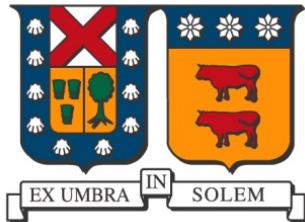


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



**“SISTEMA DE MANEJO DE ENTIDADES EN GODOT PARA
LA OPTIMIZACIÓN DE RECURSOS”**

RUBÉN ANTOINE RUIZ POMODORO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Sven von Brand L.

Profesor Correferente: ...

Diciembre - 2025

DEDICATORIA

A los desarrolladores que dedicaron esfuerzo y pasión para permitirme disfrutar de videojuegos a un precio que me hizo sentir que los estafé, por inspirarme a tomar este tema y aprender sobre algo que siempre me había interesado investigar.

AGRADECIMIENTOS

A mi pareja por siempre estar para ofrecer su apoyo, mi familia por esperar más de mí, mis amigos por incentivar me a seguir adelante y hasta mi jefe por recordarme los beneficios de sacar el título. Completar este proyecto propio ha sido difícil para mí y no lo habría conseguido sin su apoyo y la motivación que me daba cumplir con sus expectativas. Gracias.

RESUMEN

Resumen— Se creó un sistema de optimización de entidades en *Godot*, cuyo objetivo es ser una herramienta de aprendizaje para desarrolladores indie sobre las capacidades del motor de juegos en la implementación de *occlusion culling*, un sistema *HLOD*, *object pooling* y *renderizado por chunks* como formas de optimización, donde cada una corresponde a una experiencia distinta en el sistema, la cual está diseñada para mostrar cómo y cuándo es favorable su implementación. Para los resultados, se evaluó un caso control sin optimizar por cada técnica frente a su experiencia correspondiente, obteniendo que en promedio, las mejoras en el rendimiento para *FPS* y entidades respectivamente son: para *occlusion culling* un 57.05 % y un 88.33 %; para *HLOD* un 57.46 % y un 43.04 %; para *object pooling* un 42.49 % y un 92.46 %; y, finalmente, para *renderizado por chunks* un 64.15 % y un 60.82 %. Siendo lo anterior evidencia suficiente para determinar que la aplicación de cada técnica de optimización es correcta y efectiva.

Palabras Clave— Godot; FPS; Entidades; Optimización; Videojuegos.

ABSTRACT

Abstract— A system for the optimization of entities in *Godot* was created with the objective of being a learning tool for indie developers about the game engine's capabilities on the implementation of *HLOD* system, *object pooling* and *chunk rendering* optimization ways, where each one corresponds to a distinct experience in the system, which is designed to show how and when its implementation is favourable. For the results, a control case was evaluated without optimizing for each technique against its corresponding experience, obtaining that on average, the performance upgrades for *FPS* and entities respectively are: for *occlusion culling* a 57.05 % and an 88.33 %; for *HLOD* a 57.46 % and a 43.04 %; for *object pooling* a 42.49 % and a 92.46 %; and, lastly, for *chunk rendering* a 64.15 % and a 60.82 %. The former being sufficient evidence to determine that the application of each optimization technique is correct and effective.

Keywords— Godot; FPS; Entities; Optimization; Videogames.

GLOSARIO

AAA: modismo para describir a compañías y videojuegos de alto presupuesto y reconocimiento.

DI: Departamento de Informática.

Gamer: modismo para describir a jugadores de videojuegos.

Indie: desarrollador o equipo de desarrollo independiente y de bajos recursos.

NPC: personaje no jugable.

Publisher: entidad que se encarga de la publicación del videojuego en distintas plataformas.

Viewport: Pantalla en la cual el videojuego es proyectado.

Rasterización: es la tarea de tomar una imagen descrita en un formato de gráficos vectoriales (formas) y convertirla en una imagen rasterizada (una serie de píxeles , puntos o líneas que, al mostrarse juntos, crean la imagen que se representó mediante formas).

UTFSM: Universidad Técnica Federico Santa María.

Mesh: estructura geométrica 3D hecha de polígonos que conforman objetos.

Prop: utilería o accesorios utilizados para poblar una escena.

Frame: cuadro o imagen que muestra el estado del videojuego una vez que se han terminado de procesar todas las operaciones correspondientes a un instante.

Frames por segundo (FPS): unidad de medida de rendimiento que indica la cantidad de frames de juego que pueden ser procesados en un segundo.

Draw calls: de forma simplificada, es una llamada a la GPU con la información necesaria para dibujar algo en pantalla.

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	IX
ÍNDICE DE TABLAS	X
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Contexto	2
1.2 Problema	2
1.3 AAA vs indie	3
1.4 Solución propuesta	3
1.4.1 Objetivo general	3
1.4.2 Objetivos específicos	3
1.5 Impacto inicial de solucionar el problema	4
CAPÍTULO 2: MARCO CONCEPTUAL	5
2.1 Motores de juegos	5
2.1.1 Características de los motores populares	5
2.1.2 Motores de juego personalizados	6
2.2 Entorno de trabajo	6
2.2.1 Motores comerciales vs motor propio	7
2.2.2 Godot	7
2.2.3 Popularidad	7
2.2.4 Open Source	8
2.2.5 Licencia de Godot	8
2.2.6 Conceptos básicos del motor	9
2.3 Entorno 3D	9
2.3.1 Cámara	9
2.3.2 Cámara en Godot	10
2.3.3 Entidades	10
2.3.4 Entidades en Godot	10
2.3.5 Liberar y cargar recursos	11
2.4 Técnicas de optimización comunes	11
2.4.1 Fustrum culling	11
2.4.2 Cuándo aplicar Fustrum culling	11
2.4.3 Fustrum culling en Godot	12
2.4.4 Occlusion culling	12

2.4.5	Cuándo implementar occlusion culling	12
2.4.6	Occlusion culling en Godot	14
2.4.7	LOD	15
2.4.8	Cuándo implementar LOD	15
2.4.9	LOD en Godot	16
2.4.10	Object pooling	17
2.4.11	Cuándo aplicar object pooling	17
2.4.12	Object pooling en Godot	18
2.4.13	Gestión de entidades a través de data	18
2.4.14	Cuándo aplicar gestión de entidades a través de data	18
2.4.15	Gestión de entidades a través de data en Godot	18
2.4.16	Renderizado por chunks	19
2.4.17	Cuándo aplicar renderizado por chunks	20
2.4.18	Renderizado por chunks en Godot	20
2.5	Github	21
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN		22
3.1	Repositorio en Github	22
3.2	Proyecto en Godot	22
3.2.1	Assets	22
3.2.2	Menú	23
3.2.3	Búcle de juego	24
3.2.4	Player	24
3.2.5	Escena del mundo	25
3.2.6	Fustrum culling	25
3.2.7	Experiencia 1: Occlusion culling	25
3.2.8	Experiencia 2: HLOD	27
3.2.9	Experiencia 3: Object pooling	30
3.2.10	Experiencia 4: Renderizado por chunks	34
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN		37
4.1	Casos	37
4.1.1	Caso control 1: occlusion culling	38
4.1.2	Caso control 2: HLOD	39
4.1.3	Caso control 3: object pooling	42
4.1.4	Caso control 4: renderizado por chunks	44
4.1.5	Caso 1: occlusion culling	45
4.1.6	Caso 2: HLOD	48
4.1.7	Caso 3: object pooling	49
4.1.8	Caso 4: renderizado por chunks	52
4.2	Ánalisis de los datos recolectados	53
4.2.1	Análisis: Occlusion culling	55
4.2.2	Análisis: HLOD	55
4.2.3	Análisis: Object pooling	56

4.2.4 Análisis: Renderizado por chunks	56
CAPÍTULO 5: CONCLUSIONES	58
ANEXOS	63
REFERENCIAS BIBLIOGRÁFICAS	64

ÍNDICE DE FIGURAS

1	Popularidad de Godot después del anuncio de Unity.	8
2	Ejemplo de escena en <i>Godot</i>	13
3	<i>Wireframe</i> de la escena original.	13
4	<i>Wireframe</i> de la escena con <i>occlusion culling</i> activo.	14
5	<i>buffer</i> aplicado a una estructura en el editor de <i>Godot</i>	15
6	Ejemplo de <i>meshes LOD</i> generadas al importar un objeto en <i>Godot</i>	16
7	Ejemplo del <i>wireframe</i> de <i>meshes LOD</i> generadas al importar un objeto en <i>Godot</i>	17
8	Diagrama de un ejemplo del funcionamiento del <i>renderizado por chunks</i>	20
9	Representación del jugador utilizado en las experiencias.	25
10	Vista externa del mapa creado para la experiencia <i>occlusion culling</i>	27
11	Vista interna del mapa creado para la experiencia <i>occlusion culling</i>	27
12	Vista externa del mapa creado para la experiencia <i>HLOD</i>	29
13	Vista interna del mapa creado para la experiencia <i>HLOD</i>	29
14	Vista externa del mapa creado para la experiencia <i>object pooling</i>	32
15	Vista interna del mapa creado para la experiencia <i>object pooling</i>	33
16	Vista externa del mapa creado para la experiencia <i>renderizado por chunks</i>	35
17	Vista interna del mapa creado para la experiencia <i>renderizado por chunks</i>	36
18	FPS a lo largo del tiempo para caso control de <i>occlusion culling</i>	38
19	Cantidad de entidades a lo largo del tiempo para caso control de <i>occlusion culling</i>	39
20	<i>Draw calls</i> realizadas a lo largo del tiempo para caso control de <i>occlusion culling</i>	39
21	FPS a lo largo del tiempo para caso control de <i>HLOD</i>	40
22	Cantidad de entidades a lo largo del tiempo para caso control de <i>HLOD</i>	41
23	<i>Draw calls</i> realizadas a lo largo del tiempo para caso control de <i>HLOD</i>	41
24	FPS a lo largo del tiempo para caso control de <i>object pooling</i>	42
25	Cantidad de entidades a lo largo del tiempo para caso control de <i>object pooling</i>	43
26	<i>Draw calls</i> realizadas a lo largo del tiempo para caso control de <i>object pooling</i>	43
27	FPS a lo largo del tiempo para caso control de <i>renderizado por chunks</i>	44
28	Cantidad de entidades a lo largo del tiempo para caso control de <i>renderizado por chunks</i>	45
29	<i>Draw calls</i> realizadas a lo largo del tiempo para caso control de <i>renderizado por chunks</i>	45
30	FPS a lo largo del tiempo para caso de <i>occlusion culling</i>	46
31	Cantidad de entidades a lo largo del tiempo para caso de <i>occlusion culling</i>	47
32	<i>Draw calls</i> realizadas a lo largo del tiempo para caso de <i>occlusion culling</i>	47
33	FPS a lo largo del tiempo para caso de <i>HLOD</i>	48
34	Cantidad de entidades a lo largo del tiempo para caso de <i>HLOD</i>	49
35	<i>Draw calls</i> realizadas a lo largo del tiempo para caso de <i>HLOD</i>	49
36	FPS a lo largo del tiempo para caso de <i>object pooling</i>	50
37	Cantidad de entidades a lo largo del tiempo para caso de <i>object pooling</i>	51
38	<i>Draw calls</i> realizadas a lo largo del tiempo para caso de <i>object pooling</i>	51

39	FPS a lo largo del tiempo para caso de <i>renderizado por chunks</i>	52
40	Cantidad de entidades a lo largo del tiempo para caso de <i>renderizado por chunks</i>	53
41	<i>Draw calls</i> realizadas a lo largo del tiempo para caso de <i>renderizado por chunks</i>	53

ÍNDICE DE TABLAS

1	Comparación control vs <i>occlusion culling</i>	54
2	Comparación control vs <i>HLOD</i>	54
3	Comparación control vs <i>object pooling</i>	54
4	Comparación control vs <i>renderizado por chunks</i>	54

INTRODUCCIÓN

En el siguiente documento se discute el efecto que han tenido las mejoras en la tecnología en el contexto del desarrollo de videojuegos. Particularmente, en los problemas de optimización que se han producido debido a que el aumento en las capacidades de las máquinas ha permitido que las grandes empresas AAA cambien su enfoque de optimizar los videojuegos para generar un producto que alcance al mayor público posible, a tener los mejores gráficos y aumentar la rentabilidad, entregando productos incompletos para alcanzar fechas importantes como festividades. Por esta razón, se propone cambiar el enfoque a la escena indie, la cual ha entregado buenos productos consistentemente los últimos años. Específicamente, se propone el desarrollo de un sistema de manejo de entidades para la optimización de recursos como una herramienta de aprendizaje para facilitar a los desarrolladores indie los conocimientos sobre técnicas de optimización comunes y sus efectos en el desempeño.

El documento se encuentra dividido en cinco secciones principales, las cuales son la definición del problema, el marco conceptual, la propuesta de solución, la validación de la solución y las conclusiones, respectivamente. En la definición del problema, se procede a entregar el contexto en el que este ocurre, para luego explicarlo y describir a rasgos generales la solución propuesta, definiendo un objetivo general y una serie de objetivos específicos para determinar si la solución desarrollada cumple o no con ellos. Finalmente, se define cuál es el impacto esperado inicialmente al implementar la solución descrita. Pasando al marco conceptual, se introducen los conceptos importantes que son necesarios para la comprensión de la implementación del sistema propuesto. Estos conceptos van desde definir y explicar el motor de juegos elegido, el entorno utilizado para la implementación del sistema propuesto, las técnicas de optimización elegidas para la solución y la forma que se utilizará para hacer llegar la solución a los desarrolladores. Para la propuesta de solución, se explica cómo se desarrolló el sistema implementado como solución del problema, dando a conocer el repositorio creado para alojarlo y los elementos que lo componen. Dichos elementos van desde los assets utilizados, el bucle de juego definido, las escenas creadas y las experiencias diseñadas para cada técnica de optimización elegida. Luego se procede a la validación de la solución, donde se recolectan y analizan los datos obtenidos en el sistema creado. Se parte por enseñar las características del computador utilizado para las pruebas, además de definir las métricas utilizadas para medir el rendimiento en cada prueba. Luego se exponen las pruebas realizadas para cada técnica de optimización y se confeccionan gráficos y tablas para presentar los resultados, finalizando con un análisis del impacto en el rendimiento de cada forma de optimización implementada. Para finalizar, se exponen las conclusiones, donde se habla de la efectividad de las técnicas implementadas, el razonamiento detrás de las métricas elegidas y su utilidad, el cumplimiento de los objetivos propuestos y la relevancia del sistema creado en el problema original, finalizando con unas reflexiones personales sobre lo aprendido y la importancia del tema elegido.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. Contexto

Cuando se habla de optimización en el mundo gamer, en especial si nos centramos en los usuarios de computador, existe una preocupación importante por parte de los jugadores de maximizar el rendimiento que los equipos pueden proveer. Esto provocó que se abriera un mercado enfocado a jugadores, que se encargó de desarrollar nuevos componentes con especificaciones cada vez mejores, lo cual ha permitido a los desarrolladores empujar los límites de lo que es posible abarcar en un videojuego [GINX, 2023]. Desde mejoras gráficas, complejidad de mundos y mecánicas hasta cantidad de contenido y nivel de inteligencia de los NPC's, estos avances tecnológicos han masificado el mercado de videojuegos y permitido que muchos estudios de desarrollo, tanto AAA como indies, traigan innovación a la escena actual.

1.2. Problema

Sin embargo, los avances en la tecnología también trajeron consigo una baja en la calidad de los videojuegos en el mercado. En un comienzo, esto era algo que se esperaba de la escena indie, pero hoy en día son las grandes compañías AAA las que han abandonado un desarrollo óptimo en busca de maximizar la rentabilidad, aprovechándose de la potencia del hardware para que el videojuego sea “jugable” y seguir cobrando precios exorbitantes por lo que, esencialmente, es un producto mínimo viable [GINX, 2023]. Gran parte del problema viene de los Publisher [GINX, 2023], que, al buscar obtener mayores ganancias con fechas importantes como festividades, obligan a desarrolladores a publicar juegos incompletos [Nasu, 2023]. También está el hecho de que el acceso a internet ha permitido que videojuegos incompletos reciban parches y nuevo contenido meses después de su lanzamiento, “completando” así el producto [Nasu, 2023], mientras que gracias a las micro transacciones permiten que este modelo sea rentable [Strickland, 2023]. Esto representa un problema grave para los jugadores que no pueden invertir demasiado en hardware de última generación, provocando que muchos títulos no puedan ser jugados en máquinas low-end, provocando que, aparte de tener que pagar alrededor de CLP\$ 60.000 por el videojuego [Strickland, 2023], se necesite gastar cerca de CLP\$ 1.000.000 para obtener un computador capaz de ejecutarlo [Pawlik, 2023].

1.3. AAA vs indie

Es aquí donde los desarrolladores indie demuestran que es posible entregar un contenido similar de mejor calidad (mejor optimizado) por precios significativamente menores. Un caso comparativo actualmente relevante es el de los videojuegos Battlefield™ 2042 y BattleBit Remastered, en donde apreciamos cómo un videojuego indie con un equipo de sólo 3 personas logró, no solo entregar el mismo contenido que un título AAA, sino que, mientras este último fue mal recibido por no estar bien optimizado y lleno de errores, además no requiere de un equipo potente para ejecutarlo a pesar de entregar una escala similar de jugadores, recursos y mecánicas [GoingInside, 2023]. Llevando esta comparación al ámbito monetario, mientras que Battlefield 2042 tiene un precio actual de CLP\$ 46.900 en Steam [Steam, 2021] (en su lanzamiento el precio superaba los CLP\$ 60.000 [Castillo, 2021]), BattleBit Remastered tiene un precio de tan solo CLP\$ 11.700 [Steam, 2023].

1.4. Solución propuesta

Hoy en día, el problema de estudios AAA cobrando precio completo por videojuegos incompletos y mal optimizados es algo que difícilmente va a desaparecer, puesto que la industria actualmente soporta ese modelo de negocio [Strickland, 2023]. Es por esta razón que se vuelve de suma importancia dar apoyo y todas las herramientas necesarias a desarrolladores pequeños, quienes son los que están entregando en la actualidad productos de alta calidad [GoingInside, 2023], para que puedan entregar contenido a un amplio rango de jugadores a precios accesibles, sin la necesidad de mejorar las especificaciones de sus computadores para poder experimentar sus videojuegos correctamente. Por esta razón, se busca generar una experiencia que eduque a los desarrolladores en formas de optimización comunes y su uso, con el objetivo de que tengan más herramientas a su disposición al momento de trabajar en sus proyectos para lograr una mejor recepción en la comunidad.

1.4.1. Objetivo general

Se busca diseñar una experiencia en Godot que implemente un sistema de optimización para un manejo eficiente de los recursos utilizados por el videojuego.

1.4.2. Objetivos específicos

1. Definir lo que se conoce como entidades en un videojuego y su impacto en el manejo de recursos.
2. Estudiar métodos típicos de manejo de entidades y optimización.

3. Seleccionar métricas relevantes para medir el impacto que tienen las entidades en un computador.
4. Diseñar un sistema en Godot que maneje los recursos utilizados por las entidades.
5. Construir casos de prueba para los métodos de optimización utilizados, con el objetivo de medir el rendimiento del sistema.

1.5. Impacto inicial de solucionar el problema

Al ser la solución que se busca generar con este trabajo una herramienta de aprendizaje orientada a un motor de juegos específico, lo que se espera es que, no solo se eduquen los desarrolladores en formas comunes de optimización, sino que también tengan experiencia en un motor de código abierto al tener acceso al proyecto de la experiencia, y así que el trabajo de aprender a crear un videojuego sea más expedito para quienes están empezando en este rubro. Esto porque muchos de los desarrolladores de videojuegos no provienen de un trasfondo de programación, sino que pueden provenir de áreas diversas con el simple objetivo de crear algo propio. Es por esto que muchos no entienden de optimización, lo cual puede afectar en la recepción final de su producto, por lo que se espera aliviar la carga de diseño del desarrollador común.

CAPÍTULO 2

MARCO CONCEPTUAL

A continuación, se procederá a hablar del marco conceptual del sistema, en el cual se abordarán los siguientes temas relevantes para la comprensión de la solución propuesta:

- Motores de videojuegos
- Entorno de trabajo
- Entorno 3D
- Técnicas de optimización comunes

2.1. Motores de juegos

Para comenzar a ahondar en el tema de la optimización en el desarrollo de videojuegos, es de suma importancia para todo desarrollador preguntarse qué motor utilizar para comenzar a trabajar en su proyecto. Sin embargo, lo anterior provoca que surja la pregunta: ¿Qué son los motores de juegos?

Cuando se habla de motores de juegos (o *game engine* por como se conoce en inglés), se refiere a la base sobre la cual se construyen los videojuegos, es decir, un conjunto de herramientas de software o API diseñadas para optimizar el desarrollo de un proyecto. Como mínimo, esto viene en la forma de un motor de renderizado 2D o 3D (o ambos, dependiendo de las necesidades que se busquen satisfacer) para manejar la gestión de la memoria y el almacenamiento en búfer de las imágenes y los objetos 3D que se muestran en la pantalla [Universidad Europea, 2024].

2.1.1. Características de los motores populares

Si bien el mínimo que un motor de juegos debe proveer son herramientas para el renderizado gráfico 2D/3D, esto es solo el comienzo de lo que ofrecen los motores más populares como *Unity*, *Unreal Engine* y *Godot*. Hoy en día, los motores ofrecen características como [Universidad Europea, 2024]:

- Bucle de juego integrado
- Motores de animación

- Multijugador y herramientas de red
- Herramientas de realidad virtual
- Iluminación, sombreadores, sombras, materiales de objetos y otros componentes visuales
- Inteligencia Artificial (IA)

Los motores de juego han apuntado a convertirse en herramientas todo en uno que buscan tener todo lo necesario para permitir a los desarrolladores crear todo tipo de videojuegos, por lo que han optado por ser más generales en su enfoque de trabajo. Esto, si bien en la mayoría de los casos es algo beneficioso para los desarrolladores, puede llegar a ser un problema a la hora de trabajar con mecánicas muy específicas o personalizadas, donde estos motores generales no tienen una manera óptima para llevarlas a cabo, por lo que se puede optar por crear un motor propio.

2.1.2. Motores de juego personalizados

Cuando los desarrolladores no pueden implementar una mecánica u optimización necesaria para sus proyectos o si simplemente quieren probarse a sí mismos, es donde se vuelve viable la creación de un motor de juegos propio, específico para el videojuego en desarrollo. Algunas de las ventajas de hacerlo incluyen:

- Simplificación de las características del motor para acomodar el proyecto
- Control directo de la optimización de recursos
- Implementación personalizada de mecánicas

Sin embargo, el desarrollo de un motor de videojuegos no es algo simple y conlleva mucho esfuerzo y tiempo, por lo que no es una opción recomendable para gente que está comenzando en el desarrollo de videojuegos y/o que no obtiene un beneficio claro al desarrollar su propio motor.

2.2. Entorno de trabajo

Teniendo claro lo que es un motor de juegos es necesario elegir uno para la creación de la solución

2.2.1. Motores comerciales vs motor propio

Para el desarrollo de la memoria, la elección de un motor comercial es lo más beneficioso, debido a que el enfoque es ayudar a la comunidad de desarrolladores indie y, como se indicó anteriormente, la creación de un motor propio no es una solución recomendable para la mayoría de nuevos desarrolladores, además de que solo garantiza un mayor control en la optimización de casos específicos, siendo el objetivo de esta memoria una ayuda general que pueda ser aplicada en múltiples proyectos.

2.2.2. Godot

De los motores de juego disponibles, el sistema de optimización será realizado en el motor de desarrollo de videojuegos Godot, el cual fue elegido debido a su alza en popularidad entre los desarrolladores indie en el último tiempo, además de ser *open source*, lo cual puede ser útil a la hora de optimizar el uso de recursos.

En su documentación, Godot se define como un motor de juegos repleto de características, multiplataforma para crear juegos 2D y 3D por medio de una interfaz unificada. Provee un conjunto exhaustivo de herramientas comunes, para que los usuarios puedan enfocarse en crear juegos sin tener que reinventar la rueda. Los juegos pueden exportarse en un solo clic a numerosas plataformas, incluyendo las principales plataformas de escritorio (*Linux, macOS, Windows*), plataformas móviles (*Android, iOS*), así como plataformas y consolas basadas en la web [Godot Community, a].

2.2.3. Popularidad

Tal como reporta el clarín, desde el 12 de septiembre de 2023 la popularidad del motor fue en aumento, como se muestra en la figura 1, donde se aprecia que en septiembre aumentó considerablemente y se ha mantenido relevante en la actualidad. Esto concuerda con el anuncio del popular motor Unity, en donde se publicó que incluiría una tarifa por descarga, lo que no fue bien recibido por la comunidad de desarrolladores y provocó una migración masiva a Godot.

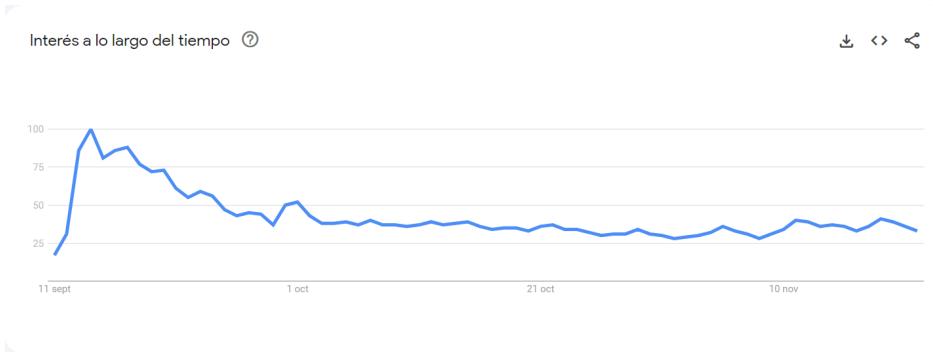


Figura 1: Popularidad de Godot después del anuncio de Unity.
Fuente: Google trends.

2.2.4. Open Source

Uno de los puntos fuertes de *Godot* por sobre otros motores de juego es el hecho de que es *open source*, lo cual da paso a que la comunidad contribuya en el mejoramiento continuo del motor, además de permitir el desarrollo de herramientas compatibles con este, como plugins para integrar nuevas mecánicas y optimizaciones, las cuales son compartidas con la comunidad. Esto implica que, además las herramientas base de optimización que provee *Godot* (y en las cuales se enfoca esta memoria), es posible implementar soporte para casos más específicos.

2.2.5. Licencia de Godot

En su documentación, *Godot* indica que es “completamente gratis y de código abierto bajo la permisiva licencia MIT (Licencia del Instituto Tecnológico de Massachusetts). Sin condiciones, sin regalías, nada. Los juegos de los usuarios son suyos, hasta la última línea de código del motor. El desarrollo de *Godot* es totalmente independiente y dirigido por la comunidad, lo que permite a los usuarios ayudar a dar forma a su motor para que coincida con sus expectativas. Está respaldado por la *Godot Foundation* (Fundación *Godot*) sin fines de lucro” [Godot Community, a].

Este es un punto importante, tomando en cuenta el revuelo que causó el anuncio de *Unity* (del cual tuvo que retractarse eventualmente debido a la gran respuesta negativa de los usuarios), ya que les da la seguridad a sus usuarios de que el acceso a sus características y los proyectos que produzcan con el motor son y siempre serán de su propiedad, cosa que se ha vuelto algo a tener en cuenta a la hora de elegir un motor de juegos, en especial si dichos usuarios son desarrolladores indie sin muchos recursos o directamente personas que están desarrollando su primer videojuego por sí mismos.

2.2.6. Conceptos básicos del motor

En *Godot* el proceso de creación de videojuegos se encuentra representado como un árbol de nodos, en donde cada nodo es un elemento del videojuego que obedece a una jerarquía padre-hijo, los cuales en conjunto construyen las escenas que conforman el producto terminado.

Siguiendo lo anterior, un videojuego en este motor se puede definir como un conjunto de escenas, en donde cada una actúa como el nodo raíz de su árbol. Dichas escenas están compuestas por múltiples entidades, ya sean props decorativos, objetos interactuables, jugadores, NPCs, el mundo, etc. Por sí solas, cada entidad es su propio árbol, donde cada nodo que la compone posee características especiales que permiten crear todo lo que el juego necesite, como la *mesh*, un cuerpo para detectar colisiones, una cámara, un agente de audio, etc.

2.3. Entorno 3D

Por otra parte, se utilizará el entorno 3D, debido a su popularidad con los desarrolladores y a que el manejo de entidades se vuelve más complejo en comparación con el entorno 2D, por lo que es un ambiente propicio para analizar los cambios en el rendimiento causados por la cantidad o el detalle de las entidades. Debido a esta complejidad, también se generan más oportunidades de optimización, lo que facilitará realizar un análisis de distintas técnicas disponibles para un mejor uso de los recursos y una mejora en el desempeño.

2.3.1. Cámara

Uno de los conceptos cruciales sobre el cual muchas técnicas de optimización giran en torno es la cámara que, en muchos casos, actúa como los ojos del jugador y es la ventana al entorno gráfico en el que se desarrollan los videojuegos. Esto se debe a que, más allá de ciertas optimizaciones de código, el proceso de renderizado gráfico es una de las áreas en donde mayor impacto se puede generar en el desempeño al implementar métodos de optimización de recursos gráficos.

La cámara se convierte en un objeto de análisis importante debido a que tener en cuenta la sección del entorno que está mostrando es la base para saber cosas como cuándo renderizar un objeto y cuándo dejar de hacerlo, cuándo es importante mostrar mayor detalle y cuándo se pueden ahorrar recursos mostrando menos detalle, cuándo un grupo de objetos se debe mostrar en su totalidad y cuándo pueden ser representados como un solo objeto.

2.3.2. Cámara en Godot

En *Godot*, dependiendo de si se está usando un entorno 2D o 3D, la cámara se representa como un tipo de nodo especial llamado **Camera2D** o **Camera3D** según corresponda. Como se mencionó anteriormente, el enfoque será el entorno 3D, ya que este permite presentar más oportunidades de optimización, por lo que este documento se centrará en el nodo **Camera3D**.

La documentación de *Godot* lo describe como un nodo especial que muestra lo que es visible desde su ubicación actual. La cámara se registra en el nodo *Viewport* más cercano (ascendiendo en el árbol de nodos) o, en caso de no haber, se registra en el *viewport* global. La cámara en este entorno viene siendo el nodo que provee las capacidades visuales 3D a un *viewport* y una escena registrada en un *viewport* sin una, no se puede mostrar [Godot Community, b].

2.3.3. Entidades

Otro concepto en torno al cual gira este proyecto son las entidades. Para el propósito de su uso en esta memoria, lo que se entiende por una entidad, de forma general, se refiere a una *mesh* que representa un objeto dentro de una escena. Esto significa que una entidad puede ser desde los suelos y paredes que componen el mapa en un nivel de un videojuego, hasta los *props* y personajes que se utilizan para poblarlo. Cabe destacar que un objeto compuesto por múltiples *meshes* puede ser considerado como más de una entidad.

El objetivo de la solución propuesta es el manejo las entidades y sus componentes de forma tal que sea posible reducir el impacto que estas tienen en el rendimiento de los videojuegos. Para lograr esto, se analizarán diferentes técnicas de optimización que involucran múltiples formas de interactuar con las entidades. Una de las más comunes es ocultarlas para que no sean renderizadas.

2.3.4. Entidades en Godot

En *Godot*, teniendo en cuenta que para el proyecto se utilizará el entorno 3D, el nodo que se utiliza comúnmente para representar una *mesh* es **MeshInstance3D**, se compone de un conjunto de polígonos, materiales, texturas y propiedades que le dan forma y atributos útiles a las entidades.

De acuerdo a su documentación, *Godot* lo define como un nodo que toma un recurso *mesh* y lo añade a la escena actual a través de una instancia de la entidad. Esta es la clase más usada para renderizar geometría 3D y puede ser utilizada para instanciar una única *mesh* en muchos lugares [Godot Community, h].

2.3.5. Liberar y cargar recursos

Como se mencionó anteriormente, una de las formas más comunes de mejorar el rendimiento de una escena es ocultar las entidades. Esto es posible gracias a que todos los nodos en Godot poseen la propiedad *visible*, la cual es un booleano que indica si un nodo y sus hijos serán renderizados o no. Cuando se habla de liberar recursos ocupados por una *mesh*, generalmente se refiere a asignar esta propiedad de su nodo **MeshInstance3D** como falsa (*false*), causando que la entidad no sea renderizada incluso estando dentro del rango de visión de la cámara. Por otro lado, se pueden volver a cargar los recursos que utiliza una *mesh* asignando esta propiedad como verdadera (*true*).

2.4. Técnicas de optimización comunes

Al momento de optimizar recursos, existen múltiples vías por las cuales se pueden liberar recursos que no se estén utilizando. Para los propósitos de esta memoria, es necesario definir las siguientes técnicas de optimización:

2.4.1. Fustrum culling

Fustrum culling es la práctica de detectar cuándo un objeto se encuentra fuera de cámara, con el objetivo de no seguir renderizando su *mesh* hasta que se encuentre nuevamente dentro del rango de visión de la cámara.

Esta técnica viene implementada de manera automática en motores de juego como *Godot*, *Unity* y *Unreal engine*, ya que es una de las maneras más generales de optimizar el uso de recursos. Debido a que en la gran mayoría de aplicaciones resulta natural pensar que este tipo de optimización implica un impacto positivo en el rendimiento, siendo lógico que lo que el jugador no vea no necesita ser renderizado, es una práctica útil para la mayoría de los desarrolladores.

2.4.2. Cuándo aplicar Fustrum culling

Como se mencionó anteriormente, la aplicación *Fustrum culling* resulta útil para la gran mayoría de proyectos y es por esto que los motores de juego modernos lo implementan automáticamente. En general, siempre que existan objetos que pueden estar fuera del rango de visión de la cámara, se recomienda la aplicación de esta técnica de optimización. Por otro lado, un ejemplo de cuándo no es necesaria su implementación es al tener un bucle de juego que transcurre completamente dentro del rango de visión de la cámara, como puede darse en videojuegos en donde todas sus escenas transcurren frente a una cámara estacionaria.

2.4.3. Fustrum culling en Godot

En su documentación, *Godot* indica que el motor implementa *frustrum culling* automáticamente con el objetivo de prevenir el renderizado de objetos que se encuentran fuera del *viewport* [Godot Community, d]. Esto resulta útil a la hora de implementar un sistema de optimización, debido a que no hay que preocuparse por la implementación de esta técnica.

2.4.4. Occlusion culling

Occlusion culling en videojuegos es la práctica de detectar los elementos que se encuentran al frente de la cámara en un ambiente 3D. Esto se hace con el objetivo de solo renderizar dichos objetos, liberando recursos visuales de los objetos que se encuentran obstruidos, es decir, aquellos que se encuentran bloqueados por las *meshes* de los que están dentro del rango de visión de la cámara.

Existen múltiples métodos para lograrlo, pero esta técnica se basa en la idea de analizar la perspectiva de la cámara para determinar los objetos que puede ver y cuáles no, renderizando todos aquellos que se encuentran dentro del área visible y removiendo del proceso de renderizado a todos aquellos que se encuentran obstruidos por alguno de los objetos visibles, resultando en un *framerate* más rápido y una mejora en el desempeño.

Es importante tener en cuenta que, en general, la implementación de *occlusion culling* lleva una carga extra de cómputo, lo que puede conllevar una disminución en el rendimiento si se implementa mal o innecesariamente.

2.4.5. Cuándo implementar occlusion culling

Occlusion culling es una técnica que brilla en escenas con una gran cantidad de entidades, las cuales se sitúan en espacios cerrados en donde la mayoría de estas se encuentran obstruidas. Esto es porque, como se mencionó anteriormente, la implementación de esta técnica de optimización conlleva un impacto en el desempeño, por lo que hay que tener en cuenta, al momento de diseñar un nivel, cómo sacar el mayor provecho para que dicho impacto sea positivo. Esto va acorde a la recomendación que hace *Godot* en su documentación, donde indica que funciona mejor en escenas en interiores, con muchas salas pequeñas en lugar de unas pocas salas grandes.

En las figuras 2, 4 y 3 se muestra un ejemplo del efecto que tiene implementar esta forma de optimización, donde se puede apreciar que, analizando el esqueleto de las *meshes* en pantalla, se disminuye la cantidad de entidades renderizadas en comparación a la escena original.

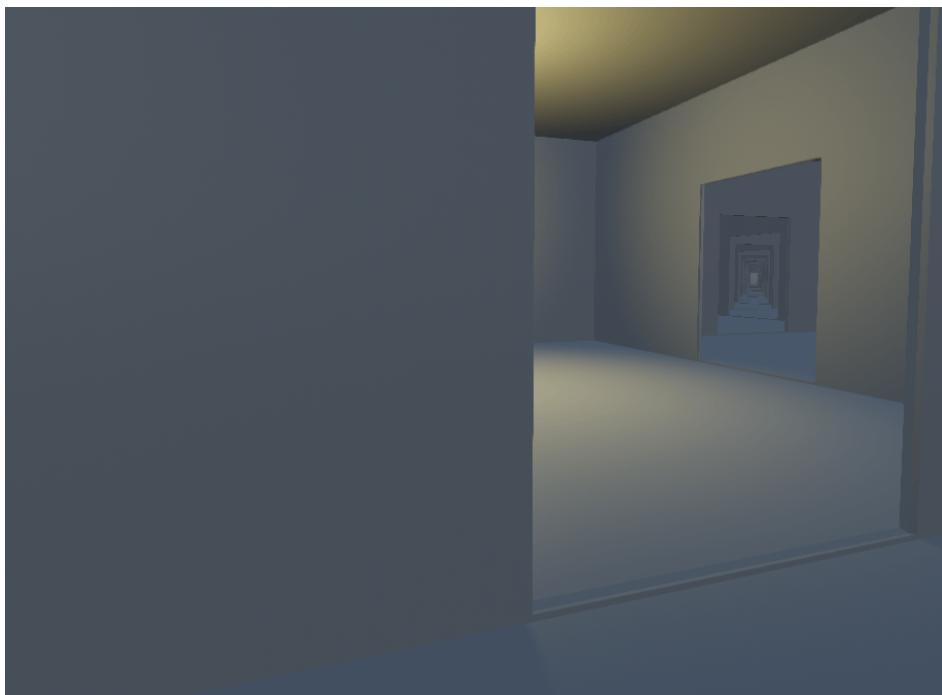


Figura 2: Ejemplo de escena en Godot.

Fuente: Godot Community.

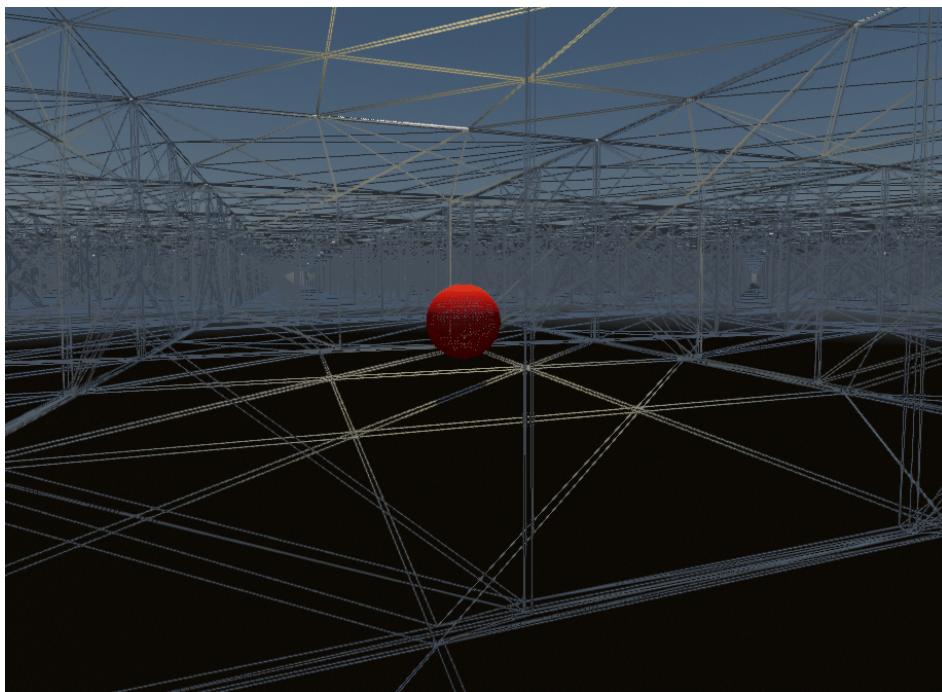


Figura 3: Wireframe de la escena original.

Fuente: Godot Community.

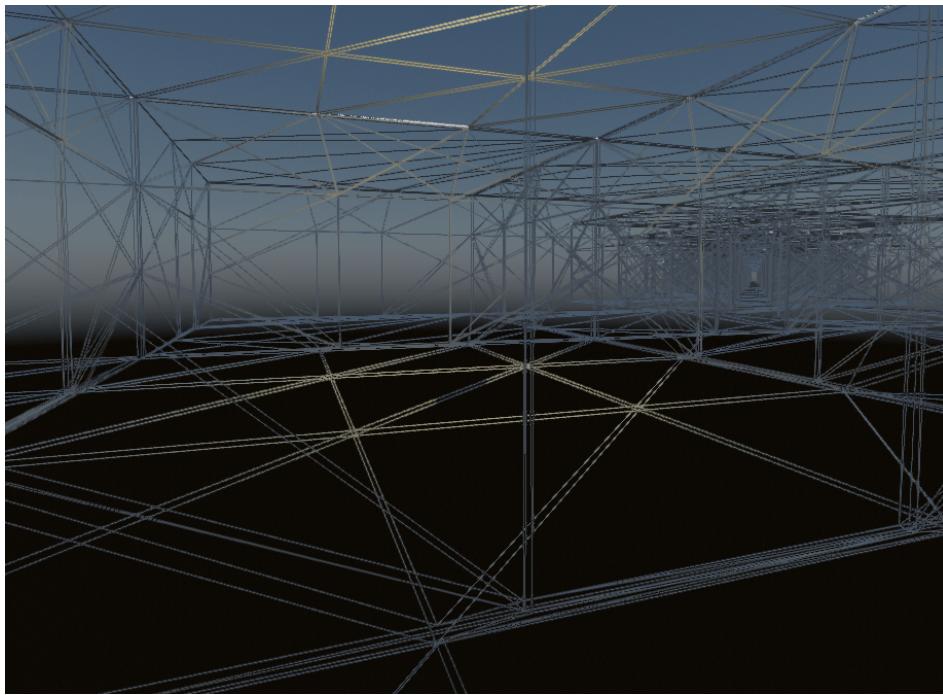


Figura 4: *Wireframe* de la escena con *occlusion culling* activo.
Fuente: Godot Community.

2.4.6. Occlusion culling en Godot

De acuerdo a la documentación, en Godot, *occlusion culling* funciona rasterizando la geometría que ocluye a un *buffer* de baja resolución en la CPU (ver figura 5). Esto se logra utilizando la librería de *raytracing* de software *Embree*.

Luego el motor utiliza este *buffer* de baja resolución para probar la *AABB* del objeto que está siendo ocluido contra la forma del oclusor. La *AABB* del objeto ocluido debe estar completamente ocluida por el oclusor para que este deje de renderizarse.

Como resultado, es más probable que objetos pequeños sean dejados de renderizar que objetos grandes. A su vez, oclusores grandes (como paredes) tienden a ser mucho más efectivos que unos pequeños (como props decorativos) [Godot Community, f].

Para activar *occlusion culling* en Godot, se necesita ir a las configuraciones del proyecto (con las configuraciones avanzadas habilitadas) y habilitarla en **Rendering >**Occlusion Culling >Use Occlusion Culling** [Godot Community, i]. Una vez activado, se debe añadir un nodo especial a la escena en la que se aplicará la técnica de optimización llamado **OccluderInstance3D**. En base a este nodo se puede elegir que Godot genere los oclusores automáticamente o se pueden generar unos personalizados.



Figura 5: *buffer* aplicado a una estructura en el editor de Godot

Fuente: Elaboración Propia.

2.4.7. LOD

Cuando se habla de *LOD* (*Level of detail*) se refiere a la práctica de aumentar o disminuir el nivel de detalle de un objeto 3D dependiendo de la distancia a la que se encuentra de la cámara. Esto se logra controlando la cantidad de polígonos que utiliza su *mesh*, ya que, mientras mayor sea esta cantidad, mayor es el nivel de detalle que esta puede mostrar.

Debido a que utilizar *meshes* demasiado definidas implica un impacto mayor en el rendimiento, es importante controlar la cantidad de polígonos que se están renderizando al mismo tiempo, lo cual puede resultar ser una tarea difícil en escenas con demasiados objetos. Para combatir esto, una buena práctica es reducir la cantidad de detalle de las *meshes* de los objetos que se encuentran a una gran distancia de la cámara, ya que en estos no se puede apreciar un gran nivel de detalle, lo cual los vuelve un impacto innecesario en el rendimiento.

Este principio se puede llevar aún más lejos si, desde cierta distancia a la cámara, se cambian las *meshes* de los objetos por unas menos detalladas que consuman menos recursos. Un ejemplo de esto sería pasar de una esfera con muchos polígonos a un cubo con una cantidad mucho menor una vez que se alcance cierta distancia en donde no se pueda apreciar bien el nivel de detalle e inclusive se puede llevar más allá, añadiendo otro nivel de pérdida de detalle en donde pase de un cubo a una imagen plana.

2.4.8. Cuándo implementar LOD

Level of detail es una técnica que es, en general, recomendable utilizar. No es extraño encontrar que los motores de juego modernos tengan algún tipo de implementación automática

de LOD, por lo que incluso en casos donde no afecta demasiado, sigue siendo recomendable usarla, ya que probablemente implicará un impacto positivo.

Ahora bien, existen casos en donde es recomendable realizar una implementación personalizada de esta técnica, sobre todo para videojuegos de mundo abierto o simplemente escenas demasiado grandes en donde la cámara puede llegar a estar muy lejos de ciertos objetos, ya que es aquí donde se puede sacar el mayor provecho de manipular el nivel de detalle de las meshes.

2.4.9. LOD en Godot

En su documentación, Godot indica que el motor provee una forma para generar automáticamente meshes menos detalladas para su uso en *LOD* al importarlas (ver figuras 6 y 7), para luego usar esas meshes *LOD* automáticamente cuando sea necesario. Esto es completamente transparente al usuario. Se utiliza la librería *meshoptimizer* para la generación de meshes *LOD* detrás de escena. El *LOD* de meshes funciona para cualquier nodo que dibuje meshes 3D, esto incluye los nodos **MeshInstance3D**, **MultiMeshInstance3D**, **GPUParticles3D** y **CPU-Particles3D** [Godot Community, g].

Además de generar meshes menos detalladas automáticamente al importarlas, también se puede controlar el grado de agresión para las transiciones *LOD* en la configuración del proyecto modificando el valor **Rendering > Mesh LOD > LOD Change > Threshold Pixels**, o bien, ajustando manualmente la propiedad **mesh_lod_threshold** encontrada en estos nodos, siendo 1 el valor por defecto. Al disminuir este valor, en el caso de ajuste manual, el grado de pérdida de detalle aumenta, siendo el efecto más visible y teniendo un mayor impacto positivo en el desempeño [Godot Community, c].



Figura 6: Ejemplo de meshes *LOD* generadas al importar un objeto en Godot.

Fuente: Godot Community.



Figura 7: Ejemplo del *wireframe* de meshes *LOD* generadas al importar un objeto en *Godot*.

Fuente: *Godot Community*.

2.4.10. Object pooling

Object pooling es una técnica de optimización orientada a la reutilización de objetos en lugar de instanciar nuevos que ocupen más memoria. El objetivo de esto es evitar la adición innecesaria de nuevas entidades que ocupan recursos y pueden reducir el rendimiento, dependiendo de la cantidad y frecuencia con la que se añaden.

Un ejemplo de estas situaciones es tener una entidad que dispara proyectiles. Si la entidad instancia un nuevo proyectil en cada disparo, se corre el riesgo de que eventualmente se acabe la memoria. Incluso cuando esto se previene liberando los proyectiles una vez impactan con algún objetivo o superficie, si aumentamos el número de entidades que disparan, el desempeño puede sufrir debido a que se están instanciando múltiples disparos simultáneos.

La solución que propone *object pooling* para esta problemática es que, al momento de instanciar la entidad, esta venga con un número de proyectiles disponibles y que, una vez disparados, en lugar de eliminarlos, estos se oculten y se reutilicen para los siguientes disparos, instanciando nuevos proyectiles solo en caso de que sea absolutamente necesario. Así no solo se soluciona el problema de generar demasiados objetos que consuman la memoria disponible, sino que también se evita que el desempeño disminuya debido a que se añadan múltiples instancias simultáneas.

2.4.11. Cuándo aplicar object pooling

Como se menciona en el ejemplo anterior, *object pooling* es una técnica sumamente útil en situaciones donde se tienen múltiples entidades que instancian otras entidades, como puede ser el caso de tener muchas instancias de un tipo de enemigo que dispara proyectiles. El darles a cada instancia una *pool* de proyectiles que puede reutilizar previene el uso innecesario de recursos.

2.4.12. Object pooling en Godot

No hay una forma específica en la que *Godot* aborde esta técnica de optimización, por lo que su implementación viene dada por parte del desarrollador y su diseño del videojuego para encontrar instancias donde su uso sea beneficioso.

2.4.13. Gestión de entidades a través de data

A diferencia de las técnicas anteriores, la gestión de *entidades a través de data* no es una forma de optimización con una aplicación tan concreta. El objetivo de gestionar correctamente la data es evitar situaciones en donde se produce un gran número de operaciones por *frame*. En especial si dichas operaciones no son óptimas o toman demasiado tiempo en procesarse.

La idea de este tipo de optimización es buscar situaciones en donde exista un gran número de operaciones simultáneas que puedan ser satisfechas con una única operación, con la finalidad de disminuir el tiempo de procesado de cada *frame* y así aumentar el rendimiento.

2.4.14. Cuándo aplicar gestión de entidades a través de data

Como se mencionó anteriormente, en general la idea es fijarse en secciones en el código o escena en donde ocurran acciones muy repetitivas que puedan ser satisfechas por una acción que obtenga el mismo resultado. Un ejemplo de lo anterior es obtener la distancia entre un conjunto de objetos en relación a la cámara. La opción directa sería que cada objeto consulte su distancia a la cámara pero, cuando el número de objetos es muy grande y/o la operación debe realizarse continuamente, el tiempo de procesamiento en cada *frame* aumenta y provoca una caída en el desempeño.

Esto puede solucionarse agrupando los objetos en sectores, los cuales calculan la distancia entre el sector y la cámara y se la proporcionan a cada objeto, lo cual disminuye el número de operaciones de una gran cantidad de objetos a un conjunto menor de sectores. Además, cabe destacar la importancia de utilizar operaciones rápidas, ya que, siguiendo con el caso anterior, si la forma de calcular la distancia no es óptima, esta aumentará el impacto en el rendimiento para cada operación realizada.

2.4.15. Gestión de entidades a través de data en Godot

Si bien *Godot* no aborda en su motor directamente una implementación de esta técnica, existe una herramienta útil para lograrlo que combina bien con la implementación de *LOD* llamada *Visibility ranges (HLOD)*. Esta herramienta permite asignarle a cualquier nodo que herede de **GeometryInstance3D** y permite controlar el rango en el que es visible el nodo.

De acuerdo a la documentación de *Godot*, el beneficio de un *HLOD* por sobre un sistema *LOD* tradicional es su naturaleza jerárquica. Una única *mesh* más grande puede reemplazar múltiples *meshes* más pequeñas, para que el número de *draw calls* pueda ser reducido a la distancia, pero las oportunidades de *culling* puedan ser preservadas al estar cerca [Godot Community, e].

Esto permite implementar la gestión de *entidades* a través de *data* al optimizar recursos por medio de la agrupación de múltiples *draw calls* en solo uno, lo cual va acorde al objetivo de la técnica de reducir el número de operaciones por *frame*.

2.4.16. Renderizado por chunks

Cuando se habla de cargar un mapa en *chunks* o pedazos, se refiere a dividirlo en secciones (comúnmente de un mismo tamaño en una grilla) las cuales son cargadas o descargadas dependiendo de la proximidad del jugador a cada sección. El objetivo de esta técnica de optimización es disminuir la carga de procesamiento del computador al restringir los recursos utilizados a las zonas en las que el jugador se encuentra actualmente.

La implementación de esta técnica involucra, como ya se comentó, dividir el mapa en múltiples *chunks*, una forma de guardar la información referente a cada uno en caso de necesitarlo y utilizar una distancia de renderizado para determinar cuáles deben estar cargados. Esta distancia es utilizada como el radio de un círculo, cuyo centro y punto de referencia es el jugador. Todo *chunk* que se encuentre dentro de la zona cubierta por el círculo se cargará y mantendrá cargado mientras que no salga del círculo. Por otro lado, toda zona que no esté dentro de su área se descargará y permanecerá descargada hasta entrar en contacto con el círculo, tal y como se aprecia en la figura 8, donde el punto azul es el jugador, los cuadros verdes y rojos son *chunks* cargados y descargados, respectivamente, y la distancia de renderizado es dos.

Un ejemplo clásico de una aplicación de esta técnica se encuentra en el videojuego **Minecraft**, el cuál divide su mapa en *chunks* de 16x16 bloques, permitiendo al jugador modificar la distancia de renderizado en *chunks*. Debido a esto, al disminuir la distancia, es posible aumentar el rendimiento sacrificando la experiencia visual del videojuego al mostrar menos del mundo al mismo tiempo.

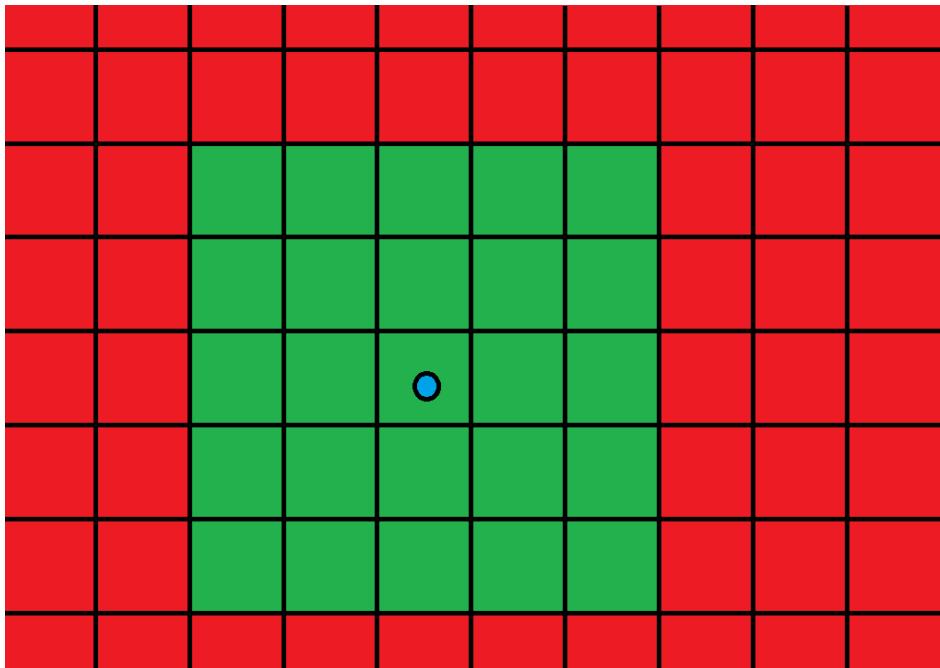


Figura 8: Diagrama de un ejemplo del funcionamiento del *renderizado por chunks*.
Fuente: Elaboración Propia.

2.4.17. Cuándo aplicar renderizado por chunks

Como se mencionó anteriormente, su uso se ha asociado a su aplicación más popular, por lo que se ha vuelto la opción común cuando se necesita un sistema de generación de mapas procedimental, en especial para mundos abiertos y extensos. Sin embargo, este no es el único caso en el que esta técnica de optimización es útil. Una de las ventajas más grandes que conlleva su implementación es la sensación de una experiencia continua sin pausas. Debido a que el mapa se carga siguiendo al jugador y sus alrededores, se elimina la necesidad de dividir el mundo en niveles y la implementación de transiciones o pantallas de carga que interrumpan la experiencia de juego.

Por lo anterior, el caso de uso principal en donde implementarchunks es útil, es en escenas con mucho contenido que buscan otorgar una experiencia sin interrupciones, por lo que puede aplicarse en diversos tipos de mapas, sin importar si dichos chunks funcionan como una grilla o a través de zonas que actúan como una transición para hacer tiempo mientras se carga la siguiente sección y se descarga la anterior.

2.4.18. Renderizado por chunks en Godot

Godot no provee una forma particular para abordar su implementación, por lo que queda a criterio del desarrollador el cómo abordarlo de manera tal que sea beneficioso para su

proyecto.

2.5. Github

Un aspecto importante de lo que se quiere lograr es el poder compartir con otros las formas de optimización utilizadas en el sistema. Es por esta razón que se utiliza el software *Github* para almacenar el proyecto y poder darle acceso a otros que estén interesados en utilizar *Godot* y quieran un acercamiento a las opciones de optimización que posee el motor.

Según su documentación, *Github* es una plataforma basada en la nube donde se puede almacenar, compartir y trabajar con otros usuarios para escribir código. En donde almacenar código en un “repositorio” en *Github* permite lo siguiente [GitHub Community, b]:

- **Presentar o compartir** el trabajo.
- **Seguir y administrar** los cambios en el código a lo largo del tiempo.
- Dejar que otros usuarios **revisen** el código y realicen sugerencias para mejorarlo.
- **Colaborar** en un proyecto compartido, sin preocuparse de que los cambios afectarán al trabajo de los colaboradores antes de que esté listo para integrarlos.

El trabajo colaborativo, una de las características fundamentales de la plataforma, es posible gracias al software de código abierto *Git*, en el que se basa *Github*. Este es un sistema de control de versiones que realiza un seguimiento de los cambios en los archivos. *Git* es especialmente útil cuando un grupo de personas está haciendo cambios en los mismos archivos al mismo tiempo. Normalmente, un flujo de trabajo basado en *Git* funciona de la siguiente manera [GitHub Community, a]:

- **Crear una rama** a partir de la copia principal de archivos en los que se está trabajando.
- **Realizar modificaciones** en los archivos de forma independiente y segura en una rama personal.
- **Git fusiona mediante combinación** y de forma inteligente los cambios específicos en la copia principal de archivos, de modo que los cambios no afecten a las actualizaciones de otras personas.
- **Git realiza un seguimiento** de los cambios hechos hasta el momento, por lo que todos los colaboradores siguen trabajando en la versión más actualizada del proyecto.

Al ser una plataforma de trabajo colaborativo, existe la opción de que un repositorio sea público, lo que permite a otros usuarios ver el proyecto. Esto es perfecto para permitir que este sistema llegue al mayor público posible.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Para dar solución al problema antes descrito, se ha desarrollado un proyecto en el motor de juegos *Godot*, el cual consta de cuatro experiencias de corta duración enfocadas en evidenciar los beneficios de cuatro técnicas de optimización. Debido a que el objetivo de esta experiencia es ser una herramienta de aprendizaje, se optó por subirla a un repositorio público en *Github*, para que pueda funcionar como demostración de las capacidades del motor, así como también como un ejemplo de uso para cada una de las técnicas que se encuentran implementadas, con el objetivo de brindar apoyo a los desarrolladores *indie* que necesiten o estén interesados en optimizar sus videojuegos.

3.1. Repositorio en Github

Para el repositorio en *Github*, se creó una cuenta con el correo proporcionado por la universidad (el nombre de usuario es *rruiz-sansano*), con la cual se creó un repositorio público para permitir el acceso a cualquier usuario interesado en la experiencia. En el repositorio se encuentra la versión más actual del proyecto, un archivo **readme.md** con una descripción simple del proyecto y este escrito para facilitar el acceso a la información necesaria. El enlace al repositorio es [el siguiente](#)

3.2. Proyecto en Godot

A continuación, se describen las experiencias diseñadas en el motor de juegos *Godot* en la versión 4.5 estable, explicando las herramientas utilizadas, la estructura del proyecto y el bucle de juego implementado en cada una, para luego enfocarse en la implementación de cada técnica de optimización utilizada.

3.2.1. Assets

Antes de comenzar a describir los aspectos técnicos, es importante mencionar que este proyecto utiliza *assets* descargados desde la página web [Itch.io](#). Los *assets* utilizados corresponden a los siguientes:

- **Free Modular Low Poly Dungeon Pack** creado por **Raphael Gonçalves (Rgsdev)**, el cual se utiliza para la creación de los mapas

- **3D Medieval Interior** creado por **Pelatho**, utilizado en la creación de *props* para poblar las escenas.
- **Studio Ghibli Inspired Asset Pack** creado por **Callum Andrews**, utilizado en la creación de un mapa
- **Free CC0 Melee Weapons Pack** creado por **3dmodelscc0**, del cuál se hace uso de una lanza para utilizar como proyectil.

3.2.2. Menú

Al momento de abrir el proyecto, lo primero que se puede ver es un menú simple con los botones **Jugar**, **Instrucciones**, **Créditos** y **Salir**. El botón **Jugar**, el botón **Instrucciones** y el botón **Créditos** llevan a tres nuevos menús, respectivamente, mientras que el botón **Salir** termina la experiencia y cierra el programa.

Dentro del menú de instrucciones se les indica a los jugadores que pueden activar y desactivar las distintas técnicas de optimización implementadas dentro del menú de selección de experiencia, al cual se llega apretando el botón **Jugar**. Además, se les indica que los controles disponibles para la experiencia son las teclas WASD para el movimiento y el mouse para mirar, los cuales son los controles estándar cuando se trata de *juegos en primera persona*. Por último, se puede apretar el botón **Volver** para regresar al menú anterior.

El menú de créditos funciona de la misma manera que el de instrucciones, con la diferencia siendo que en este se presentan los *assets* utilizados en la experiencia.

Por otro lado, dentro del menú de juego es donde se pueden activar y desactivar las técnicas de optimización para poder analizar su impacto en el desempeño al iniciar sus experiencias correspondientes a través de cuatro botones de tipo activación o *toggle*. Al lado derecho de cada uno de estos botones se encuentra uno con el texto **Jugar**, el cual da inicio a la experiencia correspondiente al texto que se encuentra al lado izquierdo de los botones, los cuales, en orden, dicen: **Occlusion culling**, **HLOD**, **Object pooling** y **Chunk**. Además, se tiene un botón para volver al menú principal llamado **Volver**.

Una vez que se selecciona una experiencia, se carga una escena intermedia que consta de una interfaz simple con un texto que dice *Cargando...*, la cual cumple con el rol de permitir que la experiencia seleccionada termine de cargarse antes de redirigir al jugador a esta.

Debido a que el objetivo del menú no es más que el de tener una forma de controlar las técnicas de optimización activas a modo de realizar análisis y comparaciones, no presenta una interfaz de usuario diseñada con algo más allá de su utilidad en mente, por lo cual puede resultar simple y algo tosco.

3.2.3. Búcle de juego

Al ser una experiencia enfocada en comparar el rendimiento al aplicar una forma de optimización frente a un caso control sin ella, no se le da mucho énfasis a la jugabilidad más allá de caminar desde un punto **A** a uno **B**, donde el objetivo real es que el jugador experimente un escenario en el cual el uso de la técnica de optimización es favorable. Es por esto que, como un mínimo incentivo para el jugador, se muestra un objetivo en pantalla a través de un nodo **CanvasLayer**, el cual permite utilizar un nodo **Label** para mostrar texto en pantalla dentro de la experiencia. A esto se le suma un nodo **Timer** para que, al iniciarla y luego de una cierta cantidad de tiempo, se oculte el texto en pantalla.

Por lo anterior, para las experiencias de *Occlusion culling*, *HLOD* y *Object pooling* se utiliza un incentivo simple de un cofre del tesoro al final de un pasillo, donde el objetivo es llegar hasta este, dando por terminada la experiencia y devolviendo al jugador al menu principal para que pruebe las demás formas de optimización y/o la repita activando/desactivando la técnica a modo de comparación.

Para el caso de la experiencia de *renderizado por chunks*, el objetivo requiere un paso extra (sin dejar de ser simple), en el cual se le pide al jugador buscar una llave que se encuentra en otra habitación, para luego regresar a la sala inicial, donde está la puerta que se abre con esa llave. Esto con el objetivo de mostrar de mejor manera la técnica de optimización de la experiencia sin complicar mucho el proceso.

3.2.4. Player

El jugador en la experiencia se encuentra representado por una escena con unos nodos **MeshInstance3D** y **CollisionShape3D** con forma de cápsula para una representación simple. Además, un nodo **CanvasLayer** con su respectivo nodo **Control** para mostrar en pantalla los *frames per second (FPS)*, la cantidad de entidades que se encuentran en pantalla actualmente y el número de *draw calls* realizadas, a través de tres nodos **Label**, con el objetivo de medir el impacto de las técnicas de optimización activas. Para lograr esto, el nodo **Control** contiene un script que obtiene esos datos a través de la clase *Performance* provista por Godot. Por último, un nodo **Camera3D** que actúa como los ojos del jugador y le permite observar su entorno. Lo anterior puede apreciarse en la figura 9.

El nodo raíz de la escena, llamado **Player**, contiene un *script* (escrito en el lenguaje de programación de Godot *GDSscript*) que contiene funciones básicas de movimiento como mirar con el *mouse*, moverse con WASD y saltar con la barra espaciadora.

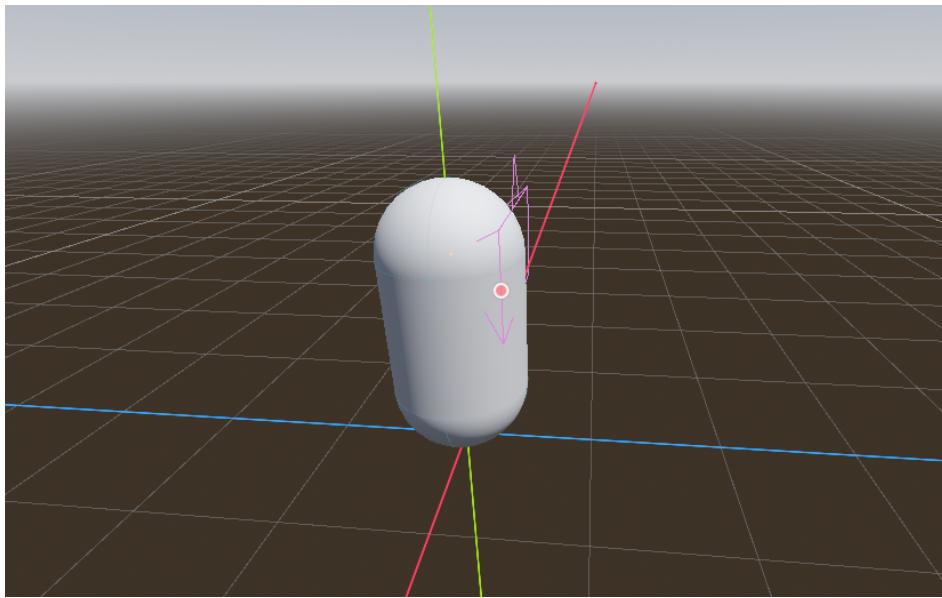


Figura 9: Representación del jugador utilizado en las experiencias.
Fuente: Elaboración Propia.

3.2.5. Escena del mundo

Para cada técnica de optimización, se utiliza un mapa diferente con entidades que se ajustan a la experiencia que se busca otorgar al jugador. Sin embargo, cada una incluye tres características en común: la representación del jugador descrita en la sección anterior, el mundo y su iluminación. Estos últimos son incluidos comúnmente como la representación de las características del ambiente y el cielo respectivamente. Si bien no son estrictamente necesarios, se utilizan para controlar la estética del mundo en la escena a través de un nodo **WorldEnvironment** para el ambiente y un nodo **DirectionalLight3D** para el sol y el cielo.

3.2.6. Fustrum culling

Tal y como se explicó en el capítulo anterior, *Godot* implementa automáticamente *frustum culling*, por lo que la experiencia no lo aborda.

3.2.7. Experiencia 1: Occlusion culling

Para presentar el potencial de optimización que trae la implementación de *occlusion culling*, se utiliza una estructura cerrada, la cual se encuentra compuesta de dos habitaciones importantes: la sala donde se encuentra el jugador y la que contiene el cofre del tesoro mencionado en la sección “Bucle de juego”. Estas salas se conectan a través de un pasillo largo el cual, a su

vez, se encuentra conectado a otras 618 habitaciones que rodean esta estructura principal.

Lo que determina el uso de la técnica consta de dos partes: que se encuentre activo el uso de *occlusion culling* en las configuraciones del proyecto como se explicó en el capítulo anterior (también puede activarse en *runtime* a través de la línea `get_tree().root.use_occlusion_culling = true`) y que se utilice al menos un nodo **OccluderInstance3D** que actúe como oclusor. Debido a esto, existen dos implementaciones paralelas del mismo mapa para satisfacer los casos en donde la técnica de optimización se encuentra activa o inactiva. La diferencia entre ambas implementaciones, además de activar/desactivar la técnica en las configuraciones del proyecto a través del menú de selección de la experiencia, radica en que, para el mapa con *occlusion culling* activo, se agrega un nodo oclusor para permitir que las paredes del mapa ocluyan lo que se encuentre detrás de ellas.

En cuanto a la construcción del mapa, se utilizaron múltiples copias de una pared, una baldosa y una antorcha para confeccionar la estructura de la escena. Para la pared y la baldosa se utilizó un nodo **MeshInstance3D** para la *mesh*, un nodo **StaticBody3D** para asignarle colisión con el jugador a través de un nodo **CollisionShape3D**. Por otro lado, la antorcha no requiere de colisión pero sí de la capacidad de iluminar, por lo que, además de su nodo **MeshInstance3D**, se utiliza un nodo **OmniLight3D** para asignarle las propiedades de emisión de luz. Se utiliza un nodo **OccluderInstance3D** con la opción “*Bake occluders*” que provee Godot para generar automáticamente los oclusores necesarios en base a las *meshes* presentes en el mapa. Para mostrar el objetivo, se utiliza un mensaje en la pantalla que se oculta luego de unos segundos, el cual está compuesto por un nodo **CanvasLayer** que permite utilizar un nodo **Control** como contenedor de un nodo **Label** para mostrar un mensaje en la pantalla, además, se utiliza un nodo **Timer** como temporizador para enviar la señal a un *script* para ocultarlo. Por último, se utilizan tres nodos **MeshInstance3D** para la construcción del cofre que cumple el rol de condición de término de la experiencia, además de un nodo **Area3D** en conjunto con un nodo **CollisionShape3D** para detectar si el jugador llegó hasta el cofre y dar por finalizada la experiencia. En la figura 10 puede apreciarse una vista externa de lo anterior, tomada desde el editor de Godot, donde cada ícono de ampolleta es una sala diferente.

Como se mencionó anteriormente, el objetivo del jugador es alcanzar el cofre del tesoro, el cual puede apreciarse en la figura 11. Sin embargo, esto no es más que un pretexto para que este experimente el efecto de implementar *occlusion culling* frente a no usarlo. Para lograr esto, la experiencia está diseñada para forzarlo a caminar a través de un pasillo largo, el cual se encuentra rodeado por cientos de habitaciones, lo que significa un impacto importante en el rendimiento. Con la técnica activa, gran parte de esos cientos de habitaciones no se encontrarán renderizadas, por lo que el impacto en el rendimiento será menor comparado con el caso de no implementarla. Es importante destacar que implementar *occlusion culling* implica un impacto considerable en el rendimiento, por lo cual es importante que la escena esté diseñada de tal manera que se justifique su uso y así aprovechar sus beneficios.

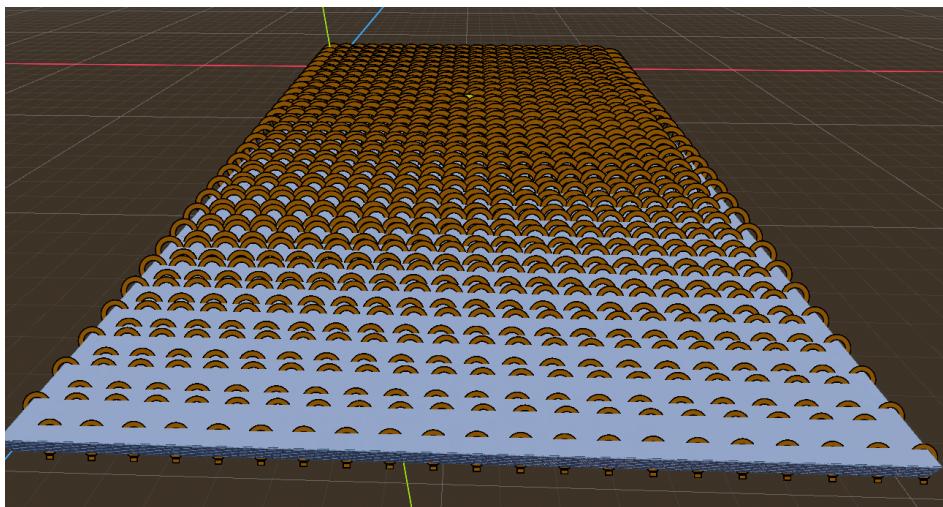


Figura 10: Vista externa del mapa creado para la experiencia *occlusion culling*.
Fuente: Elaboración Propia.



Figura 11: Vista interna del mapa creado para la experiencia *occlusion culling*.
Fuente: Elaboración Propia.

3.2.8. Experiencia 2: HLOD

Para la implementación de esta experiencia basada en la distancia entre el jugador y las *meshes*, se utiliza un mapa abierto, el cual está compuesto por un bosque frondoso con un área central rectangular sin árboles. Esta sección funciona como un pasillo, en donde a un extremo se encuentra el jugador, mientras que al otro hay un altar que contiene el cofre mencionado en la sección “Bucle de juego”.

La implementación de este sistema de optimización está dividida en dos partes: mostrar la capacidad de Godot de cambiar de forma dinámica el grado de agresión de las transiciones LOD e implementar gestión de entidades a través de *data* con *HLOD* para reducir aún más el uso de recursos, agrupando un conjunto de *meshes* en una más simple una vez alcanzada

cierta distancia del jugador. Debido a que la implementación de estos sistemas es a nivel de la *mesh*, basta con desactivar la implementación a través de código para satisfacer el caso en donde estas técnicas no son implementadas. Esto porque *Godot*, como se mencionó en el capítulo anterior, implementa transiciones *LOD* a la *mesh* al momento de importarla, por lo que viene implementado por defecto un nivel de agresividad de transición *LOD* de 1 y, en cuanto al sistema *HLOD*, basta con eliminar *meshes* utilizadas para las transiciones. Lo anterior permite utilizar un único mapa para ambas implementaciones de la experiencia.

En cuanto a la construcción del mapa, debido a que es abierto, se utiliza una plataforma simple como base, creada con un nodo **StaticBody3D** que contiene un nodo **MeshInstance3D** que no es más que un paralelepípedo rectangular verde para simular pasto y un nodo **CollisionShape3D** de la misma figura para permitir que el jugador se pare en ella. Lo anterior se rellena con un bosque compuesto de 3 tipos de árboles copiados múltiples donde cada uno se encuentra compuesto por 3 nodos **MeshInstance3D** que representan su *mesh* de cerca, distancia media y lejos respectivamente. Para cada *mesh* se configura su rango de visibilidad, pasando de 0 a 25 metros, 25 a 50 metros y de 50 metros en adelante respectivamente. Esto permite que cada árbol tenga una *mesh* detallada de cerca, la misma con un alto nivel de agresión en las transiciones *LOD* a distancia media y una *mesh* conformada por imágenes 2D del árbol al observarse desde lejos, reduciendo el nivel de detalle a medida que el jugador se aleja de estos. Por otro lado, el bosque se encuentra dividido en grupos de 32 árboles compuestos en una grilla de 16x16 metros, de los cuales 8 ubicados al centro no poseen una *mesh* intermedia o lejana, efectivamente eliminando el centro de cada bloque de árboles al estar lejos del jugador y reduciendo el número de árboles por grupo. Para el objetivo se utiliza la misma estructura presentada en la experiencia anterior. Por último, el pedestal que contiene el cofre del tesoro (el cuál está compuesto por los mismos componentes que en la sección anterior) se encuentra compuesto por una *mesh* de cerca y otra de lejos, con rangos 0 a 25 metros y 25 metros en adelante respectivamente. La *mesh* de cerca se encuentra compuesta por dos plataformas con un nodo **MeshInstance3D** cada una y cuatro pilares con un conjunto de nodos **MeshInstance3D**, **StaticBody3D** y **CollisionShape3D** cada uno. Por otra parte, la *mesh* de lejos se encuentra compuesta por las mismas dos plataformas, 4 pilares simples que solo cuentan con un nodo **MeshInstance3D** cada uno y una caja café con un nodo **MeshInstance3D** que representa al cofre del tesoro visto desde lejos. Para el caso en el que la técnica de optimización se encuentra desactivada, se desactivan los rangos de visibilidad y se eliminan las *meshes* de rango medio y lejano, provocando que todos los árboles, el pedestal y el cofre se encuentren presentes con el máximo nivel de detalle en todo momento y a cualquier distancia. En la figura 12 puede apreciarse una vista externa de lo anterior, tomada desde el editor de *Godot* mostrando el efecto de la distancia en los grupos de árboles.

Al igual que en la experiencia anterior, el objetivo de obtener el cofre del tesoro no es más que un incentivo para el jugador, que busca que experimente la diferencia entre implementar un sistema *HLOD/LOD* y no hacerlo. Para lograr esto, la experiencia está diseñada para que el jugador comience ubicado lejos del pedestal y de los árboles que tiene directamente frente a él (como se muestra en la figura 13), para que, a medida que se acerca al cofre

del tesoro, se pueda apreciar el impacto en el rendimiento al tener que renderizar más entidades de manera más detallada. Si la técnica de optimización se encuentra desactivada, el rendimiento se mantendrá consistentemente peor, ya que no habrá momentos en que las entidades presentes no sean renderizadas al máximo nivel de detalle.

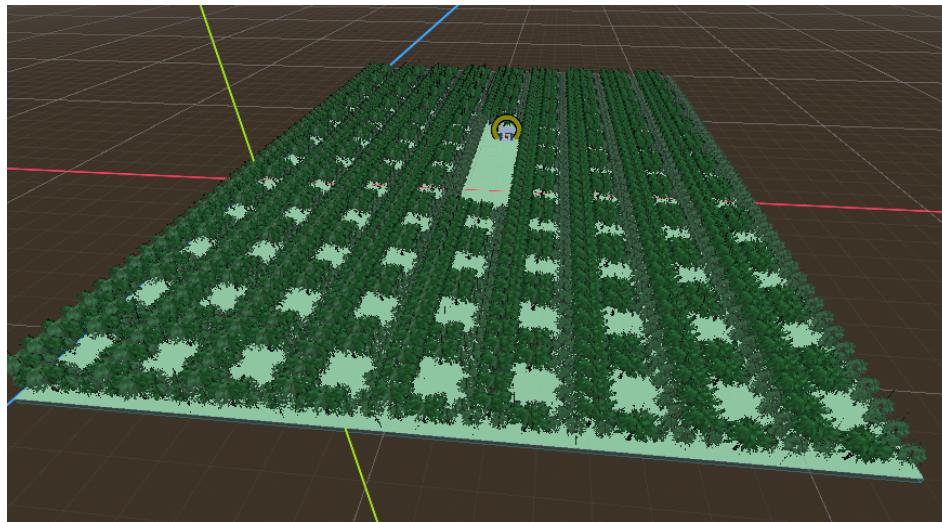


Figura 12: Vista externa del mapa creado para la experiencia *HLOD*.
Fuente: Elaboración Propia.

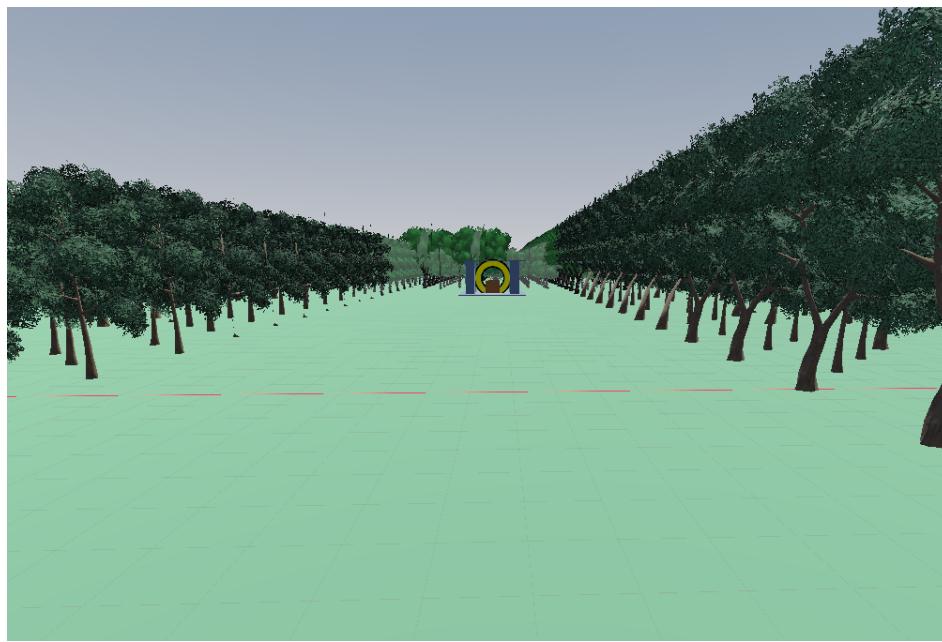


Figura 13: Vista interna del mapa creado para la experiencia *HLOD*.
Fuente: Elaboración Propia.

3.2.9. Experiencia 3: Object pooling

En la experiencia, *object pooling* se representa a través del uso de un pasillo repleto de trampas que disparan un proyectil (una lanza), del cual tienen munición infinita. El pasillo conecta con dos habitaciones que, al igual que con las técnicas de optimización anteriores, conecta al jugador con el cofre del tesoro mencionado en la sección “Bucle de juego”. Dependiendo de si la técnica se encuentra activa o no, se corre el riesgo de saturar la memoria del computador con proyectiles, debido a que no hay un límite en su uso, lo que se ve reflejado en las dos implementaciones que tiene esta mecánica.

Para la implementación cuando *object pooling* no se encuentra activo, cada vez que una trampa dispara una lanza, se crea una nueva instancia del proyectil a la cual se le asigna la posición de la trampa y se le aplica una rotación para que quede apuntando en la misma dirección que esta, para finalmente añadirlo al árbol de la escena principal y aplicarle una fuerza de impulso que permite que la lanza sea disparada hacia donde se encuentra mirando la trampa. El problema que conlleva esta implementación es que no toma en cuenta qué pasa con el proyectil una vez ya ha sido disparado, el cual quedará clavado en donde aterrizó indefinidamente. Es por esto que, con la optimización activa, se le asigna a cada trampa una cantidad de munición finita de lanzas que se instancian al inicio de la experiencia y son reutilizadas, evitando que se agreguen infinitas copias que deban ser renderizadas y saturen la memoria del computador.

La manera en que esto se encuentra implementado es que, al comenzar la experiencia, se instancian 11 proyectiles por trampa, los cuales se guardan en un arreglo. Cada trampa hace uso de temporizadores que determinan su cadencia de tiro, su periodo de enfriamiento y desde qué momento comienzan a disparar. Cuando llega el momento de disparar, habiendo obtenido la lanza previamente, se posiciona en el índice 0 de la lista de munición y, al igual que en el caso anterior, se procede a asignar la posición de la cámara, rotar, añadir al árbol y disparar el proyectil con ese índice. Esto se repite para los índices 1 a 9, actuando de la misma manera que en el caso anterior. Sin embargo, al llegar al índice 10, se utiliza una función auxiliar para ocultar la *mesh* del proyectil utilizado más antiguo (en este caso el de índice 0), luego se procede a disparar la lanza normalmente y, antes de terminar, se asigna 0 como el próximo índice a utilizar. Desde este punto, cada proyectil utilizado subsecuentemente esconderá la lanza más antigua para reutilizarla, recorriendo la lista de munición de manera cíclica, por lo que, sin importar la cantidad de veces que se dispare una nueva lanza, siempre habrá 11 proyectiles al mismo tiempo y no se consumirán más recursos por instanciar nuevas lanzas en la escena.

En cuanto a la construcción del mapa, debido a que la estructura es simple y la experiencia no requiere evitar su uso (a diferencia de los casos anteriores), se utilizan tres nodos **GridMap** para el piso y techo, paredes horizontales y paredes verticales respectivamente, utilizando una **MeshLibrary** construida a partir de las *meshes* extraídas de uno de los paquetes de assets. Las trampas no son más que un nodo base **Node3D** con un *script* para disparar los proyectiles y tres nodos **Timer** que se utilizan para enviar una señal al *script* para indicar

cuando puede disparar. Para la iluminación se utiliza la antorcha descrita en la experiencia 1, la cual es equipada a la cámara del jugador a modo de linterna y es visible en la parte inferior derecha de esta. Por otro lado, cada proyectil se encuentra compuesto por un nodo **MeshInstance3D** que representa la lanza, un nodo **CollisionShape3D** para que interactúe con el resto del mapa, un nodo **Area3D** con su respectivo nodo **CollisionShape3D** para detectar cuando el proyectil ha impactado algo y, finalmente, un nodo **VisibleOnScreenNotifier3D** para permitir llevar la cuenta de cuántas instancias de la lanza se están mostrando en pantalla. Por último, se reutiliza el cofre del tesoro y el mensaje del objetivo de las secciones anteriores. En la figura 14 puede apreciarse una vista externa de lo anterior, tomada desde el editor de Godot, dejando en evidencia la estructura simple de este mapa.

De la misma manera que en las experiencias anteriores, el cofre del tesoro, el cual puede apreciarse en la figura 15, actúa como incentivo para que el jugador interactúe con la experiencia. La diferencia radica en que, en el caso donde la técnica de optimización no se encuentra implementada, el número de entidades en la escena irá aumentando a medida que pasa el tiempo, por lo que se busca impedir que el jugador pueda terminar demasiado pronto. Es por esto que, si una lanza impacta al jugador cuando este intenta cruzar el pasillo hacia el cofre del tesoro, se utiliza código para devolver al jugador al inicio, forzándolo a tomarse su tiempo para evitar las trampas y permitiendo que las lanzas logren acumularse. Para el caso en que sí esté activado *object pooling*, el número de lanzas nunca superará las 198 (11 proyectiles de munición por 18 trampas), garantizando que su impacto en el rendimiento no aumente a medida que pasa el tiempo. También es importante considerar que la instanciación de cada lanza no implica un impacto en el rendimiento para este caso, debido a que todas fueron instanciadas a la vez al inicio de la experiencia, a diferencia del otro caso en donde se instancian continuamente.

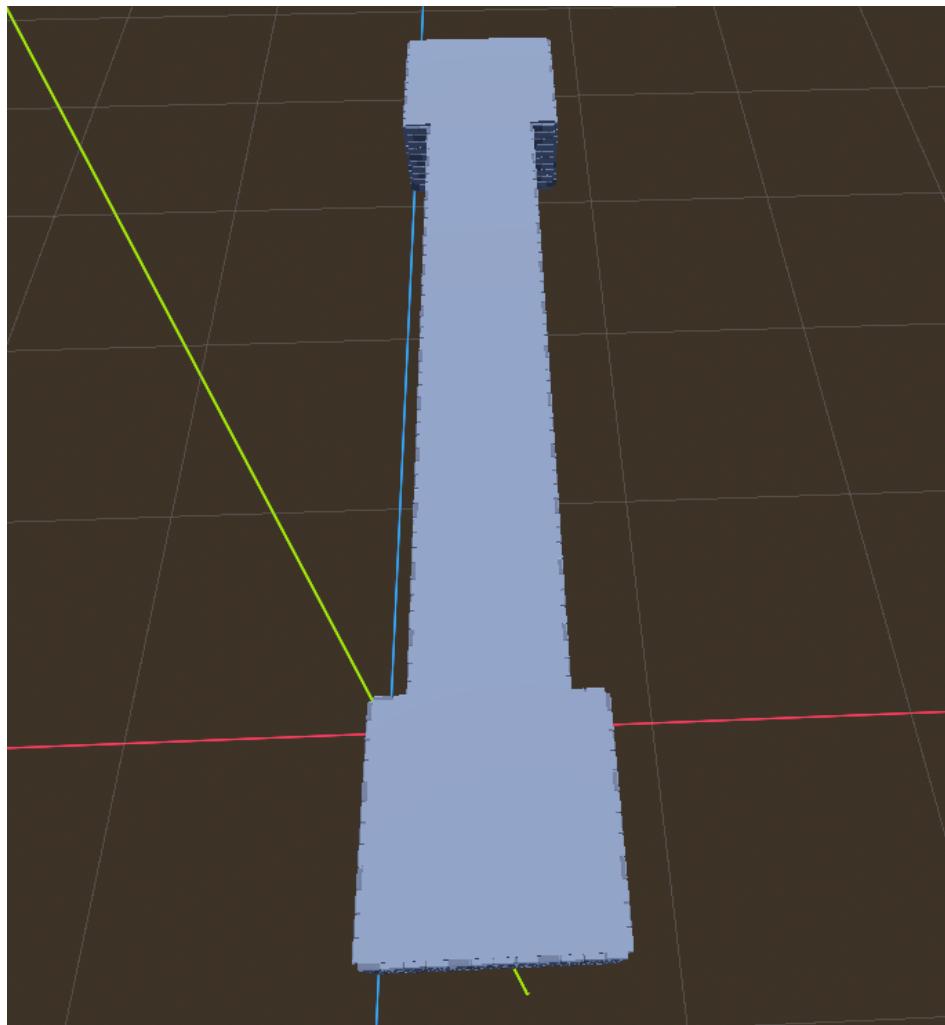


Figura 14: Vista externa del mapa creado para la experiencia *object pooling*.
Fuente: Elaboración Propia.

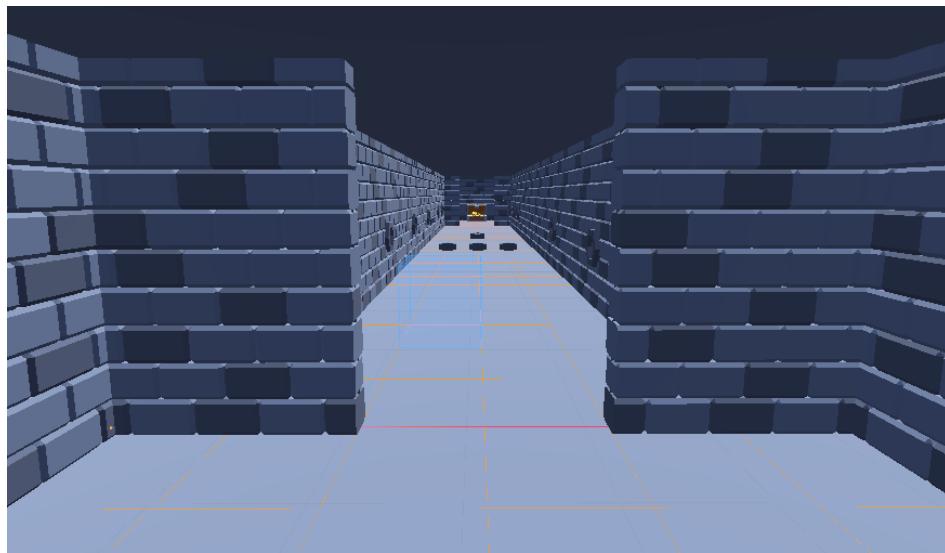


Figura 15: Vista interna del mapa creado para la experiencia *object pooling*.
Fuente: Elaboración Propia.

3.2.10. Experiencia 4: Renderizado por chunks

Para la implementación del *renderizado por chunks*, se utilizan dos salas simples conectadas por un pasillo, el cual tiene forma de “U”, por lo que no es posible observar ambas salas a la vez. A diferencia de las experiencias anteriores, esta no hace uso del cofre del tesoro. En cambio, la habitación inicial tiene una puerta cerrada, mientras que la otra tiene la llave necesaria para abrirla.

Para la implementación de la técnica de optimización, se divide el mapa antes descrito en 4 *chunks*: la sala inicial, el pasillo dividido en dos mitades y la sala con la llave. Estos son enumerados del 1 al 4, permitiendo recorrerlos de manera secuencial, positiva o negativamente, a diferencia del uso común de una grilla. La distancia de renderizado es 1, lo que significa que siempre estarán cargados el *chunk* en el que se encuentra el jugador y los inmediatamente adyacentes a él (ejemplo: si el jugador se encuentra en el *chunk* 2, además estarán cargados el 1 y el 3). Lo anterior es controlado a través de código utilizando un sistema sencillo de detección del *chunk* actual a través de una serie de áreas que envían señales cuando se entra o sale de ellas. Para el caso en donde no se utilice esta forma de optimización, existe otra implementación del mapa, la cual no tiene áreas de detección del *chunk* actual y todos los *chunks* se encuentran cargados en todo momento.

En cuanto a la construcción del mapa, al igual que en la experiencia anterior, cada *chunk* hace uso de tres nodos **GridMap** para la construcción del suelo y techo, paredes horizontales y paredes verticales, respectivamente. Para la puerta se utiliza un nodo **MeshInstance3D** para representar su *mesh*, un nodo **StaticBody3D** con su respectivo nodo **CollisionShape3D** para detectar colisiones con el jugador y un nodo **Area3D** con su respectivo nodo **CollisionShape3D** para detectar si el jugador se encuentra frente a ella. Para simular un *chunk* sobre-cargado de entidades, se utilizan distintos *props* extraídos de los paquetes de assets descargados, los cuales son una estantería, un libro, un barril y una cajonera. Cada uno de ellos está conformado por un nodo **MeshInstance3D** y un nodo **StaticBody3D** con su respectivo nodo **CollisionShape3D**, a excepción de la cajonera, la cual utiliza cinco nodos **MeshInstance3D** para representar su *mesh* compuesta por la estructura principal y sus cuatro cajones. Por otro lado, la llave utiliza un nodo **MeshInstance3D** para representar su *mesh* y un nodo **Area3D** con su respectivo nodo **CollisionShape3D** para enviar una señal cuando el jugador se acerque lo suficiente para eliminarla por código y mostrar un mensaje en pantalla de que ha sido recogida. Dicho mensaje, junto con el de que la puerta está cerrada y se debe buscar la llave, utiliza la misma estructura que los mensajes de las experiencias anteriores. Para la iluminación se utilizan múltiples copias de la antorcha descrita en las experiencias anteriores. Por último, para detectar el *chunk* actual se utiliza un sistema simple de cuatro áreas, cada una compuesta por un nodo **Area3D** y su nodo **CollisionShape3D**, los cuales envían una señal a un *script* que maneja la carga y descarga de cada *chunk*. En la figura 14 se puede apreciar una vista externa de lo anterior, tomada desde el editor de Godot, donde se muestran todos los *chunks* cargados.

A diferencia de las experiencias anteriores, el incentivo para el jugador es encontrar una llave

para abrir una puerta cerrada en la habitación inicial. Esto se debe a que, como la siguiente sala no se encuentra cargada, no es posible mostrarle el cofre del tesoro al final del pasillo. Para lograr lo anterior, el pasillo tiene forma de "U", impidiendo al jugador ver más allá de sus esquinas, por lo que actúa como una especie de transición entre ambas salas, dándoles tiempo para cargarse y descargarse antes de que el jugador pueda verlas. Esto es posible ya que, al iniciar la experiencia, se cargan los cuatro *chunks*, los cuales son instanciados o eliminados del árbol de la escena según corresponda, gracias a la información provista por las señales emitidas por las áreas que representan el espacio que ocupa cada sección del mapa. La finalidad de todo esto es que el jugador experimente el impacto en el rendimiento al llegar a la sala con la llave, porque se utilizan cientos de *props* apilados para simular el efecto de cargar un *chunk* que se encuentra saturado de entidades, tal y como se muestra en la figura 17. Para el caso en donde no se utilice esta técnica de optimización, el *chunk* saturado de entidades tendrá un impacto en el rendimiento incluso cuando el jugador se encuentre en la habitación inicial, por lo que el rendimiento promedio será menor en comparación con la experiencia optimizada. Cabe destacar que se utiliza una implementación sencilla de renderizado por *chunks* debido a que es una simulación en una escala pequeña, por lo que existen mejores maneras de obtener el mismo resultado en mapas más grandes.

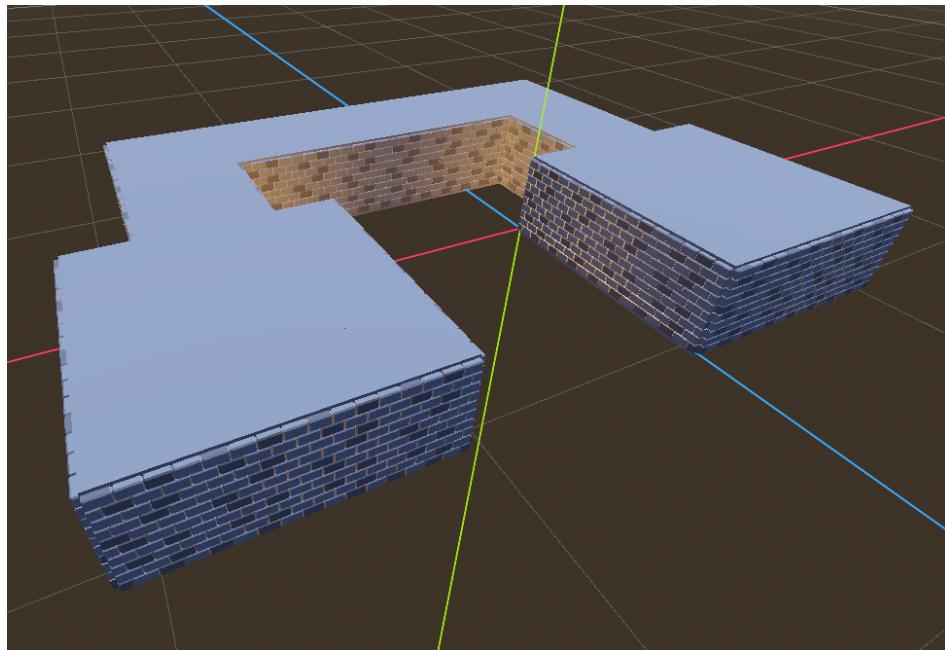


Figura 16: Vista externa del mapa creado para la experiencia *renderizado por chunks*.
Fuente: Elaboración Propia.



Figura 17: Vista interna del mapa creado para la experiencia *renderizado por chunks*.
Fuente: Elaboración Propia.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

Para verificar la veracidad de la implementación de las técnicas descritas en el motor de juegos Godot, se realizaron una serie de pruebas en las experiencias, comparando el impacto en el desempeño al tener activas las optimizaciones implementadas versus un control en el que se jugó cada experiencia sin su respectiva optimización activa. Esto es posible gracias a la implementación de la opción de activar/desactivar cada técnica implementada individualmente, lo que permite obtener métricas de desempeño para cada caso.

Las pruebas fueron realizadas en un computador con las siguientes especificaciones:

- Procesador: 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (2.69 GHz)
- RAM: 16,0 GB
- GPU: NVIDIA GeForce RTX 3060 Laptop GPU

Para evaluar el impacto en el desempeño, las métricas elegidas fueron las siguientes:

- *Frames por segundo o FPS*: una de las métricas más importantes para los jugadores a la hora de revisar el rendimiento. Indica el número de veces que se procesa la escena del videojuego cada segundo (más es mejor).
- *Entidades*: el número de objetos en pantalla para comparar el efecto de las técnicas de optimización (menos es mejor).
- *Drawcalls*: número de llamadas a la GPU para dibujar algo en la pantalla (menos es mejor)

Estas fueron medidas cada segundo de la duración de la experiencia, permitiendo la confección de los distintos gráficos de desempeño por segundo que pueden observarse a continuación. Cabe destacar que para la recolección de datos representativos, v-sync se encuentra desactivado para no limitar los *FPS* al límite del monitor.

4.1. Casos

En esta sección, se detallan los resultados obtenidos al completar cada experiencia, cuidando siempre de mantener un comportamiento y un tiempo de completado similares para cada iteración. Se preparó un caso para cada técnica implementada, comparándolas siempre con su respectivo caso control.

4.1.1. Caso control 1: occlusion culling

Para esta prueba, se busca medir el rendimiento en tres tipos de ubicaciones representativas de las diferentes vistas disponibles, esperando 10 segundos en cada una para obtener una lectura estable. La primera es desde el inicio hasta el cofre del tesoro para abordar una gran cantidad de salas en el caso sin *occlusion culling*; la segunda es frente a una pared para representar el mejor caso de aplicación de oclusión y, por último, la tercera es mirando hacia uno de los pasillos perpendiculares para representar una vista que no favorece su aplicación particularmente.

Los resultados de esta prueba se ven reflejados en las figuras 18, 19 y 20. Se destacan en el eje de los segundos los tramos de 0 a 10, de 14 a 24 y de 29 a 39, los cuales corresponden a las tres ubicaciones representativas que se eligieron para esta prueba. Para cada métrica, se obtiene la moda de cada tramo como su valor representativo, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Tramo 1: 107 FPS, 12585 entidades y 16 *draw calls*
- Tramo 2: 176 FPS, 5391 entidades y 8 *draw calls*
- Tramo 3: 154 FPS, 6002 entidades y 8 *draw calls*
- Promedio: 156 FPS, 7808 entidades y 12 *draw calls*

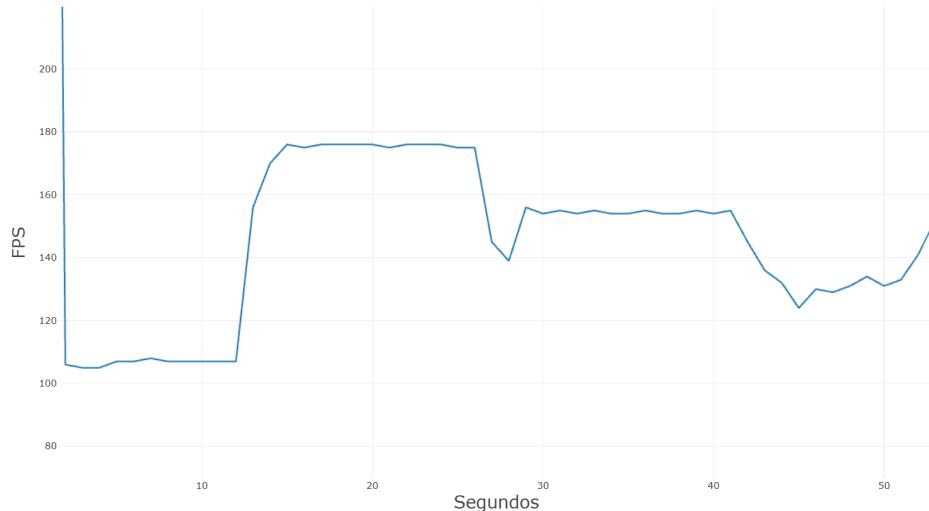


Figura 18: FPS a lo largo del tiempo para caso control de *occlusion culling*.
Fuente: Creación propia.

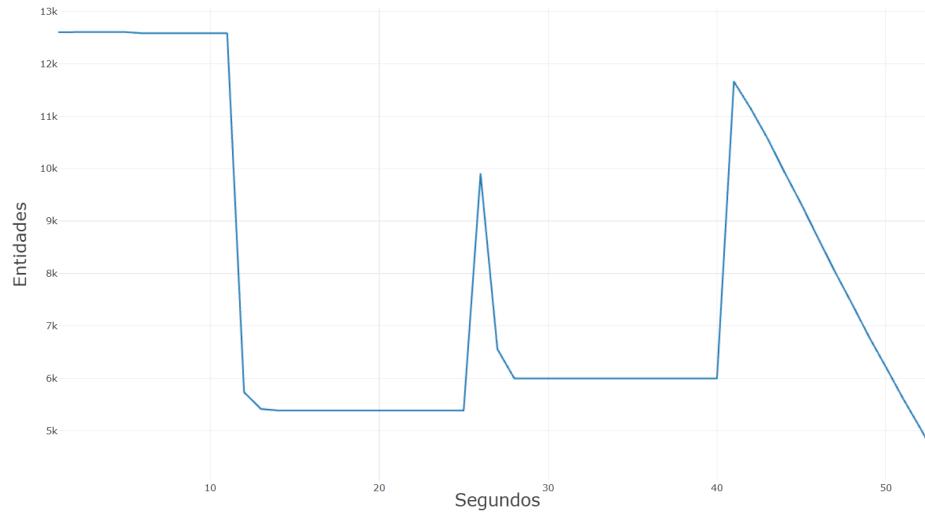


Figura 19: Cantidad de entidades a lo largo del tiempo para caso control de *occlusion culling*.

Fuente: Creación propia.

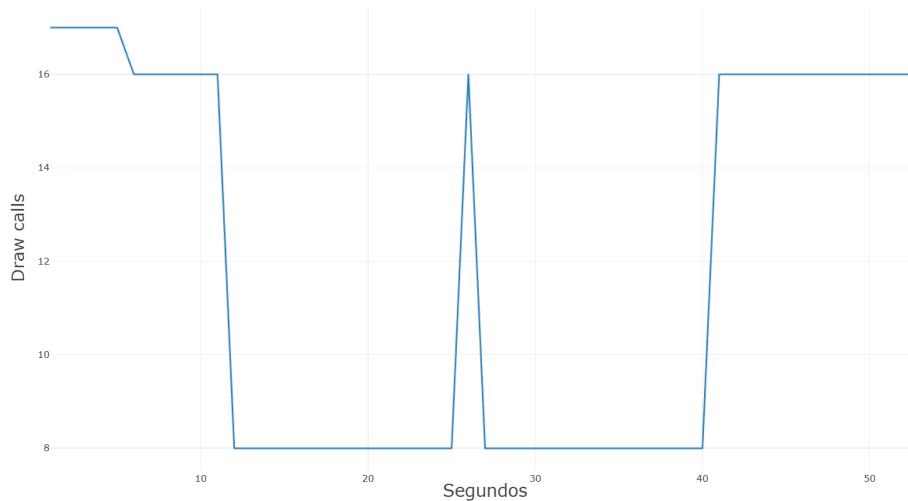


Figura 20: *Draw calls* realizadas a lo largo del tiempo para caso control de *occlusion culling*.

Fuente: Creación propia.

4.1.2. Caso control 2: HLOD

Para este caso en el que se utiliza una forma de optimización que depende de la distancia del jugador a los objetos, el objetivo de la prueba es observar el rendimiento a múltiples distancias. Para lograr esto, se esperan 10 segundos en distintas vistas de distancias variadas a modo de tener una lectura representativa de cada métrica. Los puntos observados son norte, este y sur para el inicio del pasillo; norte y sur (debido a que no cambia la distancia a ambos lados del pasillo) a la mitad; y norte al final, frente al cofre del tesoro, para incluir su *mesh* con el pedestal de cerca.

Los resultados de esta prueba se ven reflejados en las figuras 21, 22 y 23. Se destacan en el eje de los segundos los tramos de 0 a 10, 13 a 23, 26 a 36, 44 a 54, 58 a 68 y 76 a 86, los cuales corresponden a las seis vistas representativas que se eligieron para esta prueba. Para cada métrica, se obtiene la moda de cada tramo como su valor representativo, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Tramo 1: 203 FPS, 3096 entidades y 46 draw calls
- Tramo 2: 204 FPS, 1834 entidades y 37 draw calls
- Tramo 3: 221 FPS, 1994 entidades y 35 draw calls
- Tramo 4: 212 FPS, 2502 entidades y 45 draw calls
- Tramo 5: 212 FPS, 2530 entidades y 37 draw calls
- Tramo 6: 258 FPS, 1946 entidades y 45 draw calls
- Promedio: 228 FPS, 2298 entidades y 41 draw calls

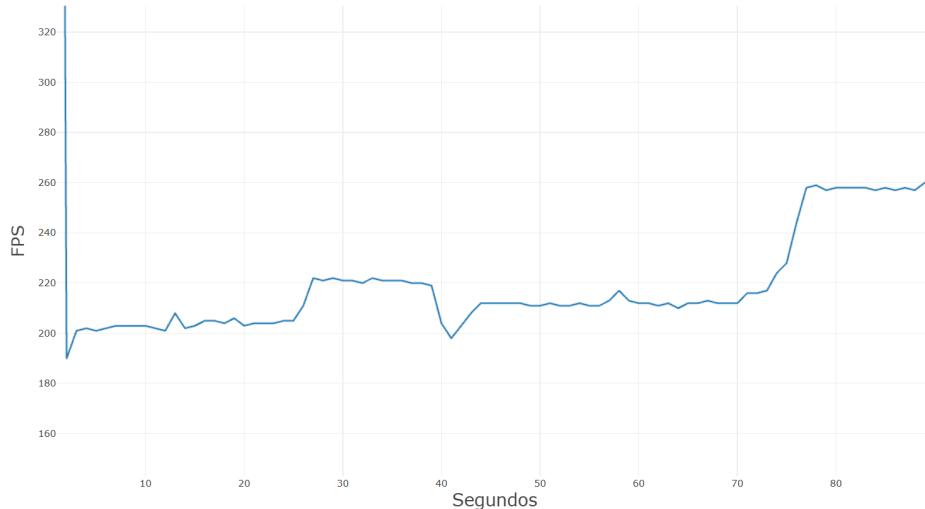


Figura 21: FPS a lo largo del tiempo para caso control de HLOD.

Fuente: Creación propia.

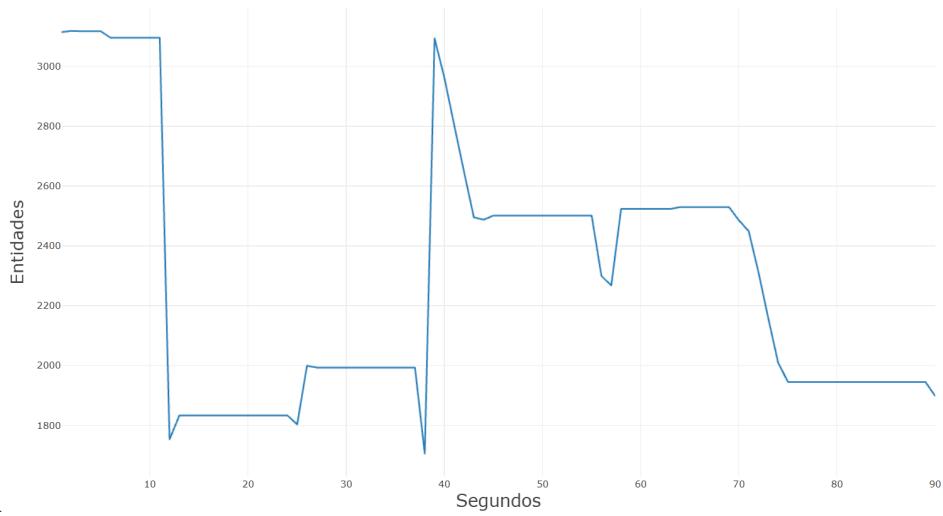


Figura 22: Cantidad de entidades a lo largo del tiempo para caso control de *HLOD*.
Fuente: Creación propia.

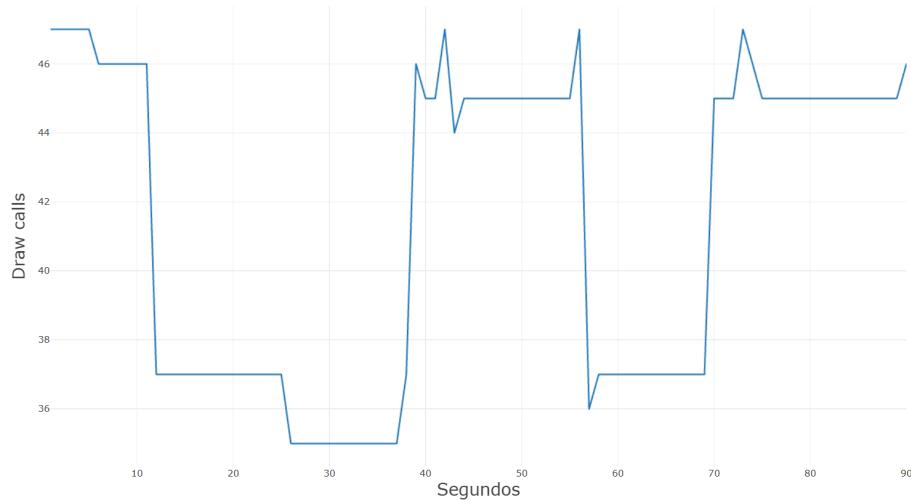


Figura 23: *Draw calls* realizadas a lo largo del tiempo para caso control de *HLOD*.
Fuente: Creación propia.

4.1.3. Caso control 3: object pooling

Debido a que, para este caso, el rendimiento empeora a medida que pasa el tiempo, se esperan 5 minutos al inicio mirando hacia el cofre del tesoro antes de intentar cumplir con el objetivo para permitir que aumente la cantidad de objetos en pantalla y, así, su impacto en el rendimiento.

Los resultados de esta prueba se ven reflejados en las figuras 24, 25 y 26. A diferencia de las demás experiencias, la prueba de este caso involucra un aumento en las entidades directamente proporcional al tiempo transcurrido; por lo que, en el eje de los segundos, existe un único tramo relevante que va de 0 a 300. Para cada métrica, se obtiene el valor al inicio del tramo y su valor final como sus valores representativos, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Inicio: 1160 FPS, 85 entidades y 38 draw calls
- Final: 499 FPS, 6169 entidades y 40 draw calls
- Promedio: 626 FPS, 3024 entidades y 39 draw calls

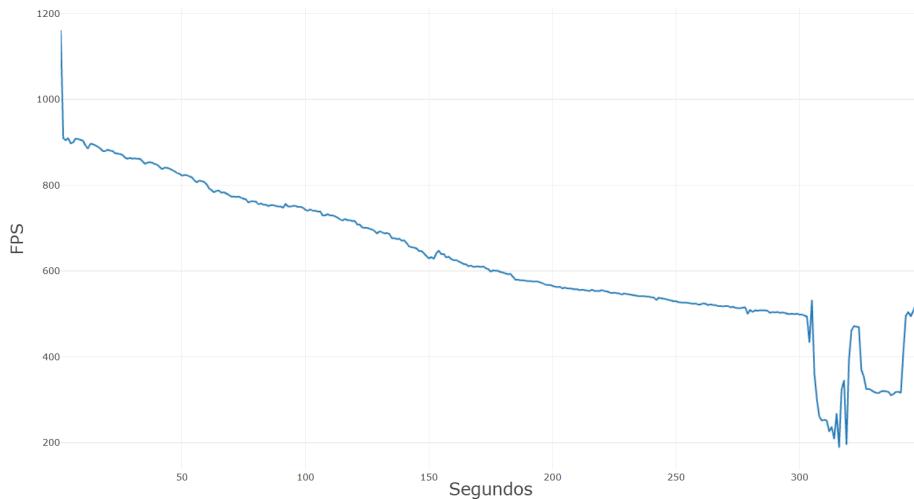


Figura 24: FPS a lo largo del tiempo para caso control de *object pooling*.

Fuente: Creación propia.

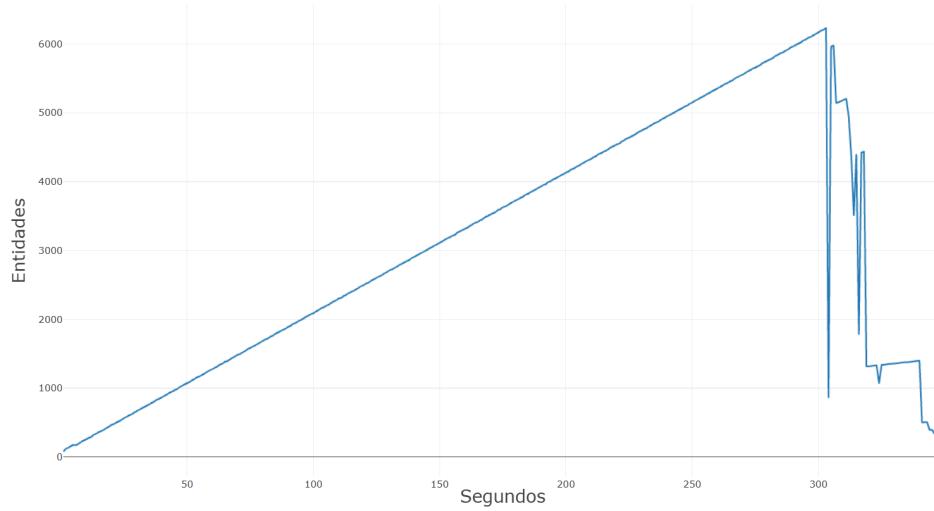


Figura 25: Cantidad de entidades a lo largo del tiempo para caso control de *object pooling*.
Fuente: Creación propia.

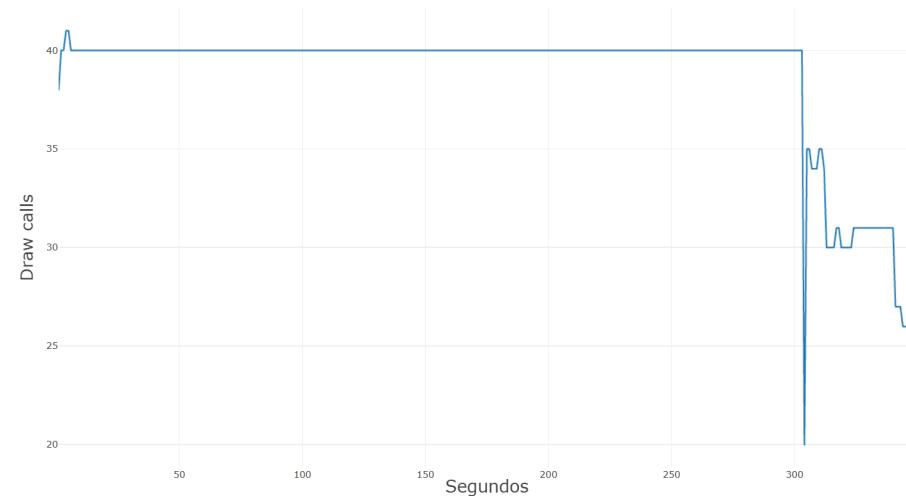


Figura 26: *Draw calls* realizadas a lo largo del tiempo para caso control de *object pooling*.
Fuente: Creación propia.

4.1.4. Caso control 4: renderizado por chunks

En este caso, existen dos puntos importantes de los cuales se pueden obtener datos representativos: la sala con la puerta y la entrada de la sala con la llave. Para aprovecharlos al máximo, se realiza la experiencia tratando de siempre mirar en dirección a la sala con la llave. Para darle tiempo a las métricas para estabilizarse, se esperan 10 segundos en ambos puntos, repitiendo el primer punto antes de finalizar la experiencia para que la fluctuación en el rendimiento sea más visible.

Los resultados de esta prueba se ven reflejados en las figuras 27, 28 y 29. Se destacan en el eje de los segundos los tramos de 3 a 13, 23 a 33 y 47 a 57, los cuales corresponden a las dos ubicaciones representativas que se eligieron para esta prueba, repitiendo la primera antes de finalizar la experiencia. Para cada métrica, se obtiene la moda de cada tramo como su valor representativo, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Tramo 1: 232 FPS, 12276 entidades y 40 draw calls
- Tramo 2: 120 FPS, 12244 entidades y 33 draw calls
- Tramo 3: 229 FPS, 12277 entidades y 41 draw calls
- Promedio: 212 FPS, 10819 entidades y 35 draw calls

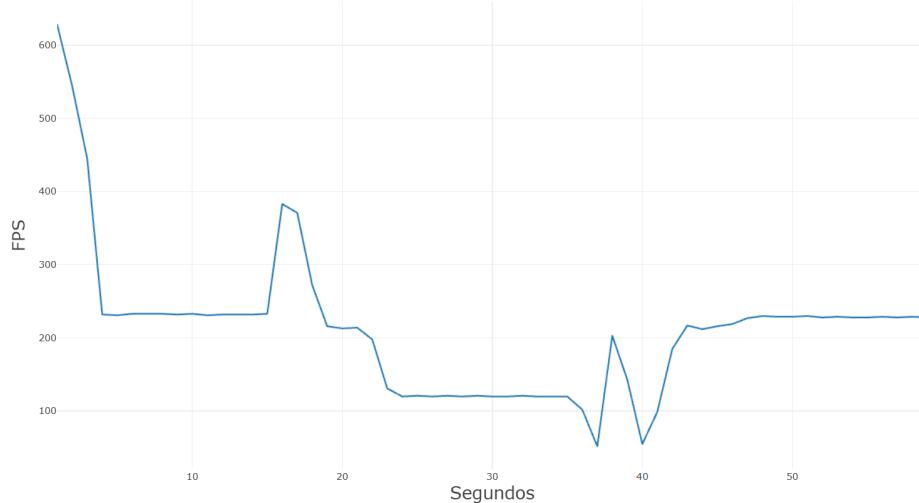


Figura 27: FPS a lo largo del tiempo para caso control de *renderizado por chunks*.
Fuente: Creación propia.

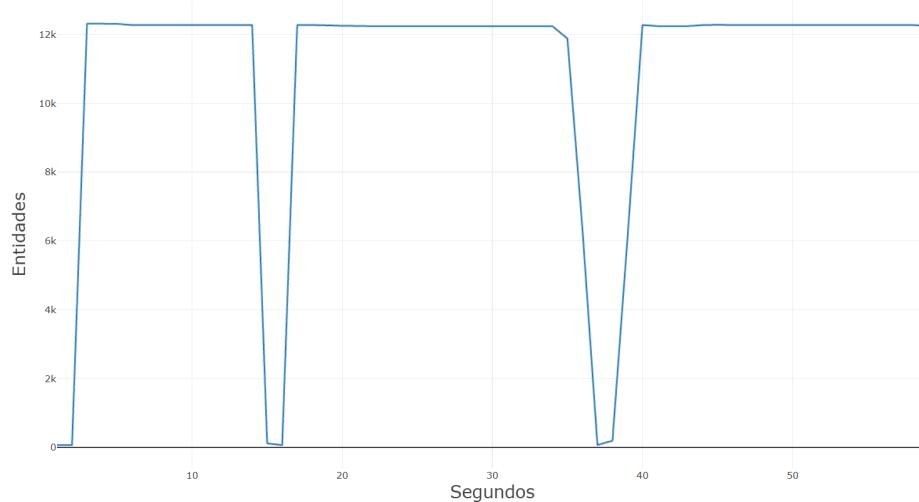


Figura 28: Cantidad de entidades a lo largo del tiempo para caso control de *renderizado por chunks*.

Fuente: Creación propia.

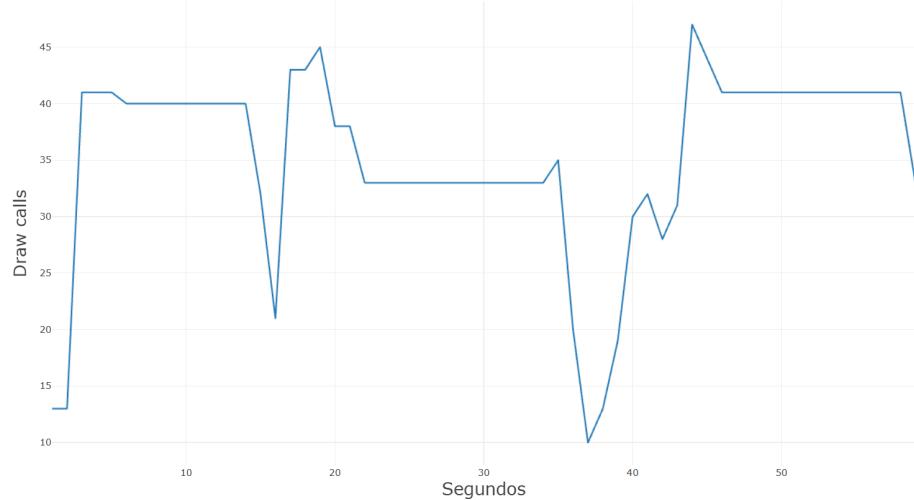


Figura 29: *Draw calls* realizadas a lo largo del tiempo para caso control de *renderizado por chunks*.

Fuente: Creación propia.

4.1.5. Caso 1: occlusion culling

Para que las medidas sean comparables, se repite la misma prueba que en el caso control. El objetivo es analizar el comportamiento del rendimiento en un caso desfavorable sin la técnica de optimización, un caso que demuestre la eficacia de implementar *occlusion culling* y un caso en donde esta forma de optimización no se vea favorecida. Es por esto que se eligieron los puntos listados en el caso control, respectivamente.

Los resultados de esta prueba se ven reflejados en las figuras 30, 31 y 32. Se destacan en el eje de los segundos los tramos de 0 a 10, de 14 a 24 y de 29 a 39, los cuales corresponden a las mismas tres ubicaciones utilizadas para el caso control. Para cada métrica, se obtiene la moda de cada tramo como su valor representativo, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Tramo 1: 140 FPS, 747 entidades y 16 *draw calls*
- Tramo 2: 379 FPS, 34 entidades y 4 *draw calls*
- Tramo 3: 236 FPS, 407 entidades y 8 *draw calls*
- Promedio: 245 FPS, 911 entidades y 11 *draw calls*

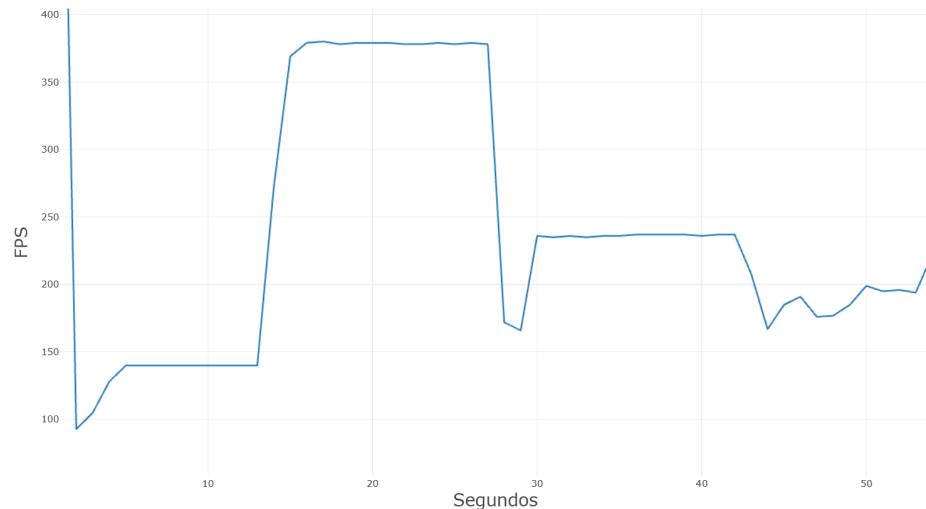


Figura 30: FPS a lo largo del tiempo para caso de *occlusion culling*.
Fuente: Creación propia.

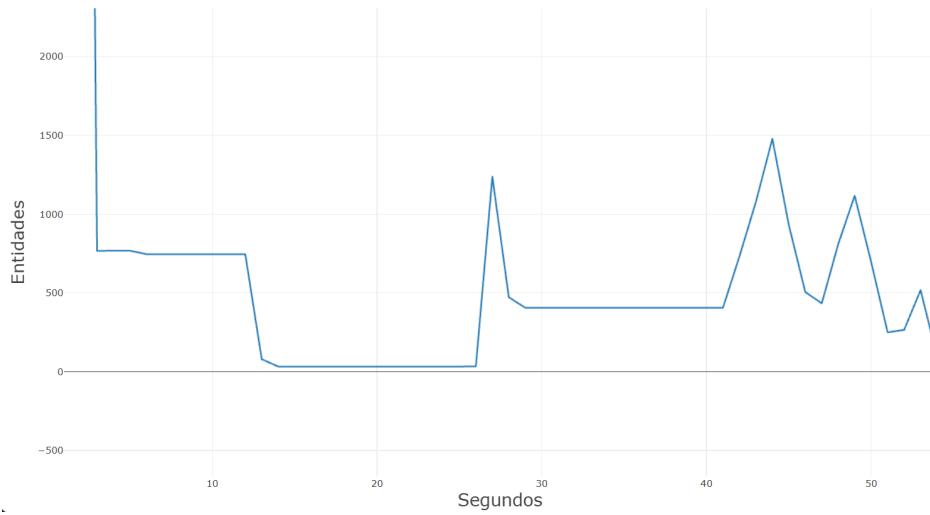


Figura 31: Cantidad de entidades a lo largo del tiempo para caso de *occlusion culling*.
Fuente: Creación propia.

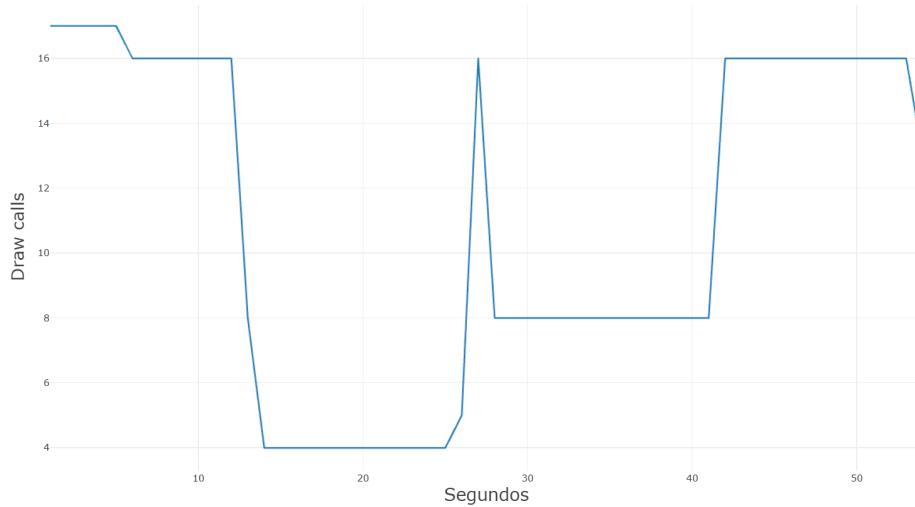


Figura 32: *Draw calls* realizadas a lo largo del tiempo para caso de *occlusion culling*.
Fuente: Creación propia.

4.1.6. Caso 2: HLOD

En este caso, se repite la prueba realizada para el caso control. Lo que se busca mostrar a través de los datos recolectados es la fluctuación del rendimiento al analizar distintos niveles de detalle y la cantidad de entidades en pantalla, utilizando los puntos de vista más representativos.

Los resultados de esta prueba se ven reflejados en las figuras 33, 34 y 35. Se destacan en el eje de los segundos los tramos de 0 a 10, 14 a 24, 28 a 38, 47 a 57, 61 a 71 y 80 a 90, los cuales corresponden a las mismas seis vistas utilizadas en el caso control. Para cada métrica, se obtiene la moda de cada tramo como su valor representativo, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Tramo 1: 386 FPS, 1631 entidades y 35 draw calls
- Tramo 2: 283 FPS, 1124 entidades y 30 draw calls
- Tramo 3: 343 FPS, 1163 entidades y 31 draw calls
- Tramo 4: 362 FPS, 1440 entidades y 35 draw calls
- Tramo 5: 352 FPS, 1343 entidades y 28 draw calls
- Tramo 6: 388 FPS, 1165 entidades y 42 draw calls
- Promedio: 359 FPS, 1309 entidades y 34 draw calls

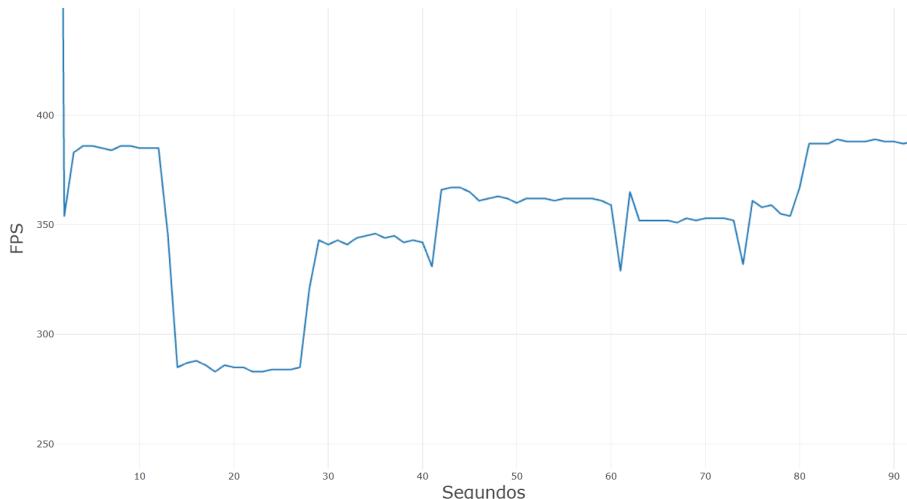


Figura 33: FPS a lo largo del tiempo para caso de HLOD.
Fuente: Creación propia.

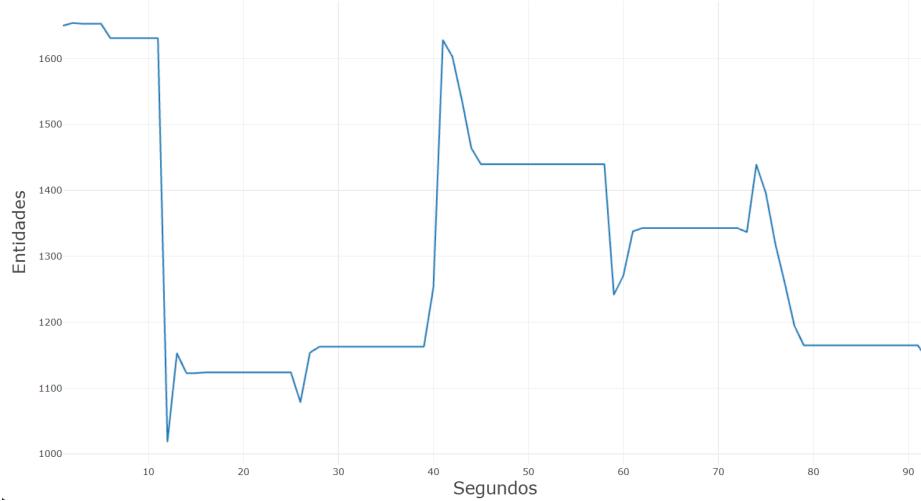


Figura 34: Cantidad de entidades a lo largo del tiempo para caso de *HLOD*.
Fuente: Creación propia.

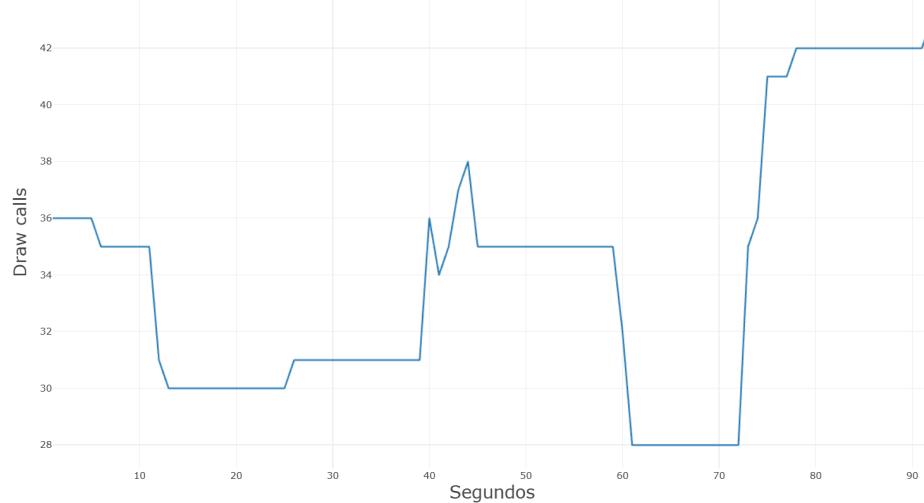


Figura 35: *Draw calls* realizadas a lo largo del tiempo para caso de *HLOD*.
Fuente: Creación propia.

4.1.7. Caso 3: object pooling

Lo que se busca evidenciar con la prueba control es la diferencia entre un caso en el que no existe un límite de entidades y uno en el que se alcanza un límite de entidades que no seguirá aumentando. Se utiliza la misma prueba que en el caso control para alcanzar ese tope de entidades y mostrar como deja de empeorar el rendimiento.

Los resultados de esta prueba se ven reflejados en las figuras 36, 37 y 38. Al igual que en el caso control, la prueba de este caso involucra un aumento en las entidades directamente proporcional al tiempo transcurrido; por lo que, en el eje de los segundos, se utiliza el

mismo tramo que va de 0 a 300. Para cada métrica, se obtiene el valor al inicio del tramo y su valor final como sus valores representativos, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Inicio: 952 FPS, 85 entidades y 38 draw calls
- Final: 898 FPS, 247 entidades y 40 draw calls
- Promedio: 892 FPS, 228 entidades y 39 draw calls

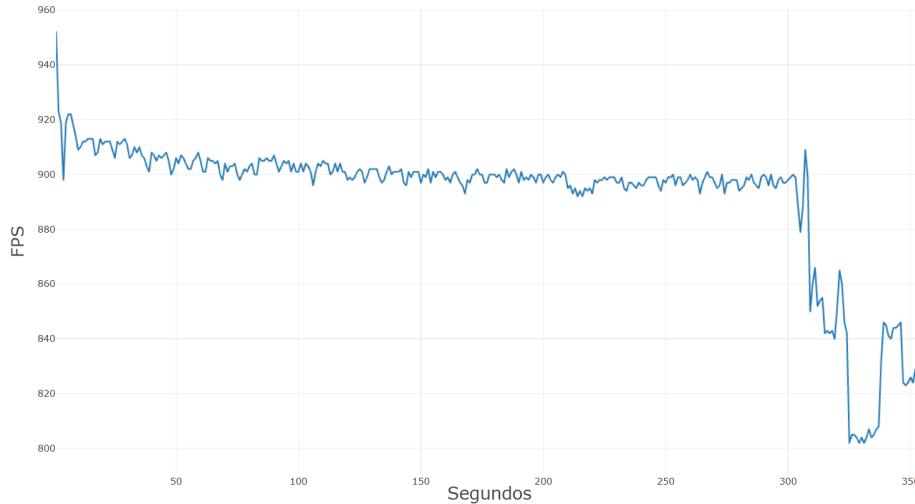


Figura 36: FPS a lo largo del tiempo para caso de *object pooling*.
Fuente: Creación propia.

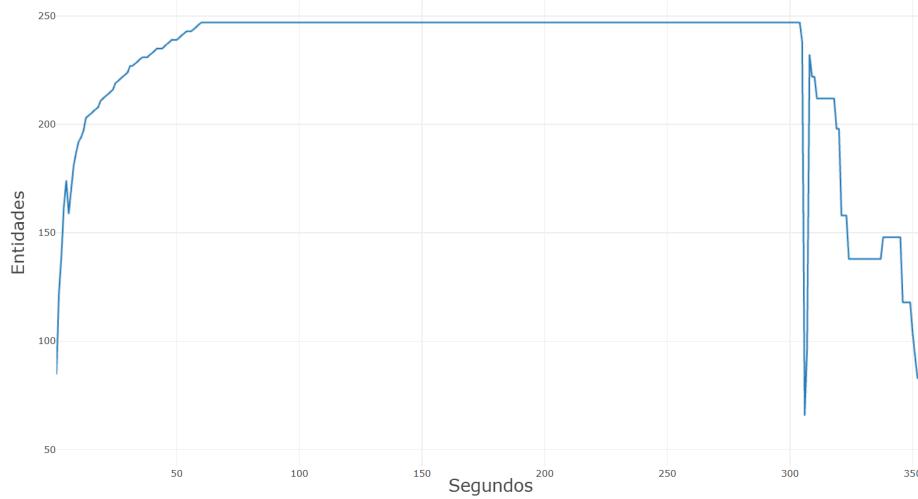


Figura 37: Cantidad de entidades a lo largo del tiempo para caso de *object pooling*.
Fuente: Creación propia.

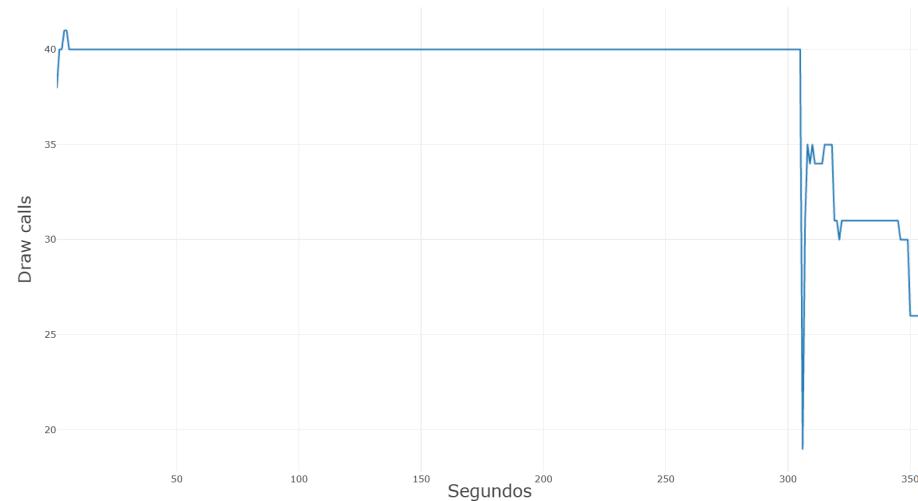


Figura 38: *Draw calls* realizadas a lo largo del tiempo para caso de *object pooling*.
Fuente: Creación propia.

4.1.8. Caso 4: renderizado por chunks

Al igual que en los otros casos, se repite la prueba de control para el caso de tener la técnica de optimización activa, con el objetivo de mostrar como aumenta y disminuye el número de entidades en pantalla de acuerdo a si se encuentra cargado o no el *chunk* de la sala con la llave. Esto se logra gracias a que se mantiene la cámara en dirección hacia dicha sala, manteniendo la máxima cantidad de entidades posible en pantalla.

Los resultados de esta prueba se ven reflejados en las figuras 39, 40 y 41. Se destacan en el eje de los segundos los tramos de 3 a 13, 21 a 31 y 45 a 55, los cuales corresponden a las mismas dos ubicaciones utilizadas en el caso control, repitiendo de la misma manera la primera antes de finalizar la experiencia. Para cada métrica, se obtiene la moda de cada tramo como su valor representativo, además del promedio del total de los datos (aproximado al entero más cercano), obteniéndose lo siguiente:

- Tramo 1: 494 FPS, 51 entidades y 16 draw calls
- Tramo 2: 182 FPS, 12241 entidades y 29 draw calls
- Tramo 3: 456 FPS, 51 entidades y 16 draw calls
- Promedio: 348 FPS, 4239 entidades y 21 draw calls

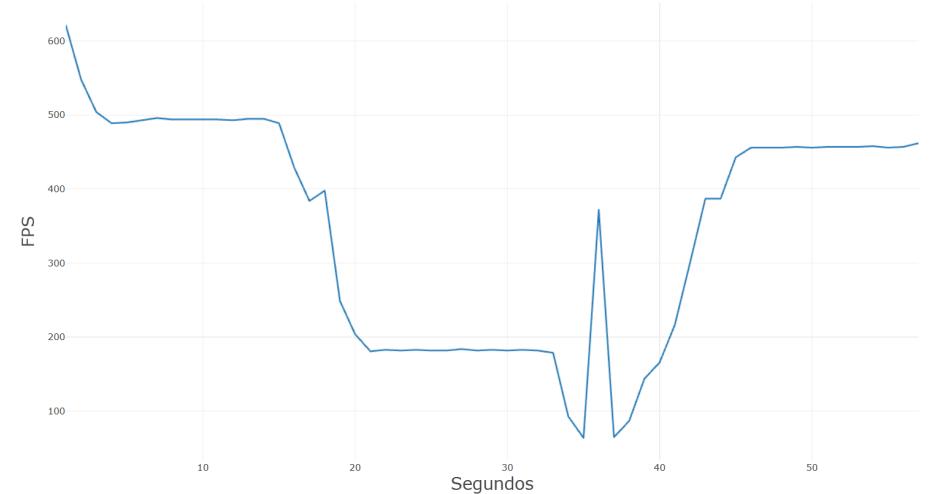


Figura 39: FPS a lo largo del tiempo para caso de *renderizado por chunks*.
Fuente: Creación propia.

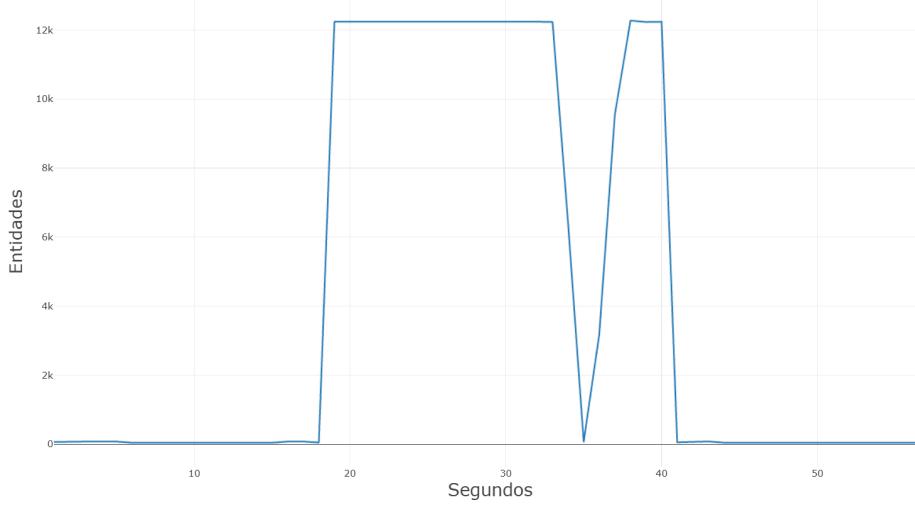


Figura 40: Cantidad de entidades a lo largo del tiempo para caso de *renderizado por chunks*.
Fuente: Creación propia.

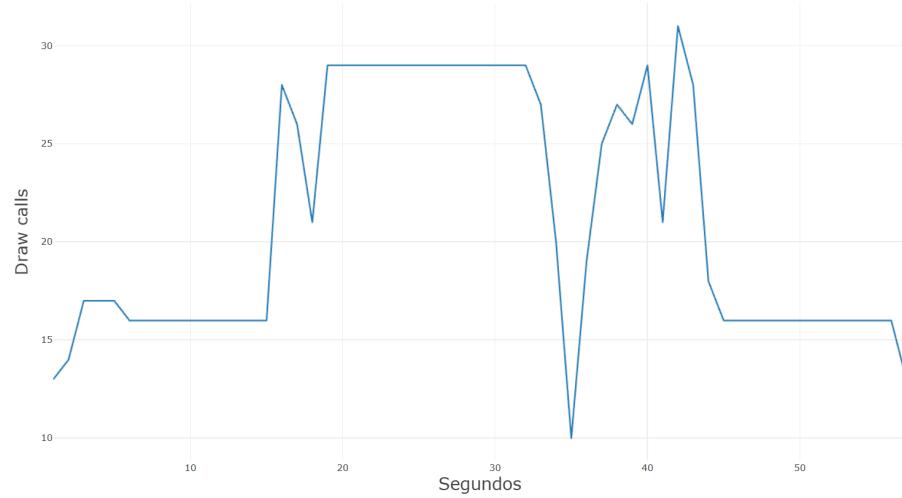


Figura 41: *Draw calls* realizadas a lo largo del tiempo para caso de *renderizado por chunks*.
Fuente: Creación propia.

4.2. Análisis de los datos recolectados

Una vez obtenidos los datos para cada técnica de optimización y sus respectivos casos de control, se confeccionan las siguientes tablas de comparación:

Tramos	FPS control	FPS	% mejora	entidades control	entidades	% mejora	Draw calls control	Draw calls	% mejora
1	107	140	30.84 %	12585	747	94.07 %	16	16	0 %
2	176	379	115.34 %	5391	34	99.37 %	8	4	50 %
3	154	236	53.25 %	6002	407	93.22 %	8	8	0 %
Promedio	156	245	57.05 %	7808	911	88.33 %	12	11	8.33 %

Tabla 1: Comparación control vs *occlusion culling*.

Fuente: Elaboración Propia.

Tramos	FPS control	FPS	% mejora	Entidades control	Entidades	% mejora	Draw calls control	Draw calls	% mejora
1	203	386	90.15 %	3096	1631	47.32 %	46	35	23.91 %
2	204	283	38.73 %	1834	1124	38.71 %	37	30	18.92 %
3	221	343	55.20 %	1994	1163	38.71 %	35	31	11.43 %
4	212	362	70.75 %	2502	1440	42.45 %	45	35	22.22 %
5	212	352	66.04 %	2530	1343	46.92 %	37	28	24.32 %
6	258	388	50.39 %	1946	1165	40.13 %	45	42	6.67 %
Promedio	228	359	57.46 %	2298	1309	43.04 %	41	34	17.07 %

Tabla 2: Comparación control vs *HLOD*.

Fuente: Elaboración Propia.

Tramos	FPS control	FPS	% mejora	Entidades control	Entidades	% mejora	Draw calls control	Draw calls	% mejora
1	1160	952	-17.93 %	85	85	0 %	38	38	0 %
2	499	898	79.96 %	6169	247	96.00 %	40	40	0 %
Promedio	626	892	42.49 %	3024	228	92.46 %	39	39	0 %

Tabla 3: Comparación control vs *object pooling*.

Fuente: Elaboración Propia.

Tramos	FPS control	FPS	% mejora	Entidades control	Entidades	% mejora	Draw calls control	Draw calls	% mejora
1	232	494	112.93 %	12276	51	99.58 %	40	16	60 %
2	120	182	51.67 %	12244	12241	0.02 %	33	29	12.12 %
3	229	456	99.13 %	12277	51	99.58 %	41	16	60.98 %
Promedio	212	348	64.15 %	10819	4239	60.82 %	35	21	40 %

Tabla 4: Comparación control vs *renderizado por chunks*.

Fuente: Elaboración Propia.

En donde se compara cada métrica del control con su respectiva forma de optimización y se obtiene el porcentaje de mejora para cada tipo de dato obtenido en la experiencia con la optimización activa. Para el caso de los *FPS*, se obtiene el porcentaje de cuánto aumentaron en comparación con el caso control. Por otro lado, las entidades miden su mejoría como el porcentaje en que se disminuyó el dato en comparación con el caso control. Por último, las *draw calls*, al igual que las entidades, miden su mejoría en base al porcentaje que disminuyó el dato en comparación con el caso control. El proceso anterior se repite para cada tramo relevante observado en la prueba, además del promedio de los datos obtenidos para una comparación general del rendimiento de la experiencia frente a su control.

4.2.1. Análisis: Occlusion culling

En la tabla 1 se puede observar una clara mejoría en los *FPS* y el número de entidades; no así en las *draw calls*. Para todos los tramos se observa una mejora de más del 90 % en la cantidad de entidades renderizadas, lo cual va acorde al objetivo de esta técnica de optimización, que es evitar el renderizado de lo que no puede verse en pantalla, siendo este su punto más fuerte. También puede observarse que para los tramos 1 y 3, que tienen la mayor cantidad de entidades renderizadas, presentan una mejora importante en los *FPS* de un 30.84 % y un 53.25 % respectivamente, gracias a que, en comparación con el caso control, el número de entidades en pantalla es sumamente bajo. Para el tramo 2 lo anterior es aún más evidente, ya que al observar directamente una pared y aprovechar al máximo la optimización (teniendo tan solo 34 entidades en pantalla), se alcanza una mejora en los *FPS* de un 115.34 %, siendo un claro indicador de la efectividad de utilizar *occlusion culling*. Por último, debido a que no hay presentes muchas entidades diferentes en el mapa, no existen muchas oportunidades de disminución en las *draw calls*, lo cual se ve reflejado en que solo para el tramo 2, el cuál limita las entidades en pantalla a la *mesh* de una pared, se experimenta una mejora del 50 %.

De acuerdo a lo anterior, se deja en evidencia que la implementación de la técnica de optimización *occlusion culling* está justificada para una experiencia de estas características e implica una mejoría en el rendimiento, aumentando en promedio los *FPS* en un 57.05 % y disminuyendo en promedio las entidades y las *draw calls* en un 88.33 % y un 8.33 % respectivamente en este caso.

4.2.2. Análisis: HLOD

De acuerdo a la tabla 2, se puede observar una clara mejoría en todas las métricas. Para los tramos 2, 3 y 6, donde se observa el bosque a una distancia corta, es donde se experimenta la menor mejoría, teniendo respectivamente un 38.73 %, un 55.20 % y un 50.39 % de aumento en los *FPS*; un 38.71 %, un 38.71 % y un 40.13 % de disminución en las entidades y un 18.92 %, un 11.43 % y un 6.67 % de disminución en las *draw calls*. Lo anterior va acorde a la idea de que, mientras más lejos se encuentra el jugador de las *meshes*, mejor rendimien-

to se obtiene con esta técnica de optimización. Por otro lado, el resto de tramos presentan una mejora en *FPS*, entidades y *draw calls* superiores al 60 %, 40 % y 20 % respectivamente. Siendo el tramo 1 el mejor para los *FPS* y las entidades, con un 90.15 % y un 47.32 % de mejoría respectivamente, mientras que para las *draw calls* se tiene el tramo 5 con un 24.32 % de mejoría (debido a que este tramo no incluye las meshes del pedestal con el cofre del tesoro).

Lo anterior deja en evidencia que la implementación de un sistema *HLOD* se encuentra justificada y significa una mejora en el rendimiento para una experiencia de estas características, aumentando en promedio los *FPS* en un 57.46 % y disminuyendo en promedio las entidades y *draw calls* en un 43.04 % y un 17.07 % respectivamente para este caso.

4.2.3. Análisis: Object pooling

La tabla 3 muestra una mejora significativa en los *FPS* y, en especial, en la cantidad de entidades, mientras que en las *draw calls* no se observa impacto alguno. Cabe destacar que, debido a la naturaleza de la implementación, tanto el control como la experiencia con la optimización activa inician en el mismo estado, por lo que no se observan diferencias en los datos obtenidos a excepción de los *FPS*, lo cual pudo ser causado por una diferencia a la hora de cargar ambas experiencias y no por algo significativo. Por otro lado, el tramo 2 muestra el estado de la experiencia luego de que hayan pasado 5 minutos, presentando una mejora del 79.96 % en los *FPS* y una disminución del 96 % en el número de entidades. Debido a que el caso control solo puede empeorar a medida que avanza el tiempo, estos porcentajes solo pueden mejorar, por lo que el impacto positivo aumenta proporcionalmente con el paso del tiempo. En cuanto a las *draw calls*, al ser solo una vista con un número estático en los tipos de *meshes* disponibles, no existen diferencias para esta métrica entre el control y la experiencia con la optimización activa, por lo cual no tiene impacto en el análisis.

Por lo anterior, puede determinarse que la implementación de la técnica de optimización *object pooling* se encuentra justificada para una experiencia de estas características e implica una mejora en el rendimiento, aumentando en promedio los *FPS* en un 42.49 % y disminuyendo en promedio las entidades en pantalla en un 92.46 %, sin afectar, en este caso, a las *draw calls*.

4.2.4. Análisis: Renderizado por chunks

Según la tabla 4 se puede observar un impacto positivo en el rendimiento en todas las métricas. Como los tramos 1 y 3 se tomaron de la misma manera y en el mismo *chunk* (a modo de identificar diferencias causadas al descargar la sala con la llave), presentan datos casi idénticos, mostrando respectivamente una mejora de un 112.93 % y un 99.13 % para los *FPS*, un 99.58 % en ambas para las entidades y una mejora de un 60 % y un 60.98 % en las *draw calls*. Por otro lado, el tramo 2 fue tomado en el *chunk* que contribuye la mayor parte del impacto

en el uso de recursos, por lo que presenta los menores índices de mejora. Respectivamente, para cada métrica presenta una mejora de un 51.67 %, un 0.02 % y un 12.12 %, en donde estas vienen dadas por la descarga del *chunk* inicial.

De acuerdo a lo anterior, se evidencia que la implementación de la técnica de optimización *renderizado por chunks* se encuentra justificada para experiencias de estas características y además implica una mejora en el rendimiento, aumentando en promedio los *FPS* en un 64.15 % y disminuyendo en promedio el número de entidades y las *draw calls* en un 60.82 % y un 40 % respectivamente en este caso.

CAPÍTULO 5

CONCLUSIONES

De acuerdo a los análisis realizados, se puede ver que existen situaciones en las que las técnicas de optimización aplicadas resultan en un impacto positivo para el rendimiento, demostrando que, dependiendo del caso, su implementación puede ser sumamente efectiva. Cabe destacar que, si bien para el propósito de este proyecto, se realizó una experiencia por implementación, las formas de optimización presentadas no son mutuamente exclusivas, por lo que perfectamente se puede implementar un sistema de optimización que las involucre todas e incluso agregar otras más. Se debe recordar que, al fin y al cabo, no son nada más que herramientas que, mientras su implementación vaya acorde con el diseño del videojuego (o viceversa), representan una mejora en el manejo de recursos para ayudar a los desarrolladores a llevar a cabo sus proyectos.

Como resumen de las características importantes de cada una de las implementaciones de las técnicas de optimización abordadas, se tiene que:

- Una implementación de *occlusion culling* es útil cuando se tiene una escena en interiores, donde existan múltiples habitaciones cuyas paredes produzcan oportunidades de oclusión para dejar de renderizar habitaciones contiguas con todas las entidades que las conforman. Es importante recordar que su implementación implica un impacto en el rendimiento, por lo que es importante evaluar que su inclusión en el proyecto signifique un aporte al rendimiento.
- La implementación de un sistema *HLOD* es útil cuando se tiene una escena en exteriores o lo suficientemente amplia para justificar que entidades que se encuentren dentro del campo de visión pierdan detalle sin afectar la experiencia del jugador, reemplazando grupos de *meshes* por una representativa con pocos detalles.
- Implementar *object pooling* es útil para escenas con entidades que se instancian constantemente, utilizando un conjunto finito de instancias de estas entidades, las cuales se reutilizan en lugar de renderizar nuevas instancias que impacten en el rendimiento.
- una implementación de *renderizado por chunks* es útil cuando se tiene un mapa grande en el cual se quiera lograr una experiencia de juego sin interrupciones, dividiendo dicho mapa en *chunks* que solo se cargan si se encuentran a una distancia del jugador denominada distancia de renderizado.

La razón por la cual se escogieron las métricas *FPS* y entidades fue debido a que representan las mejores medidas de rendimiento para esta memoria. Los *FPS* representan una medida que todo jugador y desarrollador puede reconocer fácilmente como determinante para saber si un videojuego “corre” bien o no, convirtiéndose en la métrica más importante a la hora de

hablar del rendimiento en el mundo *gamer*. Ya sea un desarrollador buscando optimizar su videojuego o un jugador queriendo mejorar su *hardware*, ambos grupos se enfocan en hacer subir este número. Por otro lado, el número de entidades es la exemplificación predilecta para demostrar que la solución propuesta hace lo que su nombre indica, que es manejar las entidades de forma tal que se mejore el desempeño. Por esta razón se determinó que era importante enseñarle al jugador como se comportaban las entidades en un caso control y así demostrar que las optimizaciones implementadas tienen un efecto en ellas, ya sea evitando que se rendericen, limitando su uso o cambiando la forma en las que son renderizadas, todo sin cambiar la experiencia del jugador de manera significativa.

A diferencia de las métricas mencionadas, el uso de *draw calls* como medidor de rendimiento fue una decisión más experimental, ya que el objetivo de su inclusión fue, más allá de medir el impacto que las técnicas de optimización tenían en ellas, hablar de ellas, su uso y qué tipo de experiencia tenía un mayor efecto en los datos obtenidos. Gracias a esto, se pudo dejar en evidencia que, debido a que *Godot* procesa múltiples instancias de una misma *mesh* en una *draw call* (o el número de *draw calls* necesario para dibujar una *mesh* y sus componentes), una de las mejores formas de mantener bajo el número de *draw calls* es evitar el uso de muchas *meshes* únicas, es decir, reutilizar *meshes* a través de múltiples instancias. Esto puede lograrse a través del uso de *assets* modulares, los cuales están diseñados para usarse como bloques de construcción repetitivos.

En cuanto a los objetivos planteados para la solución, fue posible cumplirlos en su totalidad:

1. Se dio una explicación de lo que se entendía por “Entidad” y como impacta en el rendimiento en el marco conceptual, además de usarse como métrica para determinar la eficacia del sistema producido.
2. Se eligieron cuatro técnicas de optimización populares para entregar un ejemplo a los desarrolladores de cómo implementarlas y cuándo usarlas.
3. Se utilizaron, como se describió anteriormente, dos métricas sumamente relevantes para medir el impacto de las entidades en cada escena, además de una métrica importante pero poco utilizada para mostrar su uso y las situaciones que la afectan.
4. Se diseñó y produjo un sistema en el motor de juegos *Godot*, el cual manipula las entidades de tal manera que se obtenga una mejora en el rendimiento.
5. Se construyó una experiencia para cada una de las técnicas de optimización implementadas, diseñadas específicamente para resaltar sus fortalezas y casos de uso, además de incluir la opción de desactivar las optimizaciones para tener un caso control para cada una con la cual se pueden comparar los resultados de las métricas obtenidas.

Por otro lado, es importante recordar que la solución propuesta viene dada dentro del contexto de potenciar la escena indie. Específicamente, el objetivo final es que el sistema desarrollado, en conjunto con este escrito, sea una herramienta de aprendizaje para desarrolladores que buscan optimizar los videojuegos en los que trabajan, o incluso desarrolladores

que buscan empezar a adentrarse en el mundo de los videojuegos y quieran utilizar este proyecto como ejemplo. Para lograr esto, además de explicar el funcionamiento de las técnicas de optimización utilizadas, en la propuesta de solución se buscó hacer énfasis en cómo se construyó cada escena, explicando los menús, los mapas de las experiencias e incluso la representación del jugador. Se explicó, dentro del alcance de esta memoria, los nodos utilizados y la razón de su uso, todo dentro del contexto de trabajar en el motor de juegos Godot para que quienes lean este documento puedan entender de mejor manera la estructura del proyecto provisto a través de *GitHub* y utilizarlo como referencia para sus propios proyectos.

Es debido a lo anterior que se tomó la decisión de utilizar Godot. Más allá de la controversia que tuvo *Unity*, lo que causó un aumento en el interés en Godot, uno de sus más grandes beneficios es el hecho de que, al ser de código abierto, es un motor de juegos mantenido por la comunidad, la cual trabaja constantemente para responder dudas en sus foros, tomar sugerencias e implementar funciones que sean beneficiosas para los desarrolladores. Además, permite la creación de complementos o *plugins* que la comunidad comparte para añadir nuevas funcionalidades a parte de las actualizaciones estables. Esto es importante debido a que es un ambiente propicio para que desarrolladores de todo tipo puedan crear nuevos proyectos sin miedo a restricciones o a pérdida de soporte, gracias a una documentación extensa y detallada con un repositorio público mantenido por otros desarrolladores de la comunidad. Por estas razones es que esta memoria busca ser una introducción e invitación para probar el motor y todo lo que ofrece.

A través de la solución, se pudo dejar en evidencia que, no solo es posible implementar un sistema de manejo de entidades para optimizar los recursos utilizados en un videojuego, sino que también las características que posee Godot para facilitar la implementación de sistemas como *occlusion culling* y su generación automática de oclusores, *meshes LOD* con manejo manual y a nivel de proyecto de la agresión en sus transiciones, o el control de la distancia de renderizado a través de sistemas *HLOD*, hacen de este motor una gran opción cuando se trata de optimizar y personalizar proyectos. Godot demuestra ser un motor poderoso y una opción competente entre los motores más populares, con una interfaz sencilla y no más difícil de aprender.

Hoy en día, potenciado con el avance de la inteligencia artificial, se ha visto un deterioro en la calidad de los videojuegos AAA, como ya se mencionó en la definición del problema. Sin embargo, cada vez más se ven nuevos equipos de desarrolladores indie obteniendo reconocimiento, ya sea por sus ideas innovadoras, interacciones con la comunidad e incluso nominaciones a premios. Todo lo anterior sirve para reforzar la idea de que es importante fomentar y ayudar a los nuevos desarrolladores a completar sus proyectos y que estos estén en un estado apropiado para ser publicados. Es por esto que, para una futura ampliación de esta memoria, se podría incluir una guía que entre en detalle sobre cómo comenzar a desarrollar un videojuego en Godot, incluyendo:

- Metodologías típicas
- Opciones de construcción de mapas

- Iluminación
- Buenas prácticas
- Familiarización con los nodos más usados
- Plantillas básicas para géneros conocidos (primera persona, tercera persona, RTS, etc)

Además, se podría aumentar el alcance al entorno 2D y las oportunidades de optimización que este presenta, y/o incluir un caso en el que una experiencia tenga todas las técnicas de optimización utilizadas. Esto sería un aporte importante para los nuevos desarrolladores, ya que los ayudaría a disminuir la cantidad de ensayo y error necesaria para el aprendizaje de cualquier motor de juegos, debido a la gran cantidad de características que estos poseen.

A modo de reflexión personal, considero de gran importancia los trabajos de este estilo, debido a que, a la hora de crear este sistema de optimización, hubo mucho que no sabía o que tenía una idea incorrecta sobre cómo funcionan ciertas características de Godot. El proceso de creación del proyecto involucró no solo entender cómo funcionaban las técnicas de optimización implementadas, sino que también averiguar cuando estas son útiles para aplicarlas correctamente y cómo diseñar un nivel que muestre sus fortalezas. El sistema pasó por múltiples iteraciones debido a que siempre hay un detalle que se puede mejorar o una característica que reemplaza secciones completas de código. Si bien aprendí muchas cosas en la confección de estas experiencias simples, aún hay muchas características que no se abordaron dentro del alcance de la memoria, tales como distintos tipos de nodos que podrían haber significado un mayor aumento en el rendimiento (como puede ser el correcto uso del nodo **MultiMeshInstance3D** en relación a *HLOD* o las *draw calls*). Por lo anterior, a pesar de que su documentación se encuentre bien detallada, considero que el siguiente paso para esta memoria es la creación de la guía para Godot antes descrita, porque la consolidación de esa información significaría para los nuevos usuarios el empezar a experimentar mucho antes y facilitar el aprendizaje, el cuál puede ser intimidante por la cantidad de características.

Otra razón por la que considero que más personas deberían probar Godot es que hoy en día existe bastante contenido de calidad, como videos tutoriales para aprender paso a paso, series en donde se cubren distintos tópicos de un género y múltiples introducciones a características del motor, por lo que se puede ver que la comunidad está activa y hay un interés en la creación de contenido educativo. Siendo este un motor de juegos que funciona de manera similar a *Unity* (por lo que pasar de un motor a otro no implica comenzar desde cero) y no incluye planes o tarifas de servicio, es una gran opción para aprender a utilizar y realizar proyectos.

Finalmente, considero que, ya sea en Godot, *Unity*, *Unreal engine* o cualquier otro motor, resulta sumamente importante apoyar a los nuevos desarrolladores y estudios emergentes. No solo se tienen ceremonias como los *Game Awards*, sino que basta con ver cuáles fueron los videojuegos más esperados y populares del último año (2025) para darse cuenta que la gente desea el contenido más experimental y valora el esfuerzo de los desarrolladores. En un

ambiente AAA en donde cada vez más se empujan los límites de lo que se les permite hacer a las grandes empresas, teniendo ejemplos de franquicias aclamadas que en cada instalación incluyen microtransacciones más agresivas, precios más altos e incluso tratando de justificar el uso de contenido generado por inteligencia artificial, se vuelve de vital importancia que, como consumidores, demostremos nuestro apoyo a aquellos videojuegos que demuestran haber sido creados con cariño y pasión por el proyecto en el que desarrolladores, diseñadores, animadores, actores de voz y todos quienes trabajaron en su creación, lo hayan hecho porque creían en la calidad del contenido que publicaron.

Espero poder haber enseñado algo o compartido alguna información útil, pero como mínimo quisiera extender la invitación de, sin importar cuál sea tu motor preferido o si es tu primera vez, que te animes a crear tu propio videojuego, que experimentes el arduo pero satisfactorio trabajo que es llevar tus ideas a algo concreto y que te atrevas a compartir tus proyectos con la comunidad. Todos debemos empezar en alguna parte y, sin importar que tan simples o complejas puedan ser, tus ideas pueden convertirse en el próximo videojuego favorito de alguien. En un mercado saturado de críticas y decepción hacia las grandes empresas, el esfuerzo de los desarrolladores como nosotros es algo que la gente aprecia cada día más.

ANEXOS

Para esta sección, se incluyen canales de *Youtube* los cuales suben contenido útil para aprender sobre las características de *Godot*:

- Majikayo Games
- Single-Minded Ryan
- LegionGames
- FinePoitCGI
- Godotneers
- Ryan Games

Todos los canales mencionados tienen múltiples tutoriales para diversos temas relacionados a *Godot* por lo que son recursos útiles a la hora de aprender a utilizar el motor de juegos.

REFERENCIAS BIBLIOGRÁFICAS

- [Castillo, 2021] Castillo, A. (17/11/2021). Battlefield 2042: Fecha de lanzamiento, precio y tráileres. https://as.com/meristation/2021/11/17/noticias/1637154397_666171.html. Accedido: 09/10/2023 18:35 PM.
- [GINX, 2023] GINX (10/06/2023). Why AAA Games Are Declining. <https://www.youtube.com/watch?v=90uMuuDgBCw>.
- [GitHub Community, a] GitHub Community. Acerca de Git. <https://docs.github.com/es/get-started/start-your-journey/about-github-and-git#about-git>. Accedido: 06/07/2025 16:00 PM.
- [GitHub Community, b] GitHub Community. Acerca de GitHub. <https://docs.github.com/es/get-started/start-your-journey/about-github-and-git#acerca-de-github>. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, a] Godot Community. About Godot Engine. <https://docs.godotengine.org/en/stable/about/introduction.html#about-godot-engine>. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, b] Godot Community. Camera3D. https://docs.godotengine.org/en/stable/classes/class_camera3d.html. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, c] Godot Community. Configuring mesh LOD performance and quality. https://docs.godotengine.org/en/stable/tutorials/3d/mesh_lod.html#configuring-mesh-lod-performance-and-quality. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, d] Godot Community. Culling. <https://docs.godotengine.org/en/stable/tutorials/performing-culling.html#how-it-works>. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, e] Godot Community. How it works. https://docs.godotengine.org/en/stable/tutorials/3d/visibility_ranges.html#how-it-works. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, f] Godot Community. How occlusion culling works in Godot. https://docs.godotengine.org/en/stable/tutorials/3d/occlusion_culling.html#how-occlusion-culling-works-in-godot. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, g] Godot Community. Introduction. https://docs.godotengine.org/en/stable/tutorials/3d/mesh_lod.html#introduction. Accedido: 06/07/2025 16:00 PM.
- [Godot Community, h] Godot Community. MeshInstance3D. https://docs.godotengine.org/en/stable/classes/class_meshinstance3d.html. Accedido: 06/07/2025 16:00 PM.

[Godot Community, i] Godot Community. Setting up occlusion culling. https://docs.godotengine.org/en/stable/tutorials/3d/occlusion_culling.html#setting-up-occlusion-culling. Accedido: 06/07/2025 16:00 PM.

[GoingInside, 2023] GoingInside (06/09/2023). How This Indie Game KILLED Battlefield. <https://www.youtube.com/watch?v=B0OX8WH3bP0>.

[Nasu, 2023] Nasu (28/01/2023). The Absolute State of Unfinished AAA Game Releases. <https://www.youtube.com/watch?v=fW9Sw48mNMI>.

[Pawlik, 2023] Pawlik, M. (07/04/2023). How Much Does a Decent Gaming PC Cost? Complete Guide. <https://www.linkedin.com/pulse/how-much-does-decent-gaming-pc-cost-complete-guide-mike-pawlik/>. Accedido: 09/10/2023 18:34 PM.

[Steam, 2023] Steam (15/06/2023). BattleBit Remastered. https://store.steampowered.com/app/671860/BattleBit_Remastered/. Accedido: 09/10/2023 18:36 PM.

[Steam, 2021] Steam (19/11/2021). Battlefield™ 2042. https://store.steampowered.com/app/1517290/Battlefield_2042/. Accedido: 09/10/2023 18:35 PM.

[Strickland, 2023] Strickland, D. (22/05/2023). Gamers disapprove of \$70 AAA game price, but it's unlikely to change. <https://www.tweaktown.com/news/91263/gamers-disapprove-of-70-aaa-game-price-but-its-unlikely-to-change/index.html>. Accedido: 09/10/2023 18:32 PM.

[Universidad Europea, 2024] Universidad Europea (11/03/2024). ¿Qué es un motor de juegos? <https://creativecampus.universidadeuropea.com/blog/motor-juegos/>. Accedido: 06/07/2025 16:00 PM.