejados por el *job* tienen que ser **"*Blittable*"** [38], que son todos aquellos tipos de datos que se representan igual en memoria tanto en código gestionado como en código

sin gestionar. El código gestionado es aquel que debe ser manejado por una máquina virtual (como ocurre con **C#**), mientras que el código no gestionado es aquel que se ejecuta directamente en CPU (como ocurre en **C**) [41].

- **Burst Compiler:** ***"****Burst* es un compilador que traduce de *IL/.NET bytecode* a código nativo altamente optimizado usando *LLVM*" [31].

DOTS también cuenta con otros cinco paquetes, aunque su uso no ha sido necesario para nuestra herramienta. Estos son:

- **Unity Physics:** *"Nuevo motor de Física construido con la tecnología de pila de tecnología basada en datos (DOTS)"* [31].

- **Unity NetCode:** *"Ofrece predicción del lado del cliente, servidor acreditado e interpolación"* [31].

- **DSPGraph:** *"Es la base del próximo sistema de audio de pila de tecnología basada en datos (Data-Oriented Technology Stack, DOTS), actualmente en vista previa. Nuestro nuevo motor de audio de nivel bajo funciona con el compilador Burst y es totalmente extensible en C#, lo que permite que los programadores de audio y los desarrolladores de sistemas de audio creen su propio sistema de audio personalizado".* [31].
- **Animación de Unity:** ***"****Ofrece funcionalidades básicas, como la mezcla de animación, IK (Inverse Kinematics), movimiento raíz, capas y máscaras, y está previsto incluir más funciones"* [31].

- **Tiempo de ejecución DOTS:** *"Con Project Tiny, el tiempo de ejecución altamente modular de Unity impulsado por DOTS, próximamente disponible, podrás crear juegos instantáneos, pequeños, ligeros y rápidos."* [31].

REPRESENTACIÓN DEL ESPACIO Y OCCLUSION CULLING

Buscamos hallar una solución que ayude a generar y analizar un sistema de reaparición, para ello se necesita poder manejar algo tan afín a la reaparición como es la representación del espacio. Para llevar esto acabo, se valoran distintas opciones: *Octree* [42], *Navigation Mesh* [43], Partición binaria del espacio [44], etc.

No obstante, más adelante se centra la investigación sobre la visibilidad. En este punto, debemos ver que se ha investigado en dicho campo hasta el momento. No hizo falta llegar a tiempos contemporáneos, ya que en el siglo XX ya se habían desarrollado algoritmos de ***culling.*** Estos trabajan conceptos similares a los que se pretendían manejar.

El ***culling*** es una técnica usada en la generación de gráficos por ordenador que consiste en determinar, mediante una serie de algoritmos, qué polígonos son visibles y cuáles no. Pretende ahorrar cálculo computacional descartando polígonos que no son visibles [45].

La gran mayoría de algoritmos de *occlusion culling* se recogen en la siguiente tabla [4] (véase Tabla 3), y han sido clasificados en base a sus atributos:

	Type	**Accuracy**	**Preprocessing**	**Online processing**	**Aliasing artifacts**	**Dimension**	**Fusion**	**Scene Type**	**Temporal Coher.**
Aspect Graph	Cell-polygons	Exact	YES (major)	NO	NO	3D	YES	Polygon soup	---
Backface	Point-polygon	Overest.	NO	NO	NO	3D	NO	Polygon soup	NO
View-frustum	Point-cell	Overest.	YES (minor)	YES	NO	3D	NO	Polygon soup	NO
Hierarchical Z- buffer	Point-cell	Overest.	YES (minor)	YES	YES	3D	YES	Renderable objects	YES
Hierarchical Occlusion Maps	Point- cell	Overest.	YES	YES	YES	3D	YES	Renderable objects	YES (occluder-selection)
Visibility Octree	Cell-cell	Overest.	YES (major)	NO	NO	3D	NO	Object List (interior defined)	---
Directional Discretized Occluders	Cell-cell	Overest.	YES	YES	NO	3D	YES	Connected polygons	NO
Densely Occluded Scenes	Cell-cell	Overest.	YES	NO	NO	2.5D	NO	Convex polygons (city block)	---
Cull Maps	Cell-cell	Overest.	YES	NO	NO	2.5D	YES	City block polygons	---
Blocker Extension	Cell-cell	Overest.	YES	NO	NO	2.5D	YES	City block polygons	---
Shadow Culling	Point-cell	Overest.	YES	YES	NO	3D	NO	Polygon soup	NO
Portals (Luebke)	Point- cell	Overest.	NO	YES	NO	2.5D	NO	Axis- aligned cells	NO

Portals (Teller)	Cell-cell (+point- cell)	Overest.	YES (major)	NO (+YES)	NO	2.5D	NO	Axis-aligned cells	YES
RT Occlusion - Large Occluders	Point- cell	Overest.	YES	YES	NO	3D	YES	Convex-connected polygons	YES
Extended Projections	Cell- object	Overest.	YES	NO	NO	3D	YES	Convex-concave polygon objects	--
Virtual Occluders	Cell- object	Overest. (2D), Approx (2.5D)	YES	YES	NO	2.5D	YES	Polygon objects	---
Occlusion Trees	Point- cell	Overest.	YES	YES	NO	3D	YES-*OBSP* NO-*MOBSP*	Convex polygons	YES
Linearized Aspect Graph	Point- cell	Overest.	YES	YES	NO	3D	NO	Convex connected polygons	YES
LOD Occluders	Point-cell	Approx.	YES	YES	NO	3D	YES	Polygon-soups	NO
Platinga	Cell-object	Approx.	YES	NO	NO	2.5D	NO	Polygon-soups	---
Hierarchical Back-Face	Cell-cluster	Approx.	YES	YES	NO	3D	NO	Polygon-soups	---
OpenGL Culling	Point-cell	Approx.	YES	YES	NO	3D	YES	Renderable-objects	NO
PLP	Point-cell	Approx.	YES	YES	YES	3D	NO	Polygon soups	NO
cPLP	Point-cell	Overest.	YES	YES	NO	3D	YES	Polygon soups	NO
Hoops	Cell-cell	Overest.	YES	NO	NO	3D	YES	Object list	---
Hierarchical Terrain Vis.	Cell-quad	Overest.	YES (major)	YES	NO	2.5D	---	Terrain Quads	NO
Selective Refinement	Cell- object	Overest.	YES (major)	YES	NO	3D	YES	Convex-concave polygons	NO
Lazy Occlusion Grid	Point- object	Overest.	NO (minor)	YES	NO	3D	YES (image space)	Renderable objects	NO

Tabla 3.- Métodos de culling. Occlusion Culling Algorithms: A Comprehensive Survey [4].

Posteriormente comenzamos a filtrar estos métodos para encontrar aquel que fuera más afín a nuestro proyecto. Para ello se siguió el siguiente razonamiento:

Queremos saber si un jugador puede ver a un agente cuando este reaparece. Lo que supone realizar los cálculos de visibilidad cada vez que haya una reaparición, es decir, tenemos

que hacer **varias veces** el cálculo de la visibilidad. Debido a que esto sería muy costoso, propusimos una aproximación totalmente **preprocesada**. Con una solución así, puede interpretarse de una sola vez toda la escena. Gracias a esto, podemos conocer de antemano la visibilidad de cualquier reaparición.

Lo siguiente que consideramos fue que, debido a que los escenarios de videojuegos son cada vez más amplios, necesitaríamos soluciones apropiadas para grandes tamaños. La columna *"Type"* indica que tipo de representación utiliza cada método; debido a las necesidades de nuestro problema, lo mejor era utilizar la opción **celda a celda.**

Finalmente, se tiene en cuenta que el nivel objetivo será **tridimensional,** lo que descarta todos aquellos que no cumplan con ese requisito.

Tras la revisión detallada, nos quedan cuatro posibilidades:

- *Aspect Graph*
- *Visibility Octree*
- *Extended Projections*
- *Hoops*

Analizamos en profundidad cada uno de ellos y acabamos decantándonos por aquel que se acercaba más a nuestras necesidades, el ***Visibility octree.*** Este método utiliza una estructura para guardar el espacio que deriva de un ***octree***. Un *octree* [42] es una estructura de datos en la que un "nodo" tiene un conjunto de datos, y este "nodo" se divide en exactamente ocho nodos hijo. Suele ser usada para la representación y división de espacios tridimensionales. Su funcionamiento es el siguiente:

Un volumen inicial representa el espacio a ser trabajado, y este espacio se divide en ocho "octantes" volumétricos, cada uno de esos ocho octantes se subdivide en otros ocho etc. (véase Figura 23)

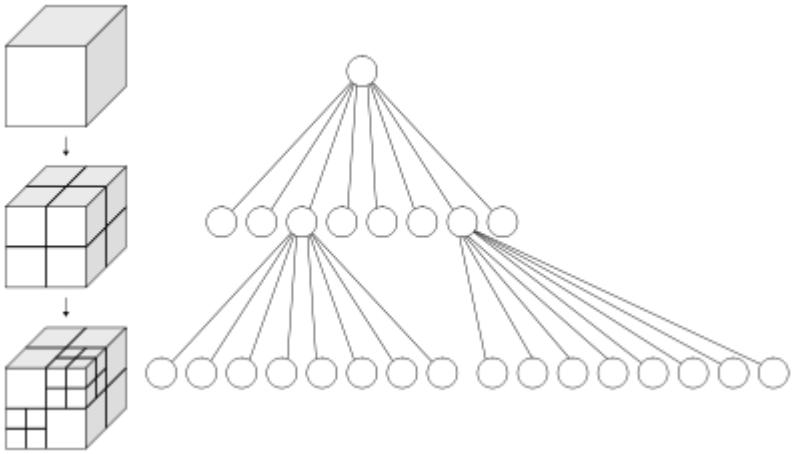

Figura 23.- Esquema del funcionamiento de un octree. Wikipedia [42].

Esta estructura fue utilizada por primera vez en gráficos 3D por ***Donald Meagher*** en **1980** ***"Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer"*** [42].

VISIBILITY OCTREE

El ***Visibility octree*** [5] fue descrito por C. Saona-Vázquez, I. Navazo, P. Brunet. Se presenta como una estructura de datos preparada para acelerar la navegación en escenas muy complejas desde un punto de vista gráfico. Es un *"conservative visibility algorithm"* que computa y almacena jerárquicamente la estructura en un estado preprocesado del programa.

Ellos describen el **problema que solucionan** de la siguiente forma:

"The problem can be specified as follows: given a set P of static polyhedral objects and a dynamic viewpoint O, we want to compute the set V(O) of visible polygons from every feasible O position. This computation must be done quickly, in constant or at most logarithmic time.

The ideal algorithm would take the desired frame rate as an input and would be able to guarantee it."

Destaca de su investigación el siguiente párrafo. En nuestro caso, nos hizo reflexionar sobre la oportunidad de conseguir una **aproximación suficientemente buena como para ser aceptable**.

"Research in the field of visibility computation is characterized by a relaxation of the concept of visibility. In this redefinition, the set of visible polygons from any viewpoint is a superset of the set of real visible polygons. The algorithm presented in this paper pertains to the family of weak visibility algorithms."

Así pues, acaban por definir el algoritmo de manera más extensa:

"The visibility octree is an adaptive data structure that stores potentially visible sets at its terminal nodes. Unlike uniform grid structures, its size depends on the nature of

the scene. The octree is computed in a top-down fashion.

To compute a node, the algorithm first selects some convex occluders. Afterwards, it culls the scene and computes the set of potentially visible polygons from the node region. If the set is not small enough the node is subdivided if further occlusion can be expected."

Matemáticamente se justifica con los siguientes preceptos [5]:

Paso 1:

Definition 1. *The shadow from a point p of a set* $A \in R^n$ *is*

$$S(p, A) = \{q \in R^n | \overline{pq} \bigcap A \neq \theta \wedge q \notin A\}$$

where $\overline{pq}$ *is the segment between p and q.*

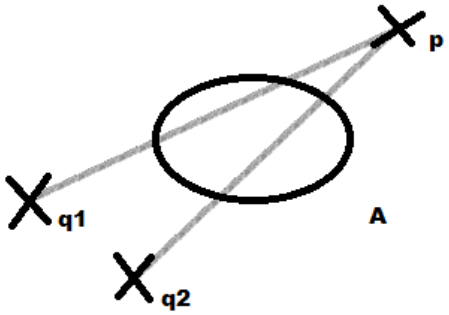

Figura 24.- Sombra de un conjunto A desde un punto p.

En la definición se describe cómo se calcula **la sombra desde un punto *p* para un conjunto *A*:**

La sombra es cualquier punto ***q*** tal que el segmento formado por este hipotético punto y el punto ***p*** corte el conjunto ***A***, el punto ***q*** no puede pertenecer al conjunto ***A*** (véase Figura 24).

Paso 2:

Se resuelve **cómo calcular la sombra desde un conjunto de puntos P para un conjunto A**, demostrando matemáticamente que el resultado es igual a realizar la **intersección de las sombras generadas desde los puntos que conforman P** (véase Figura 25).

Definition 2. *The shadow from a set P of a set* $A \in R^n$ *is*

$$S(P, A) = \{q \in R^n | \forall p \in P : \overline{pq} \bigcap A \neq \theta \wedge q \notin A\} = \bigcap_{p \in P} S(p, A)$$

where $\overline{pq}$ *is the segment between p and q*

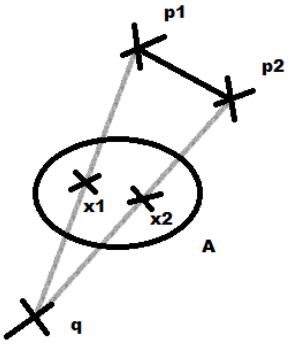

Figura 25.- Sombra de un conjunto A desde un conjunto de puntos P.

Paso 3:

En el paso previo a la demostración final que permite dar validez a los resultados del ***Visibility octree*** y se prepara el camino con el siguiente lema

Lemma 1. *If* $A \in R^n$ *is a convex set and there is a hyper-plane that separates A from the segment,* $s = \overline{p1p2}$ *then*

$$S(s, A) = S(p1, A) \bigcap S(p2, A)$$

Se debe mencionar como algo importante que el conjunto convexo y el segmento están separados por un plano obligatoriamente.

Paso 4:

Se define el teorema clave para que, como se manifiesta anteriormente, *Visibility octree* y *Spawntaneous: Visibility Module* funcionen.

Theorem 1. *If* $A \in R^n$ *is a convex set and there is a plane that separates A from the closed and bounded polyhedron* $C \subseteq R^3$

$$S(C,\ A) = \bigcap_{i=1}^{n} S(p_i\ , A)$$

where p_i *are the n vertices of polyhedron C.*

Se plantea como calcular la **sombra producida desde cualquier punto dentro de un poliedro** cerrado *C* y delimitado para un conjunto convexo *A*. Esta equivale a **la intersección de todas las sombras generadas desde cada vértice de este poliedro C para el mismo conjunto convexo A**. Se permite por tanto abarcar la visibilidad para grandes volúmenes simplemente calculando esa misma visibilidad para los vértices que delimitan ese volumen, reduciendo así el **número de cálculos.** Esto nos proporciona una solución aceptable y con capacidad de ser preprocesada.

Una vez revisada la parte matemática, se revisa la parte algorítmica en sí, podemos destacar los siguientes puntos, de los cuales hemos extraído información para nuestro proyecto:

- La selección de oclusores se debe realizar en base a ciertos criterios como la convexidad o el tamaño.

- Utilizar un *octree* como estructura de datos presupone una mejora en las búsquedas de datos.

- Por último, el concepto de *"Sample"* se utiliza para representar resultados intermedios de la visibilidad.

PRUEBA DE CONCEPTO

Realizamos un **prototipo** para poner a prueba el concepto de sombra que aparece en el *Visibility octree*, esto lo hacemos con los siguientes objetivos:

- Se quiere constatar las lecciones aprendidas del *Visibility octree* de una forma práctica.
- Se pretende obtener **una idea real y tangible** de lo que es el **módulo de visibilidad** antes de embarcarse en su desarrollo.

La **prueba de concepto** (véase Figura 26) se lleva a cabo en una escena aparte, en ella **se sitúan dos cubos**. Uno realiza la función de **conjunto convexo** y el otro de **poliedro cerrado y delimitado**; ambos se encuentran **separados por un plano en el espacio**.

Con estas premisas se hace posible valorar el teorema que determina **la sombra producida desde un poliedro por un conjunto convexo.**

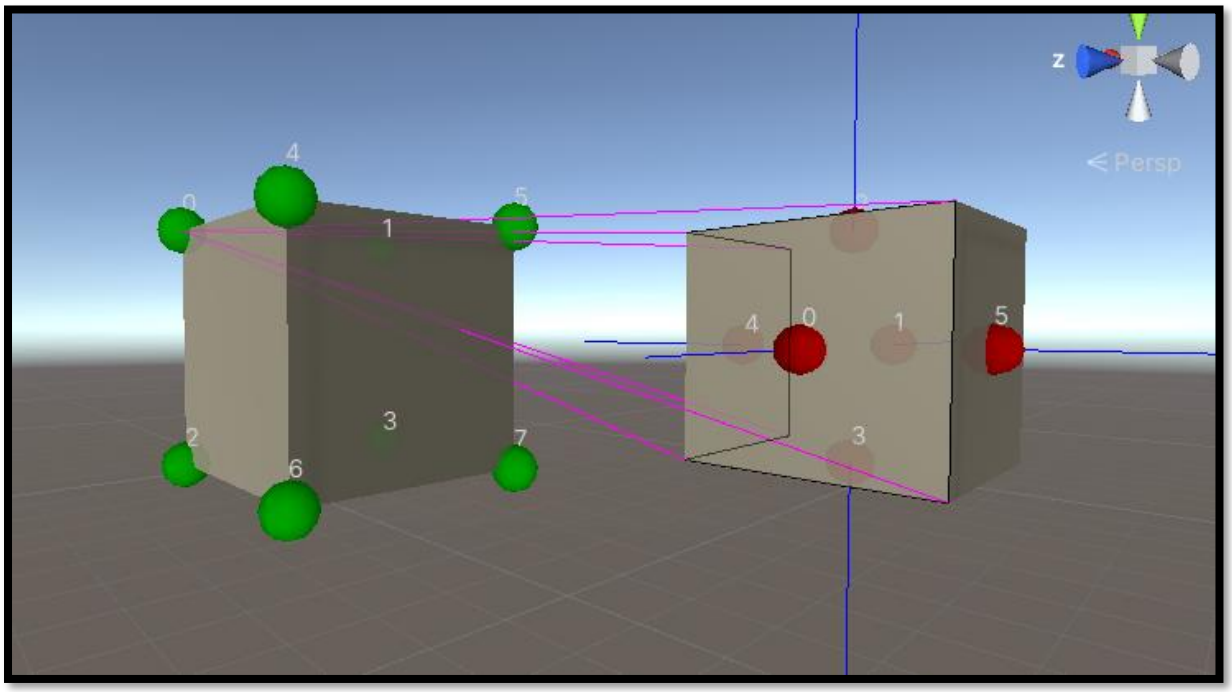

Figura 26.- Prueba de concepto de visibilidad.

En la imagen superior, Figura 26, se muestra una representación gráfica de la escena, en ella, se aprecia un cubo cuyos vértices están marcados por esferas verdes. Este cubo simboliza el poliedro. Por otro lado, el cubo con esferas rojas en los puntos centrales de las caras constituye el conjunto convexo.

La prueba se basa en las explicaciones pormenorizadas de **cómo calcular el *shadow frusta*[1] de un oclusor convexo O desde un punto p**

[1] Shadow frustum (pl. frusta): subespacio generado por la proyección resultante desde un punto por un oclusor y que se encuentra delimitado por un plano. No es visible desde el punto desde el que se proyecta.

Una vez se realizan todos los pasos del algoritmo, se obtienen unos resultados. Estos aparecen en las siguientes imágenes: Figura 27, Figura 28 y Figura 29.

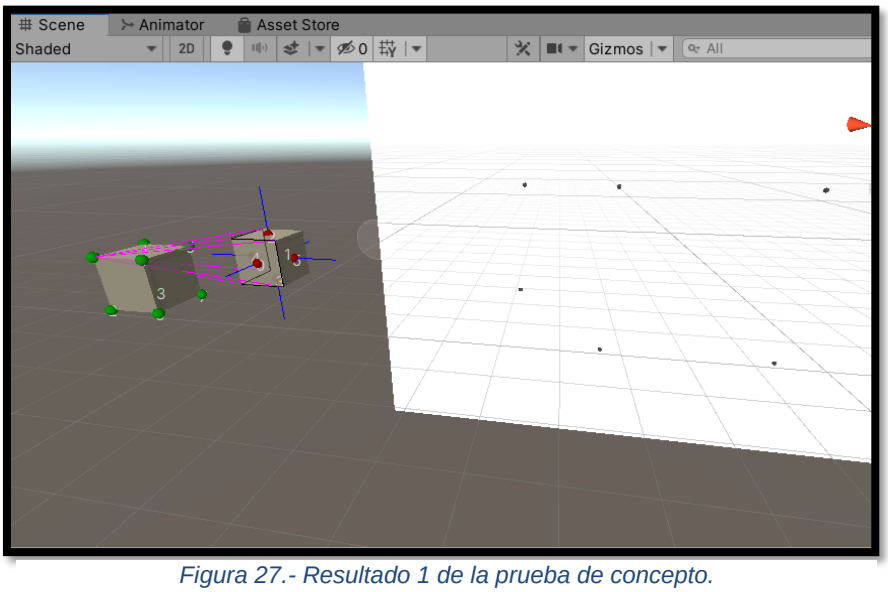

Figura 27.- Resultado 1 de la prueba de concepto.

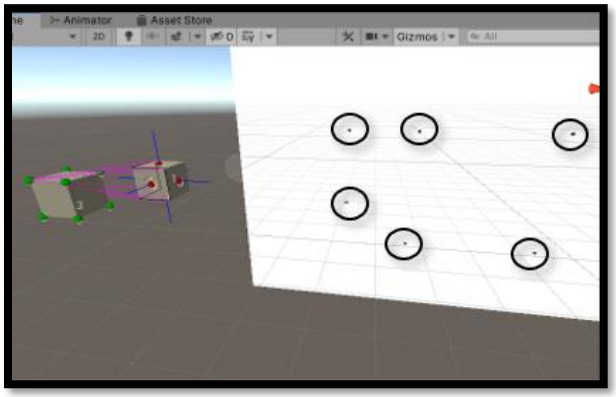

Figura 28.- Resultado 2 prueba la de concepto.

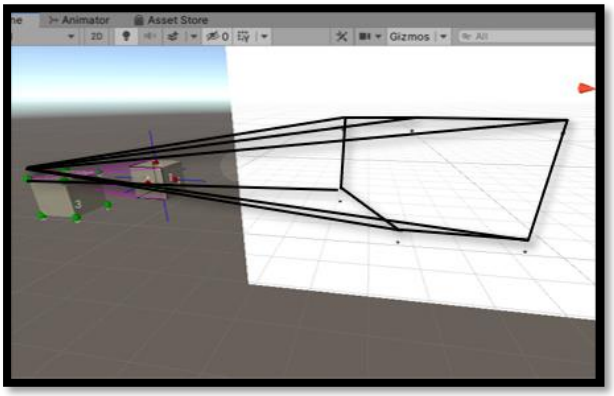

Figura 29. - Resultado 3 de la prueba de concepto

Estos tres resultados aportan la siguiente información:

1. Primer resultado, Figura 27, no presenta ninguna modificación, es el producto de la prueba.
2. Segundo resultado, Figura 28, se han marcado los puntos de la intersección de las rectas sombra con un plano en el espacio.
3. Tercer resultado, Figura 29, se ha dibujado la representación de la sombra generada por el conjunto convexo desde un vértice del poliedro. Esta representación aparece en forma de *frustum* y para ello se han unido los puntos anteriores entre ellos y se han dibujado las rectas sombra.
   Este supone el resultado más importante ya que encarna de forma más exacta la sombra producida.

Si la misma operación se realiza para cada vértice y se hace la intersección de los *frusta,* se consigue **la sombra que genera el oclusor desde ese poliedro.** Esto es lo que explica el teorema.

Si realizamos esto desde todos aquellos poliedros que necesitemos para recubrir la escena (de un tamaño determinado) y hacia todos los oclusores, **podemos calcular la visibilidad de toda una escena.**

OCLUSORES

CONVERSIÓN DE GAMEOBJECTS A ENTITIES

Primera Aproximación: métodos de *Authoring*

Como deseamos utilizar la tecnología Data-Oriented Technology Stack (DOTS) [6], tenemos que **transformar** los GameObjects utilizados por el motor Unity a entidades de ECS, ya que la mayoría de usuarios sigue utilizando un diseño tradicional. En especial, se pretende transformar aquellos con el rol de oclusores en el proceso de visibilidad. **GameObject** es el concepto más importante del motor Unity, representa cualquier objeto dentro del juego desde personajes, objetos coleccionables, cámaras, luces, etc.

Las entidades **(*Entities*)** son uno de los tres conceptos más importantes de la arquitectura Entity Component System (ECS) [11]. Se refieren a la representación de algo y, por tanto, se puede llegar a decir que las entidades son los GameObjects del patrón de arquitectura ECS, de forma matizada.

En un primer momento se investigan varios métodos de conversión conocidos como **"*Authoring*"** [46], se valora utilizar dos de estos métodos:

- Implementar la interfaz ***IConvertGameObjectToEntity***
- ***Subscenes***.

Se realizan varias pruebas con ellos y se acaba optando por aplicar la solución ***Subscenes***.

Ahondando en el método de *Subscenes*, surgen ciertos inconvenientes. Este método se basa en añadir una escena como parte de otra. Esta escena es la encargada de convertir los GameObjects que contenga en entidades. Para ello hace uso de un componente Subscene.

Las subescenas deben ser almacenadas, lo que nos obliga a **gestionar directorios.** También nos vemos obligados a **eliminar y crear la subescena constantemente,** esto es causado por el hecho de tener que procesar distintos oclusores cada vez**.**

Se le suma a esto un constante trasiego de llamadas para activar, marcar como editado o desactivar las escenas.

Por último, el uso de las subescenas esta ideado para el ***Play Mode***. Para adaptarlas al Editor debemos recurrir a **DOTS *Live Conversion*** [47].

Segunda Aproximación: Entidades Específicas

Más adelante en el desarrollo y debido a los problemas de las subescenas, se inicia un replanteamiento de la estrategia de **conversión de GameObject a Entidad**. En estos momentos las entidades y componentes generados por el método de subescenas no proporcionan la información adecuada para calcular la visibilidad. Muchos de los componentes que se obtienen son irrelevantes y son dependientes directamente del GameObject**.**

En definitiva, lo que se quiere realmente es obtener cierta información muy concreta de los GameObjects, **no nos interesa** una conversión **real y fidedigna** del objeto.

Los GameObjects guardan su información en componentes, para calcular la visibilidad se hace imprescindible la información proporcionada por el **componente MeshFilter** [48] **y Transform** [49].

Se gesta una **nueva forma de trabajo** para la obtención de entidades oclusoras, esta basa su metodología en la **creación de entidades propias con componentes propios.** Estos componentes almacenan la información realmente útil para nuestro programa

SELECCIÓN DE OCLUSORES

Automatic Selection Primitivo

En un primer momento, la única forma disponible para obtener una serie de GameObjects para el rol de oclusores es a través de un primitivo método ***checkOccluders()***. Esta función solo toma aquellos GameObjects dentro de los límites (Bounds) del GameObject *Scene Scope*.

Selection Tool

Considerando la intencionalidad del programa, es un error no dar al usuario mayor control sobre qué GameObjects ejercen de oclusores. Surge así la necesidad de una herramienta que permita al usuario elegir cuales son los oclusores. La herramienta se cimenta entonces en el evento de selección de Unity, y utiliza una serie de técnicas de exclusión e inclusión comparando la selección actual con la anterior.

En ese momento aún es una herramienta tosca y con fines de desarrollo. Muestra de ello es que se requiere mantener pulsado la tecla **control** mientras se hace clic en un GameObject para que empiece o deje de formar parte de la lista de oclusores.

La aplicación debe mostrar de manera clara cuando está activada, para esto se opta por cambiar la visualización de la escena cuando esta se activa y volver a deshabilitarlo cuando se desactiva.

Por último, también se decide pintar un cubo rojo con los Handles [50] en la posición donde se encuentra un oclusor para mostrar un indicador visual al usuario.

Nuevas Restricciones al Automatic Selection

Según el *artículo* de ***Visibility octree***. El primer paso para que su solución sea aceptable es obtener una lista de oclusores que sean óptimos. En particular ellos asumen que los oclusores son óptimos si cumplen las siguientes limitaciones:

- Se baraja solo aquellos oclusores que sean convexos
- Los oclusores tienen que ser suficientemente grandes como para generar oclusión. Se toma como medidas objetivas áreas y volúmenes. Después se compara con un elemento válido, en particular el octante más pequeño producido por el *octree*.

DEFINIENDO ENTIDADES OCLUSORAS

Primera Aproximación: Nuevos Conceptos y Primer Arquetipo

Para la primera aproximación se empieza trabajando con conceptos próximos al de entidad. Aparecen dos conceptos claves en ECS, por un lado, la clase **World** y por otro, la clase **EntityManager**. Un objeto **World** [51] contiene tanto un EntityManager como un conjunto de ComponentSystems. El EntityManager se encarga de gestionar todas las entidades que se hallan en el mismo, y el conjunto de ComponentSystem otorga funcionalidad al ser los encargados de modificar la información de los componentes.

Optamos por crear un **mundo (World) exclusivo para el módulo**:

World w = new World("VisibilityModule")

Después, se aboga por definir un **arquetipo** de oclusor para que estos se hallen próximos en memoria. Gracias a la conversión de oclusores, se conocen ya gran cantidad de componentes que forman el arquetipo de oclusor.

Las entidades de los oclusores se crean en el **hilo principal**, y se generan tantas como GameObjects hay en la lista de oclusores. Se crean en el hilo principal debido que, crear entidades en un hilo adicional solo supone una necesidad de sincronización.

A continuación, se introduce el **primer sistema de la visibilidad, el *OccluderCreateSystem*.** La estrategia para obtener entidades oclusoras se convierte en la siguiente: las entidades se crean en el hilo principal, pero se dejan vacías. Es decir, solo se reserva la memoria, pero no se guarda ningún dato.

El sistema recibe una lista de Bounds y otra de matrices localToWorldMatrix. Usando el tipo de trabajos en paralelo de DOTS ***Entitlies.ForEach*,** se recorren las entidades oclusoras y **se rellenan** con los datos de las listas de *Bounds* y localToWorldMatrix*.*

Segunda aproximación: Elegir bien los componentes del arquetipo

Se hace necesario elegir correctamente los componentes que guardan la información de un oclusor, hasta entonces, la mayoría de los componentes salen de aquellos generados al **usar** las **subescenas**. Este el caso del componente **RenderBounds** y del componente **LocalToWorld**.

Las operaciones que se realizan para el cálculo de la visibilidad marcan el devenir de nuevos componentes. El factor clave de la visibilidad son las caras de los poliedros, dichos poliedros se obtienen a partir de los “Bounds" de las mallas poligonales (*Mesh*) de los oclusores. Bounds es una estructura de Unity y consta de un punto central y un vector que representa la dimensión de cada uno de sus ejes. Combinado con la matriz de LocalToWorld permiten representar rotaciones, escalados, etc.

Como se dice anteriormente, las caras o *faces* de estos poliedros toman un importante rol. Se opta por crear una serie de componentes que almacenen la información de las caras necesaria para calcular la visibilidad.

Los componentes que se crean son:

- ***OccluderPlaneCenterPointBufferElement***
- ***OccluderPlaneNormalBufferElement***
- ***OccluderFaceVertexEntityRefBufferElement***

Todos ellos son parte de una lista o *buffer*, cada elemento guarda información de una cara y así, en su totalidad, guardan la información de todas las caras. En los dos primeros componentes, se almacenan la información del **punto central** y de la **normal** de la cara de forma respectiva.

OccluderFaceVertexEntityRefBufferElement es un componente diseñado para guardar los vértices de una determinada cara. Sin embargo, debido a restricciones de ECS, esto no puede hacerse forma directa. ECS no permite que un componente guarde listas de un valor, en su lugar, debe guardar componentes **IBufferElementData.** Estos permiten guardar una lista de varios componentes del mismo tipo. Si ese Buffer ya guarda una lista con las caras, ¿cómo podría guardar cada cara una lista de sus vértices? Necesitábamos una lista de listas, y no existe equivalente en ECS.

La solución a este problema pasa por el diseño de una nueva entidad llamada ***FaceVertex***. Cada unidad del primer DynamicBuffer guarda una referencia a una entidad *FaceVertex*, y esta entidad guarda la lista de vértices en componentes *OccluderFaceVertexBufferElement.*

CUMPLIMENTAR LA INFORMACIÓN DE LAS ENTIDADES OCLUSORAS

Asignación de la Información a los componentes de las entidades

En este apartado se profundiza la asignación de valores a componentes.

Una entidad oclusora obtiene sus valores de su equivalente GameObject:

- Los MeshFilter.Bounds de los GameObjects oclusores aportan el RenderBounds de las entidades oclusores.
- Los Transform.localToWorldMatrix de los GameObjects oclusores aportan los componentes LocalToWorld de las entidades oclusores

En un comienzo se tienen dos listas, una de las matrices LocalToWorld y otra de Bounds. Ambas se consiguen del conjunto de GameObjects de la lista de oclusores, y para aplicar técnicas de concurrencia en la asignación de los datos a las entidades, se hace necesario usar colecciones como los NativeArray.

Una vez se cuenta con los datos en estas nuevas colecciones, se vuelve necesaria una referencia a las entidades oclusoras a la que asignar los datos. Esto se resuelve haciendo uso de otra de las características estrellas de ECS, las llamadas EntityQueries, que son consultas que permiten recopilar referencias a todas las entidades que cumplan ciertos requisitos. En el caso de las entidades oclusoras, se busca toda entidad con un componente de tipo *OccluderPlaneNormalBuffer*.

Para realizar la asignación de valores en paralelo se usa la construcción **Entities.ForEach**. Dentro de Entities.Foreach se declara la ejecución a nivel de entidad, solo se describe qué operaciones se realizan a una entidad y se aplica lo mismo a todas las de la *query*. Aquí podemos definir qué componentes son de escritura o cuales solo son de lectura, lo cual mejora el rendimiento.

Otro problema al que nos enfrentamos es cómo poder acceder a la información de los vértices. Recorriendo los oclusores solo contamos con la referencia a la entidad *FaceVertex, por esto*, se usa la clase BufferFromEntity [52]. Esta clase permite obtener un DynamicBuffer de la entidad que se indique.

Por último, se debe especificar cómo ejecutar el ***Entities.ForEach*** añadiendo al final ***"ScheduleParallel (this.Dependency)"***, esto proporciona una ejecución en paralelo y un JobHandle [53]. Este JobHandle puede determinar el orden de ejecución respecto a otros métodos, de este modo, aseguramos que hasta que no se haya terminado de ejecutar todos los hilos no empieza a ejecutarse el siguiente.

Revisión de la Información Asignada

A mitad del desarrollo, se consigue arreglar el cálculo de los valores referentes a vértices, normales y otra serie de conceptos en el espacio. Aunque se contaba con la matriz que permite pasar de coordenadas locales a globales, nos supuso un desafío aplicarla ya que no nos dimos cuenta de estos errores hasta realizar cálculos con oclusores rotados o fuera del centro de coordenadas.

Debido a temas de rendimiento y del compilador Burst la información matemática utilizada se maneja en estructuras de la librería matemática “Math”. Por ejemplo, se sustituyen las clásicas *Vector3* de Unity por *float3*. Dichos cambios también dificultaron el correcto cálculo de valores en un principio.

OCTREE

COMMIT INICIAL OCTREE

En el **primer *commit*** se explora la idea de hacer un ***octree* adaptado al patrón de arquitectura de ECS**, y empiezan a gestarse unos componentes iniciales que dan forma a la entidad del octante.

Cabe recordar que **no hay ningún otro** ejemplo de *octree* hecho con **DOTS** en internet, **a excepción de uno** de un usuario llamado ***Antypodish.*** Este utiliza enfoques anticuados porque usa versiones anteriores de DOTS. Su *octree* está desarrollado para ser usado en tiempo de ejecución, lo que convierte nuestro *octree* en **el único** en ser desarrollado para ser ejecutado en el editor con **DOTS.**

En este punto comenzamos a investigar y a diseñar nuestras entidades y sus componentes, aunque estos responden a un enfoque orientado hacia la **legibilidad**. El objetivo principal en esta versión era **aprender** cómo funciona esta tecnología, en general estos componentes responden al desconocimiento que tenemos en este punto sobre esta tecnología.

A continuación, se listan los componentes de esta versión:

- ***Vertices*:** Este componente tiene la función de guardar los ocho vértices del volumen de un octante, posee ocho *float3* (colección de tres valores de coma flotante) cada uno con su nombre completo para una mayor legibilidad.

- ***Size*:** Tamaño del octante (número entero).

- ***Depth*:** La profundidad del octante, se consideraban todavía distintos métodos de búsqueda y navegación dentro del *octree*, ya que se desconocía cómo se haría al final.

Las "referencias" que se explican a continuación no son en si referencias sino una forma de identificar entidades. La idea era usar un ***"string"*** para identificar cada división del espacio y así poder trazar la herencia, respondiendo al siguiente patrón:

- **"0"** <- raíz del *octree*, principio de todos los *strings*.
- **"[a-h]"** <- división del octante, respondiendo al orden de esta.

La cadena estaba formada por cada división desde la raíz hasta llegar al objetivo. Por ejemplo "0ac" respondía a una segunda generación, que parte del primer hijo de la raíz (tomando la raíz como generación cero).

- ***childrenRef:*** inicialmente era un *NativeString128* ya que los componentes deben ser ***"Blittable"*** [38]. Era un DynamicBuffer con tamaño 8 que iba a guardar los *strings* de cada uno de sus hijos. Un *Dynamic Buffer* es una colección diseñada para ser manejada como un componente, es considerada dinámica, ya que, no hace falta determinar su tamaño a la hora de crearla [54].

Si no se determina su tamaño, cuando se van añadiendo elementos, se copian los elementos de este *Buffer* a otro de mayor tamaño.

- ***parentRef:*** un ***NativeString128*** que guarda el *string* identificativo del padre.
- ***selfRef:*** el propio ***NativeString128*** identificativo del octante.

PRIMERA VERSIÓN FUNCIONAL DEL *OCTREE*

En este *commit* se sube una versión del *octree* que utiliza la clase **JobComponentSystem**, que fue durante mucho tiempo la **clase utilizada en ECS para manejar *Jobs*** [55]. En esta versión, al iniciar el sistema, se crea una referencia al **EntityManager.** Un EntityManager es el manejador de entidades dentro de un *World* [51], y posee referencias a todas ellas [33]. Tras crear esa referencia, se crea un **arquetipo** con los componentes primitivos. Un arquetipo es un patrón que será usado cuando crees una entidad, y se corresponde con el conjunto de componentes que tendrá.

Para controlar el flujo de ejecución de esta versión del programa se utiliza un booleano [56].

Cada ciclo de actualización se comprueba si el programa debe continuar. Esto se hace con el booleano mencionado anteriormente *("flag").* En la primera iteración se crea la raíz (en estos momentos con unos valores predeterminados para que la ejecución del programa fuera controlada), en las iteraciones subsiguientes, se crean ocho entidades y se vinculan mediante los *strings* a la raíz. Cada una de estas entidades se crea en el hilo principal, pero sus componentes son añadidos y calculados en un *job*. Añadir o quitar componentes en un hilo alternativo obliga a que deba ser sincronizado con el hilo principal mediante un **EntityCommandBuffer (ECB)** [57]. El uso de esta clase supone un gran **cuello de botella** ya que el sistema debe esperar a que se complete la sincronización.

Esto incurre en una **limitación** por el número de *jobs* esperando a ser ejecutados, que puede ser muy grande tras 3-4 generaciones.

CAMBIO EN LA MANERA DE INDICAR LA JERARQUÍA

Más adelante cambiamos la forma de referenciar entidades, tras experimentar vimos que se pueden vincular las entidades directamente. También se elimina la referencia a la propia entidad, ya que su único propósito era la trazabilidad en las pruebas.

La **nueva jerarquía** respondía a la siguiente descripción: una entidad padre guarda como componente directamente la entidad de su hijo, como una entidad es un identificador, no se realiza una copia de esta, sino una **referencia**. Para navegar por la jerarquía solo hay que comprobar cuál de los hijos posee el valor deseado. El nuevo sistema para vincular de una manera jerárquica las entidades de los octantes se implementó usando el método anterior para generar el *octree*.

Esta versión de desarrollo fue la primera que guarda cierta relación con el producto final, ya que las siguientes variaciones se empezarían a construir en torno a esta versión del diseño. Por fin, adaptamos una estructura de datos basada en **herencia** a un patrón de arquitectura basado en **composición.**

CAMBIOS EN LA GENERACIÓN DE LOS OCTANTES

Posteriormente se cambió la manera de generar los hijos de una entidad y el sistema de generar *jobs.* En lugar de añadir *jobs* individuales a la pila de dependencias, se añade un conjunto de *jobs* utilizando la clase ***NativeList.*** De este modo, cada uno de los *jobs* que generan hijos de la misma entidad comienzan a la vez. Esta decisión de diseño supone una mejora de rendimiento cuando la carga computacional es baja, pero cuando aumenta (por ejemplo 512 tareas), el rendimiento empeoraba debido a los picos de ejecución (sobrecarga momentánea) que sufría el ordenador.

Tras esto comenzamos a pintar una nueva imagen en el horizonte del desarrollo. Se dan pinceladas a los componentes, sistemas y funciones para responder a las necesidades reales del diseño más que a la legibilidad.

SYSTEMBASE

Debido a que ECS en Unity sigue en una fase muy temprana de desarrollo, cuando hay actualizaciones, estas pueden suponer cambios muy significativos. Precisamente, durante el desarrollo de la aplicación nos encontramos con uno de estos casos. La clase manejadora de *jobs* cambió a **SystemBase,** dejando obsoleta la clase JobComponentSystem [58].

También quedaron obsoletos métodos y prácticas anteriores, como era el caso de algunos *jobs* que utilizamos en versiones previas.

Esto supuso un cambio en la orientación del desarrollo ya que ciertos *jobs* enfocados a colecciones quedaron obsoletos en favor de otros que estaban enfocados a ***"chunks".*** Estos *jobs* funcionan con conjuntos de entidades que comparten componentes en vez de con colecciones genéricas. Sin embargo, el cambio no nos llevó a usarlos de manera inmediata, sino que volvimos al más básico, el **IJob**, que ejecutaba una tarea cuando le era indicado.

A lo largo del desarrollo anterior fuimos cambiando el **punto de sincronización** un poco a ciegas. En aquel momento, no había documentación al respecto determinando qué punto podría ser más útil, o incluso cuando se ejecutaba cada punto. Los puntos disponibles eran los siguientes: ***Initialization, Simulation y Presentation***. La sincronización se realizaría en un momento relativo a estas fases (antes del inicio, durante el desarrollo, entre *frames*, después de que acabaran, etc.) [59].

WORKTAG

Más adelante en el desarrollo se abandonó la ejecución controlada con valores predeterminados y se implementó una forma para sacar estos valores de un GameObject. Esto supuso el primer paso para mezclar el mundo de MonoBehaviour con el de ECS en el *octree*.

Esta versión también destaca por introducir el componente llamado ***WorkTag***, este sirve para identificar que octante debe ser **trabajado** y expandido. Está vacío, de manera que **no ocupa espacio.** Este componente solo sirve para que las entidades que lo tienen se encuentren en otra posición de memoria.

La explicación de que no usáramos un componente con un valor determinado para identificar que octantes deben ser expandidos, es la siguiente: **es más rápido** buscar entidades que tengan un determinado componente que buscar todas las entidades para comprobar cuales tienen un determinado valor. Esto se debe a que la primera búsqueda solo consiste en guardar una colección con las entidades que tengan un componente, mientras que la segunda, debe además buscar dentro de esa colección.

También implementamos las llamadas ***"Queries"*.** Una *query* es una búsqueda dentro de todas las entidades existentes de un *manager* para que obtener una colección con las entidades que cumplen con ese criterio [60]. En este momento, para controlar la ejecución del programa, se comprueba a la par que haya al menos una entidad con el componente *WorkTag* (haciendo una *query*, mediante la función ***"*RequireForUpdate*"*** [61]) y el tamaño objetivo de los octantes. De este modo, se respetan las condiciones establecidas en la configuración del módulo.

Al usar ***WorkTag***, el flujo de ejecución cambia, ahora, cuando se terminan de crear los ocho hijos de un octante, se le quita el componente ***WorkTag***. Cada ciclo de actualización se hace una *query* para ver que octantes deben ser procesados y se trabaja sobre esa colección.

SEGUNDA VERSIÓN FUNCIONAL DEL *OCTREE*

Debido a que cambiamos la forma de generar la jerarquía, había que identificar que parte del octree se corresponde con el octante generado. Pero esa identificación sólo es necesaria a la hora de generar los datos del nuevo octante, por eso se añadió una variable dentro del componente ***WorkTag*** (ya no está vacío), que solo existe durante el procesado de las entidades.

El flujo de ejecución no podía seguir siendo controlado con un booleano, cambiamos la implementación para usar un bucle basado en las generaciones que se iban a procesar. Para saber el número de generaciones se diseñó la siguiente **fórmula**:

$$log(min(\frac{SceneScope.extents.x \cdot 2}{OctreeVoxel.extents.x \cdot 2}), (\frac{SceneScope.extents.y \cdot 2}{OctreeVoxel.extents.y \cdot 2}), (\frac{SceneScope.extents.z \cdot 2}{OctreeVoxel.extents.z \cdot 2})),2)$$

Debido a que los objetos no deben tener necesariamente el mismo tamaño en todas las dimensiones, se selecciona el valor mínimo de dividir el tamaño inicial y el tamaño objetivo en cada dimensión.

En esta versión, el proceso para generar los hijos pasa a ser el siguiente**: en el hilo principal** se generan los ocho hijos y se vinculan al padre. Tras esto se pone **en paralelo un hilo a procesar** las entidades que tengan el componente *WorkTag*. El *job* utilizado en este momento es un ***IJobChunk***, que funciona de manera similar a un ***foreach*** [62], pero dentro de un ***"chunk"*** [34].

Se pasó a usar ***ScheduleParallel*** en vez de ***Schedule*** para permitir la ejecución del *job* en varios hilos (un conjunto de elementos por hilo). ***Schedule*** y sus variantes es la forma de añadir a la pila de ejecución una tarea para que la ejecute un hilo [63].

Finalmente, aquí es cuando **se pinta el *octree* por primera vez**, para ello, se buscan todos los octantes de máxima profundidad y se pinta un *Handle* [50] en su lugar. Esto puede verse en la Figura 30.

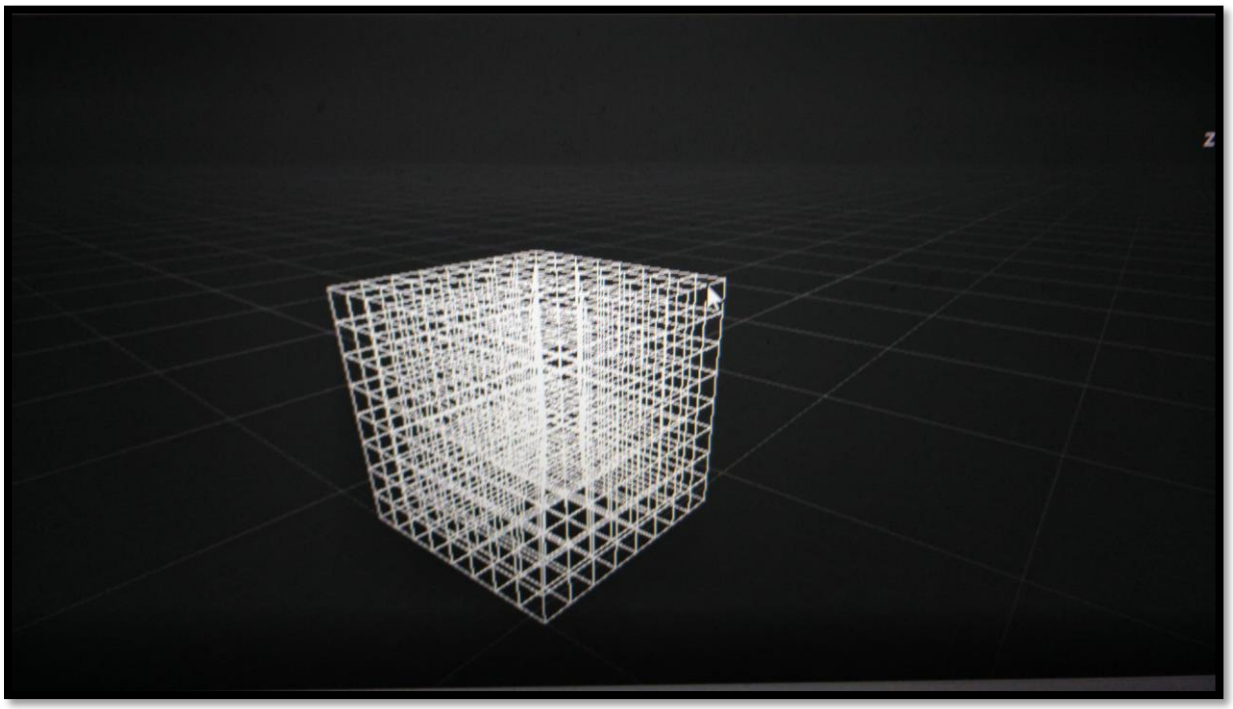

Figura 30.- Pintado inicial del octree.

Como se puede apreciar, esto dificulta entender el resultado y no nos deja claro cómo debe interactuar el usuario con el o*ctree*.

TERCERA VERSIÓN FUNCIONAL DEL *OCTREE: ENTITIES.FOREACH()*

Uno de los cambios introducidos en las actualizaciones grandes que sufrió **ECS** son las ***queries*** con la siguiente estructura:

"Entities.WithAll<TComponent>().ForEach((param)=>{doStuff()})"

Funcionan de la siguiente manera: se buscan todas las entidades que tengan el componente indicado (TComponent) y se trabaja sobre ellos (con la función *doStuff()*) [64].

Lo importante de esa forma de hacer *queries* es el parámetro de la *función lambda* [65]**,** el acceso a este parámetro está **muy optimizado** y es inocuo en el rendimiento del programa. [64] Es por esto por lo que los parámetros a ser modificados en la *función lambda* suelen estar incluidos en los parámetros. Sin embargo, estas búsquedas introducen un punto negativo, dentro de estas *queries* no pueden hacerse búsquedas de entidades usando como filtro el tipo de componente que aparezca como parámetro en la *función lambda.* Esto es para evitar que las búsquedas te devuelvan dos colecciones idénticas, lo que no está permitido [64].

Debido a que hasta este punto necesitábamos acceder a valores de la entidad padre para rellenar valores de la entidad hijo, no podíamos poner parámetros para trabajar con ellos en las *queries*. Nos vimos obligados a usar el parámetro *"Entity e"*, que es una referencia a la propia entidad.

En aras del rendimiento, para quitar el componente **WorkTag** de los octantes en el hilo principal empleamos una tarea con el parámetro:

"WithoutBurst().WithStructuralChanges().Run()".

Poner esto al final de una tarea significa que **no se usa el compilador Burst** (debido a que no soporta esas tareas), que **esta tarea realiza cambios estructurales** (lo que supone que automáticamente manejara la sincronización) **y debe ejecutarse en el hilo principal** (debido a la incompatibilidad con las funciones) [58]. Esto, aunque es ejecutado en el hilo principal, demostró ser una forma muy optimizada de trabajar sobre un conjunto de entidades, ya que **es más rápido que hacerlo directamente**.

Cuando los oclusores estuvieron preparados (aunque en una fase temprana), se comenzaron a vincular a las entidades. Al principio no se filtraban, ya que aún no se tenía claro la forma en la que se haría. A medida que fue avanzando el desarrollo en este campo, fueron integrándose con el *octree,* creando componentes nuevos que contendrían su información. Eventualmente si se llegó a conseguir un filtrado de oclusores en la división de octantes, consiguiendo hacer herencia, para que cada nueva división solo tenga que comprobar los de su padre.

CUARTA VERSIÓN FUNCIONAL DEL *OCTREE:* COLECCIONES ADICIONALES

Debido al problema anteriormente mencionado**,** buscamos alternativas para poder utilizar parámetros en Entities.ForEach. Una de estas alternativas fue colocar los componentes de la entidad padre necesarios en una colección, y acceder a ellos con un identificador dentro de la *función lambda.* Esto aumentó el rendimiento, pero aun así **no era perfecto** porque creaba la necesidad de tener una serie de **colecciones muy grandes.** Además, el usar esta técnica obliga a hacer el acceso normal a un componente, que supone buscar un componente de una entidad concreta dentro de la colección de todas las entidades y esto no está optimizado. El identificador utilizado en este proceso también se añadió dentro del componente ***WorkTag***, ya que solo se necesitaría esa información durante el procesado.

Estas colecciones debían usarse con ***"WithReadOnly()"*** como argumento en el ***"ForEach()"*** para que pudieran accederse de manera segura en paralelo. Este parámetro garantiza que los datos de esas colecciones sólo serán usados para **leerlos,** lo que asegura la **integridad** de estos.

Debido a necesidades del diseño, algunos componentes sufrieron una serie de cambios, como por ejemplo el tamaño pasó de ser un valor *"int"* (número entero) a ser *"extents"* de los *"Bounds"* (límites) del objeto, para poder calcular intersecciones [66]. Otro componente que sufrió cambios es ***"depth"***, que **fue eliminado** a lo largo del desarrollo ya que, a medida que iba madurando el diseño de la búsqueda y del movimiento dentro del *octree*, se fueron eliminando componentes que solo servían al control y a la legibilidad. La profundidad solo nos ayudaba a la hora de trazar el progreso del sistema. También se optó por no borrar el componente **WorkTag** en el último nivel de profundidad, ya que nos era útil a la hora del cálculo de la visibilidad para encontrar los octantes que se usarían para proyectar.

Finalmente, en esta versión cambia el pintado del *octree* ya que el anterior no era nada eficiente. Explicaremos a continuación los cambios realizados a la navegación y al pintado:

Comenzaremos por la navegación, cuando el *octree* está activo se ve la raíz. Si se hace clic dentro de ella se buscan sus hijos y se pintan, si se hace clic en uno de sus hijos pasan a pintarse solo los hijos de ese hijo, y así sucesivamente. De este modo sólo tenemos que pintar ocho elementos a la vez como máximo, lo que supone que el pintado se vuelve inocuo para el rendimiento. Si se hiciera clic fuera de los octantes representados volveríamos atrás en un nivel.

El pintado funciona como una pila de elementos a representar, los cuales van cambiando en función del octante en el que hayamos pulsado. Simplemente se pintan *Handles* en las posiciones de los octantes de la pila. (véase Figura 31)

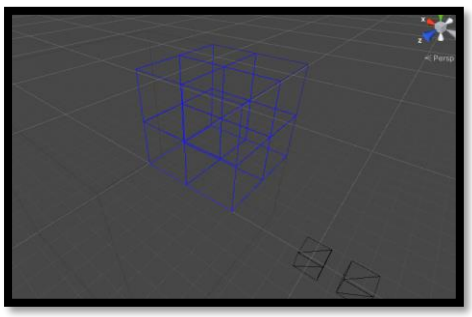

Figura 31.- Segundo tipo de pintado de octree en la escena.

QUINTA VERSIÓN FUNCIONAL: INSTANCIAR EN LUGAR DE CREAR

En la búsqueda por el rendimiento descubrimos una forma de realizar **"herencia"**, sin la necesidad de colecciones adicionales. Para ello, instanciamos entidades en lugar de crearlas. Esto supone que, cuando se genera un "hijo", este es una **copia** del padre, por lo que ya posee los valores heredados de su padre y no necesitamos perder recursos en:

1. Crear las entidades y generar cambios estructurales.

2. Hacer búsquedas adicionales para encontrar los valores de otras entidades.

Esto supuso la mejora de rendimiento más significativa que experimentamos en el diseño final.

VISIBILIDAD

La visibilidad de la escena se calcula en el *script VisibilityCreateSystem**,*** este sistema incluye todos los métodos y *jobs* para el correcto cálculo de la visibilidad.

El cálculo de la visibilidad recae en los conceptos matemáticos del *Visibility Octree,* tomamos inspiración de su explicación sobre cómo calcular la sombra de un oclusor convexo desde un punto. Para calcular la sombra de un oclusor desde un punto, primero debe conocerse que caras son visibles desde este. Después, hay que seleccionar la **silueta** del oclusor, es decir, el conjunto de aristas que no se encuentren compartidas por dos o más caras. Con el propósito de calcular la silueta implementamos la clase ***Edge***. Esta clase ordena los dos puntos que conforman la arista de la misma forma, mediante el método *hashcode*; permitiendo así discernir que la arista AB es igual a la BA.

Si se une el punto con estas aristas obtenemos el ***frustum*** que representa la sombra generada, obtendremos lo que llamamos ***Sample*****.** Un *Sample* es por tanto una entidad que contiene información de la sombra de un oclusor desde un vértice del *voxel*.

Se necesita generar una cantidad variable de *Samples* en función del caso, y se opta por el EntityCommandBuffer (en adelante ECB) para predisponer una generación de entidades acordes. Las entidades son creadas en el hilo principal ya que, de ser generadas en un hilo paralelo, deberíamos esperar a la sincronización del ECB.

PUESTA EN MARCHA DEL ECB

El primer problema surge al intentar realizar toda la visibilidad en una función propia del SystemBase. Para que el ECB se sincronice es necesario que se produzca un cambio de ciclo de reloj, y, para que este se lleve a cabo, se tiene que mover el código al método *Update* del SystemBase. El *Update* no se ejecuta en modo Editor [59]. Hay que añadirlo a ejecución creando una Action de C# [67].

SAMPLES ÚNICOS

Durante el desarrollo nos percatamos de que los *voxels* que comparten vértices generan de entre 2 a 8 copias por cada *Sample* único. Entendemos por *Sample* único aquel con cierto vértice y cierto oclusor. Nos centramos en solventar este problema, y para ello se plantean las siguientes soluciones:

- Intentamos hacer uso de un *HashMap* [68] que tuviera como clave una estructura vértice-oclusor. Debido a restricciones de escritura-lectura y paralelismo se descarta.
- También se prueba a borrar aquellos repetidos, pero es más costoso eliminarlos que tenerlos en memoria.

MÚLTIPLES SAMPLES, CÁLCULO ÚNICO

Tras experimentar con las anteriores soluciones, seguimos enfrentándonos a la misma problemática. Necesitamos reducir el número de operaciones, pero sabemos que reducir el número de *Samples* no es una opción. Pensamos en aprovechar que ECS agrupa la memoria en *chunks* [34]. Así, usando los denominados **SharedComponent** [69] podemos agrupar por *chunks* las entidades que tienen el mismo valor. Se procede a reconvertir el componente asociado al vértice y al oclusor en un SharedComponent*;* colocando así en el mismo *chunk* todos aquellos *Samples* iguales. Pese a que tener muchos *chunks* relativamente vacíos no es algo positivo, nos permite procesar una entidad por *chunk* y copiar el resultado en el resto. Así, sacrificando memoria, evitamos **repetir** cálculos.

INTERFAZ DE USUARIO

ANTECESOR DE LA INTERFAZ

En un primer momento, la interfaz responde a necesidades del desarrollo, sin tener en cuenta la usabilidad. Esta interfaz podría sufrir una serie de cambios a causa de necesidades que pudieran emerger a lo largo del periodo de desarrollo. Para desarrollarla usamos la clase GUILayout (y EditorGUILayout), que es el método clásico para programar interfaces de usuario [70], [71].

Esta interfaz, o precursor de la misma, se programó en código, es decir, toda la jerarquía de elementos se hacía al vuelo, generando contenedores y escribiendo en los mismos los elementos deseados y sus funciones. Todo gracias a las clases mencionadas anteriormente: *EditorGUILayout* y *GUILayout.* Estas se encuentran bien asentadas ya que son bastante maduras. Sin embargo, esto solo era por conveniencia, ya que, si queríamos innovar, debíamos innovar en todo lo posible.

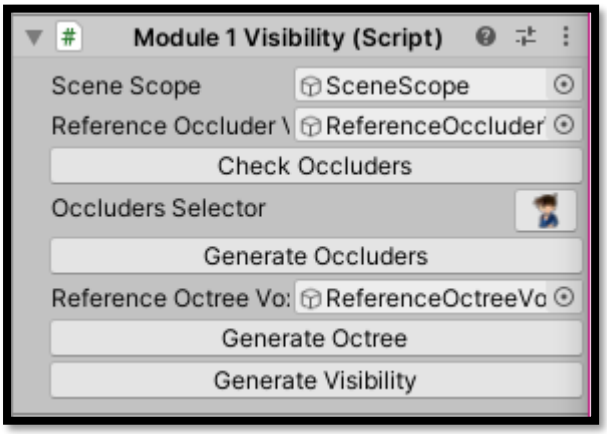

Figura 32.- Primera interfaz.

La primera interfaz implementada (véase Figura 32) consistía en una serie de botones colocados en serie respondiendo el orden al flujo de ejecución del programa, se procede a explicar todos los campos de esta versión:

- ***Scene Scope:*** Este primer campo se correspondía con el GameObject que serviría como volumen contenedor de la escena. Aunque como ya se mencionó anteriormente, no sería utilizado realmente hasta más adelante en el desarrollo.

- ***Reference Occluder Voxel:*** El GameObject actualmente conocido como *"Occlusion Reference".* Ambos valores de referencia (tanto el de los oclusores como el de la proyección) tenían nombres provisionales que, en lugar de ser cercanos al usuario, eran cercanos a la teoría.

Por el momento pensábamos únicamente con volúmenes y no con objetos como tal, por lo que, todo seguía muy conceptual.

- ***Check Occluders:*** Este botón servía para controlar el flujo del programa, y su correspondencia con la interfaz actual es el botón *"Automatic Selection"*. Su función es la de seleccionar automáticamente todos aquellos oclusores que sean más grandes que el tamaño definido por el ***"Reference Occluder Voxel"***.

- ***Occluders Selector:*** Esto no es sino una versión primitiva de ***"Tool Panel".***

Originalmente los controles usados al tener la herramienta activa eran bien distintos, principalmente orientados a ser usados por nosotros, no por un usuario de la herramienta. Si se hace clic sobre cualquier otro elemento se pierde de vista el inspector, y por tanto los instrumentos de este. Tampoco mostraba ningún tipo de indicación de controles, ya que, los usuarios de esa versión, nosotros, debíamos conocerlos de memoria. Estos eran los siguientes:

1. Para **añadir oclusores** lo mejor era hacer **control + clic** en la jerarquía, ya que el *Scene Scope* tampoco se encontraba deshabilitado, lo que dificulta mucho interactuar con los elementos de la escena. Si el añadir el objeto se hace correctamente, adquiere un marco rojo y un mensaje en la **consola** nos avisa del resultado.
2. Para rotar la cámara había que pulsar **alt + clic** y arrastrar en la dirección que queramos.
3. Para **quitar oclusores**, hay que realizar las mismas tareas que para añadirlos, pero sobre un GameObject añadido a la escena. Aparecerá un mensaje en consola indicando el éxito o el fracaso de esta operación, además de que el objeto en cuestión pierde su característico borde rojo.

- ***Generate Occluders:*** Al pulsar este botón los objetos seleccionados se convierten en entidades. No hay ningún indicador visual que avise del éxito de esta operación. Para ver los resultados debemos entrar en el ***"debugger"*** de entidades, para ver cuales existen (véase Figura 33).

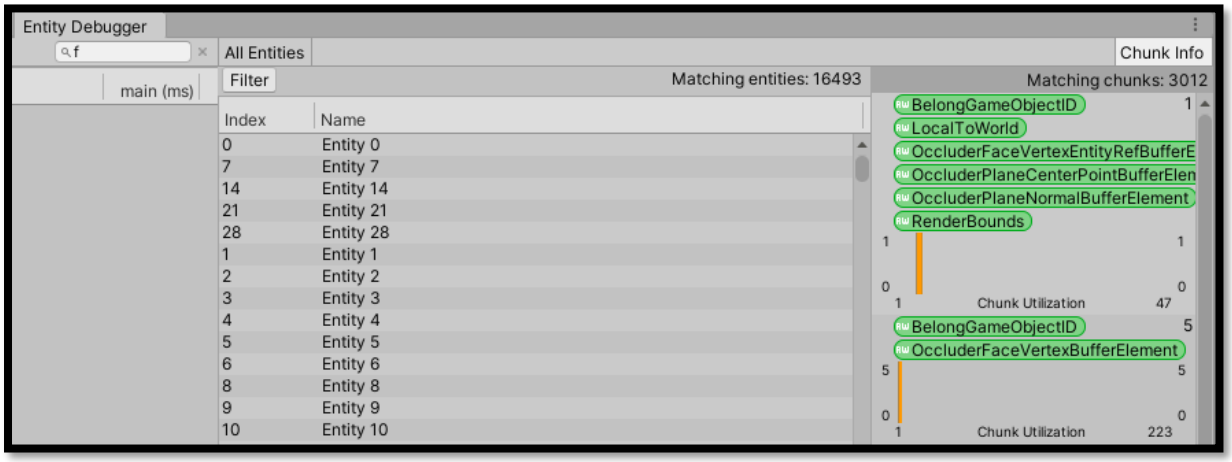

Figura 33.- Entity Debugger.

- ***Reference Octree Voxel:*** GameObject que servía para saber cómo de grande debían ser los octantes de máxima profundidad. Este objeto sería de vital importancia en versiones de desarrollo posteriores.

- ***Generate Octree:*** Si pulsamos este botón se generará el *octree*, de nuevo, para ver los resultados debemos entrar en el *"debugger".* En versiones tempranas el *octree* no aparecía ni siquiera dibujado. La explicación de que este botón se encuentre separado de *"Generate Visibility"* es el desarrollo simultáneo que ambos métodos estaban experimentando. Además, podríamos controlar mejor el flujo de ejecución del programa y ver con mayor claridad los tiempos de ejecución.

- ***Generate Visibility:*** Al pulsar este botón se genera la visibilidad, de nuevo, no se muestra ningún estímulo que indique el resultado.

Una vez generada la visibilidad, si vuelve a activarse el *"Occluders Selector"* veremos como el *Scene Scope* toma un color azul (véase Figura 34).

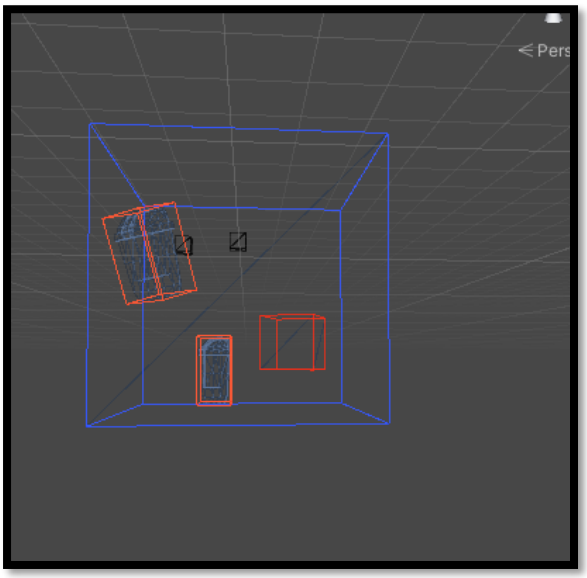

Figura 34.- Scene Scope activo.

Para **navegar** por los resultados hay que hacer **control + clic** dentro del o*ctree* para ir bajando por las generaciones hasta llegar al octante proyector (véase Figura 35).

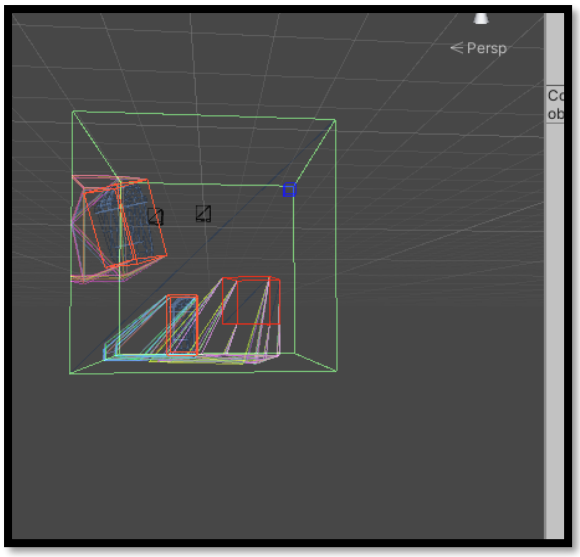

Figura 35.- Visibilidad de un octante.

Como puede apreciarse en la Figura 35, esta versión ya implementa la visualización ***"Wireframe"*** como modo de enseñarnos que la herramienta se encuentra activa. Considerábamos esto como un estímulo visual a la par de como una forma clara de visualizar los resultados de la visibilidad.

Debido a que en estos momentos la visibilidad se imaginaba como una **subescena** donde desarrollar estos cálculos, era necesario activar la herramienta *"Live Conversion in Edit Mode".* Esta herramienta se encuentra en la barra de menú *"DOTS -> Live Link Mode -> Live Conversion in Edit Mode".* (véase Figura 36)

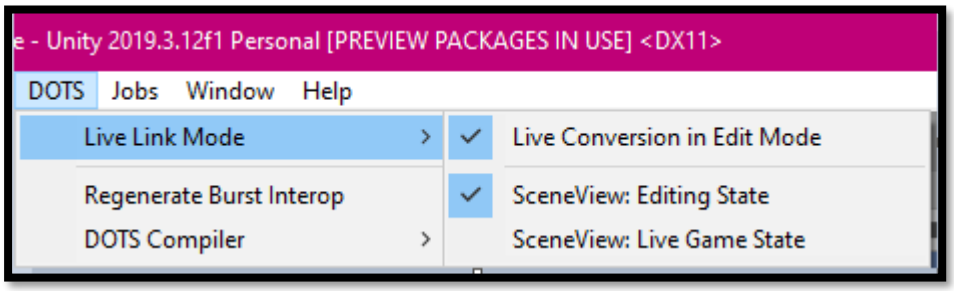

Figura 36.- Live Link Mode

Esta herramienta era necesaria (en su momento) para pasar de GameObject a Entidad.

DISEÑO DE UN PROTOTIPO: AVANZANDO EN EL DESARROLLO

Más adelante, para poder activar desde nuestra herramienta la función DOTS Live Conversion, se añade un botón al principio del editor. El orden de la interfaz en estos momentos sigue correspondiéndose directamente con el flujo de funcionamiento del programa.

Decidimos formalizar el diseño, empezando a pensar en una interfaz usable por el usuario, surge el siguiente **esquema** (véase Figura 37). El campo *"Occluder Selector"* desaparece, ya que empezamos a plantearnos si el usuario tiene la necesidad de usarlo o no.

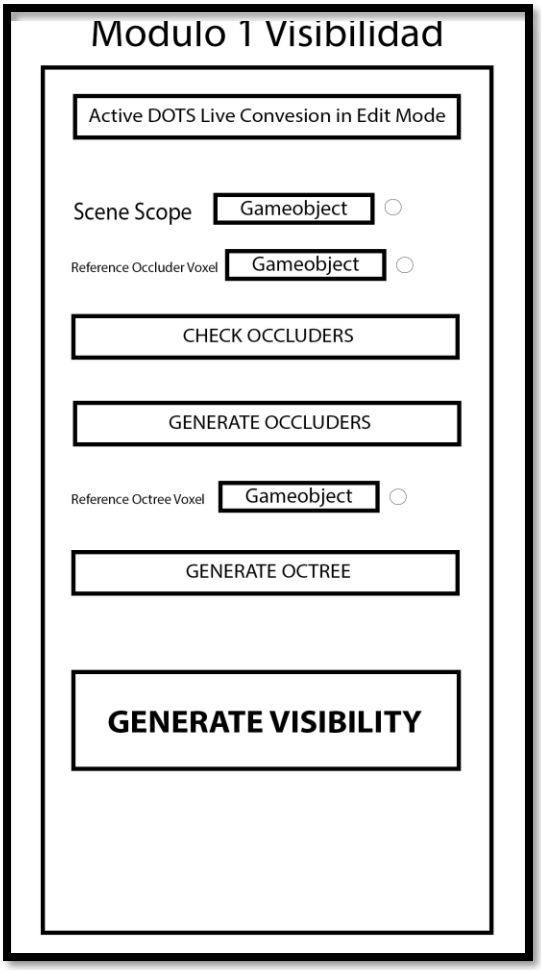

Figura 37.- Esquema antecesor a la interfaz moderna.

Otro hecho que motiva su retirada es que, como el resto de la interfaz está en base al flujo de ejecución del programa, tener un campo que puede ser usado en distintos momentos resulta en una **mala práctica**. Es por esto por lo que este esquema plantea una versión en la que el acceso a los resultados es automático. La visibilidad con ***"wireframe"*** es exclusiva de los resultados.

PRIMERA VERSIÓN FUNCIONAL HECHA CON *UIELEMENTS*

UIElements es la nueva clase constructora de interfaces, bien para su uso en tiempo de ejecución o bien para construir editores [21]. En un primer momento, su uso en tiempo de ejecución era exclusivo para ECS, y su sintaxis es relativamente simple. Pese a esto, como suele pasar con la tecnología experimental de Unity, su documentación es bastante **ligera**, a menudo invitando a la investigación debido a la falta de datos.

Esta clase usa un **UXML** para generar su jerarquía [23], lo que supone un contraste con la manera anterior de generar un editor. Anteriormente se creaba el contenedor y se programaba el elemento con todo lo necesario, ahora, mediante el paquete UIBuilder [20], se genera en una herramienta con una **interfaz gráfica** (véase Figura 38). Aquí podemos generar una interfaz sin funcionalidad, pero con jerarquía y estilo plenamente desarrollados.

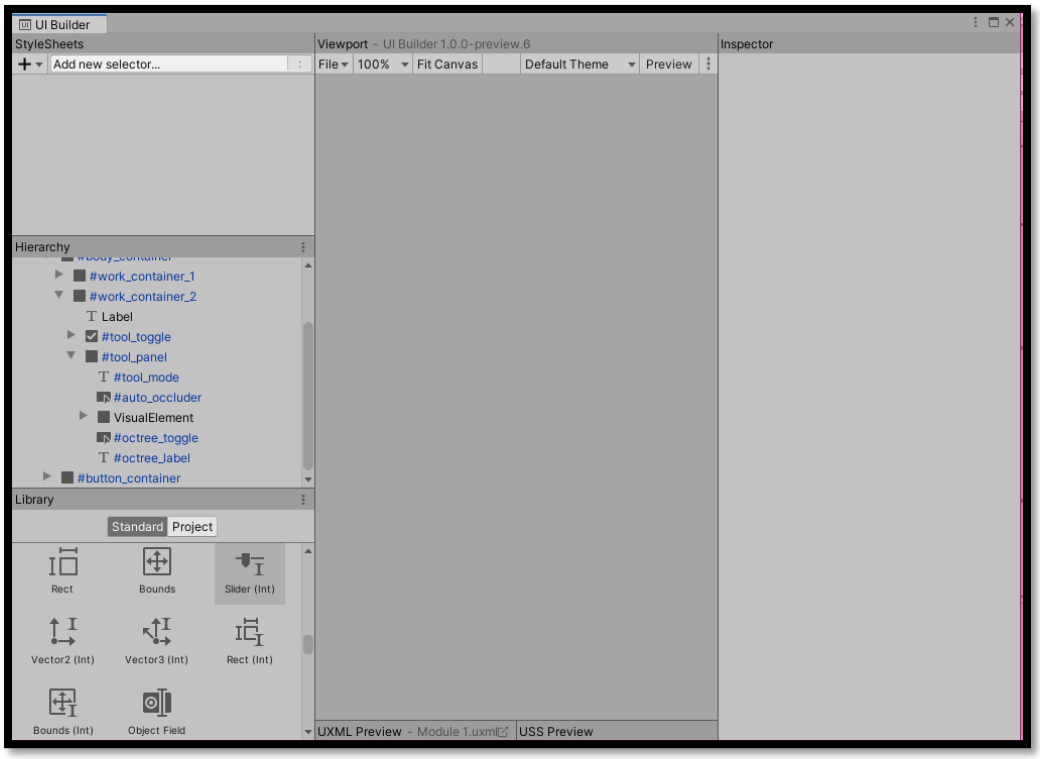

Figura 38.- Interfaz gráfica de UIElements.

Hay un cambio en la forma de trabajar, ahora hay hacer ***"UQueries"*** [72] en la jerarquía de la interfaz para poder acceder a los elementos y modificarlos.

Esta clase está en **desarrollo** todavía, por lo que aún carece de muchos elementos, lo que suponía un reto ya que, debido a esto, la interfaz resultante podría no haber sido lo suficientemente buena.

Esta es la primera versión cuyo desarrollo se orienta ya hacia el usuario, eliminando campos que carecen de interés para él.

A continuación, se listan los elementos de la anterior interfaz y se explicarán los cambios que sufren para convertirse en los de esta versión (véase Figura 39).

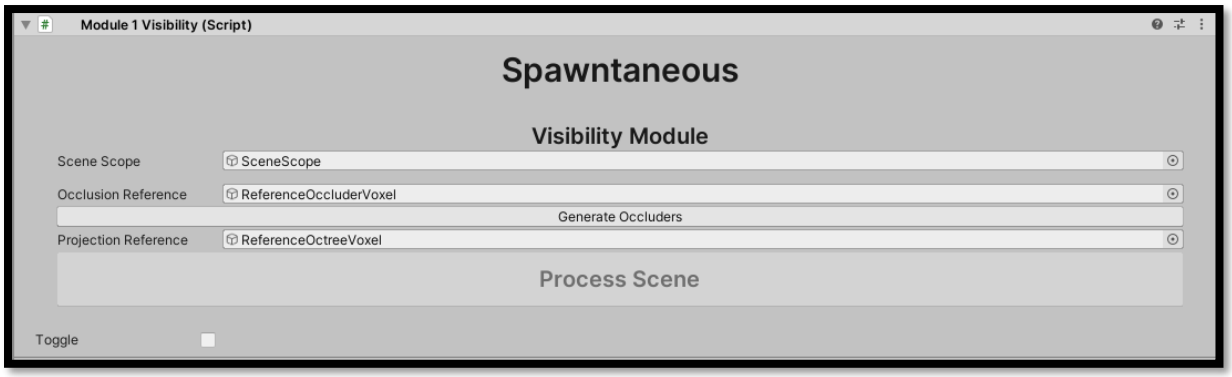

Figura 39.- Primera versión de la interfaz hecha en UIElements.

- ***Scene Scope:*** se mantiene en el mismo sitio, ya que es la base de nuestro programa.

- ***Reference Occluder Voxel:*** Pasa a llamarse ***"Occlusion Reference"***, para que tenga una mayor legibilidad para el usuario.

- ***Check Occluders:*** Este campo desaparece, aunque será restaurado después. En estos momentos carece de funcionalidad.

- ***Occluder Selector:*** Eliminado también, ahora su función se realiza automáticamente tras procesar la escena.

- ***Generate Occluders:*** este botón continúa aquí, aunando la funcionalidad anterior con la de *Check Occluders*, ya que, consideramos que un usuario podría hacer ambas a la vez (en estos momentos tenemos fe plena en la selección automática).

- ***Reference Octree Voxel:*** de nuevo, otro campo que cambia de nombre debido a la legibilidad. ***"Projection Reference"*** es mucho más claro.

- ***Generate Octree:*** desaparece, debido a que consideramos que un usuario no debe perder el tiempo entre la generación del *octree* y la de la visibilidad. El propósito de este *octree* es exclusivamente el de servir para el cálculo de la visibilidad.

- ***Generate Visibility:*** se convierte en ***"Process Scene"***, de este modo dejamos claro que este botón hará todos los cálculos necesarios.

- El campo que aparece como ***“Toggle”*** es el precursor de la herramienta que sustituirá al anteriormente conocido como ***"Occluder Selector"***.

SEGUNDA VERSIÓN FUNCIONAL CON *UIELEMENTS*: PANELES

A medida que crecía esta interfaz, también crecían los elementos que había en ella. Es aquí donde toma forma la interfaz final, si bien todavía debe pasar por algunos cambios, este es un gran paso para alcanzar dicho resultado. (véase Figura 40 y Figura 41)

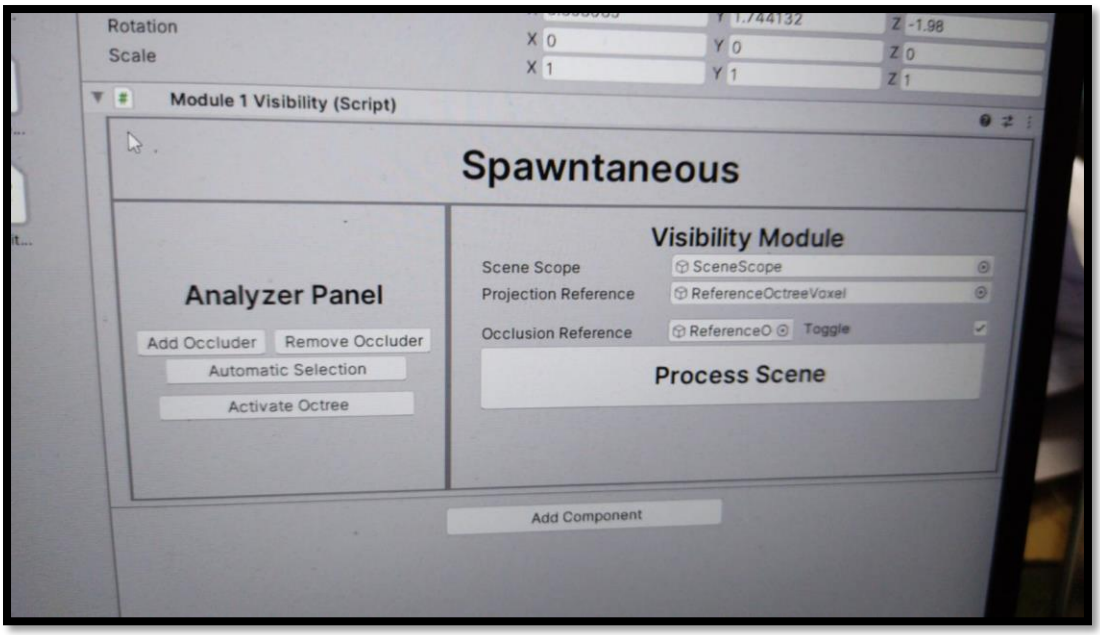

Figura 40.- Versión de la interfaz con el panel a la izquierda.

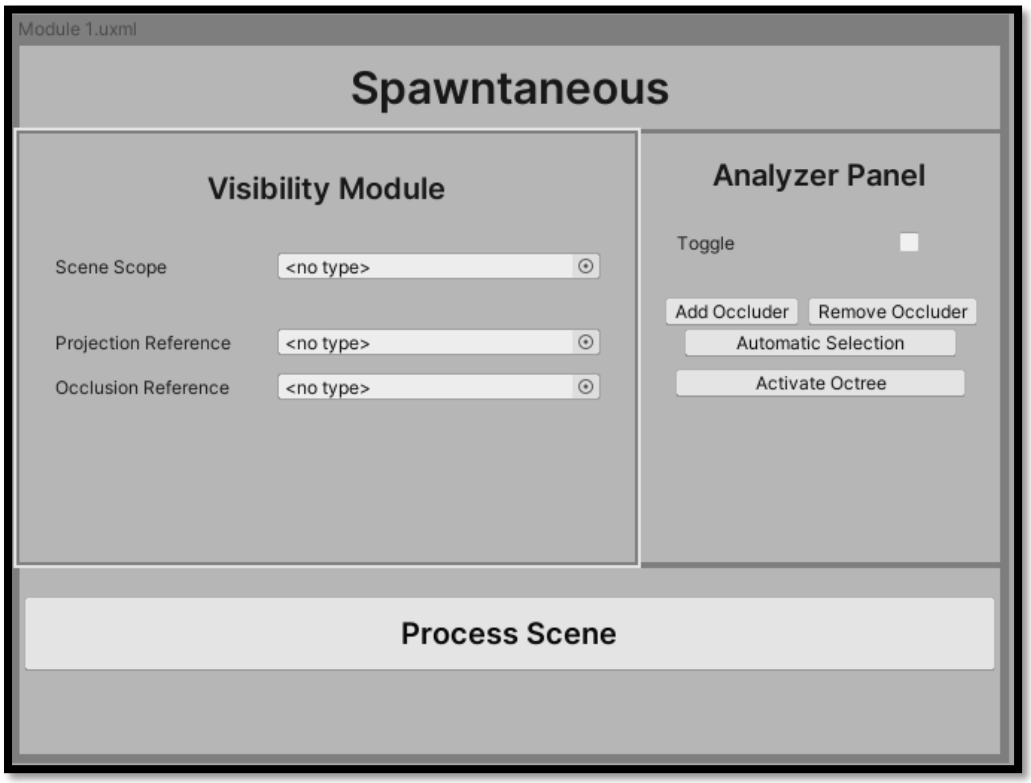

Figura 41.- Versión de la interfaz con el panel a la derecha.

Esta interfaz consta de **tres** paneles:

1. **Un panel denominado *"Visibility Module"***, que se corresponde con los objetos que conforman la configuración del módulo: *"Scene Scope"*, *"Occlusion Reference"* y *"Projection Reference"*.

2. **Un panel llamado *"Analyzer Panel"***, que tiene un interruptor, que, de estar apagado, oculta todos sus botones, y, de estar encendido, muestra los instrumentos disponibles para el usuario. El motivo de que se encuentre hecho con un interruptor es que, de no ser así, tendríamos que bloquear los instrumentos que contiene cuando no fuera posible su uso. Creemos que la mejor interfaz posible debería mostrarnos en cada momento lo que **podemos hacer**, no una serie de funciones no disponibles. Sin embargo, en esta temprana versión, el botón *"Process Scene"* si se bloquea cuando no tenemos una selección o cuando nos falta alguno de los tres objetos de configuración, lo que supone una incongruencia con nuestras intenciones. Dentro de este panel pueden verse una serie de botones que corresponden al siguiente orden: añadir-eliminar oclusores primero, selección automática de oclusores segundo, activar-desactivar *octree* tercero. Esto se debe a que pretendíamos unir los elementos comunes, no veíamos como una buena decisión colocarlos de tal manera que los paneles siguieran este orden: un botón, dos botones, de vuelta a un botón.
Los botones ***"Add Occluder"*** y ***"Remove Occluder"*** se corresponden con la anterior funcionalidad de hacer **control + clic** para añadir o eliminar oclusores de la selección. Estos eliminan la necesidad de pulsar control para llevar a cabo su función. Ahora es solo con **clics**, de dejarlo de la anterior forma, estaba garantizado que habría algún usuario que intentaría hacer solo clics, y que no vería ningún resultado. Errores de ese estilo debían ser **erradicados**.
El botón ***"Automatic Selection"*** es el que realiza la función anterior de ***"Check Occluders"***.
"Activate-Deactivate Octree" sirve exclusivamente el propósito de que no se meta de por medio el *octree* a la hora de ver los oclusores; permitiendo así su desactivación para una **mayor visibilidad**; de nuevo, buscando una mejora de la usabilidad.
En esta versión desaparece el botón *"Generate Occluders"*, ya que caemos en que, para el usuario medio, comprobar que sus oclusores se han convertido en entidades, le es indiferente, ya que busca que se **procese su escena**.

3. **Un tercer panel** que no tiene nombre muestra el botón *"Process Scene",* y cuando se calcula la visibilidad, las barras de progreso del programa. (véase Figura 42)

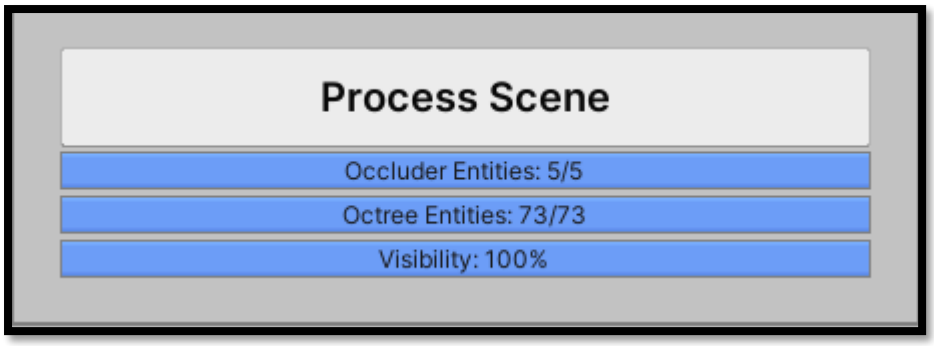

Figura 42.- Barras de progreso.

Como se puede ver en la Figura 40, y en la Figura 41, dudábamos si debíamos colocar el segundo panel a la izquierda o a la derecha. Finalmente decidimos colocarlo a la derecha porque, debido a que somos de una cultura que escribe de **izquierda a derecha**, nos parecía extraño que el "segundo" panel (siguiendo el orden de uso) fuera el izquierdo, iba en contra de nuestra lógica. Decisiones como esa, responden exclusivamente a la **usabilidad**. Siendo la usabilidad máxima prioridad en estos momentos.

Esta interfaz es un gran salto hacia delante ya que reduce el número mínimo de botones necesarios para pulsar a **tres**: el *"Toggle"*, para activar la herramienta, el *"Automatic Selection"*, para poder tener alguna selección, y el *"Process Scene"*, para generar el cálculo.

Simplifica enormemente el uso de la herramienta.

En estos momentos, la interfaz no es adaptable o ***"responsive"***, es decir, no se ajusta bien al espacio disponible y puede generar errores visuales.

TERCERA VERSIÓN FUNCIONAL CON *UIELEMENTS*

Todos los errores anteriores, ya sean de lógica, o de programación, son arreglados en esta versión. Es sin lugar a duda la interfaz más cercana a la iteración final. De aquí al producto final hay muy pocas diferencias (véase Figura 43).

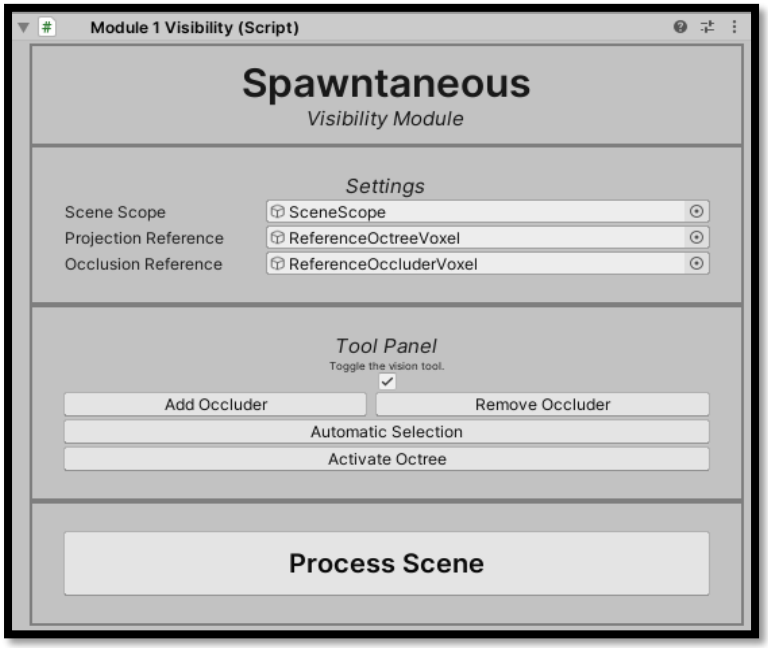

Figura 43.- Versión avanzada de la interfaz hecha en UIElements.

Esta versión presenta cambios en los nombres de elementos:

- Se da un **título y un subtítulo** a la herramienta, para que sea fácilmente identificable.

- **Settings:** el panel anteriormente conocido como ***"Visibility Module"***, no tiene sentido que este panel tuviera ese nombre, ya que toda la herramienta es el módulo de visibilidad. Los elementos de este panel no sufren ninguna modificación.

- ***Tool Panel:*** un nombre más apropiado que ***"Analyzer Panel"***, ya que la funcionalidad que ofrece no es solo la de análisis, además, contiene las herramientas para interactuar con la escena que el programa nos ofrece.

- Se añaden ya **indicadores** que explican qué hacen los elementos, en concreto, se explica que hace el interruptor, para no confundir al usuario. Esto supone una pincelada del cuadro final, que contará con indicadores para no perder al usuario.

- La interfaz por fin se hace **adaptable**, como podemos apreciar en la Figura 43, cuando la interfaz se queda sin espacio, el ***"Tool Panel"*** se mueve hacia abajo. Se genera una experiencia de usuario adaptable, permitiendo usar el espacio que el usuario considere, sin dañar su experiencia.

VERSIÓN FINAL: USABILIDAD

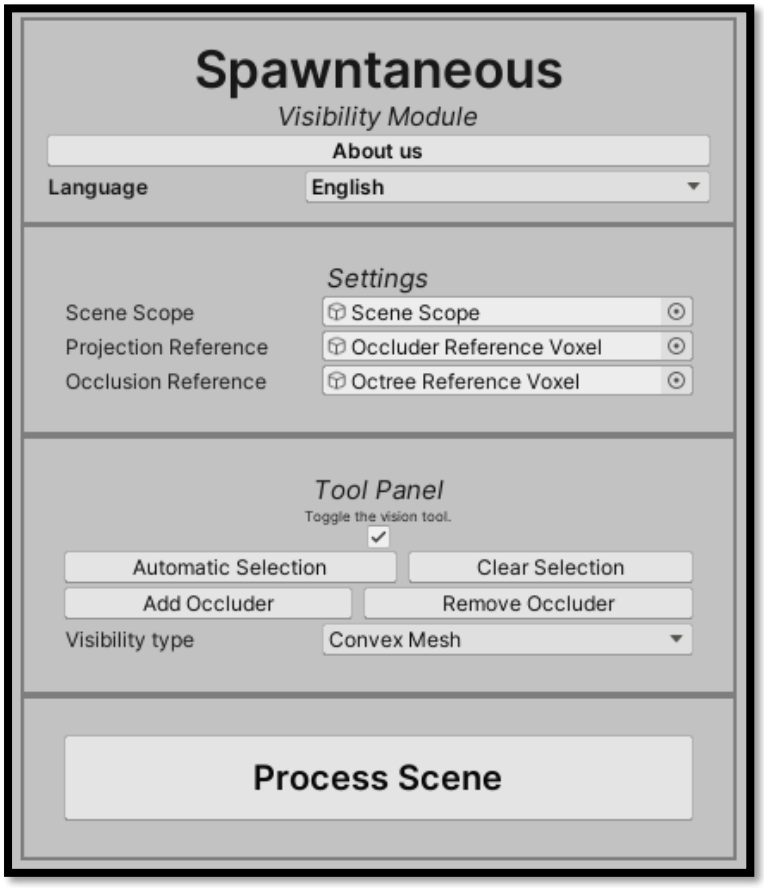

Figura 44.- Versión final de la interfaz.

Esta versión está dedicada exclusivamente a cambios en aras de la **usabilidad:** se añaden indicadores contextuales, mensajes emergentes y textos explicativos.

Debido a que la clase **UIElements** sigue en desarrollo hay muchos elementos que no se encuentran disponibles en esa clase por defecto, lo que nos llevó a buscar soluciones alternativas. Se añaden elementos que se consideran esenciales (como mensajes de alerta) que deben hacerse con **GUILayout** porque no están disponibles en UIElements. En la manera que nos ha permitido la clase, esta interfaz es **autocontenida.**

En esta versión no hay botones bloqueados, y el número mínimo de botones necesarios para que funcione es uno. Solo hace falta pulsar ***"Process Scene"***, y el usuario más sencillo habrá terminado de usar la herramienta, sin embargo, para un usuario que requiera de una mayor precisión, las **opciones** son mucho mayores, desde distintos modos de pintado, hasta selección manual de oclusores.

Uno de los cambios realizados en este momento es que dejamos a un lado la visión ***"wireframe"*** en favor de ***"shaded wireframe"*** para obtener una mayor visibilidad.

Se añade por primera vez en el trabajo la opción de que la interfaz pueda ser visualizada en **dos** idiomas, español e inglés, dedicando un elemento de decisión para ello. Aquí identificamos un problema serio, los cambios que se hacen a la jerarquía del **UXML**, como puede ser hacer elementos visibles o invisibles, solo se guardan si se mantienen valores destinados

para ello en el objeto al que pertenezca la interfaz, debido a esto, si cambias el idioma de la interfaz y quitas el foco a este inspector, ese cambio se pierde.

Para solucionar esto, nos hemos visto obligados a mantener una búsqueda de inconsistencias mientras se esté usando la ventana.

Mientras está visible, comprueba que el subtítulo y el lenguaje coinciden, de este modo no nos vemos obligados a saturar de campos la jerarquía, ya que actualmente parece ser el único modo de hacer esto.

Se añaden mensajes emergentes cuya única finalidad es la de evitar errores del usuario.

Finalmente, Para mejorar el flujo de uso de la interfaz, se hace invisible el botón *Activate Octree* hasta que haya un *octree* que activar/desactivar.

De este modo, alcanzamos un producto del que podemos estar **orgullosos**.

USABILIDAD

Nuestra interfaz ha sido diseñada con la **usabilidad como prioridad**. Todas las decisiones tomadas a lo largo del desarrollo de esta han sido tomadas bajo este criterio.

Queríamos aunar lo positivo que tiene una interfaz simple para algunos usuarios, con la sensación de satisfacción que para otros usuarios supone el tener un mayor control**.** Surge así nuestro planteamiento de un flujo divergente en el uso de la aplicación. Los usuarios que quieran una interfaz sencilla la tendrán, pero a la vez, intentamos satisfacer a aquellos usuarios que prefieran realizar cambios más avanzados.

Teniendo en cuenta las limitaciones que la herramienta empleada para diseñar la interfaz nos ha impuesto, hemos dado este control de manera opcional, para que un usuario casual no se sienta abrumado.

El uso de la interfaz está optimizado de tal modo que con **un solo clic** el programa calcula la visibilidad; es decir, el usuario puede simplemente ver cómo su proyecto es procesado sin tener que involucrarse, pero, de querer hacerlo, puede quitar un poco de control al algoritmo y hacer el mismo la selección de oclusores.

Para comenzar, la interfaz se encuentra disponible tanto en **español** como en **inglés**, para poder llegar al máximo número de usuarios y que no haya ningún problema por la diferencia de idiomas (véase Figura 45 y Figura 46).

Figura 45.- Ejemplo de la interfaz en inglés.

Figura 46.- Ejemplo de la interfaz en español.

Se ha buscado ofrecer **ayuda** al usuario de manera constante, **integrada** en la interfaz, pero de manera **no intrusiva**. Para esto, se ha pensado que la mejor manera de hacerlo sería con mensajes emergentes (véase Figura 47).

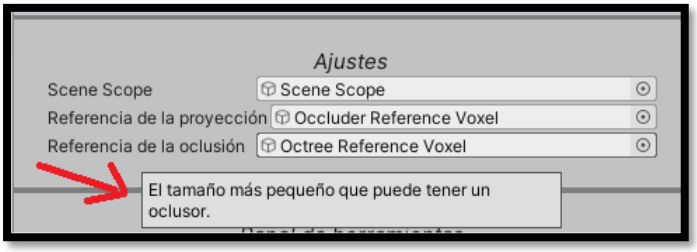

Figura 47.- Mensaje emergente.

Estos mensajes muestran una breve explicación acerca de lo que hace cada elemento de la interfaz, y solo son visibles si el ratón se queda encima de los elementos para que aparezca este mensaje.

Creemos que esta es una manera muy **intuitiva** de presentar la ayuda, sin tener que recurrir a documentación externa. Unity ya integra de manera nativa muchos elementos que cuentan con mensajes similares; por lo que esta práctica no será desconocida para el usuario.

Los nombres de los elementos de la interfaz son lo más intuitivos posibles, siendo quizás el más opaco *"Scene Scope"*, pero este término es explicado con su correspondiente mensaje emergente; además, este parámetro, al igual que los otros dos GameObjects del panel de ajustes, no tienen por qué ser cambiados, tan solo escalados.

Hemos intentado mostrar de manera clara qué se puede hacer en cada momento, para esto, no hemos bloqueado ningún botón de la interfaz, salvo por imperativo de la ejecución, es decir, exclusivamente en el caso de que falte algún GameObject del panel de ajustes. En este caso, se bloquea el botón "Procesar escena"**.** Aun así, de ser este el caso, se muestra un mensaje de aviso flotante. Es decir, prevemos el error y no dejamos que el usuario lo cometa. (véase Figura 48)

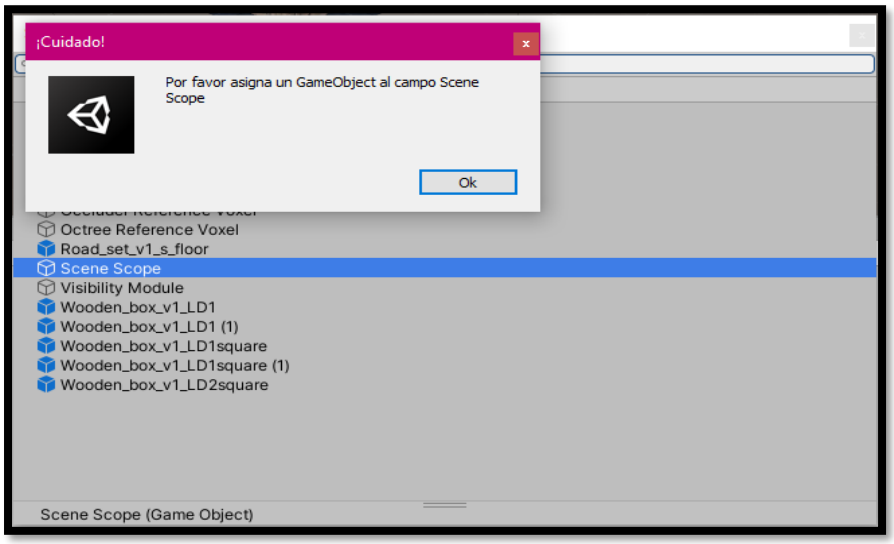

Figura 48.- Mensaje de confirmación.

El que el panel de herramientas tenga una visibilidad por demanda, se debe también a esto: cuando la herramienta está activa aparecen las opciones del panel de herramientas, pero solo porque están disponibles.

Por el mismo motivo, el botón Activar-Desactivar *Octree* solo aparece si existe un *octree*, de no existir, es **invisible**.

De nuevo, solo mostramos las opciones disponibles. No queremos que el usuario tenga que preocuparse de qué puede o no puede hacer. Todas las opciones que están disponibles en un determinado momento se muestran, ocultando aquella que no lo están.

La interfaz es adaptativa o ***responsive***. De esta forma, el usuario puede emplear el espacio que considere oportuno para ver la escena. Asimismo, el espacio restante es suficiente para mostrar los elementos de forma clara y concisa. La interfaz se comprime, adoptando una colocación vertical. Tal y como se puede ver en las imágenes a continuación: Figura 49 y Figura 50.

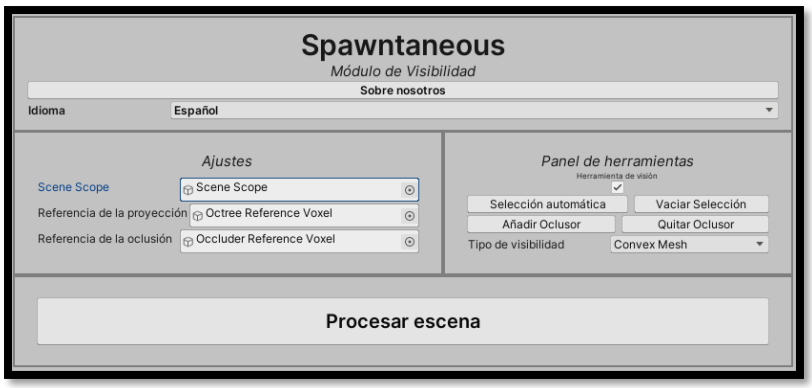

Figura 49.- Interfaz expandida horizontalmente.

Figura 50.- Interfaz contraída verticalmente.

El orden de colocación de los botones corresponde al flujo normal de ejecución del programa. Queremos dejar claro que el panel de herramientas es, en principio, opcional**.** Si bien es obligatorio para trabajar con el resultado. En el caso de que la interfaz tenga espacio suficiente, esto es más representativo, ya que, en lugar de encontrarse como un paso intermedio opcional, aparece como un paso a la derecha. Es decir, como un camino **alternativo**.

Dentro del panel de herramientas encontramos seis opciones: las que se ejecutan de manera automática y reescriben la selección de oclusores se encuentran las primeras. En segundo lugar, se encuentran los botones que modifican la selección de manera manual.

En tercer lugar, activar o desactivar el *octree*, y, finalmente, la elección de cómo queremos ver nuestra visibilidad (véase Figura 51).

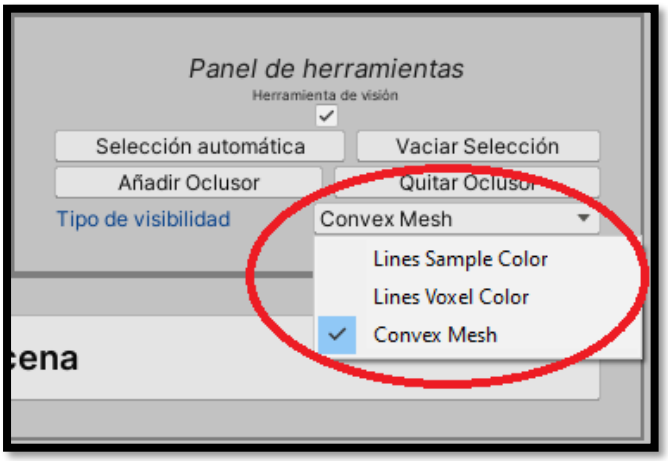

Figura 51.- Opciones de visibilidad.

A continuación, se explicará el orden de los instrumentos del panel de herramientas:

Un usuario casual, primero seleccionará de manera automática los oclusores, tras esto, puede que quiera modificar de manera manual esta selección. No tiene sentido hacer una selección manual para que luego se pierda debido a la selección automática. Sin embargo, estos dos botones también comprueban que no han sido pulsados por error, detectando cuando hay algún oclusor seleccionado, y pidiendo confirmación al usuario sobre lo que quiere hacer (véase Figura 52).

Tras esto se encuentra el botón de activar/desactivar *octree*, esto se hará una vez se haya procesado la escena, no tiene sentido que se encontrara antes.

El último elemento es por **lógica** el tipo de visibilidad elegida, ya que es la forma de visualizar los resultados finales; es lo último que haces en el flujo de ejecución.

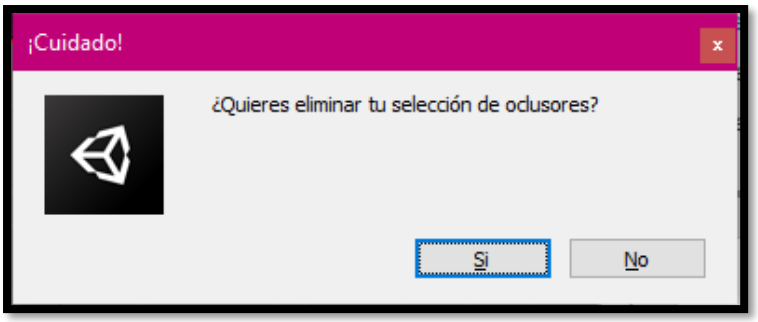

Figura 52.- Mensaje de confirmación del usuario para vaciar su selección.

Creemos que con tres paneles la interfaz es lo suficientemente profunda como para ofrecer todas las posibilidades disponibles, pero no para desbordar un solo panel y agrupar todas las opciones en el mismo sitio.

Tampoco supone una interfaz complicada en la que se tenga que comprobar cinco o seis sitios para encontrar lo que se quiere hacer**.**

De hecho, el usar paneles aporta otra ventaja y es que nos ayuda a encapsular los elementos afines. Los objetos que constituyen los ajustes se encuentran juntos, del mismo modo que todas las herramientas.

Si bien es cierto que el tercer panel cuenta con una dualidad de elementos, aparece un botón y tres barras de progreso. Sin embargo, están directamente relacionadas, ya que el pulsar el botón genera esas barras de progreso.

Finalmente, hemos creado una ventana muy simple que sirve para que los usuarios comprueben la visibilidad generada, y así se cercioren de que los cálculos de nuestro algoritmo son correctos. Esta ventana contiene una breve explicación sobre la prueba (véase Figura 53).

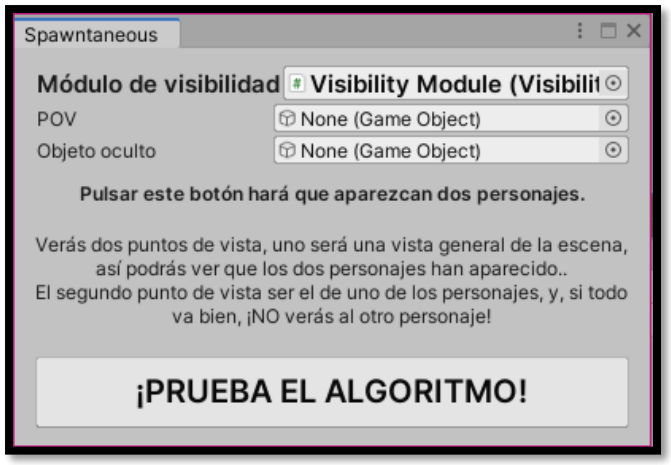

Figura 53.- Ventana de la prueba del algortimo.

Para usuarios conocedores del programa hemos incorporado **atajos de teclado**, actualmente solo existen dos: Uno para crear el objeto de visibilidad (**control +** `) y otro para crear la ventana que para realizar pruebas (**tecla de alternativa +** `).

Estos atajos de teclado están pensados para un **teclado internacional**, en esa disposición, pueden ser activados de manera cómoda con la mano izquierda, dejando la mano derecha disponible para el uso del ratón.

Para dotar de una **coherencia estética** a la herramienta con el resto del entorno de Unity, no se han generado hojas de estilo, sino que, se ha dejado de tal modo que la estética de la herramienta cambia acorde a la de Unity (véase Figura 54).

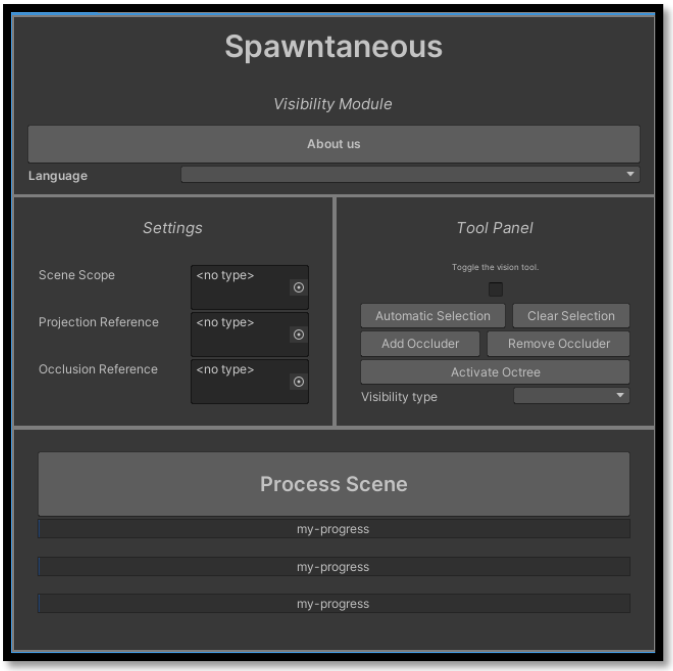

Figura 54.- Preview de la interfaz usando el modo oscuro de Unity.

La herramienta da visibilidad de su estado de ejecución, mientras se está ejecutando, aparecen unas barras de progreso que muestran el trabajo del programa. Cuando este acaba, muestra un mensaje flotante al usuario informando de ello.

En la vena de proteger al usuario de los errores que este pueda cometer, cuando se detecta que la carga computacional elegida es demasiado alta, se pide confirmación al usuario de si quiere procesar la escena, ya que puede tardar en exceso.

Finalmente, cabe comentar que la herramienta muestra un indicador visual de cuando está activa su funcionalidad: el modo de renderizado pasa a ser *"Shaded Wireframe"*. El inspector se bloquea en la herramienta, para no perder de vista las opciones.

RESULTADOS

Spawntaneous es una herramienta auxiliar para el desarrollo de videojuegos, su función principal es servir de soporte para los diseñadores de niveles en varios aspectos. La aplicación desarrollada se encarga de procesar una sección del espacio de cierta escena para calcular y representar de una manera clara la visibilidad de esta.

En nuestro trabajo, interpretamos como visibilidad la abstracción matemática de la real, esta dice lo siguiente: dado un conjunto de obstáculos en un espacio (en este caso tridimensional), se dice que dos puntos son visibles si la línea que los une no se corta con ningún obstáculo [13].

Para poder realizar los cálculos de la visibilidad de una escena nos inspiramos en la aproximación postulada en el documento ***"The visibility octree: a data structure for 3D navigation" C. Saona-Vázquez, I. Navazo, P. Brunet*** [5]. Parte de su investigación demuestra que es posible calcular la sombra producida por un conjunto convexo desde un poliedro teniendo en cuenta sólo la producida desde cada uno de los vértices del poliedro y haciendo su intersección. Este teorema nos permite reducir de manera significativa el número de cálculos a realizar, ya que nos permite calcular una buena aproximación de la visibilidad de una escena.

Debido a la necesidad representar la escena con un conjunto de poliedros, utilizamos un *octree* para que divida la escena de manera ordenada y regular. La herramienta muestra de manera clara los resultados de los cálculos, y el usuario puede usarlos como base para construir un sistema de reaparición, o para comprobar si sus métodos de reaparición existentes son correctos.

Para el desarrollo de este trabajo, se ha tenido en cuenta el rendimiento y la supervivencia de este hasta su posible obsolescencia. Para garantizar ambos, se ha desarrollado usando la tecnología **Data-Oriented Technology Stack**, lo que nos ha permitido producir una aplicación muy optimizada y novedosa.

La interfaz de usuario también se ha programado en base a una tecnología en desarrollo, **UIElements** [21]. Esta clase también es nueva y las interfaces producidas con ella son más eficientes. Su uso significa aún más nuestro compromiso con la innovación y el rendimiento en este trabajo.

El funcionamiento de nuestro programa puede dividirse en una serie de fases, a continuación, procederemos a explicar cada una de ellas:

Fase de editor:

1. El usuario debe instanciar el "*Visibility Module"* en su escena. Esto se consigue haciendo clic en el menú GameObject y seleccionando *Spawntaneous, Visibility Module* (véase Figura 3). Se dispone también de un atajo de teclado (control + `).

2. Se genera así un GameObject llamado ***"Visibility Module"***, este objeto contiene el *script* que maneja la funcionalidad de nuestro programa. También contiene tres GameObjects que sirven para configurar el funcionamiento del programa, aunque el papel de estos objetos puede ser jugador por cualquier GameObject que contenga el componente MeshRenderer.
   Estos son:
   a. Scene Scope: Sirve como volumen contenedor de la escena, todo lo que este dentro de este objeto se considera como espacio a procesar. El usuario debe cambiar su tamaño y debe colocarlo de tal modo que contenga en su interior aquella parte de la escena de la que desee calcular su visibilidad.
   b. Projection Reference: Las dimensiones de este objeto se usan para decidir las de los octantes proyectores. El usuario debe cambiar su tamaño para que se acople a sus necesidades; similarmente, podría usar el modelo del jugador como Projection Reference, lo que ajustaría automáticamente su envergadura.
   c. Occlusion Reference: Este objeto solo es considerado cuando la selección de oclusores se hace de manera automática, durante este evento, se seleccionan todos aquellos objetos que sean más grandes que este objeto. El usuario debe ajustar el tamaño de este objeto a sus necesidades, pudiendo usar el modelo de los enemigos para que se ajuste directamente.

3. Tras generar esos objetos, el usuario debe elegir los GameObjects que serán considerados como oclusores. Para realizar esta tarea, se dan dos opciones:
   a. Selección automática: Esta se da en dos vertientes, en una de ellas, el usuario elige hacerla activando el "Panel de Herramientas" y seleccionado la opción homónima. En la otra, esta selección se hace de manera transparente al usuario, ya que este pulsa directamente el botón "Procesar escena".
   b. Selección manual: Pueden añadirse o eliminarse GameObject haciendo uso de instrumentos dedicados para ello, localizados ambos dentro del "Panel de Herramientas". El funcionamiento de ambos es sencillo y semejante, tan solo hay que seleccionar la acción que queramos desempeñar y hacer clic sobre el objeto que deseemos.

   Una vez elegidos los oclusores, todos ellos se muestran dentro de un volumen envolvente (*Aligned-Axis Bounding Box*, AABB) [73], representado con cubos de color rojo (véase Figura 55).

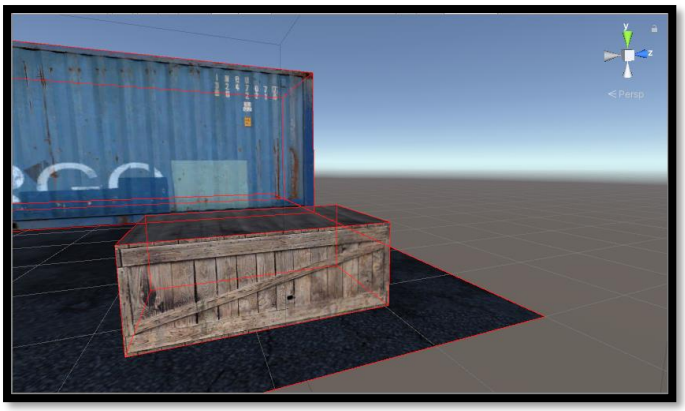

Figura 55.- Recubrimiento mínimo de GameObjects.

4. Habiendo ya elegido los oclusores, hay que pulsar el botón “Procesar escena” para que comiencen los cálculos. Entonces, se aplica una fórmula para saber cuántas generaciones del *octree* van a existir:

$$log(min(\frac{SceneScope.extents.x \cdot 2}{OctreeVoxel.extents.x \cdot 2}), (\frac{SceneScope.extents.y \cdot 2}{OctreeVoxel.extents.y \cdot 2}), (\frac{SceneScope.extents.z \cdot 2}{OctreeVoxel.extents.z \cdot 2})),2)$$

   Si se detecta que hay más de cinco generaciones, no se deja continuar. Esto es así porque actualmente hemos establecido como un límite seguro cinco generaciones y treinta oclusores.

Esto supone el fin de esta fase.

Fase de creación de entidades oclusoras:

Por cada oclusor se generan una serie de entidades que guardarán ciertos datos sobre él. En concreto surgen siete entidades por cada oclusor: una entidad que hace referencia al oclusor en sí, y seis entidades *“FaceVertex”,* las cuales guardan información de las caras del volumen contenedor del oclusor.

Así, la información almacenada de los oclusores es la siguiente:

a. Los vértices de cada cara.
b. Las normales de cada cara.
c. Los puntos centrales de cada cara.
d. La matriz que convierte sus coordenadas locales a globales.
e. El tamaño máximo en cada eje (*Bounds*).

Una vez creadas las entidades oclusoras, es el momento de generar el *octree* de la escena.

Fase de creación del *octree*:

1. Primero se genera una entidad que se corresponde con la raíz del *octree,* por motivos de claridad, la llamaremos *RootEntity*. En ella almacenaremos lo siguiente:
    a. Los **vértices** del volumen contenedor (*Scene Scope*).
    b. Los límites (***Bounds***) del volumen contenedor.
    c. Una referencia al "**padre**" a partir del cual haya sido generado. En esta entidad este valor está vacío, pero se incluye de igual manera para que este agrupada en los mismos bloques de memoria que el resto de las entidades del *octree*.
    d. Ocho referencias a sus **hijos**, una por cada uno.
    e. Una referencia a cada uno de los **oclusores** que no deba considerar para su visibilidad.
    f. Una colección que almacenará los oclusores que, si deba tener en cuenta, esto será usado únicamente por los octantes proyectores, pero se añade aquí para situar esta entidad en el mismo bloque de memoria.
    g. Una referencia a su visibilidad, aunque, al igual que en el elemento anterior, solo se da este componente para que esté en el mismo bloque de memoria que las demás.

2. Por razones del cálculo de visibilidad, se crea una entidad *RootInfoEntity* que posee cierta información de la raíz del *octree*, en concreto posee lo siguiente:
    a. La entidad *RootEntity*.
    b. Los límites (***bounds***) del volumen contenedor.
    c. Los **puntos centrales** de sus caras.
    d. La **normal** de sus caras.

3. Terminada ya la creación de dichas entidades, comenzamos con la generación de divisiones subsecuentes:
    a. Primero, hacemos una búsqueda (*query* [60]) para encontrar todas las entidades que tengan el componente *WorkTag*.
    b. Sobre cada entidad de la colección resultante de la búsqueda anterior, se realizan las siguientes operaciones:
        i. Se crean ocho "**hijos**", que en realidad son copias de la propia entidad.
        ii. Se vinculan poniendo a la entidad de la pila como padre de la entidad generada.
        iii. Se elimina el componente *WorkTag* de la entidad de la pila.
    c. Una vez realizadas esas operaciones, realizamos de nuevo una búsqueda para encontrar las entidades que tienen el componente *WorkTag*, y, en un hilo paralelo calculamos los siguientes valores:
        i. Su nuevo tamaño, que es la mitad del tamaño del padre.
        ii. Su nuevo punto central.
        iii. Sus nuevos vértices.
        iv. Los oclusores que este no deba tener en cuenta.

Este proceso se repite hasta la última generación, y, como puede denotarse de la descripción anterior, garantiza que los octantes proyectores pueden ser identificados porque aún conservan el componente *WorkTag*.

Finaliza así la creación del *octree*.

Fase de cálculo de la visibilidad:

1. Comenzamos creando las entidades de los ***Sample*** (entidad que contiene información de visibilidad). Para llevar esto a cabo, se debe hacer lo siguiente:

    a. Se recorre la lista de entidades que mantienen el componente *WorkTag*.
    b. Por cada entidad de esa lista se recorre su lista de oclusores.
    c. Por cada oclusor se recorre la lista de vértices de la entidad.
    d. Almacenamos en una entidad *Sample* la dupla resultante del vértice y el oclusor.

    Para mejorar el rendimiento de este proceso se utilizan técnicas de paralelismo. Se ha añadido un diagrama auxiliar para entender mejor el bucle anterior (véase Figura 56).

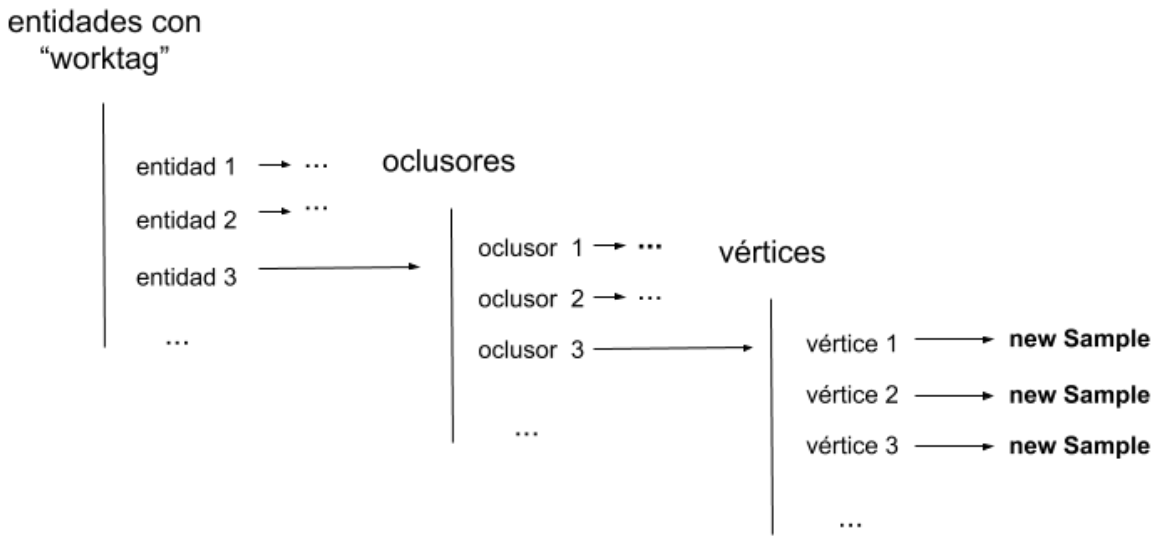

Figura 56.- Explicación del bucle de cálculos de visibilidad.

2. Debemos solventar el problema de los *Samples* **duplicados**, consideramos que la mejor opción es la de calcular la visibilidad en uno y **copiarla** en el resto de replicas. Para facilitar esta tarea, hemos colocado en los mismos fragmentos de memoria los *Samples* idénticos. Esto se hace de la siguiente manera:
    a. Primero debemos calcular la visibilidad de un *Sample*, y se hace así:
        i. Primero, averiguamos que planos son visibles desde el vértice proyecto.
        ii. Nos quedamos sólo con las aristas de la silueta del oclusor.

iii. Por cada una de ellas se generan dos rectas, una que surge de la unión del punto inicial de la arista con el vértice y otra que surge de la unión del punto final con el vértice (véase Figura 57).
iv. Hallamos las intersecciones de cada recta con los planos de las caras de "*RootInfoEntity*", de estas, solo guardamos la más cercana al proyector y que se encuentre en el sentido de la proyección.
v. Debemos calcular el conjunto de las intersecciones y puntos de las aristas, ya que esto constituye el ***frustum* sombra** de ese vértice.

b. Una vez calculado el de un *Sample*, se copia y se dan esos valores a todos sus duplicados.

3. Para calcular el *frustum* sombra de un oclusor para un proyector, debemos hacer la **intersección** de los *frusta* de sus vértices.

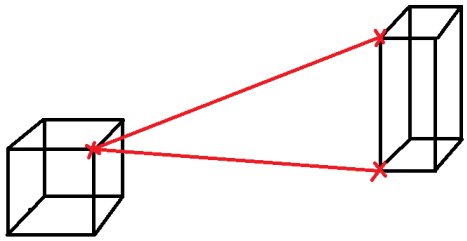

Figura 57.- Explicación de los resultados de un Sample.

Una vez generada la visibilidad de la escena, devolvemos el control al usuario y este recibe la posibilidad de navegar por el *octree*.

Fase de resultados:

En esta fase, el usuario puede ver los resultados de los cálculos realizados anteriormente, y adquiere la capacidad de navegación a través de nuestra partición espacial. A continuación, detallaremos cómo funciona la navegación.

1. Comenzamos en la raíz, aparecerá un cubo de color azul que abarca todo el volumen envolvente. Para avanzar, debemos hacer clic dentro de él.
2. Al hacer esto, se muestran sus ocho hijos. Si queremos seguir avanzando, debemos hacer clic dentro de alguno de ellos.
3. De hacerlo, se pintarán los hijos del octante en el que hayamos hecho clic, y desaparecerá el resto del *octree*. Se verá algo semejante a la Figura 58.
4. Una vez se llega a la máxima profundidad, se ven los resultados de la visibilidad para ese octante proyector. Si en algún momento quisiera volver atrás, solo debe hacer clic fuera de los octantes activos.
5. De hacer clic fuera de los octantes que aparecen en azul, volveremos hacia atrás, es decir, si estamos en un hijo, volveremos a su padre. En el caso de estar visualizando la visibilidad, dejaremos de verla y en su lugar, se nos mostrará el octante en el que estábamos y sus siete hermanos. Ver Figura 59 y Figura 58.

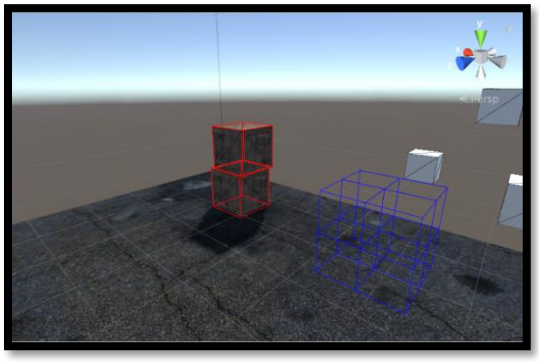

Figura 58.- Representación de resultados.

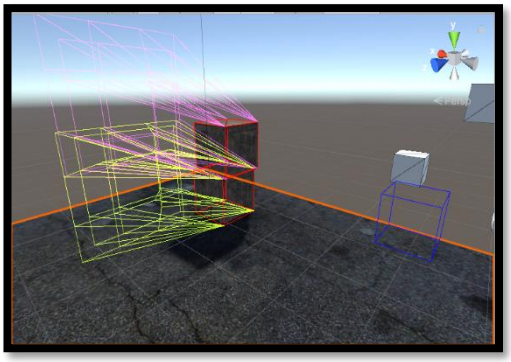

Figura 59.- Resultados de Visibilidad en una Escena

De estos resultados se puede afirmar lo siguiente: cualquier objeto situado dentro de la intersección de los *Samples* de un oclusor es invisible desde cualquier punto del octante proyector.

La aplicación cuenta con su propia interfaz, que ha sido diseñada siguiendo principios de usabilidad y se ha programado con **UIElements**. Está integrada en el inspector del **GameObject *Visibility Module***, y ofrece una funcionalidad muy sencilla, a la par que completa.

De desearlo, la ejecución del programa puede ser tan simple como colocar los **GameObjects *Scene Scope***, ***Occlusion Reference*** y ***Projection Reference*** y pulsar el botón de procesar. Sin embargo, podemos involucrarnos más utilizando las opciones disponibles en el panel de herramientas, estas nos permiten: añadir o quitar oclusores de manera manual, seleccionar automáticamente los oclusores, vaciar la lista de estos y seleccionar el tipo de representación de la visibilidad.

La visibilidad puede ser representada de tres modos: **Un color por cada *Sample*** (véase Figura 10), **una malla por cada *Sample*** (véase Figura 11), y, por último, **un color por cada oclusor** (véase Figura 12).

La interfaz pone a disposición del usuario técnicas para que la experiencia de usuario sea placentera, entre otras:

- El usuario puede seleccionar su idioma (en principio, español e inglés).
- Tiene explicaciones emergentes de lo que es cada elemento del programa.
- Además, se cuenta con una serie de comprobaciones que impiden la ejecución de errores del usuario en el programa.

Para demostrar todo nuestro trabajo hemos diseñado una pequeña **prueba** que sitúa dos objetos en la escena. Nos permite comprobar desde el punto de vista de uno de ellos que el otro objeto no es visible. También, vemos un plano general para demostrar que los dos objetos están en la escena al mismo tiempo. Esta prueba es accesible por una ventana adicional (que también cuenta con un atajo de teclado dedicado).

Como nota final, los resultados obtenidos sirven para calcular la visibilidad de un determinado entorno tridimensional, y estos cálculos pueden ser aplicados por los desarrolladores para:

- Comprobar la colocación de elementos en su escena que tengan un carácter oculto, tales como secretos, coleccionables, etc.
- Pueden apoyarse en la visibilidad para comprobar que su sistema de reaparición es correcto. Siempre y cuando se ejecute en el editor.
- Pueden construir un sistema de reaparición en torno a la visibilidad obtenida. Aunque para esto debe conseguir una permanencia de los resultados.
- Estudiar elementos del escenario tales como pueden ser coberturas.

CONCLUSIONES

De usar los paquetes Entity Component System (ECS), Jobs y Burst se desprenden una serie de conclusiones:

1. **ECS** fue el paquete más **problemático**, debido a que sigue en desarrollo, la documentación existente es muy laxa. Y sus problemas no terminan ahí, sino que aún carece de numerosas operaciones que fueron prometidas por los desarrolladores. Debido a esa falta, en ocasiones nos hemos visto obligados a usar las alternativas disponibles que, en lugar de aumentar el rendimiento, lo disminuyen.
   En retrospectiva, este paquete nos ha supuesto las complicaciones de:
   a. Aprendizaje autodidacta, la documentación invitaba a la investigación.
   b. Renovación constante de conocimientos. Los estándares iban cambiando.
   c. Búsqueda de alternativas a operaciones no soportadas.
   En contraposición, presenta la gran **ventaja** de que la mayoría del paquete aporta un rendimiento mucho más superior al del clásico MonoBehaviour, y, debido a su estructuración de los datos, facilita el desarrollo multihilo.

2. **Jobs*,*** al ser el más longevo de los tres paquetes, no nos dio ningún problema. De hecho, nos aportó una gran **ventaja**, y es que cuenta con mecanismos automáticos para proteger los datos en operaciones concurrentes y evita las temidas condiciones de carrera [38].

3. **Burst** nos presentó un único **problema**: la falta de compatibilidad con numerosas instrucciones de programación. Lo que obliga a prescindir de él en varias ocasiones, y, por lo tanto, disminuye el rendimiento. Unity promete ampliar la compatibilidad de este compilador, pero actualmente no tenemos forma de exprimirlo al máximo.

Los objetivos planteados al principio del trabajo han sido cumplidos; a continuación, se detallan las conclusiones obtenidas respecto a los objetivos.

1. **Aprovechar la tecnología Data-Oriented Technology Stack (DOTS) de Unity para asegurar la eficiencia del software.**
   Tal como se dijo al principio, este objetivo implica la necesidad de obtener un gran rendimiento en la aplicación y, a la par, obtener conocimiento en un campo desconocido.
   Primero abarcaremos la necesidad de rendimiento: al usar DOTS hemos conseguido desarrollar un programa altamente optimizado, llegamos a generar un gran número de entidades y realizamos una mayor cantidad de cálculos sobre ellas. Estos números en ocasiones pueden llegar a casi un millón de entidades, y los cálculos desbordan con creces el millón; aun así, los tiempos de ejecución no son excesivamente altos.
   Respecto al crecimiento personal, hemos obtenido **conocimientos** en un campo que nos era completamente desconocido. Esto nos ha situado por delante de

muchos desarrolladores que se verán obligados a aprender este patrón de diseño en el futuro para poder mantenerse competitivos [12].
En nuestra opinión este objetivo ha sido **cumplido.**

2. **Dar al usuario una herramienta auxiliar para el desarrollo en Unity.**
   El producto conseguido debía tener una buena usabilidad y debía ser útil para el desarrollo de sistemas de reaparición. Nuestra aplicación permite el análisis de la visibilidad de una escena, y, efectivamente, aporta esos cálculos como elemento auxiliar al desarrollador. También, en lo que se refiere a usabilidad, contamos con una serie de características que la garantizan, como son: una interfaz adaptativa, sencilla y bilingüe, además de unos resultados comprensibles y unos controles intuitivos.
   Sin embargo, respecto a este objetivo en particular, hay una serie de **puntos negativos**:
   a. No existe permanencia de resultados.
   b. No puede ser usado en *Play Mode*.

   Todo ello se debe a que, desde un primer momento, enfocamos el programa al Editor y no consideramos que las pruebas pudieran ser realizadas en *Play Mode*
   A pesar de ello, un usuario puede comprobar eventos, puntos de reaparición e incluso el propio sistema de reaparición siempre y cuando lo ejecute en el Editor.
   Consideramos este objetivo como **cumplido en su mayor parte.**

3. **Desarrollar una herramienta que permita el cálculo de la visibilidad de un espacio dado.**
   El programa debía ser capaz de calcular la visibilidad de todos los puntos posibles de una escena. La aplicación permite realizar esto, aunque esta capacidad puede verse truncada por las limitaciones de memoria, ya que, en ciertas ocasiones debido a la gran cantidad de cálculos, el programa podía saturarse. Sin embargo, una escena excesivamente grande y compleja puede ser procesada si esta se subdivide en varios módulos, por lo que este objetivo ha sido **cumplido.**

Finalmente, debemos indicar que el diseño de la aplicación siguió y cumplió con todos y cada uno de los requisitos funcionales y no funcionales.

La comunicación entre los miembros del TFG fue fluida, lo que nos permitió proporcionarnos tanto material como demás consejos o trucos de DOTS. Esto resultó vital para el desarrollo del trabajo. Esto se organizó mediante metodologías ágiles que nos ayudaron a mantenernos en contacto y mediante conversaciones informales y de tecnologías de comunicación unificadas

En resumen, creemos haber cumplido la mayoría de los objetivos de los que partimos al comenzar el trabajo, y nos encontramos contentos con el resultado.

LÍNEAS FUTURAS

1. Queremos conseguir pintar correctamente la intersección de los *frusta* generados por los *Samples* de un poliedro hacia un oclusor. La **técnica *Constructive Solid Geometry***, de aquí en adelante **CSG**, es una técnica que permite la creación de modelos malla complejos a partir de otros modelos realizando las **operaciones booleanas** tales como **intersección**, diferencia, unión, etc. Existen algunas herramientas y librerías (API) en C# que otorgan esta funcionalidad en Unity; tal es el caso de la herramienta **ProBuilder** [74], queactualmente en una versión muy temprana, o **pb_csg** [75], que es una librería a partir de la cual se construye la herramienta anterior.

   En nuestra implementación llegamos a obtener las mallas de los *Samples* sin llegar a hacer su intersección a nivel lógico, mostrar dicha intersección podría facilitar al usuario el entendimiento de lo que está viendo y un resultado mejor. Sin embargo, las librerías **CSG** actualmente disponibles aún se encuentran en un estado demasiado precario con gran cantidad de problemas.

2. El compilador **Burst** aún **se encuentra en proceso de desarrollo** y es incapaz de procesar ciertas operaciones que se realizan a nivel de código, como consecuencia resulta imposible aprovechar su rendimiento en algunos momentos. En un futuro, pretendemos actualizar el código a la par que Burst empiece a aceptar ciertas instrucciones. Mejorando aún más el rendimiento de nuestra aplicación.

3. Ampliar la funcionalidad de la aplicación para que sea compatible con *Play Mode*. Además de conseguir una permanencia de los resultados del cálculo de la visibilidad al pasar a *Play Mode*. Los datos podrían ser guardados en algún fichero o base de datos, de esta forma la escena podría ser relanzada sin necesidad de realizar los cálculos de visibilidad otra vez.

4. Mantenemos el compromiso de que ***Spawntaneous* sea potencialmente una herramienta mucho más amplia constituida por módulos autocontenidos.**

5. Mejorar la herramienta para facilitar al usuario la creación de sistemas de reaparición basados en exclusiva en *SpawnTaneous*. Actualmente se dan recursos que requieren de cierto esfuerzo por parte del desarrollador para aplicarlo.

6. Implementar una integración mayor con la librería **Hybrid Renderer**. Su primera versión ya no se encuentra en desarrollo, pero ofrece mayor funcionalidad en su versión segunda. En un futuro nos interesaría hacer una migración a esa segunda versión y además implementar toda la parte gráfica usando **Hybrid Renderer** para conseguir un mayor rendimiento.

7. Actualizar el código del programa para que funcione con **Unity 2020** y las nuevas versiones de DOTS. Hemos alcanzado el límite superior de DOTS en Unity 2019 y las versiones más recientes presentan mejoras en el rendimiento a la par que diferencias en la implementación del código. En principio nuestra implementación no queda obsoleta, ya que los métodos que iban a ser deprecados aparecían como tales en la documentación.

La finalidad de esta actualización sería conseguir nuevas funcionalidades prometidas por los desarrolladores.

8. Actualizar a versiones posteriores de **UIElements** (disponibles en Unity 2020). Las nuevas versiones de UIElements implementan elementos ya existentes en GUILayout, y permiten desarrollar una interfaz mejor.

9. Aún hay errores en la interfaz. Las barras de progreso actualmente son más bien para que el usuario pueda ver algo, no muestran el progreso sino más bien el resultado. En el futuro nos gustaría tener alguna forma de que ese código pudiera repintar la interfaz mientras se realizan los cálculos de visibilidad.

10. El tamaño máximo de generaciones de un *octree* no se corresponde con la capacidad real de la escena, y en el futuro nos gustaría tener alguna forma de calcular el tiempo que tardaría en procesar esa escena para poder avisar al usuario.

11. Actualmente la aplicación se encuentra alojada en un repositorio público de Github, sin embargo, nos gustaría que estuviese también disponible en la plataforma OfiLibre (https://ofilibre.gitlab.io/).

LICENCIA Y CÓDIGO

Licencia seleccionada

Reconocimiento-CompartirIgual 4.0 Internacional

¡Esta es una licencia de Cultura Libre!

Spawntaneous by Luis Miguel Moreno López, Ángel Noguero Salgado and Daniel Palacios Alonso is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Based on a work at https://github.com/aaangell/Spawntaneous.

Código disponible en https://bit.ly/3dOPbfT

BIBLIOGRAFÍA

[1] «Spawning (video games)», *Wikipedia*. ene. 23, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Spawning_(video_games)&oldid=937250335.

[2] M. Booth, «The AI Systems of Left 4 Dead», p. 95.

[3] «Gamasutra - Examining Game Pace: How Single-Player Levels Tick». https://www.gamasutra.com/view/feature/132415/examining_game_pace_how_.php=?print=1 (accedido oct. 17, 2020).

[4] I. Pantazopoulos y S. Tzafestas, «Occlusion Culling Algorithms: A Comprehensive Survey», *J. Intell. Robot. Syst.*, vol. 35, pp. 123-156, oct. 2002, doi: 10.1023/A:1021175220384.

[5] C. Saona-Vázquez, I. Navazo, y P. Brunet, «The visibility octree: a data structure for 3D navigation», *Comput. Graph.*, vol. 23, n.º 5, pp. 635-643, oct. 1999, doi: 10.1016/S0097-8493(99)00087-4.

[6] Unity Technologies, «DOTS, el nuevo conjunto de tecnología orientada a los datos multiproceso de Unity». https://unity.com/es/dots (accedido jun. 18, 2020).

[7] «Multihilo», *Wikipedia, la enciclopedia libre*. oct. 22, 2019, Accedido: oct. 16, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Multihilo&oldid=120649334.

[8] Unity Technologies, «Entity Component System | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/index.html (accedido oct. 14, 2020).

[9] Unity Technologies, «Unity - Manual: What is a job system?» https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemJobSystems.html (accedido oct. 12, 2020).

[10] *Getting started with Burst - Unite Copenhagen*. 2019.

[11] «Entity component system», *Wikipedia*. may 23, 2020, Accedido: jun. 18, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Entity_component_system&oldid=958452302.

[12] Unity Technologies, «What is DOTS and why is it important?», *Unity Learn*. https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important (accedido oct. 12, 2020).

[13] «Visibility (geometry)», *Wikipedia*. feb. 21, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Visibility_(geometry)&oldid=941875012.

[14] «Visual Studio Code», *Wikipedia, la enciclopedia libre*. oct. 11, 2020, Accedido: oct. 14, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=129985434.

[15] «Unity (motor de videojuego)», *Wikipedia, la enciclopedia libre*. oct. 04, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Unity_(motor_de_videojuego)&oldid=129792359.

[16] «OpenGL Overview». https://www.opengl.org/about/ (accedido sep. 15, 2020).

[17] «Direct3D», *Wikipedia, la enciclopedia libre*. jun. 19, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Direct3D&oldid=127057651.

[18] Unity Technologies, «DOTS Hybrid Renderer | Hybrid Renderer | 0.8.0-preview.19». https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.8/manual/index.html (accedido oct. 12, 2020).

[19] «Job (computing)», *Wikipedia*. abr. 26, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Job_(computing)&oldid=953282874.

[20] Unity Technologies, «Unity - Manual: UI Builder». https://docs.unity3d.com/Manual/com.unity.ui.builder.html (accedido oct. 20, 2020).

[21] *UIElements, a new UI system for the editor - Unite LA*. 2018.

[22] U. Technologies, «Unity - Manual: Important Classes - MonoBehaviour». https://docs.unity3d.com/Manual/class-MonoBehaviour.html (accedido oct. 19, 2020).

[23] Unity Technologies, «Unity - Manual: The UXML format». https://docs.unity3d.com/Manual/UIE-UXML.html (accedido sep. 09, 2020).

[24] Unity Technologies, «Unity - Manual: Styles and Unity style sheets». https://docs.unity3d.com/Manual/UIE-USS.html (accedido oct. 16, 2020).

[25] «Introduction - - SourceGear DiffMerge». https://sourcegear.com/diffmerge/webhelp/chapter_introduction.html (accedido oct. 16, 2020).

[26] «3d Math functions - Unify Community Wiki». https://wiki.unity3d.com/index.php/3d_Math_functions (accedido oct. 16, 2020).

[27] «MIConvexHull». https://designengrlab.github.io/MIConvexHull/ (accedido sep. 18, 2020).

[28] «Convex hull algorithms», *Wikipedia*. oct. 11, 2020, Accedido: oct. 13, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Convex_hull_algorithms&oldid=982947094.

[29] «Creating a convex MESH (not collider) - Unity Answers». https://answers.unity.com/questions/983425/creating-a-convex-mesh-not-collider.html (accedido oct. 03, 2020).

[30] Unity Technologies, «Unity - Manual: Asset packages». https://docs.unity3d.com/Manual/AssetPackages.html (accedido oct. 12, 2020).

[31] Unity Technologies, «Paquetes DOTS | Unity». https://unity.com/es/dots/packages (accedido oct. 05, 2020).

[32] Unity Technologies, «ECS concepts | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/ecs_core.html (accedido oct. 11, 2020).

[33] «Struct EntityManager | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/api/Unity.Entities.EntityManager.html (accedido oct. 16, 2020).

[34] Unity Technologies, «Components | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/ecs_components.html (accedido oct. 11, 2020).

[35] Unity Technologies, «Systems | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/ecs_systems.html (accedido oct. 11, 2020).

[36] Unity Technologies, «Unity - Manual: C# Job System Overview». https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemOverview.html (accedido oct. 12, 2020).

[37] Unity Technologies, «Unity - Manual: What is multithreading?» https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemMultithreading.html (accedido oct. 12, 2020).

[38] Unity Technologies, «Unity - Manual: The safety system in the C# Job System». https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemSafetySystem.html (accedido oct. 12, 2020).

[39] «Dependencia de datos», *Wikipedia, la enciclopedia libre*. jul. 10, 2019, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Dependencia_de_datos&oldid=117316057.

[40] «Bloqueo mutuo», *Wikipedia, la enciclopedia libre*. feb. 12, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Bloqueo_mutuo&oldid=123492485.

[41] «Blittable types», *Wikipedia*. dic. 17, 2019, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Blittable_types&oldid=931206358.

[42] «Octree», *Wikipedia, la enciclopedia libre*. sep. 26, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Octree&oldid=129562505.

[43] «Navigation mesh», *Wikipedia*. jun. 10, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Navigation_mesh&oldid=961710679.

[44] «Partición binaria del espacio», *Wikipedia, la enciclopedia libre*. ago. 02, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Partici%C3%B3n_binaria_del_espacio&oldid=128190127.

[45] «Hidden-surface determination», *Wikipedia*. ago. 26, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Hidden-surface_determination&oldid=975071898.

[46] «Creating gameplay | Entities | 0.7.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.7/manual/gp_overview.html (accedido oct. 07, 2020).

[47] «Game Object Conversion and SubScene», *Game Torrahod*, feb. 07, 2020. https://gametorrahod.com/game-object-conversion-and-subscene/ (accedido oct. 16, 2020).

[48] «Unity - Scripting API: MeshFilter». https://docs.unity3d.com/ScriptReference/MeshFilter.html (accedido oct. 16, 2020).

[49] Unity Technologies, «Unity - Scripting API: Transform». https://docs.unity3d.com/ScriptReference/Transform.html (accedido oct. 16, 2020).

[50] U. Technologies, «Unity - Scripting API: Handles». https://docs.unity3d.com/ScriptReference/Handles.html (accedido oct. 16, 2020).

[51] «World | Package Manager UI website». https://docs.unity3d.com/Packages/com.unity.entities@0.0/manual/world.html (accedido oct. 12, 2020).

[52] «Struct BufferFromEntity<T> | Package Manager UI website». https://docs.unity3d.com/Packages/com.unity.entities@0.0/api/Unity.Entities.BufferFromEntity-1.html (accedido oct. 16, 2020).

[53] U. Technologies, «Unity - Scripting API: JobHandle». https://docs.unity3d.com/ScriptReference/Unity.Jobs.JobHandle.html (accedido oct. 16, 2020).

[54] Unity Technologies, «Dynamic Buffers | Package Manager UI website». https://docs.unity3d.com/Packages/com.unity.entities@0.0/manual/dynamic_buffers.html (accedido oct. 12, 2020).

[55] Unity Technologies, «JobComponentSystem | Entities | 0.4.0-preview.10». https://docs.unity3d.com/Packages/com.unity.entities@0.4/manual/job_component_system.html (accedido oct. 12, 2020).

[56] «Tipo de dato lógico», *Wikipedia, la enciclopedia libre*. jul. 03, 2020, Accedido: oct. 16, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Tipo_de_dato_l%C3%B3gico&oldid=127456925.

[57] Unity Technologies, «Using Entity Command Buffers – Unite Copenhagen 2019», 01:13:32 UTC, Accedido: jun. 04, 2020. [En línea]. Disponible en: https://es.slideshare.net/unity3d/using-entity-command-buffers-unite-copenhagen-2019.

[58] Unity Technologies, «Class SystemBase | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/api/Unity.Entities.SystemBase.html (accedido oct. 12, 2020).

[59] Unity Technologies, «Class EntityCommandBufferSystem | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/api/Unity.Entities.EntityCommandBufferSystem.html (accedido oct. 12, 2020).

[60] Unity Technologies, «Using an EntityQuery to query data | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/ecs_entity_query.html (accedido oct. 12, 2020).

[61] «Class ComponentSystemBase | Package Manager UI website». https://docs.unity3d.com/Packages/com.unity.entities@0.0/api/Unity.Entities.ComponentSystemBase.html (accedido oct. 16, 2020).

[62] «Foreach», *Wikipedia, la enciclopedia libre*. abr. 16, 2020, Accedido: oct. 16, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Foreach&oldid=125249114.

[63] Unity Technologies, «Unity - Manual: Scheduling jobs». https://docs.unity3d.com/Manual/JobSystemSchedulingJobs.html (accedido oct. 12, 2020).

[64] «Using Entities.ForEach | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/ecs_entities_foreach.html (accedido oct. 12, 2020).

[65] «Expresión lambda», *Wikipedia, la enciclopedia libre*. oct. 14, 2020, Accedido: oct. 16, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Expresi%C3%B3n_lambda&oldid=130060798.

[66] Unity Technologies, «Unity - Scripting API: Bounds». https://docs.unity3d.com/ScriptReference/Bounds.html (accedido oct. 12, 2020).

[67] «C# Actions», *Unity Learn*. https://learn.unity.com/tutorial/c-actions (accedido oct. 20, 2020).

[68] «Tabla hash», *Wikipedia, la enciclopedia libre*. ago. 14, 2020, Accedido: oct. 16, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Tabla_hash&oldid=128468095.

[69] «Interface ISharedComponentData | Entities | 0.14.0-preview.19». https://docs.unity3d.com/Packages/com.unity.entities@0.14/api/Unity.Entities.ISharedComponentData.html (accedido oct. 16, 2020).

[70] Unity Technologies, «Unity - Scripting API: GUILayout». https://docs.unity3d.com/ScriptReference/GUILayout.html (accedido oct. 12, 2020).

[71] Unity Technologies, «Unity - Scripting API: EditorGUILayout». https://docs.unity3d.com/ScriptReference/EditorGUILayout.html (accedido oct. 16, 2020).

[72] Unity Technologies, «Unity - Manual: UQuery». https://docs.unity3d.com/Manual/UIE-UQuery.html (accedido oct. 16, 2020).

[73] «Bounding volume», *Wikipedia*. mar. 08, 2020, Accedido: oct. 12, 2020. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Bounding_volume&oldid=944488629.

[74] «ProBuilder». https://unity3d.com/es/unity/features/worldbuilding/probuilder (accedido oct. 20, 2020).

[75] «pb_CSG: A free boolean operations (CSG) library for Unity : Unity3D». https://www.reddit.com/r/Unity3D/comments/31jsfn/pb_csg_a_free_boolean_operations_csg_library_for/ (accedido oct. 20, 2020).