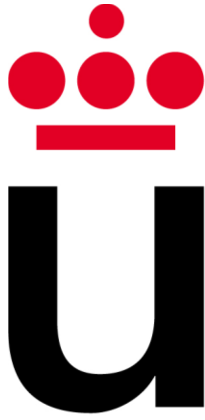


Universidad
Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DE INFORMÁTICA

Iluminación global dinámica a tiempo real y geometría virtualizada en el motor gráfico Unreal Engine 5

TRABAJO FIN DE GRADO

AUTOR: Alexander Teodoro Matos

TUTOR: Rubén Morante González

Agradecimientos:

A mi tutor Rubén Morante González por apoyarme desde los principios de mi carrera como artista y animarme para finalizar esta etapa en el grado.

A mi madre María Catalina Teodoro Gómez, que me ha dado la oportunidad de estudiar lo que he querido.

A mis compañeros de carrera que siempre están para apoyarme.

A los docentes e investigadores del departamento de Grupo de investigación de alto rendimiento de Computación Avanzada, Percepción y Optimización de la Universidad Rey Juan Carlos. Haber coincidido con ellos ha sido un privilegio y un honor.

Resumen:

En el campo de los gráficos por ordenador, el realismo siempre ha sido un desafío. Las técnicas que se han ido desarrollando a lo largo de las décadas han ido desbloqueando elementos del realismo que antes no se podían ofrecer en un medio virtual. Llegados a este punto, el límite está más en el presupuesto relacionado con el medio donde se va a producir: cine, videojuegos, infoarquitectura, etc. Las técnicas necesarias para alcanzar ese realismo, o una imagen bastante convincente, están ya creadas. Debido a las limitaciones técnicas en los gráficos a tiempo real, que necesita sacar muchos fotogramas por segundo, no es viable emplear las mismas técnicas que se emplearían en un render para cine, que puede emplear minutos u horas en sacar un fotograma.

En este proyecto se va a analizar el recorrido que están teniendo las tecnologías relacionadas con el comportamiento de la luz en una escena, y con la gestión del número de polígonos en pantalla. Qué significan los conceptos de geometría virtualizada (*virtualized geometry*) e iluminación global dinámica, por qué no son algo totalmente nuevo, pero sí que son un avance significativo en la manera de trabajar.

Posteriormente, se utiliza el motor gráfico *Unreal Engine 5* para simular la producción de un cortometraje. Para ello, se pondrán a prueba las nuevas implementaciones de este motor para la geometría virtualizada (*Nanite*), que carga en tiempo de ejecución la geometría requerida y luz global dinámica (*Lumen*), que simula rebotes de luz en tiempo real. Este procedimiento rompe con la manera tradicional de tener múltiples modelos de detalle, y una iluminación precalculada en memoria.

Abstract:

In the realm of computer graphics, achieving realism has long been a formidable challenge. Over the years, various techniques have been developed to unlock aspects of realism that were previously unattainable in virtual media. Currently, the main limitation lies in budget constraints, affecting industries such as cinema, video games, and arch viz, but in a different way. The technology required to produce highly realistic or convincing images already exists. However, real-time rendering, which demands a high frame rate (such as 30 or 60 frames per second), poses technical limitations that prevent the use of the same techniques employed in cinematic productions, where minutes or even hours can be spent rendering a single frame.

This project aims to analyse the advances in technologies related to the behaviour of light in a scene and the management of the number of polygons in screen. The study will specifically explore the concepts of virtualized geometry and dynamic global illumination, recognizing that while they are not entirely novel, they represent a noteworthy advancement in current pipelines.

Subsequently, Unreal Engine 5 will be employed to simulate the production of a short film. This will involve comprehensive testing of the engine's new features, namely Nanite for virtualized geometry, which dynamically loads the required geometry during runtime, and Lumen for dynamic global illumination, which accurately simulates real-time light reflections. This approach signifies a departure from the conventional method of storing multiple precalculated models and light reflections in memory.

Tabla de contenido

I. INTRODUCCIÓN 1

1.1. MOTIVACIÓN DEL PROYECTO 1

1.2. PLANTEAMIENTO Y OBJETIVOS DEL PROYECTO 2

1.3. ORGANIZACIÓN DEL DOCUMENTO 3

II. ESTADO DEL ARTE 5

2.1. Iluminación global 5

2.2. Iluminación global con ray-tracing 7

2.2.1. La aparición del Path tracing 10

2.2.2. Ray Tracing y Path Tracing en la industria 10

2.3. Iluminación Global en Unreal Engine 11

2.3.1. Iluminación global precalculada y almacenada en *lightmaps* 11

2.3.2. Iluminación global dinámica 13

2.3.3. Lumen 14

2.4. Futuro de la iluminación a tiempo real 21

2.5. Geometría virtualizada 21

2.5.1. Arquitectura REYES 22

2.5.2. Mapas de Normales, Bump y Displacement 23

2.5.3. Geometría Virtualizada: Nanite 25

III. HERRAMIENTAS Y ENTORNO DE DESARROLLO 31

3.1. 3DS Max 31

3.2. V-Ray 31

3.3. ZBrush 32

3.4. Adobe Substance 32

3.5. Quixel Mixer 32

3.6. Adobe Photoshop 32

3.7. Adobe After Effects 33

3.8. Unreal Engine 33

3.8.1. Quixel Megascans 34

IV. DESARROLLO 35

4.1. ANÁLISIS DE REQUISITOS 35

4.1.1. ANÁLISIS VISUAL 35

4.1.2. LIMITACIONES DEL MOTOR 37

4.1.2.1. Problemas de rendimiento 38
4.1.3. FLUJO DE TRABAJO 39
4.1.4. OBJETIVOS 40
4.1.5. ENTORNO DE TRABAJO Y REQUISITOS MÍNIMOS 40
4.1.6. OBJETIVOS DE RENDER 41
4.2. DESARROLLO DE LAS ESCENAS 42
4.2.1. RECOPILACIÓN DE REFERENCIAS Y PLANTEAMIENTO DE LA ESCENA 42
4.2.2. DESARROLLO DE UN CONCEPTO 44
4.2.3. GUIÓN GRÁFICO Y CÁMARA 46
4.2.4. MODELADO 47
4.2.5. TEXTURIZADO/MATERIALES 48
4.2.6. ILUMINACIÓN 50
4.2.7. ANIMACIÓN Y EFECTOS 51
4.2.8. RENDER 51
4.2.9. COMPOSICIÓN Y MAQUETACIÓN 52
4.3. ITERANDO SOBRE LA IDEA ORIGINAL 52
4.4. EVALUACIÓN DEL SISTEMA 53
V. CONCLUSIONES 57
5.1. PROBLEMAS ENCONTRADOS 58
5.1.1. Problemas con niebla y media textura 59
5.1.2. Problemas de antialiasing 59
5.1.3. Nanite y Path Tracing/Ray Tracing 60
5.1.4. Los detalles se perdían en la oscuridad 61
5.2. TRABAJO FUTURO 62
ANEXO A: REFERENCIAS 63
Referencias 63
ANEXO B: Estadísticas de render 67

Tabla de ilustraciones

Figura 1: Set de rodaje de producción virtual en Mandalorian. Se emplean pantallas de LED en un entorno dentro de Unreal para conseguir la integración de iluminación y el fondo en pantalla.......... 1
Figura 2: Luz directa y luz indirecta provenientes de una fuente de luz. [3] 5
Figura 3: Reflexiones difusas, sangrado de la luz, oclusión ambiental, y otros efectos de GI. [3] 6
Figura 4: Una escena del videojuego Assassin's Creed Unity, donde se aprecia una iluminación convincente proveniente de únicamente una luz direccional y el cielo 7
Figura 5: Ilustración en Treatise on Measurement, donde se aprecia un hombre pintando un laúd.......... 8
Figura 6: A la izquierda, el resultado del método de ray casting de Arthur Appel. A la derecha, un gráfico de su funcionamiento. 9
Figura 7: Render de Turner Whitted de Ray Tracing de unas esferas. Se aprecian en el render la capacidad de translucencia en la bola del centro y la reflexión en la bola de la derecha. [17].... 9
Figura 8: A la derecha, el tamaño de una luz afecta al borde de las sombras. A la izquierda, efectos de motion blur. [18] 9
Figura 9: De izquierda a derecha Bichos: Una aventura en miniatura, Shrek y Monster House.10
Figura 10: El espacio de UVs de esta pieza superpone las 4 caras verticales en el mismo espacio, de manera que se puede aplicar la misma imagen a las 4 caras y optimizar en memoria. [23].......... 11
Figura 11: Ejemplo de UVs para mapeado de luces o lightmap. Se tienen que separar todas las caras del modelo, evitando cualquier tipo de solapamiento. [23] 12
Figura 12: A la izquierda un escenario donde se aprecia la iluminación final, que fusiona los mapas de textura y de luz. A la derecha, el lightmap con toda la información de luz de este modelo. 12
Figura 13: En una escala de Rojo>Amarillo>Verde>Azul>Azul oscuro, se puede visualizar la densidad de las UV del lightmap en cada objeto de la escena. 13
Figura 14: El pipeline de Lumen. Resuelve primero screen tracing, luego emplea o software o hardware raytracing, y si un rayo sigue sin chocar contra una geometría samplea el skylight (la luz del cielo).......... 14
Figura 15: : Ejemplo de ray marching, en el paso 1 un signed distance field desde el primer punto en un rayo trazado, la operación devuelve una distancia al punto más cercano del círculo. Más adelante reevalúa el distance field en varios puntos. 15
Figura 16: Las piezas que componen el pipeline de ray tracing por software. 15
Figura 17: Comparativa de añadir o quitar screen tracing del pipeline. Especialmente notable en las reflexiones. [24] 16
Figura 18: Comparación de la imagen final con una representación de los mesh distance fields. [25] 16
Figura 19: Representación de un objeto en distance fields. Al ser una textura volumétrica, la resolución es importante para representar un objeto, suponiendo un impacto en memoria. [25] 16
Figura 20: : Comparativa entre mesh distance fields y global distance fields. [25].......... 17

Figura 21: A la izquierda, la visualización de un modelo y las tarjetas que representan dónde se ha realizado dicha captura. A la derecha, el atlas donde se almacenan todas las capturas. [24] 17
Figura 22: Pipeline de Ray tracing por hardware. [26]........ 18
Figura 23: A la izquierda el modelo de Nanite en su nivel de detalle original. A la derecha el modelo que Lumen toma de Nanite para el hardware ray tracing. [27] 19
Figura 24: A la derecha, reflejos utilizando sólo el surface cache. A la izquierda, reflejos utilizando el hit lighting. Los esculares salen a la luz gracias a este sistema. [24]........ 19
Figura 25: Se ven unos claros beneficios entre no emplear ray-tracing en la lejania a poder hacerlo sobre un modelo simplificado. [26] 20
Figura 26: Quake II (1997) Modificado por NVIDIA para demo de path tracing a tiempo real. . 21
Figura 27: Pipeline de REYES. [30]........ 23
Figura 28: Proceso de mapeo de normales. Se requiere de un modelo de alta densidad de polígonos y su versión reducida. Resultado a la derecha. Imagen de Paolo Cignoni. 24
Figura 29: Imagen de Chetvorno. Vectores de normales en una superficie curva. 24
Figura 30: Comparativa entre un mapa de relieve y un mapa de desplazamiento. Imagen de la escuela Photigy. 25
Figura 31: Explicación de la jerarquía mediante DAG. Cuando el clúster padre da demasiado error (tiene en cuenta la distancia y ángulo) se dibuja el nodo hijo. Esta operación es local y se realiza en paralelo sin tener que recorrer toda la estructura. [34] 27
Figura 32: Funcionamiento de la construcción de la jerarquía de clústeres y la estructura de grafo acíclico dirigido. [34]........ 27
Figura 33: Señalado en rojo, el área que delimita un triángulo. En este caso, dos píxeles. El rasterizador va a testear entre esos dos cuáles necesita pintar, siendo el de la izquierda el que pinta. [34]........ 28
Figura 34: Este triángulo tiene una longitud de 16 píxeles, y va a iterar el test sobre los 128 que delimita. Esto es menor eficiente que el caso anterior, el rasterizador por software deja de admitir triángulos hasta los 31 píxeles de longitud de alto o ancho. [34] 29
Figura 35: Árbol de LODs. [34] 29
Figura 36: La luz que ilumina desde la derecha de este sujeto resalta los detalles de la piel y las arrugas. Si la luz estuviera de frente la piel se vería lisa. [35]........ 36
Figura 37: Buen ejemplo de una comparativa entre un mapa de normales y una topología extruida con relieve real. Las sombras son distintas y si se fuese a poner la cámara a ras del suelo una piedra no ocluiría otra a no ser que la forma esté en la escena de verdad. [36]....... 36
Figura 38: Tráiler de videojuego Elden Ring. Si se unen los conceptos anteriores y se mantiene un ángulo adecuado entre la luz y las caras del objeto. Se resaltan los detalles de las formas de la escena........ 37
Figura 39: Caso en el que muchos objetos están apilados encima de otros. En este caso se está tirando de objetos de librería y se han apilado varios trozos de tierra uno encima de otro. A la izquierda, el modo de visualización de Nanite donde se diferencian. [27] 39
Figura 40: Abajo a la derecha, el modelo original con millones de polígonos. Arriba y a la izquierda, la versión de pocos polígonos para Unreal Engine 4........ 43
Figura 41: Pizarra de referencias de elementos arquitectónicos........ 44
Figura 42: Pizarra de referencias de iluminación........ 44
Figura 43: Lluvia de ideas en la creación del concepto. 45

Figura 44: Concepto final. 46
Figura 45: Propuesta de diversos planos 46
Figura 46: Demostración de la profundidad de campo en la implementación de cámara anamórfica de Unreal Engine 5. [42]... 47
Figura 47: A la izquierda, artefacto de la dinastía Ming (1368-1644) expuesto en el museo de arte de Cleveland, creative commons. A la derecha, la reconstrucción 3D de esa superficie ornamentada.. 48
Figura 48: Pequeña función mediante nodos en Substance Designer para extraer detalle de grano fino del mapa de normales y fusionarlo en el desplazamiento. 49
Figura 49: Generador de materiales. Lee las propiedades del relieve y aplica diferentes materiales y propiedades a las zonas cóncavas, convexas, más ocluidas, más expuestas......... 50
Figura 50: Sistema de partículas en TyFlow para la generación de rocas que chocan precipitándose en el escenario. 51
Figura 51: Concepto original 52
Figura 52: Revisión de la composición. 53
Figura 53: Modo de visualización Overdraw. Los valores claros indican superposición de geometría. 54
Figura 54: Modo de visualización Clusters. Se recuerda que 1 clúster = 128 triángulos 55
Figura 55: Modo de visualización Triangles 55
Figura 56: Imagen final. 57
Figura 57: Se observan los círulos en el borde del modelo que no corresponden con nada. 60
Figura 58: Escena con sombras por ray tracing... 60
Figura 59: La misma escena empleando el comando de Nanite.. 61
Figura 60: Comparativa aplicando diferentes valores a la escena de Lumen, a la izquierda se aprecian los ornamentos del altar en la sombra.. 62
Figura 61: Estadísticas utilizando software ray tracing. 67
Figura 62: Estadísticas utilizando hardware ray tracing.. 67
Figura 63: Estadísticas utilizando hardware ray tracing y ray traced shadows 68
Figura 64: Estadísticas utilizando hardware ray tracing y ray traced shadows cuando la memoria virtual se satura. 68

I. INTRODUCCIÓN

1.1. MOTIVACIÓN DEL PROYECTO

Tras cursar el grado de videojuegos y trabajar 3 años en la industria de los efectos especiales y la publicidad, he podido profundizar y experimentar el render offline y el render a tiempo real. He podido ver lo diferentes que son ambas tecnologías y las virtudes y carencias que supone la utilización de cada una de ellas. También, he podido observar que hay una tendencia por intentar encontrar un punto medio. Las tecnologías que se sacan en el mundo de los videojuegos no son muy innovadoras en resultados si lo comparas con lo que se tiene ya en el render offline, es más, llega 20 años después. Sin embargo, tienen tiempos de render un millón de veces más rápidos. Cada vez que se abre una puerta se desbloquean muchísimas posibilidades.

Estos elementos propios del mundo del cine son cada vez más necesarias en el mundo de los videojuegos, donde se espera y se desea una mayor calidad cinematográfica, demandada tanto por el público como por los desarrolladores y productores.

Por contrapartida, en las producciones con render offline se busca conseguir la posibilidad de ver en tiempo real los resultados del trabajo en la pantalla, obteniendo de esta manera, un *feedback* instantáneo. Pero montar una escena para tiempo real lleva más trabajo, es menos intuitivo y el resultado es peor.

Figura 1: Set de rodaje de producción virtual en Mandalorian. Se emplean pantallas de LED en un entorno dentro de Unreal para conseguir la integración de iluminación y el fondo en pantalla.

Con la tendencia de implementar tecnologías de *ray-tracing* en los últimos años, se abren nuevos caminos que conducen a los estudios a invertir en tecnología a tiempo real para producciones audiovisuales. Así toma fuerza la denominada producción virtual [1].

Por si fuera poco, Unreal Engine saca las tecnologías de **Nanite**, que permiten millones de polígonos en la escena sin pérdida de rendimiento y gestionado automáticamente por el motor. Complementariamente, **Lumen**: una luz global dinámica que permite un comportamiento físicamente intuitivo de la luz, con rebotes y reflexiones, que se pueden actualizar en tiempo de ejecución. Estas implementaciones, aun estando en desarrollo, suponen la desaparición de una gran parte de la carga y preparación previa que ha sido siempre necesario en las escenas a tiempo real. No solo eso, sino que muchos efectos que antes eran impensables ahora son posibles.

Se considera interesante investigar estas tecnologías y conocerlas en su contexto. De esta manera sacar el mayor partido posible a los puntos fuertes que tienen e intentar mantenerse a la vanguardia de estas novedosas técnicas. Los primeros beneficiados de esto son las producciones audiovisuales, cuya tendencia en la forma de trabajar con la aplicación de estos avances va encaminada a como se trabaja en la realización de largometrajes, y al ser una tecnología joven y en constante cambio, es más apropiado en una producción corta que en una producción larga como un videojuego. A pesar de todo, ya hay juegos en desarrollo que se van a beneficiar de esto [2]. Con ello, la necesidad de estandarizar una nueva dinámica va en aumento.

1.2. PLANTEAMIENTO Y OBJETIVOS DEL PROYECTO

Los objetivos de este proyecto son investigar las tecnologías de Nanite y Lumen en profundidad, entender cómo funcionan y los casos en los que desempeña bien y en los que es mejor no utilizarla.

Adicionalmente, es conveniente analizar estos avances en su contexto histórico para así poder entender cuáles son los pasos que está dando la industria y hacia dónde se encamina.

Finalmente, realizar una pequeña escena para poner a prueba su funcionamiento en un simulacro de producción virtual o una cinemática. De esta manera, analizar el flujo de trabajo y juzgar si es posible trabajar de manera fluida para este tipo de producciones con la tecnología actual.

1.3. ORGANIZACIÓN DEL DOCUMENTO

El proyecto está dividido en cuatro apartados que abarcan las distintas etapas por las que se ha tenido que pasar para lograr el desarrollo de este.

- En primer lugar, en el apartado II ESTADO DEL ARTE, se analiza el recorrido de estas técnicas a lo largo de la historia y cómo funcionan actualmente en el contexto de render en tiempo real.
- Seguidamente, en el apartado III HERRAMIENTAS Y ENTORNO DE DESARROLLO, se hace un desglose del software que se va a utilizar para la producción.
- En el apartado IV DESARROLLO, se describen los pasos realizados para el desarrollo del experimento y se analizan los resultados de render.
- Para concluir, en el apartado V CONCLUSIONES, se reflexiona sobre el proceso y resultados obtenidos, se enumeran los problemas encontrados y se describe el trabajo futuro.

II. ESTADO DEL ARTE

A continuación, se van a definir las técnicas que se quieren aplicar, así como un recorrido de sus implementaciones pasadas hasta la actualidad. Adicionalmente, se van a analizar motores de render y el hardware necesario para ponerlo en práctica.

2.1. Iluminación global

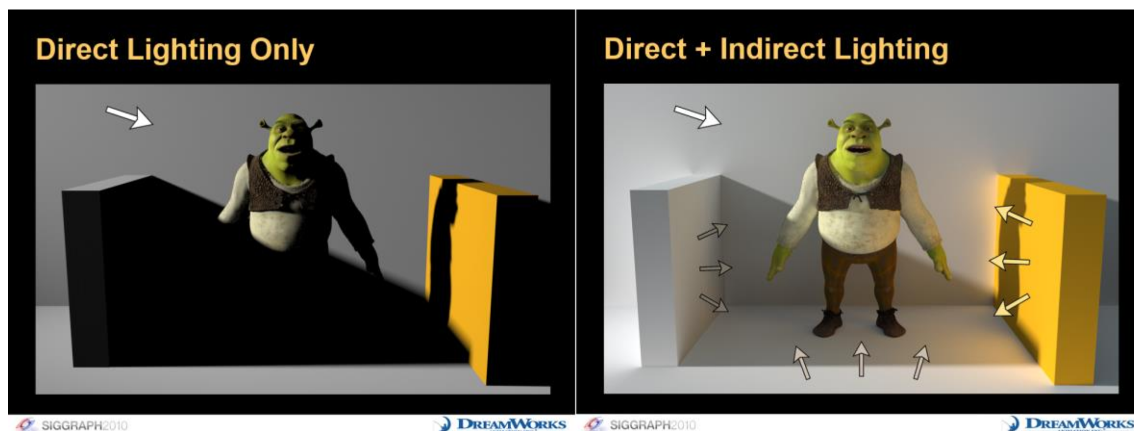


Figura 2: Luz directa y luz indirecta provenientes de una fuente de luz. [3]

La **iluminación global** (o GI, por *global illumination*) es el conjunto de algoritmos que simulan cómo la luz interactúa entre los objetos de una escena. Estos algoritmos se basan y simplifican la Teoría de Transferencia Radiativa [4]. Tienen en cuenta no sólo la luz directa de una fuente de luz, también la que los objetos proyectan y reciben en consecuencia por fenómenos como la reflexión o refracción.

En la Figura 2 se observa que con una sola fuente de luz se consigue una mayor riqueza gracias a esta técnica. A medida que se complica la escena resulta imposible replicar este resultado con luces directas. Se pueden apreciar en la imagen de la derecha los rebotes de la luz entre los objetos: nótese los tintes amarillos proyectados sobre la pared gris.

En el contexto de render a tiempo real se refiere inicialmente como luz global a la manera de conseguir estos rebotes de luz, llamado inter-reflexiones difusas, pero una luz global completa supondría lograr también otros efectos como la oclusión ambiental, subsurface scattering y cáusticas entre otros.

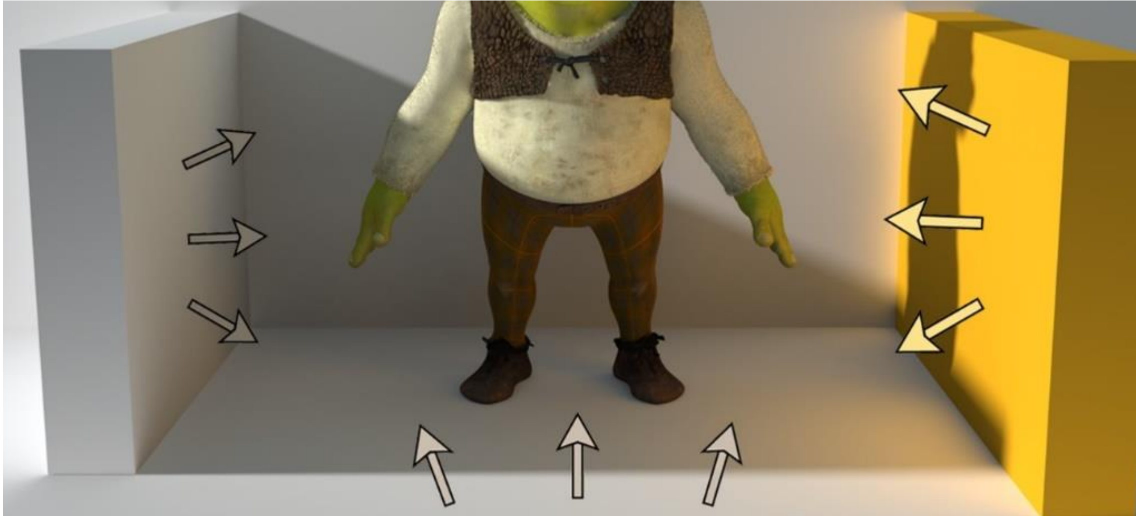


Figura 3: Reflexiones difusas, sangrado de la luz, oclusión ambiental, y otros efectos de GI. [3]

Como resultado de esta transferencia de luz en la escena, se consigue una mejor integración de los objetos en el entorno, consiguiendo una mejor unidad espacial.

A lo largo de la historia se han utilizado diversos métodos para conseguir GI en tiempo real. Que varían en función de las limitaciones del proyecto: si se necesita iluminación estática o dinámica, limitaciones de memoria, de rendimiento, fidelidad, etc. A continuación, se enumeran algunas de las técnicas de GI en función de si son estáticas o dinámicas [5].

Iluminación estática:

- Rellenar con un color ambiental
- Precalculada, almacenada en la geometría (por ejemplo, color de los vertices)
- Precalculada, almacenada en *lightmaps* [6]
- Precalculada, almacenada en *cube map* o *environment map* [7] [8]
- Precalculada, almacenada en volumenes [9]

Pseudo-Dinámica:

- Geomerics Enlighten [10]

Iluminación dinámica:

- Luces puestas a mano que simulen la iluminación indirecta
- *Cube Map Relighting* [11]
- Luces puntuales virtuales por medio de mapas de sombras reflectivos [12]
- Volúmenes de propagación de luz [13]
- En espacio de imagen [14]
- Basada en *Voxel*
- Sparse Voxel Octree Global Illumination (SVOGI) [15]

En conclusión, se encuentra una gran variedad de métodos para conseguir algo que es ya una necesidad a la hora de iluminar, que facilitan al artista la dirección de una escena y que proporciona una mejor lectura del espacio.

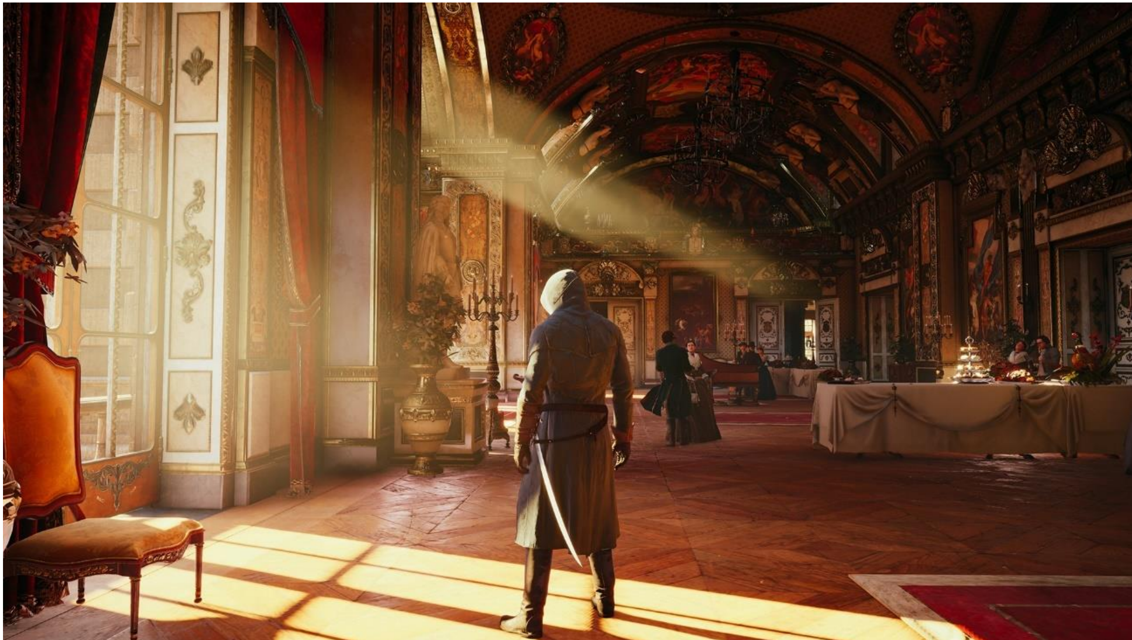


Figura 4: Una escena del videojuego Assassin's Creed Unity, donde se aprecia una iluminación convincente proveniente de únicamente una luz direccional y el cielo

Aún con todo, la iluminación global sigue siendo un problema difícil de resolver en tiempo-real. En render offline se consigue con el debido coste computacional, pero en tiempo real por lo general es muy caro de resolver con los mismos medios, ya debe de funcionar un millón de veces más rápido. Cada vez aumenta más necesidad de una calidad visual cinematográfica y junto a ello surge la producción virtual, que mezcla cine y tiempo real.

Por eso mismo, es importante un desarrollo que acerque más estos dos medios entre ellos. Se prefiere robustez antes que fidelidad. Interfaces simples antes que complejas. Poder iterar rápidamente en lugar de tener que realizar cómputos prolongados [5].

2.2. Iluminación global con ray-tracing

"It's really interesting to look back at the late 80's. That was when ray tracing and artificial intelligence research really took off. At the time most of us believed that ray tracing and neural nets – which is the technology that underlies deep learning- would be the future. But the problem was that we needed about a factor of a million more processing power than the CPUs of the day in order to make that technology useful for consumers. It took several decades to finally get to the point where we have GPUs that close

that gap and let us make these ideas from the 80's a reality." - Morgan McGuire [16]

El ***ray tracing*** es una tecnología que se ha estado desarrollando desde hace décadas y, desde entonces, no ha hecho más que optimizarse. Como consecuencia, se introduce su uso en el mundo del cine y el render *offline*. Serán 20 años después cuando esta tecnología empieza a disfrutarse en el render a tiempo-real.

El concepto de *ray tracing* no es algo estático, si no que ha ido evolucionando en el tiempo:

La historia de este concepto comienza en el siglo 16 con Alberto Durero y su *Treatise on Measurement*. En la ilustración se observa un método de representar un objeto 3D en una superficie 2D usando cuerdas desde un punto.

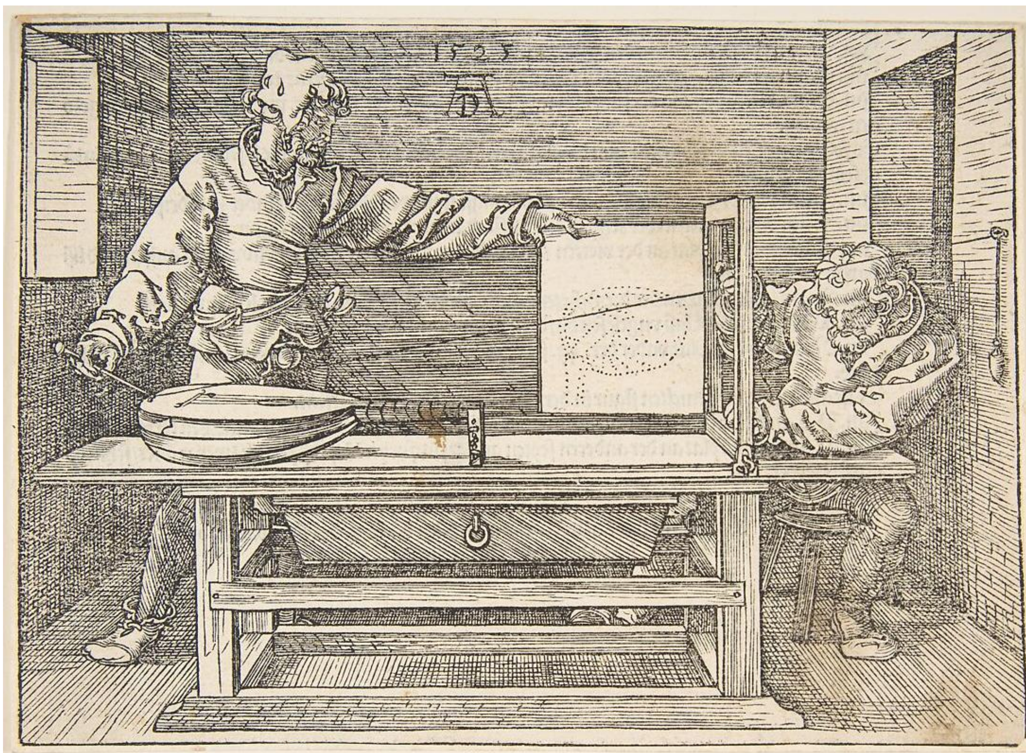
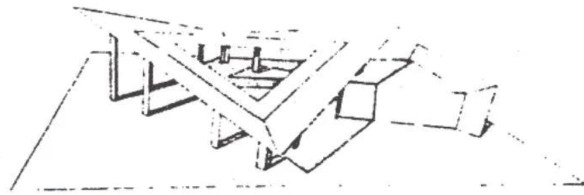


Figura 5: Ilustración en Treatise on Measurement, donde se aprecia un hombre pintando un laúd.

En 1969, Arthur Appel de IBM aplicó este concepto a los gráficos por ordenador con el nombre de ***ray casting***. Consiste en lanzar un rayo por cada píxel del puerto de vista y encontrar el objeto más cercano que bloquea la trayectoria. Luego lanza otro rayo hacia el punto de la luz para saber si está en la sombra o no.



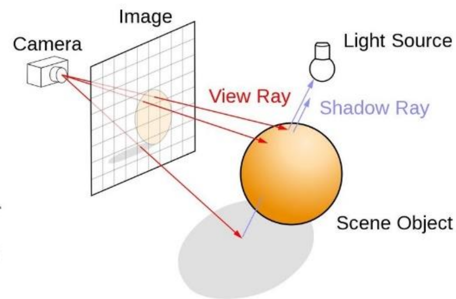


Figura 6: A la izquierda, el resultado del método de ray casting de Arthur Appel. A la derecha, un gráfico de su funcionamiento.

A partir de ahí, ***Turner Whitted*** demostró que esta idea puede aplicarse también para capturar reflexiones, sombras y refracciones.

Figura 7: Render de Turner Whitted de Ray Tracing de unas esferas. Se aprecian en el render la capacidad de translucencia en la bola del centro y la reflexión en la bola de la derecha. [17]

En 1984, ***Robert Cook*** trajo el **DRT (*Distribution Ray Tracing o Ray Tracing de distribución*)**, llamado así por realizar el muestreo de rayos en una distribución aleatoria. Se pueden lanzar varios rayos aleatorios en su emisión inicial y por cada rebote. De esta manera se consiguen varias técnicas cinematográficas: ***Motion blur, Depth of field***, penumbras, translucencia, reflexiones borrosas, etc.

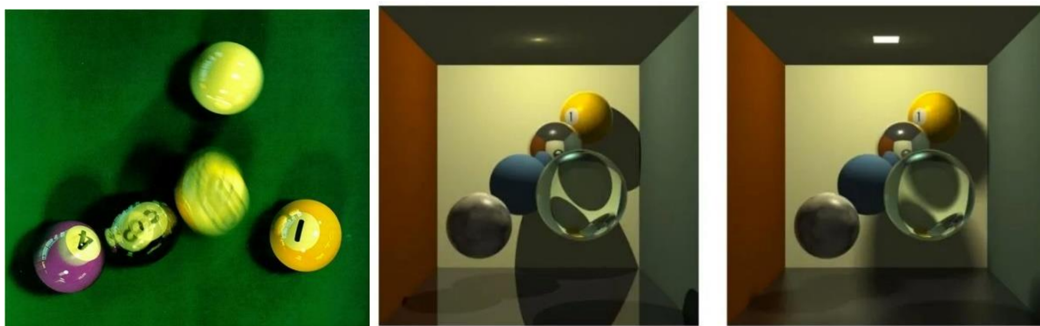


Figura 8: A la derecha, el tamaño de una luz afecta al borde de las sombras. A la izquierda, efectos de motion blur. [18]

Dos años más tarde, Jim Kajiya publica “***The Rendering Equation***” [19], donde se inspira en la teoría de transporte radiativo [20] y lo aplicó a la luz para describir como ésta se esparce por la escena virtual. Introduciendo con ello el algoritmo de ***path tracing***.

2.2.1. La aparición del Path tracing

Path tracing utiliza métodos de recursividad y **Montecarlo** para resolver la ecuación del renderizado, al igual que el Ray Tracing de distribución. Este algoritmo, en su forma más simple, consiste en trazar varios rayos desde el punto de vista al píxel (*eye rays*) y de ahí elegir la dirección de los rebotes de manera estocástica en función de las propiedades del material con el que choca: si es especular o mate, refleja o refracta. Además, se calculará el aporte de luz directa en cada rebote si es necesario. Se repite este proceso hasta que el rayo no choca con nada, llega al máximo de rebotes o se termina por probabilidad usando Ruleta Rusa. [21] [22]

2.2.2. Ray Tracing y Path Tracing en la industria

Las técnicas de Ray Tracin revolucionaron el mundo de los gráficos por ordenador. Empezó en el cine, con Bichos: Una aventura en miniatura en 1998. En el que se utilizaba el pipeline tradicional REYES pero se implementaba ray-tracing para solucionar ciertos problemas como la refracción. De esta manera, se pueden sacar similitudes con la situación de los gráficos a tiempo real 20 años después. Por ejemplo, Nanite realizando la función de REYES y Lumen como solución de Ray Tracing a tiempo real.

A la vez que se estudiaban diversas metodologías híbridas con ray tracing, se investiga lograr un flujo de trabajo de path tracing puro. Así sale el corto Bunny de Blue Sky Studios en 1998 y en 2006 Monster House, la primera película con esta tecnología.

Figura 9: De izquierda a derecha Bichos: Una aventura en miniatura, Shrek y Monster House.

Con el paso del tiempo se ve como la industria del cine opta por **path tracing como método favorito** para el render offline. Esto anima a especular si en un futuro se puede esperar lo mismo para el render a tiempo real.

2.3. Iluminación Global en Unreal Engine

Para entender el contexto de las ventajas y el avance que supone el desarrollo de las tecnologías basadas en Ray-Tracing, es necesario entender cómo se solía implementar la iluminación global hasta ahora. Se va a hablar primero de la iluminación global estática y posteriormente de dinámica, donde se incluye Lumen.

Previamente a la incorporación de ray tracing en el flujo de trabajo. La iluminación dinámica solo estaba disponible como **luz directa, sin rebotes de luz (Figura 2)**. La GI, con rebotes de luz, que ofrecía este motor gráfico **era precalculado y almacenado en lightmaps** [3] (o mapas de luces), previamente enumerado en la sección 2.1.

2.3.1. Iluminación global precalculada y almacenada en *lightmaps*

Precalcular la iluminación en UE significa realizar un proceso de **mapeado de luces o *light baking***. Almacenar en caché información de la luz mediante el uso de mapas de texturas. Este método se usa para conseguir resultados de alta calidad y es ideal para proyectos en los que la luz no es tan necesario que sea dinámica. El proceso de mapeado de luces requiere de mucha planificación y es de las fases más complicadas a la hora de generar un escenario 3D.

En primer lugar, cada modelo necesita tener configurado un canal adicional en el espacio UV. En la Figura 10 se observan las UVs para las texturas y en la Figura 11 las UVs adicionales para el mapeado de luces.

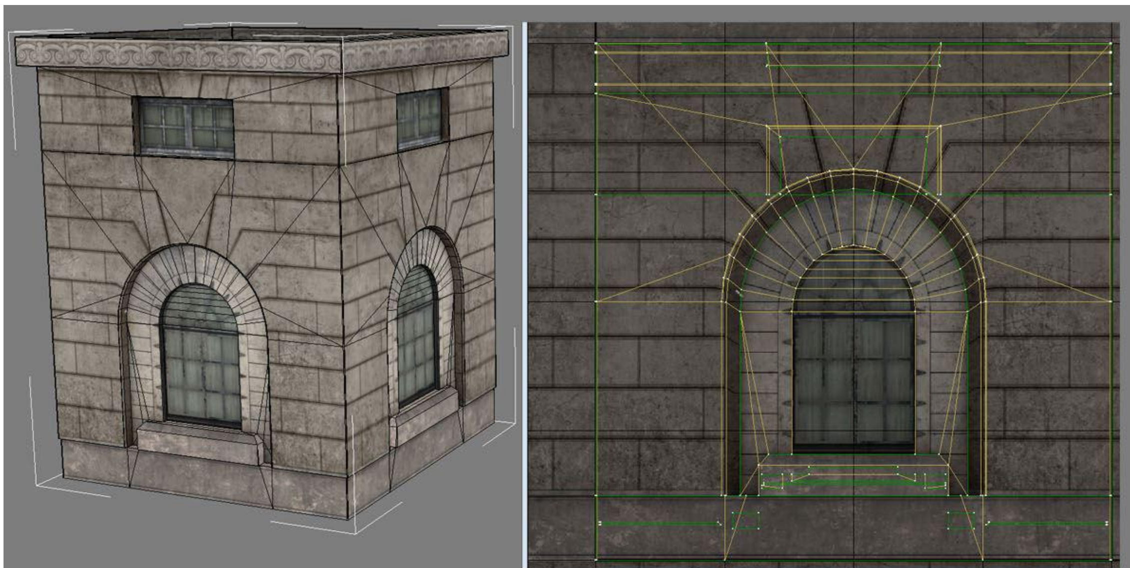


Figura 10: El espacio de UVs de esta pieza superpone las 4 caras verticales en el mismo espacio, de manera que se puede aplicar la misma imagen a las 4 caras y optimizar en memoria. [23]

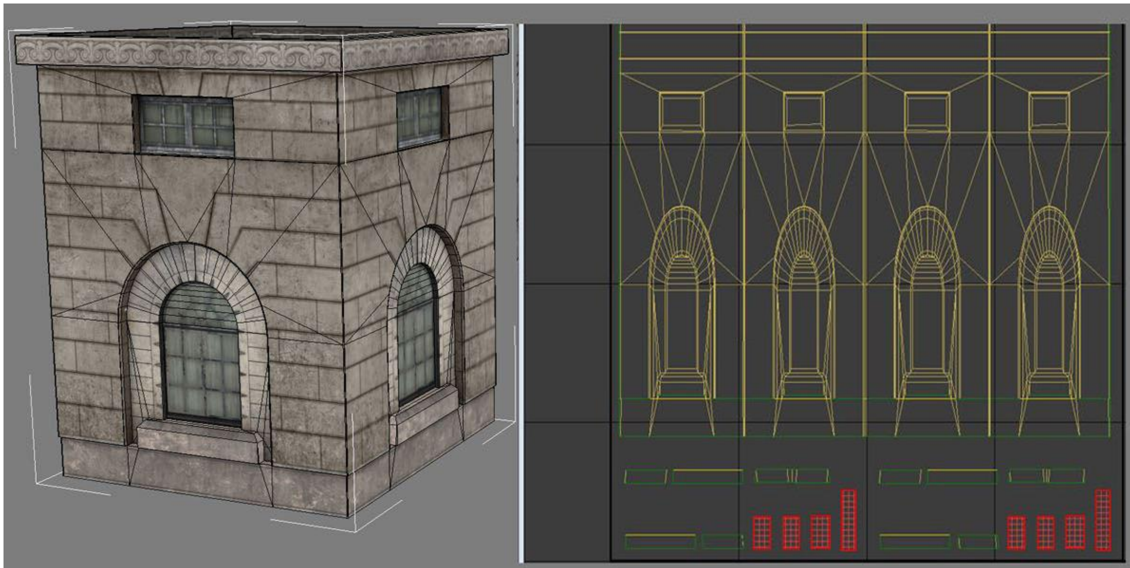


Figura 11: Ejemplo de UVs para mapeado de luces o lightmap. Se tienen que separar todas las caras del modelo, evitando cualquier tipo de solapamiento. [23]

Una vez terminado de ajustar los UVs de cada modelo, se compone la escena con sus objetos y luces y se renderiza el mapa de luces. El resultado es un mapa de con la información de iluminación como se aprecia en la Figura 12. En el caso de Unreal Engine, el motor junta toda la luz de la escena en un gran mapa al que se accederá para procesar cada objeto.

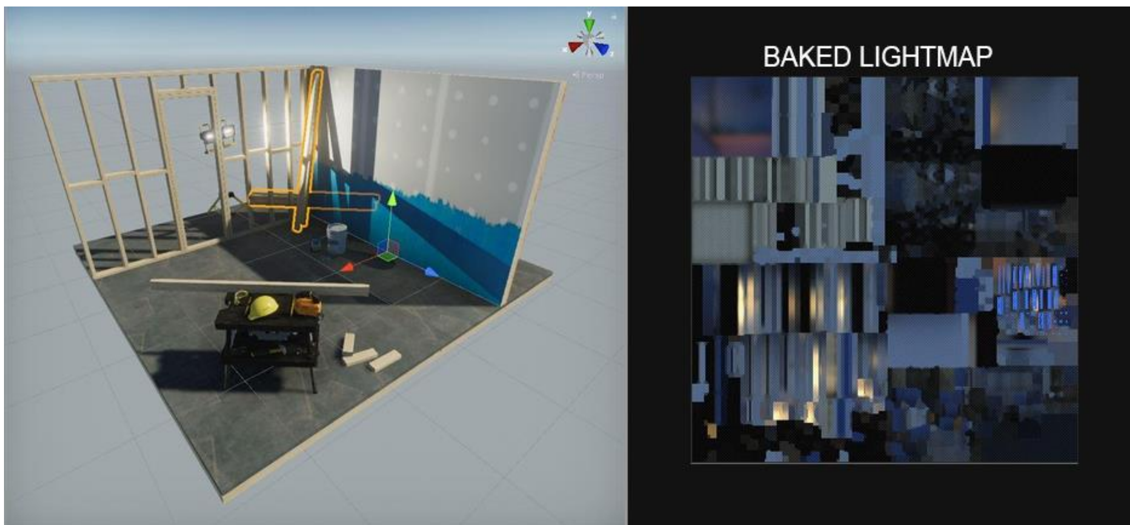


Figura 12: A la izquierda un escenario donde se aprecia la iluminación final, que fusiona los mapas de textura y de luz. A la derecha, el lightmap con toda la información de luz de este modelo.

Adicionalmente, es común realizar este proceso varias veces puesto que tienden a aparecer artefactos y errores inesperados. También es necesario ajustar la resolución local que tendrá cada objeto en el contexto global. Objetos cercanos tendrán más resolución en su mapa de luces que los objetos más lejanos de la cámara.

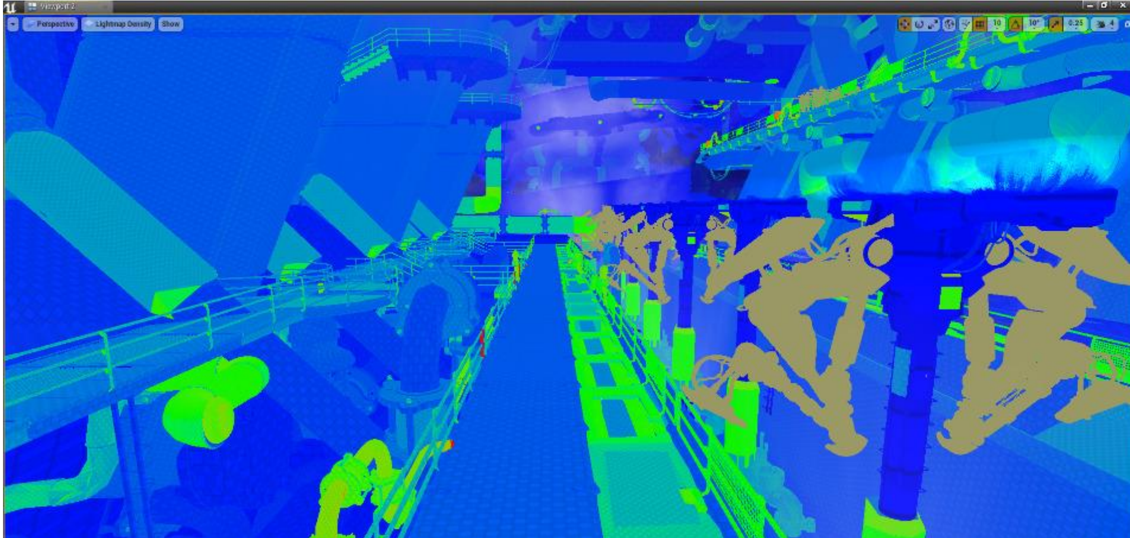


Figura 13: En una escala de Rojo>Amarillo>Verde>Azul>Azul oscuro, se puede visualizar la densidad de las UV del lightmap en cada objeto de la escena.

Esta solución de iluminación precalculada tiene sus ventajas y desventajas.

Ventajas:

- Poco costoso en render. El coste en rendimiento se resume en la memoria que se requiera para almacenar las texturas.
- Se controla la calidad mediante la resolución de los mapas y las UV relativas en cada objeto.
- Se puede usar en combinación con otras luces dinámicas.

Desventajas:

- Requiere de planificación previa.
- Costes de producción para adaptar los modelos a este sistema.
- Los cálculos para la generación de los mapas son caros cuando se requiere una calidad final de producción y pueden alargarse horas (por ello existen granjas de render para este proceso). Por lo que es necesario realizar previos de poca calidad para poder trabajar de manera ágil, sin ver el resultado final.
- Los artefactos y errores tienden a salir con frecuencia. Especialmente en objetos complejos.

2.3.2. Iluminación global dinámica.

Para conseguir Iluminación global a tiempo real Unreal Engine 5 dispone de varias soluciones:

- **Lumen** está diseñado para las consolas más potentes y ofrece un sistema de iluminación de reflexiones totalmente dinámico. Usa varios métodos de *ray tracing*. Es la técnica en la cual se centra este proyecto.
- **Screen Space Global Illumination** (SSGI) es un efecto de postproceso que genera un efecto de iluminación global basado en los objetos visibles en la cámara. Es un método barato que se puede usar de manera complementaria a otros métodos de iluminación precalculada o dinámica. Sin embargo, tiene muchas limitaciones y los objetos que no se ven en cámara no se tienen en cuenta para el cálculo, por lo que cuando la cámara se mueve aparecen y desaparecen objetos.
- **Ray Tracing Global Illumination** (RTGI). Actualmente está deprecado y puede que no se incluya en futuras versiones. Emplea el *hardware* para calcular la iluminación global, dispone de dos métodos:
    - **Por fuerza bruta:** Parecido a la manera en la que un render *offline* resuelve la iluminación, es lento.
    - **Final Gather** usa un algoritmo en dos pasos para distribuir puntos de sombreado a lo largo de la escena y se conectan en un historial. Menor calidad, pero mejor rendimiento.

2.3.3. Lumen

Lumen es una tecnología basada en varias técnicas de *ray tracing* que permite su uso en arquitecturas gráficas que no estén especializadas en *ray tracing*. Y sin necesidad de una GPU de la más alta gama. Aunque también dispone un pipeline específico para sacarle partido a este tipo de arquitecturas que sí que soporta *hardware ray tracing*.

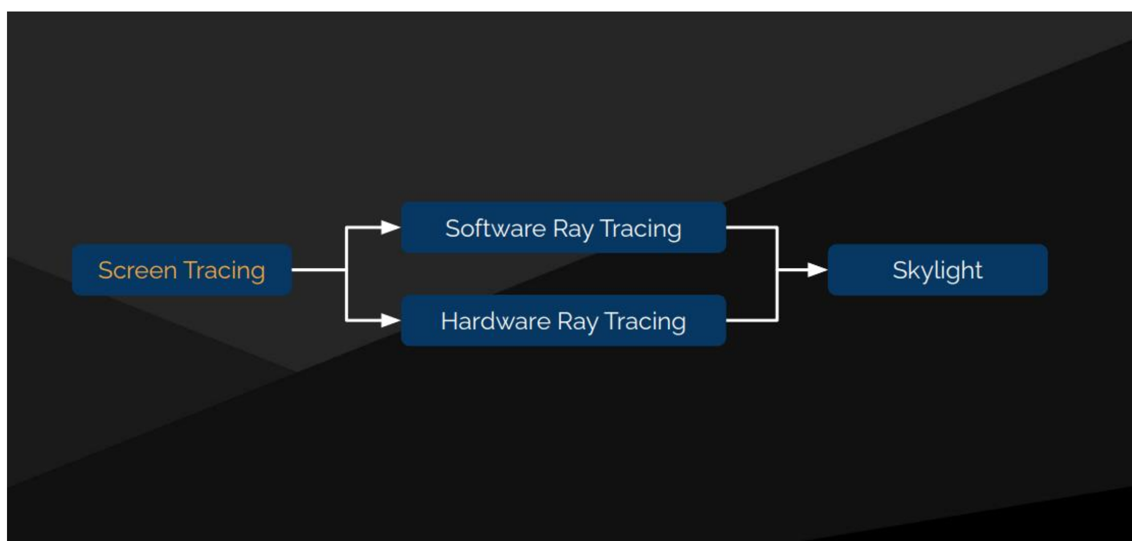


Figura 14: El pipeline de Lumen. Resuelve primero screen tracing, luego emplea o software o hardware raytracing, y si un rayo sigue sin chocar contra una geometría samplea el skylight (la luz del cielo).

Por defecto, Lumen emplea *ray-tracing* por *software*, para dar el mayor soporte y porque tiene menor coste de render que su versión de *hardware*. Es un método híbrido de varias

formas de *ray-tracing*, empleando cada una donde mejor se desempeña y haciendo uso de "signed distance fields".

Signed distance field es clave para entender cómo funciona el *ray tracing*, es una función que toma como input una posición y devuelve como output la distancia desde esa posición al punto más cercano de una figura determinada. En *ray tracing*, cuando trazas un rayo desde la cámara, esta técnica te permite saber qué rayos chocan con un objeto en la escena y cuáles no. A lo largo del rayo, se va reevaluando el SDF (a esto se le llama *ray marching*).

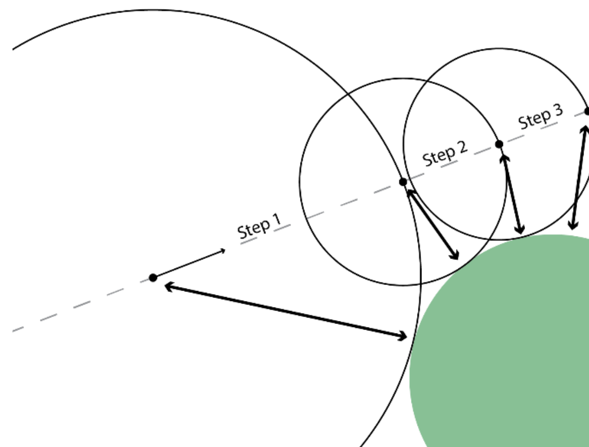


Figura 15: : Ejemplo de ray marching, en el paso 1 un signed distance field desde el primer punto en un rayo trazado, la operación devuelve una distancia al punto más cercano del círculo. Más adelante reevalúa el distance field en varios puntos.

El pipeline completo de este método de ray-tracing por software es el siguiente:

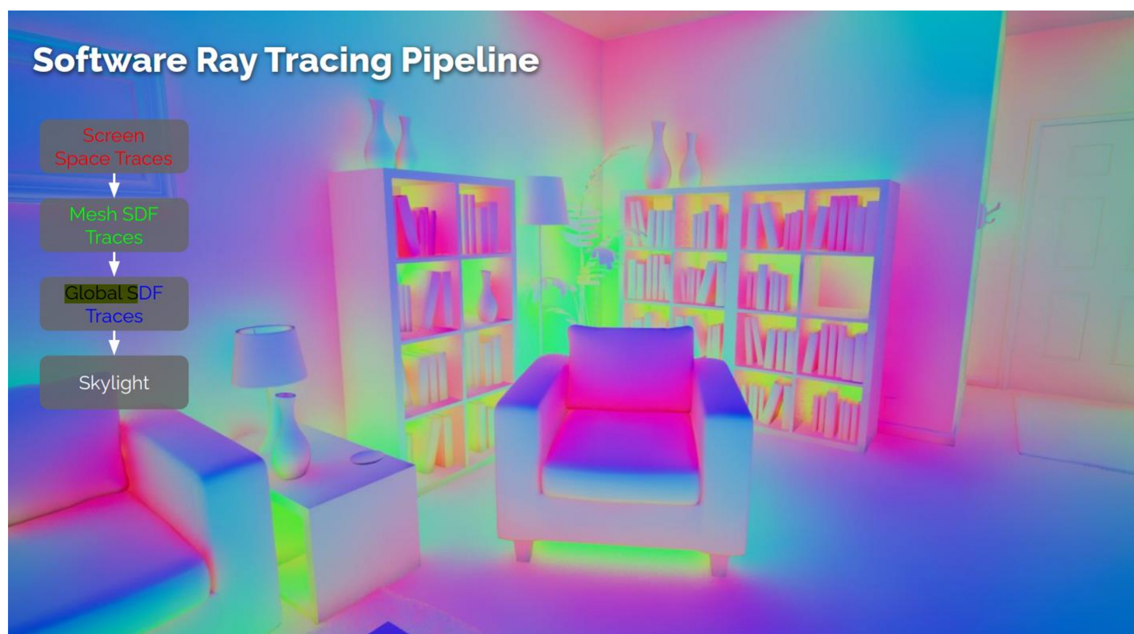


Figura 16: Las piezas que componen el pipeline de ray tracing por software.

Primero realiza ***screen tracing***, que realiza el proceso de *ray tracing* contra el *buffer* de profundidad. En otras palabras, **procesa lo visible en la pantalla**.

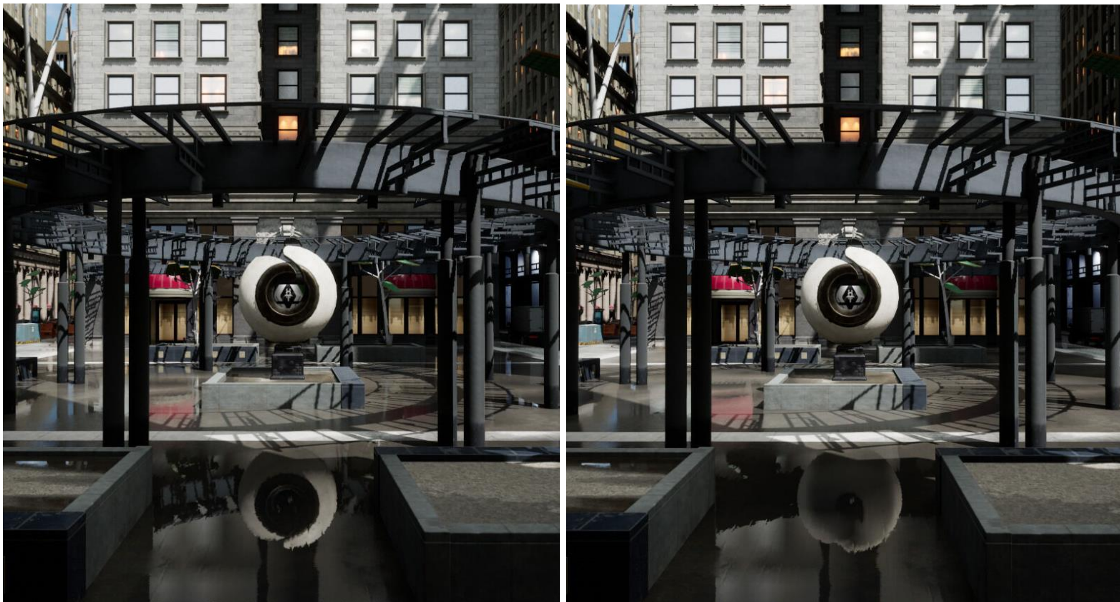


Figura 17: Comparativa de añadir o quitar screen tracing del pipeline. Especialmente notable en las reflexiones. [24]

Los rayos que no se resuelvan de esta manera se solucionan realizando el trace contra los ***mesh distance fields***.

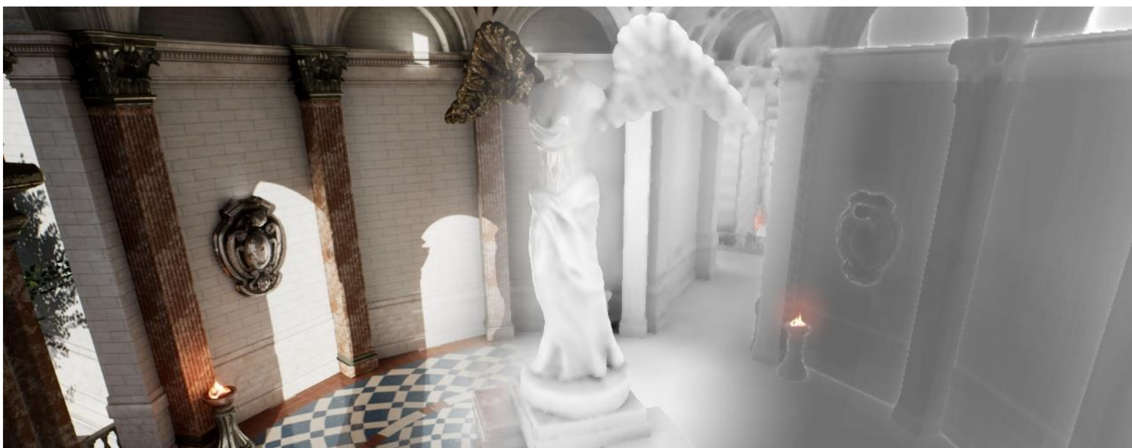


Figura 18: Comparación de la imagen final con una representación de los mesh distance fields. [25]

Los *mesh distance fields* son una manera de **representar un objeto mediante una textura volumétrica**. De esta manera no tiene apenas coste saber la posición más cercada a la superficie de un objeto, puesto que se almacena en textura.

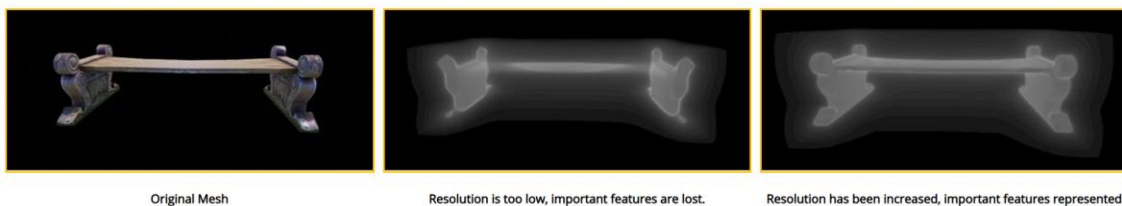


Figura 19: Representación de un objeto en distance fields. Al ser una textura volumétrica, la resolución es importante para representar un objeto, suponiendo un impacto en memoria. [25]

Por optimización, se utilizan los *mesh distance fields* contra los objetos cercanos. Contra los objetos más lejanos, se reemplazan por los ***global distance fields,*** que son una versión de menor coste que compacta los objetos de la escena.

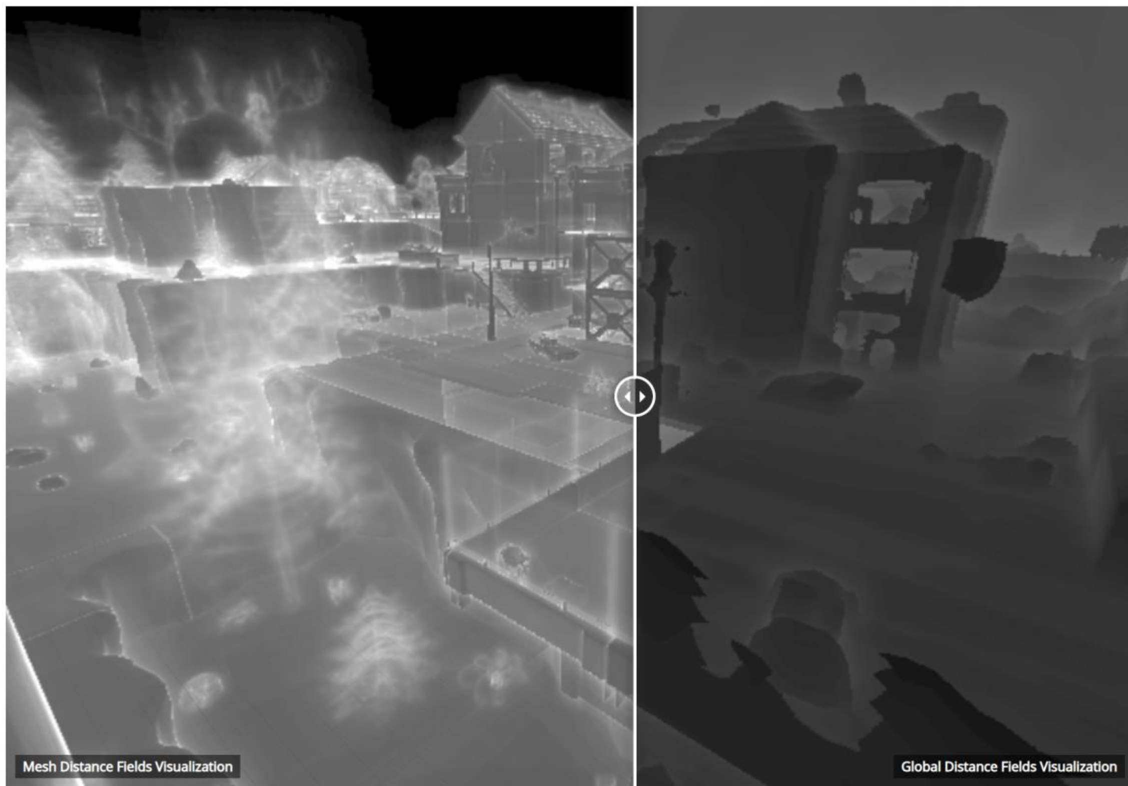


Figura 20: : Comparativa entre mesh distance fields y global distance fields. [25]

Sin embargo, lo que tiene este tipo de representación es que **no aporta información importante como las propiedades del material o** cuál es **la luz** que llega a ese punto (y rebota a nuestro ojo).

Para conseguir esa información, se crea el ***Surface cache***. El *Surface cache* consiste en capturar una geometría desde diferentes puntos de vista y almacenar esas capturas en un atlas.

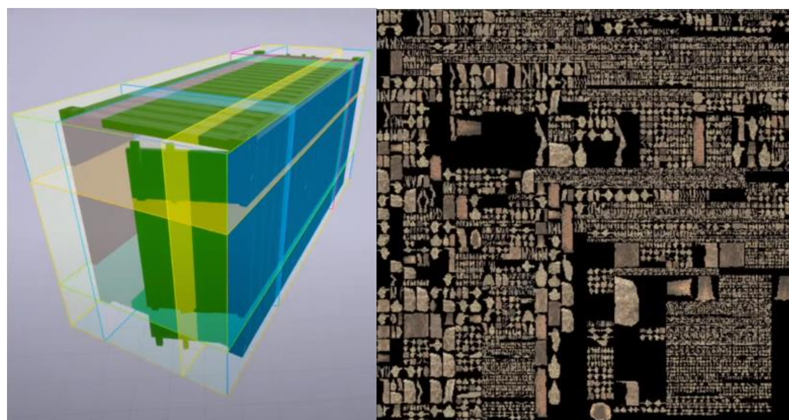


Figura 21: A la izquierda, la visualización de un modelo y las tarjetas que representan dónde se ha realizado dicha captura. A la derecha, el atlas donde se almacenan todas las capturas. [24]

Este proceso de captura se realiza de manera continua mientras la cámara se mueve por el escenario. Cuando la cámara se acerca a un objeto, se recaptura en un atlas de alta resolución. Cuando la cámara se aleja se captura en un atlas de baja resolución.

Este método tiene la desventaja que, si se tiene un objeto complejo con muchas caras internas, es posible que queden zonas sin ser representadas. Puede pasar con el interior de un edificio, que tiene varias habitaciones y ventanas. En estos casos, este método va a tener fallos. Para ellos, es necesario tener los **elementos complejos divididos en módulos**. De esta manera, el Surface cache procesará cada parte por separado sin errores.

Ventajas del *ray tracing* por software frente a *hardware ray tracing*:

- Son un método ideal para calcular la **GI y reflejos poco nítidos**.
- No requieren ningún tipo de hardware en específico.
- **Sistema versátil** que encaja con otros sistemas como el de colisión o de partículas.
- Es **lo que mejor escala** en escenas donde hay mucha geometría con **mucho solapamiento**, gracias al global *distance field* que colapsa todo.

Desventajas:

- Los *distance fields* no son adecuados para conseguir reflejos nítidos de espejo.
- No soporta geometría animada con deformaciones (personajes).

Para suplir las limitaciones de este método, Lumen dispone de una **solución por hardware**, que es una vía un poco más experimental que la anterior.

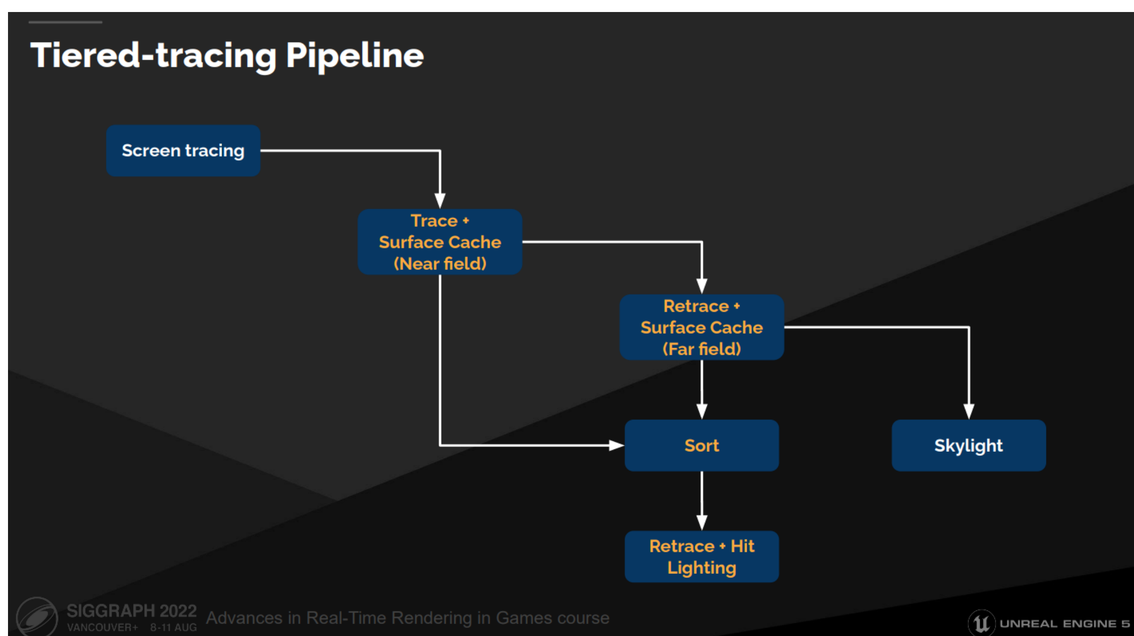


Figura 22: Pipeline de Ray tracing por hardware. [26]

El pipeline de este método dispone de dos vías distintas en función de lo que se necesite. Al igual que con el pipeline de software, primero lleva a cabo el screen tracing, luego traza contra los objetos cercanos y posteriormente los lejanos. **La mayor diferencia** es que el trazado por hardware no utiliza mesh distance fields si no que realiza un ray trace puro **contra los triángulos**

de la geometría. Esto permite soportar ciertas cualidades, como la **geometría animada** (personajes). Por otro lado, es el número de polígonos eleva los costes del trazado. Si se utiliza junto con Nanite, cargará la versión de la geometría con menor detalle posible (Figura 23).

Figura 23: A la izquierda el modelo de Nanite en su nivel de detalle original. A la derecha el modelo que Lumen toma de Nanite para el hardware ray tracing. [27]

Este pipeline **emplea la Surface cache** de manera complementaria para abaratar en costes.

Opcionalmente, para conseguir **reflejos de espejo de la mejor calidad**, se puede emplear el modelo de ray tracing **"hit-lighting"**, modificación del pipeline Sorted-Deferred tracing heredado de Unreal Engine 4 [28]. Este modelo tiene unos **costes mucho más altos**, por lo que sólo se utiliza en reflejos. Permite que en las reflexiones los objetos animados reciban luz directa y en general mejora la calidad de los reflejos. (Figura 24).

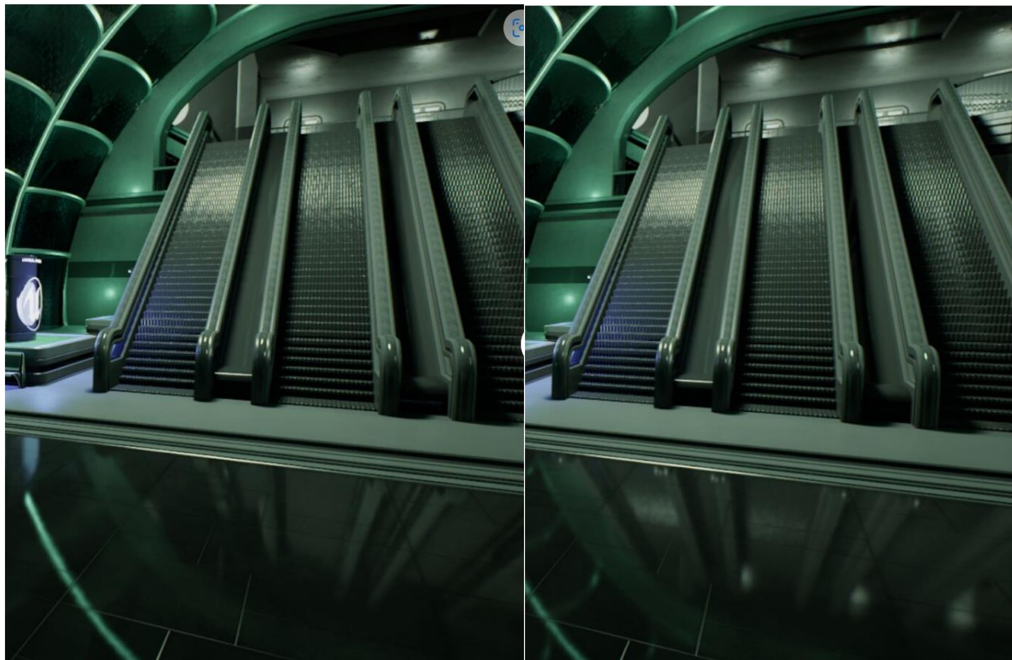


Figura 24: A la derecha, reflejos utilizando sólo el surface cache. A la izquierda, reflejos utilizando el hit lighting. Los esculares salen a la luz gracias a este sistema. [24]

Otra diferencia importante respecto al pipeline de software es la **representación far field** de la escena, que está para resolver las grandes cantidades de instancias de objetos en la

lejanía que aportan menos a la escena. Para ello, en lugar de utilizar global distance fields, emplea el sistema de partición de Unreal (World Partition System) y su representación de nivel de detalle jerárquico (HLOD), esto significa que **emplea un sustituto de la geometría a partir de cierta distancia para poder realizar el ray trace contra ello**.

Figura 25: Se ven unos claros beneficios entre no emplear ray-tracing en la lejania a poder hacerlo sobre un modelo simplificado. [26]

En conclusión, el pipeline de ray tracing por hardware consiste en: Screen tracing, después un tracing de los elementos cercanos (near field, con Nanite o no) con la Surface cache, luego contra los elementos lejanos (far field) con la Surface cache, los rayos que no rebotan después de eso se evalúan contra el skylight (cielo). Finalmente, de manera opcional, se puede hacer otro trace e invocar el hit-lighting para obtener mejores reflejos.

Ventajas del ray tracing por hardware:

- Fidelidad en reflejos de espejo
- Los personajes y geometría con deformaciones pueden aportar de la GI de manera significativa y aparecer con luz directa en las reflexiones.
- Consigue la mejor calidad posible.

Desventajas

- Es más costoso que el pipeline de software, en el mejor de los casos tiene un coste parecido, pero no menor. Sobre todo, cuando hay mucho solapamiento de geometría los tiempos de render suben descontroladamente.

2.4. Futuro de la iluminación a tiempo real

Si se analiza la evolución de las técnicas de iluminación y render en la industria del cine, al comparar con la del videojuego se puede observar que se suelen repetir los mismos patrones varios años después.

Si ahora el render a tiempo real se encuentra en la fase de pipelines híbridos de rasterización + *ray tracing*, se puede entonces especular la posibilidad de que en el futuro se adopte un pipeline de path tracing puro, como en la industria del cine en este momento. Es un campo que ya está en investigación y empresas como NVIDIA ya proporciona demos y herramientas para empezar a visualizar esta tecnología.

Figura 26: Quake II (1997) Modificado por NVIDIA para demo de path tracing a tiempo real.

2.5. Geometría virtualizada

A la hora de entender Nanite y la geometría virtualizada es pertinente hacer un recorrido de su contexto en la historia para entender qué avances supone respecto a lo que ya se dispone.

Nanite entra en juego en un momento en el que la industria disfruta de las ventajas de la fotogrametría, que pone a disposición del artista geometría escaneada con mallas de millones de polígonos. Así mismo, los softwares de generación de contenido 3D son capaces de producir también millones de polígonos. Adicionalmente, los juegos con mundos abiertos son tendencia requieren de miles o millones de instancias de geometría.

Por todo esto, es ideal tener la **capacidad de renderizar sólamente los polígonos que se necesiten** en la escena para reducir costes sin perder detalle. Significa que una malla de 2 millones de polígonos no necesita tanto detalle si ocupa 20x20 píxeles de la pantalla. No va a tener suficiente muestreo para poder ser representado con fidelidad, y el coste es muy alto. De esta manera, la clave está en renderizar como mucho un polígono por píxel. **Así surge Nanite en 2021, pero no es la primera tecnología con esta filosofía.**

En **1987 Cook et al. Introdujo la arquitectura REYES** en el contexto del render offline para conseguir resultados foto realísticos. [29]

2.5.1. Arquitectura REYES

An architecture is presented for fast high-quality rendering of complex images. All objects are reduced to common world-space geometric entities called micropolygons, and all of the shading and visibility calculations operate on these micropolygons. Each type of calculation is performed in a coordinate system that is natural for that type of calculation. Micropolygons are created and textured in the local coordinate sysem of the object, with the result that texture filtering is simplified and improved. Visibility is calculated in screen space using stochastic point sampling with a z buffer. There are no clipping or inverse perspective calculations. Geometric and texture locality are exploited to minimize paging and to support models that contain arbitrarily many primitives. - Robert L. Cook [29]

La **Arquitectura REYES** (“**Render Everything You Ever Saw**”) es un método para renderizar imágenes fotorrealistas cuando las capacidades de cómputo y memoria son limitados. Renderiza superficies utilizando una subdivisión que se adapta a las necesidades del momento, subdivide una forma en parches hasta que dichos parches abarcan aproximadamente un píxel o menos. Luego, estas formas se dividen en un grid de quads para ser rasterizados fácilmente.

Este proceso está dividido en cinco partes: Bound, Split U/V, Dice, Shade y Sample.

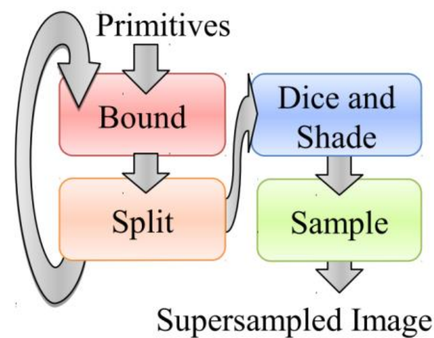


Figura 27: Pipeline de REYES. [30]

Bound y Split se ejecutan en bucle, *Bound* realiza un descarte de la geometría (bien porque el parche es muy pequeño o porque no se ve por la cámara) y Split divide el parche en dos y los devuelve a *Bound* a testear de nuevo. Finalmente, el parche se envía a Dice, que lo divide en micropolígonos. Éstos se procesan en *Shade*, que realiza el sombreado. *Sample* lleva a cabo el rasterizado final.

Este pipeline, que cumple el cometido para el que se ha desarrollado, si se fuese a aplicar al render a tiempo real supondría varios problemas. Por ejemplo, el bucle de *Bound* y *Split* es recursivo.

REYES asentó las bases del motor Renderman de Pixar, que tiene detrás un sinfín de películas. Con el tiempo, empezaron a salir pipelines híbridos que mezclaban REYES con *ray tracing* para la iluminación global o los reflejos. Años después se empieza a sustituir este pipeline por uno de *path tracing* por simplicidad. [31]

2.5.2. Mapas de Normales, Bump y Displacement

Mientras tanto, en el mundo de los gráficos a tiempo real se requiere una solución para meter detalles en los modelos sin emplear geometría. De manera tradicional, la solución ha sido resolverlo mediante texturas.

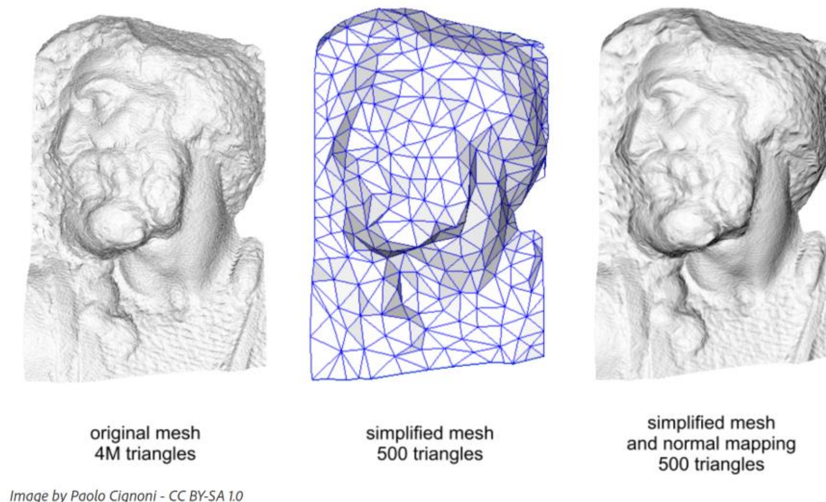


Figura 28: Proceso de mapeo de normales. Se requiere de un modelo de alta densidad de polígonos y su versión reducida. Resultado a la derecha. Imagen de Paolo Cignoni.

Los **mapas de normales (*normal maps*) y *bump maps*** tienen la capacidad de dar una **sensación de profundidad y relieve** en modelos de poca densidad alterando la dirección de las normales de un modelo.

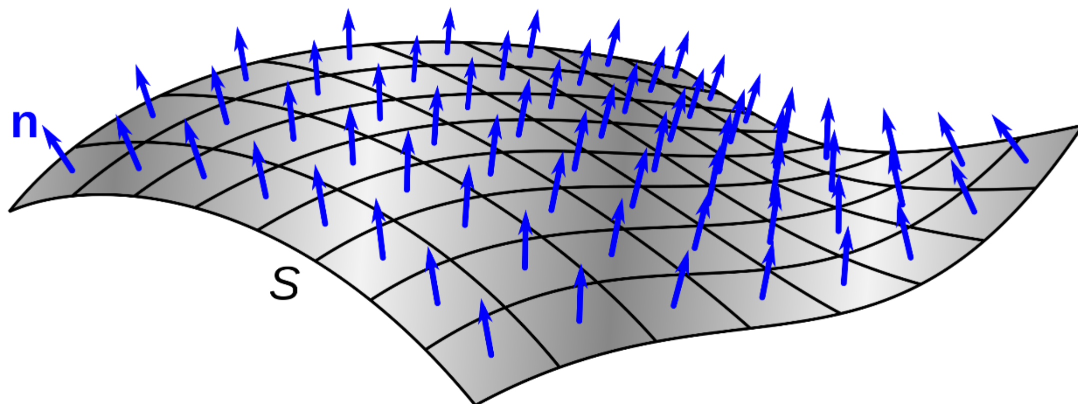


Figura 29: Imagen de Chetvorno. Vectores de normales en una superficie curva.

Es un método muy eficiente pero que tiene sus desventajas. Al igual que con los *lightmaps*, el proceso de mapeado es tedioso.

- Hay que prepara dos mallas distintas a la hora de generar un objeto, uno con pocos polígonos (*low poly*) y otro con el nivel de detalle deseado (*high poly*). El modelo *low poly* tiene que cumplir con ciertos criterios y requisitos para evitar distorsiones y artefactos.
- El proceso de mapeado suele llevar varias iteraciones hasta conseguir el definitivo.
- Un cambio en el modelo *high poly* requiere empezar este proceso de nuevo.
- Modificar las normales no modifica la forma real del modelo, por lo tanto, se pierden formas que afectan tanto a la silueta, perspectiva y render final.

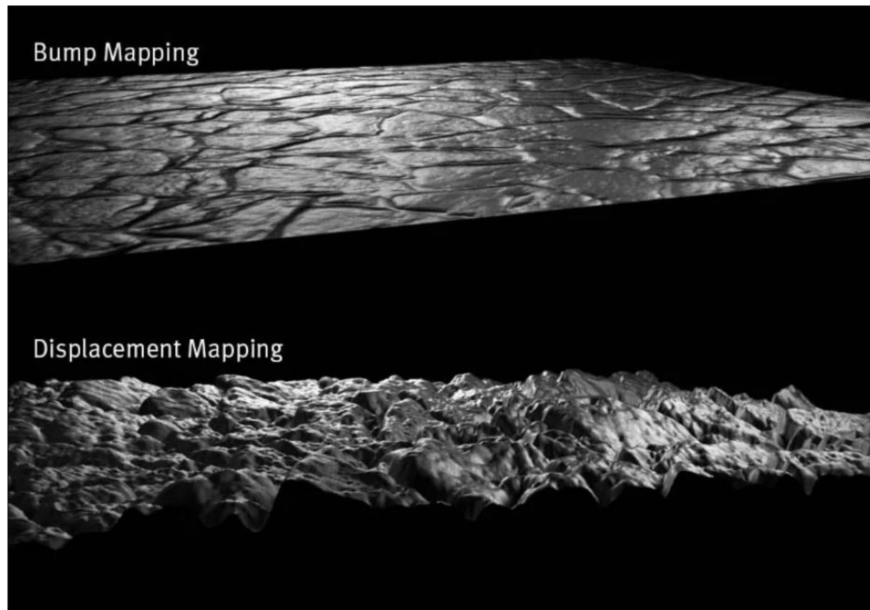


Figura 30: Comparativa entre un mapa de relieve y un mapa de desplazamiento. Imagen de la escuela Photigy.

Para suplir esta carencia están los ***displacement maps***. Que desplazan la geometría en el sentido de sus normales. Para ello es necesario subdividir la geometría, ya sea dinámicamente o dividirlo antes de la ejecución.

Las desventajas es que su aplicación es limitada: editas los vértices en un solo sentido, arriba o abajo. Dicho así, no puedes transformar una esfera en un anillo. Tiene repercusiones en memoria y al final también estás creando geometría.

Aun así, todas estas soluciones tienen sus casos particulares en los que suponen una ventaja frente a otras soluciones y no tienen pinta de que vayan a desaparecer (totalmente) en unos años.

2.5.3. Geometría Virtualizada: Nanite

Nanite se crea con la motivación de sobrepasar las limitaciones técnicas más restrictivas del momento:

- Número de polígonos
- Llamadas a CPU (*Draw Calls*) [32]
- Memoria

Que conllevan la pérdida de tiempo en optimización y la pérdida de calidad de los modelos para adaptarlo al presupuesto de la escena.

Con sistemas como REYES detrás, esta solución también se inspira en la idea de las texturas virtualizadas: Tener una textura (en este caso geometría) en la resolución que se necesite, e ir cargando más o menos resolución sobre la marcha en función de lo que se requiera. El problema está en que, al contrario que con las texturas, hacer lo mismo con la geometría no

es solo un problema de gestión de memoria. A continuación, se explica los elementos principales que caracterizan Nanite:

2.5.3.1. Pipeline controlado completamente por GPU

Para conseguir su cometido, Nanite **almacena en la GPU una versión completa de la escena**. A pesar de ello, no se carga la escena completa en cada frame, solamente lo que cambia se actualiza. De esta manera, todos los índices y vértices están almacenados en el mismo sitio. Adicionalmente, es la GPU la encargada de hacer el culling (desechado de la geometría que no se necesita, incluye frustum culling y occlusion culling) de instancias y la rasterización de los triángulos.

De esta manera se consigue **cargar toda la geometría en una llamada indirecta** (indirect draw call). Antes de Nanite, cada objeto en escena requería una draw call y si había muchos objetos en escena los costes se disparaban.

2.5.3.2. Cluster culling de triángulos

Para realizar este descarte de tantos millones de triángulos de manera eficiente, se **agrupan en clústeres (128 triángulos por clúster)**. Por cada clúster se crea bounding data (o bounding box), como una caja que delimita los confines que ocupa ese clúster. De esta manera se pueden realizar las operaciones de frustum culling (qué objetos están dentro del área delimitado por la cámara) y culling de oclusión (qué clústeres están ocluidos por otro objeto y se pueden descartar) basándose en esa información del espacio que ocupan los clústeres.

2.5.3.3. Visibilidad evaluada por pixel y desconectada de la evaluación de materiales

Para ello, se realizan los pases de evaluación de materiales en una fase separada de la rasterización. Se realiza una llamada por cada material (de nuevo, no por cada objeto) y el resultado se escribe en el G-Buffer [33].

2.5.3.4. Jerarquía de clústeres

Se sustituyen los tradicionales LOD (*Level of detail*, el artista crea varias versiones de un modelo con más o menos geometría en función de lo que se necesite) y en su lugar Nanite automatiza una jerarquía de clústeres. Se tiene un DAG (Grafo acíclico dirigido), puede imaginarse como un árbol donde el padre es la versión simplificada de los hijos. Por cada clúster se coje el del nivel necesario.

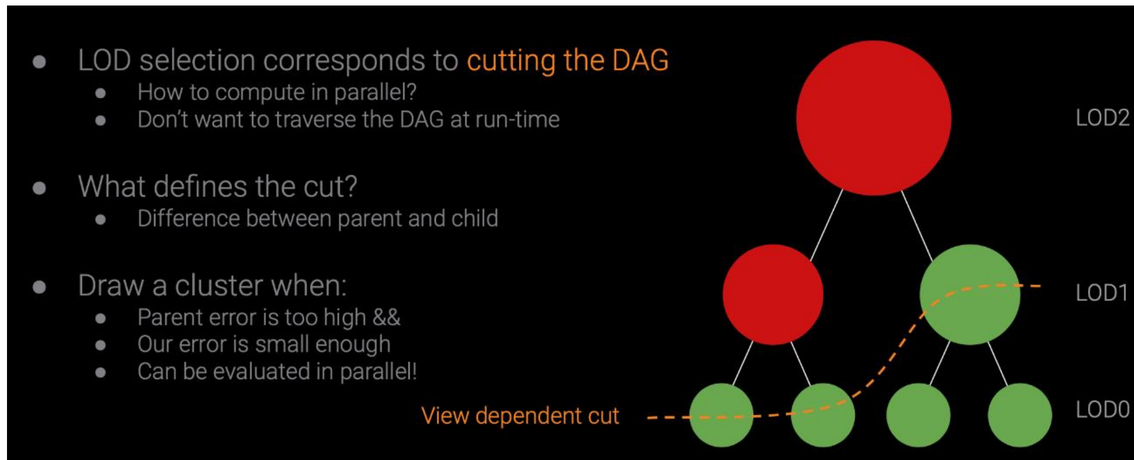


Figura 31: Explicación de la jerarquía mediante DAG. Cuando el clúster padre da demasiado error (tiene en cuenta la distancia y ángulo) se dibuja el nodo hijo. Esta operación es local y se realiza en paralelo sin tener que recorrer toda la estructura. [34]

De esta manera, en lugar de sustituir el modelo entero en función de si está más o menos lejos de la cámara, se sustituye el modelo por trozos. Así es como se consigue que los clústeres tengan un tamaño uniforme en la escena.

2.5.3.5. LODs automatizados

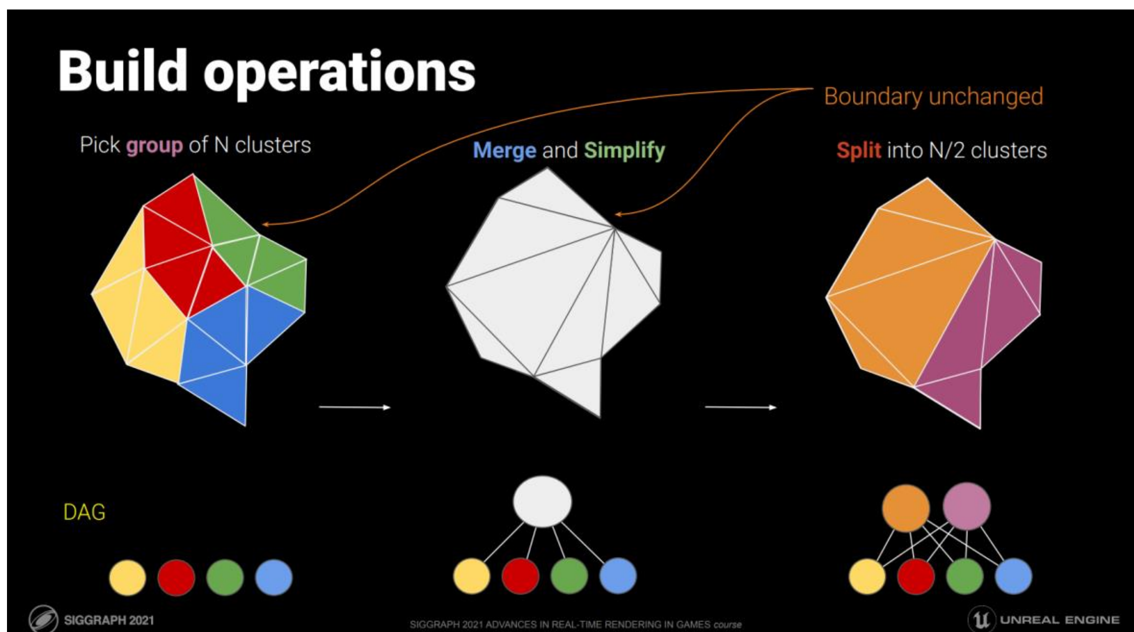


Figura 32: Funcionamiento de la construcción de la jerarquía de clústeres y la estructura de grafo acíclico dirigido. [34]

Como se observa en la figura 32, para construir esta estructura de clústeres se realizan una serie de operaciones. Se construyen primero los clústeres de los nodos hoja (128 triángulos = 1 clúster) y luego:

Mientras el número de clústeres sea mayor que 1:

- Agrupar N clústeres y conseguir un borde compartido limpio
- Fusionar los triángulos en una lista compartida
- Simplificar al 50% de los triángulos

Dividir la lista compartida de triángulos en N/2 clústeres (se siguen manteniendo 128 triángulos en cada uno).

Esta construcción previa es una solución eficiente y automatizada que ahorra horas al artista, ya que con el método de LODs tenía que producir varias versiones de un modelo. Esta práctica desaparece para los objetos compatibles con Nanite.

2.5.3.6. Rasterizador especializado

Por cómo funciona un rasterizador por hardware, centrado en paralelizar operaciones por píxel dentro de un triángulo, la eficiencia de éstos en un caso donde tienes triángulos del tamaño de un píxel es mala. Simplemente no está diseñado para eso. Para ello se diseña un rasterizador por software especializado, cuyo funcionamiento es parecido a un shader de vértices.

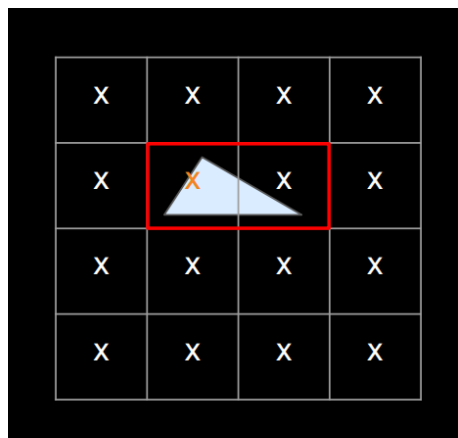


Figura 33: Señalado en rojo, el área que delimita un triángulo. En este caso, dos píxeles. El rasterizador va a testear entre esos dos cuáles necesita pintar, siendo el de la izquierda el que pinta. [34]

Realiza las operaciones de transformación adecuadas, calcula en qué píxeles cae un triángulo y por cada píxel comprueba si su centro está contenido entre las tres aristas y lo pinta si es así.

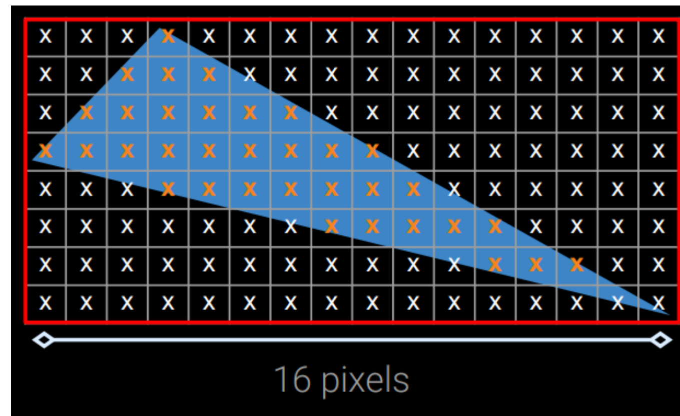


Figura 34: Este triángulo tiene una longitud de 16 píxeles, y va a iterar el test sobre los 128 que delimita. Esto es menor eficiente que el caso anterior, el rasterizador por software deja de admitir triángulos hasta los 31 píxeles de longitud de alto o ancho. [34]

Hay casos en los que un triángulo es grande, como ha sido siempre. Puede ser porque el artista lo haya querido así, al final una cara plana no tiene por qué necesitar subdivisiones. Para eso Nanite es capaz de usar el rasterizador por hardware únicamente para esos clústeres, que es más eficiente.

2.5.3.7. Geometría virtualizada

El concepto de geometría virtualizada es un análogo de las texturas virtualizadas. Esto significa que Nanite va a tener cargado sólo lo que necesita y va a descartar o pedir al disco a demanda.

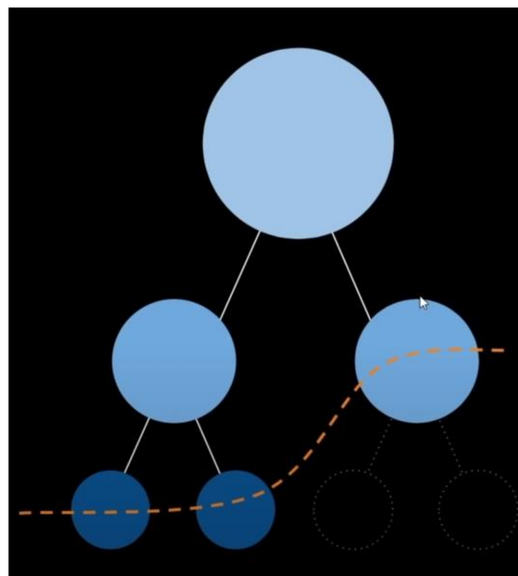


Figura 35: Árbol de LODs. [34]

Cuando se usa un nodo padre, se trata como nuevo nodo hoja y los hijos se descartan. Si se necesitan más tarde se vuelven a pedir al disco. O si se tiene un nodo, pero no se usa en un tiempo, también se puede quitar para ahorrar espacio.

Ventajas de emplear Nanite:

- Escenas con millones de polígonos en escena a tiempo real. Estas mallas pueden ocupar menos memoria y espacio en el disco que una malla tradicional con sus mapas de normales (una malla de un millón de triángulos en Nanite pesa 14MB y un mapa de normales pesa 22MB. [27]
- Rendimiento más dependiente de la resolución de salida que de la complejidad de la escena.
- Draw calls más eficientes
- LODs automáticos
- Los artistas no tienen que perder tiempo realizando mapeados innecesarios y menos optimización requerida. Es posible importar objetos de fotogrametría o esculturas de alta poligonización directamente y poder visualizarlos.
- El número de polígonos en escena no es una limitación

A pesar de todo, tiene sus limitaciones. Las cuales, al ser una tecnología en desarrollo, irán cambiando con el tiempo. Por ahora la mayor limitación es que **no soporta elementos con deformación no rígida**. Esto quiere decir que no soporta la mayor parte de animaciones de personajes.

III. HERRAMIENTAS Y ENTORNO DE DESARROLLO

En este capítulo se habla del software utilizado para resolver cada etapa del proceso.

3.1. 3DS Max

Es, junto a Maya, Blender y Houdini, uno de los paquetes de herramientas preferidos para la **producción de contenido 3D**. Es capaz de realizar prácticamente todas las tareas necesarias en la cadena de montaje. Su uso principal es el modelado, pero se puede animar, texturizar, crear materiales, iluminar, render...

Por esto mismo, se va a hacer uso de esta herramienta como punto de inicio en la producción. Saltando a otros programas cuando éstos supongan una clara ventaja en algún aspecto.

La principal desventaja de 3DS Max en este flujo de trabajo es a la hora de procesar modelos orgánicos de alta resolución. Para ello se utilizan otros softwares que suplan estas carencias.

3.2. V-Ray

V-Ray aporta la potencia del render offline con el cual se va a intentar competir dentro de Unreal Engine, para así intentar igualar los acabados del render offline a los del tiempo real. A pesar de que 3DS Max contiene todo lo necesario para la previsualización de una escena, se va a emplear el plug-in de V-Ray para iluminar y renderizar la imagen conceptual.

Dispone de sus propios elementos especializados para render: Materiales, Luces, Cámaras, geometría, efectos atmosféricos y efectos de cámara. Una solución muy útil para estos casos en los que se va a llenar la escena de millones de polígonos es el ***VrayProxy*****, que crea una versión simplificada de los modelos complejos para que poder trabajar en el puerto de vista** de 3DS Max. Ya que 3DS Max no puede gestionar muchos polígonos en escena. En tiempo de ejecución V-Ray reemplaza ese VrayProxy por modelo el original.

V-Ray se utiliza en varios campos, teniendo especial importancia en la infoarquitectura. Se ha preferido la utilización de esta herramienta frente a las demás debido al conocimiento previo que ya se posee sobre la misma.

3.3. ZBrush

Es la solución por excelencia para el modelado inorgánico y esculpido de modelos, pero no se limita a sólo esto. Tiene la capacidad de soportar escenas con millones de polígonos sin perder fluidez. De esta manera es la pieza ideal para dar un aspecto detallado a los modelos.

Es más limitado que en 3DS Max en cuanto a herramientas para texturizar o render, pero es capaz de gestionar millones de polígonos con fluidez.

3.4. Adobe Substance

El paquete de Substance, recientemente adquirido por Adobe, es una de las soluciones más potentes y estandarizadas para la generación de mapas de textura y materiales. El paquete de Substance incluye varios programas, siendo los que se van a utilizar:

Substance Designer: Generación procedural de materiales. Permite un pipeline no destructivo. Se pueden crear mapas desde cero o herramientas para procesar mapas ya existentes.

Substance Painter: Creación de texturas para un modelo. Permite un pipeline no destructivo. Emplea mapas o materiales ya creados para aplicarlos a un modelo. Adicionalmente, permite hacer mapeado de normales y otros mapas complementarios como el de curvatura. Esta última cualidad es útil en los modelos de alta resolución, porque permite crear generadores automáticos de texturas que sepan dónde generar desgaste o acumulaciones de polvo y musgo en la arquitectura (véase 4.2.5).

3.5. Quixel Mixer

De la misma manera que Substance, Quixel Mixer es una solución que ofrece una manera similar de producir materiales. Mezcla funcionalidades de Substance Designer y Substance Painter, pero de una manera más simple y menos potente. La razón por la que se incluye este programa en la lista es porque dispone de una amplia biblioteca de recursos basados en fotogrametría que Painter no ofrece y son críticos para conseguir foto realismo. Para materiales sencillos es posible que Quixel Mixer sea más rápido.

3.6. Adobe Photoshop

Solución clásica para edición y generación de imágenes. En esta línea de trabajo se va a emplear como método auxiliar para la producción de texturas cuando el software necesario no

tenga la flexibilidad o potencia adecuada. Sin embargo, se recomienda su uso como último recurso, ya que Substance Designer y Substance Painter permiten un flujo de trabajo no destructivo de manera más nativa.

También es útil para la dirección artística de la imagen, permitiendo abocetar y retocar el fotograma final para previsualizar el aspecto que se busca.

3.7. Adobe After Effects

Potente, flexible y simple. Permite realizar la composición final de una secuencia de imágenes. Tiene capacidades de animación y algunas herramientas para 3D y plugins. Es perfecto para maquetación, composición y post procesado de la imagen final del render.

3.8. Unreal Engine

Es el motor gráfico que se va a utilizar para este proyecto. Pertenece a Epic Games. Cuenta con todas las herramientas necesarias para producir una escena a tiempo real sin necesidad de *plug-ins* de terceros. En el contexto de este proyecto, se dispone de:

- La capacidad de trabajar en una línea de tiempo con distintos planos y poder animar la escena mediante *keyframes*.
- Cámara virtual que se puede ajustar para trabajar con ella como si fuera una cámara real. Con opciones para cámara anamórfica.
- Solución de iluminación realista con Lumen.
- Soporte para millones de polígonos con Nanite.
- **Herramientas de postprocesado** para tratar el fotograma final a tiempo real. Aunque para este proyecto se pueden resolver parte de estos postprocesos en After Effects.
- Potenciar la lógica de la escena mediante código. Todo puede ser programable y Unreal ofrece soluciones para poder automatizar o editar elementos de la escena mediante código: como la animación de una luz, generación procedural del escenario o gestión de cámaras. Se pueden utilizar BluePrints (programación mediante nodos) o C++.
- Integración de audio en la escena, con la capacidad de situarlo en el espacio 3D.
- Capacidad de renderizar la escena en formatos adecuados para una producción audiovisual, como **EXR** o MOV

Con todos estos elementos en cuenta, es posible realizar la totalidad del trabajo en Unreal Engine, haciendo opcional la exportación a otros programas como After Effects para el postproceso. Se dispone de herramientas de modelado, materiales, iluminación, gestión de escena, cámara, render y composición.

3.8.1. Quixel Megascans

Gracias a la colaboración entre Quixel y Epic Games se dispone de manera gratuita la extensa librería de Megascans para su utilización en Unreal Engine. En el abanico de productos se encuentran objetos 3D, texturas, materiales y demás recursos que provienen de escaneo por fotogrametría. Desempeña un buen papel a la hora de realizar una maqueta con objetos de calidad, pero también sirve para ligar los diferentes objetos en una escena. Hay mucha información que se puede sacar de un producto escaneado, ya sea de la topología o de las texturas. De esta manera, se va a extraer esta información para la generación de contenido 3D propio.

IV. DESARROLLO

En este capítulo se incluyen las etapas que se han realizado para conseguir la imagen final de este proyecto. Parte de un estudio visual y técnico, un planteamiento inicial del flujo de trabajo adaptado a la tecnología a estudiar, los problemas que han aparecido en dicho flujo y, por último: un análisis veraz de las capacidades de la geometría virtualizada y la iluminación global dinámica en el contexto de la producción de esta pieza audiovisual.

4.1. ANÁLISIS DE REQUISITOS

El estudio se compone de las siguientes fases:

- Análisis visual.
- Limitaciones del motor.
- Flujo de trabajo.
- Objetivos.
- Plataforma y perfil de entorno.
- Objetivos de render.

4.1.1. ANÁLISIS VISUAL

Al considerar una evaluación exhaustiva de las capacidades de Unreal Engine, resulta conveniente someterlo a situaciones en las cuales históricamente un motor en tiempo real no era capaz de producir resultados aceptables.

La posibilidad de incluir millones de polígonos en escena supone poder disfrutar de detalles precisos y formas que no son necesarias de simular con una textura o mapa. Un mapa de normales empieza a romper visualmente cuando:

- Se espera que un relieve de un mapa de normales proyecte una sombra u ocluya visualmente a otro.

- Se aprecia la silueta de un objeto, en cuyo caso el mapa de normales no tiene efecto alguno.

De esta manera, intentar ver el relieve en un plano frontal no tiene mucho sentido, porque en ese caso los mapas de normales funcionan bien. En su lugar, se van a poner objetos con superficies lo más paralelas posibles a la cámara y/o a la luz, buscando así un ángulo imposible para el render tradicional en tiempo real.

En cuanto a la iluminación, tradicionalmente se elige entre: sombras realistas y suaves que no se mueven o sombras dinámicas que causan bordes definidos y de poca calidad, sin rebotes.

Por ende, se plantea una escena donde hay un alto contraste: zonas de luz y zonas de oscuridad. De esta manera se aprecia mucho el fallo de la iluminación, por lo que la luz adquiere suma importancia. La escena transcurre en un interior, puesto que es más difícil iluminar realísticamente un interior al no disponer de luz y reflejos del cielo que oculten una luz poco realista. Para poder resaltar más los accidentes del terreno y las formas, se disponen las luces en un ángulo cercano a la paralela con la tangente de la superficie de los objetos en escena.

Figura 36: La luz que ilumina desde la derecha de este sujeto resalta los detalles de la piel y las arrugas. Si la luz estuviera de frente la piel se vería lisa. [35]

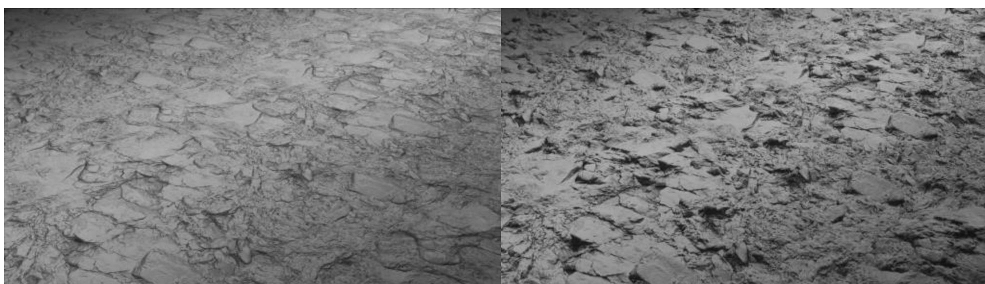


Figura 37: Buen ejemplo de una comparativa entre un mapa de normales y una topología extruida con relieve real. Las sombras son distintas y si se fuese a poner la cámara a ras del suelo una piedra no ocluiría otra a no ser que la forma esté en la escena de verdad. [36]

Figura 38: Tráiler de videojuego Elden Ring. Si se unen los conceptos anteriores y se mantiene un ángulo adecuado entre la luz y las caras del objeto. Se resaltan los detalles de las formas de la escena.

En la Figura 38, se muestra una escena construida de forma que resalta los detalles en geometría y animaciones de luz. Este tipo de escena no se puede construir en un render a tiempo real.

4.1.2. LIMITACIONES DEL MOTOR

Tener un conocimiento previo de las limitaciones con las que se trabaja es importante para adaptar el proyecto a éstas desde el principio. No es algo trivial solventar problemas de rendimiento una vez que la producción está en su fase final, o que existan problemas de compatibilidad con el motor.

De esta manera se hace un desglose de las funcionalidades no soportadas tanto por Nanite como Lumen en el contexto de este proyecto, también se mencionan las que suponen un impacto en rendimiento:

Nanite no permite:

- Geometría que se deforme, como animaciones esqueletales.
- Emplear offsets de posición global dentro de los materiales (World Position Offset). Por lo que no se puede desplazar la superficie de la geometría de manera dinámica. Esto se emplea, por ejemplo, para **animar vegetación de manera procedural**. [37]
- Geometría tipo Spline [38].
- **Color almacenado en los vértices de una instancia de geometría**. Una malla puede tener información en cada vértice, pero no se puede variar de una instancia a otra. Esto se emplea, por ejemplo, para enmascarar una geometría e integrar mejor los distintos materiales que se pueden poner sobre ella.
- El número máximo de instancias en escena es de 16 millones.
- **La información de las tangentes por vértice no se almacena**. Por lo que es interesante asegurarse de obtener una buena visualización de un material sin

depender de ajustar bien las tangentes (como se hace normalmente en el modelado inorgánico). Teniendo un número adecuado de polígonos esto no debería ser un problema.
- Solo soporta materiales de tipo opacos, los translúcidos o enmascarados no los soporta.
- Materiales de doble cara.
- **MSAA (Multisampling antialising).**
- Canales de iluminación.

Lumen tiene unas limitaciones similares a Nanite. No permite:

- Geometría que no sea estática, instancias estáticas ni terrenos.
- Desplazamiento de vértices en la posición global (World Position Offset)
- **Materiales transparentes**, que son ignorados por los distance fields.
- Materiales enmascarados por opacidad, que se tratan como opacos.
- Los objetos grandes y complejos se verán mal representados (un edificio, una montaña) ante lo cual necesitan ser **divididos en elementos más simples** para su almacenaje.
- Paredes de menos de 10cm no las trata bien, o superficies finas en general.
- Evitar que se muestre la parte de detrás de las caras de los polígonos.
- Implementación Single layer water [24]
- Hardware Raytracing contra la malla de Nanite de alta calidad (de manera nativa, véase 5.2.3).

4.1.2.1. Problemas de rendimiento

El mayor problema de rendimiento que se puede encontrar al trabajar con Nanite y Lumen es a la hora de no facilitar un trabajo adecuado al ***occlusion culling*** [39]. La filosofía de Nanite consiste en intentar no renderizar más de un polígono por píxel, pero si el motor no es capaz de descartar los polígonos no visibles que están ocluidos esta regla se rompe. Esto ocurre cuando se apilan muchas superficies encima de otras y el motor no sabe cuál escoger. Pasa normalmente con la vegetación.

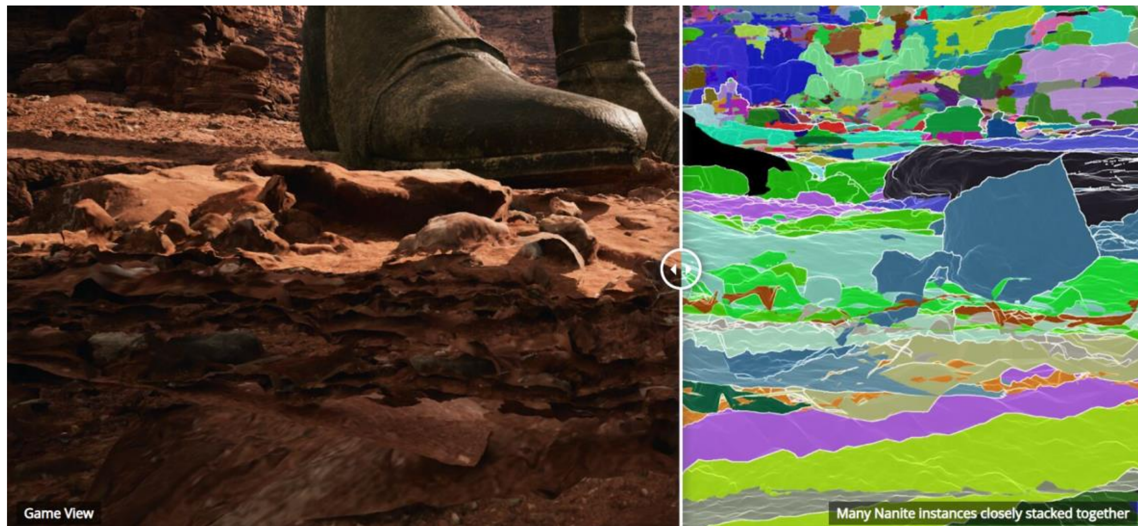


Figura 39: Caso en el que muchos objetos están apilados encima de otros. En este caso se está tirando de objetos de librería y se han apilado varios trozos de tierra uno encima de otro. A la izquierda, el modo de visualización de Nanite donde se diferencian. [27]

El segundo problema es cuando Nanite **no es capaz de reducir la geometría** acorde con la distancia. Puede darse con un conglomerado de formas geométricas que no están unidas, como la hierba o el pelo. El motor simplemente deduce que no puede simplificar más la geometría y carga demasiados polígonos en escena. Este problema también se puede dar cuando una geometría no tiene las normales suavizadas. Al intentar representar cada eje como un borde definido, el sistema intenta representar cada cara con fidelidad y no lo reduce bien. Se pueden emplear bordes que no estén suavizados si se hace en las zonas que se necesiten y no en toda la geometría. Hacer esto, por regla general, siempre ha sido un error independientemente del pipeline, aunque no se usara Nanite. Por tanto, no supone ningún compromiso nuevo.

Con esta lista de limitaciones y optimizaciones, se puede llegar a **la conclusión** de que Nanite y Lumen va a tener buen soporte en escenas con **objetos rígidos, detallados, y que no sean finos** (elementos arquitectónicos, rocas, escombros). Por otro lado, se evitarán elementos transparentes (cristal) y agua que necesite translucencia, al igual que personajes animados.

4.1.3. FLUJO DE TRABAJO

En este apartado se define las diferentes fases del flujo de trabajo para la producción de esta escena.

Gracias a Lumen y Nanite, se pueden plantear unas metodologías más típicas de la animación tradicional. Se diferencian 3 fases principales.

- **Preproducción**: Donde se planea el proyecto, se incluye la **recopilación de referencias y generación de un concepto**. Esta etapa puede ser más compleja si hubiese diferentes escenas, historia, guion, etc.

- **Producción**: Cuando se producen los elementos finales de la escena. Esta fase puede heredar objetos de la preproducción en el caso de objetos de librería que ya son definitivos, o que se puedan iterar sobre ellos. Aunque en preproducción se puede haber adelantado trabajo, en esta fase es cuando cobran verdadera importancia: **modelado, texturizado/materiales, iluminación, animación, efectos, render**.
- **Postproducción**: Si está integrada dentro del motor de Unreal Engine, es una fase que se diferencia poco de la producción. La capacidad de ver a tiempo real la imagen significa que se puede navegar entre estas 3 fases según sea necesario. Incluye la **gestión de color, efectos, composición y maquetación**.

4.1.4. OBJETIVOS

Los objetivos de esta prueba son:

- Destacar los relieves y las diferentes formas de la geometría, para ver dónde Nanite abre nuevas puertas en relación con soluciones anteriores.
- Poner a prueba cómo efectúa la gestión de la geometría con la distancia.
- Incluir mallas con geometría densa, subiendo el total de polígonos en escena a varios millones.
- Una iluminación que dependa de una iluminación global efectiva.
- Animar las luces en escena para poner a prueba el dinamismo de ellas.
- Realizar todo lo anterior sin perder de vista qué soportan y qué no soportan estas implementaciones por el momento.

4.1.5. ENTORNO DE TRABAJO Y REQUISITOS MÍNIMOS

El proyecto se lleva a cabo en un PC de sobremesa con las siguientes especificaciones:

- Windows 10 64-bit versión 22H2.
- AMD Ryzen 9 5900X 3.7 GHz.
- 64 GB RAM.
- 1 TB SSD.
- NVIDIA RTX 3070 con controladores de la versión 535.98.

Este equipo está dentro de la media de un PC de desarrollo en Epic [40] y por encima de los requisitos recomendados para un uso genérico de Unreal. Y en cuento a Nanite y Lumen se requiere un mínimo de:

Iluminación global y reflexiones con Lumen:

- Software Ray Tracing: Tarjetas gráficas con DirectX 11 y soporte para Shader Model 5
- Hardware Ray Tracing: Windows 10 con DirectX 12. Tarjeta gráfica NVIDIA serie RTX-2000 o más reciente, o serie AMD RX-6000 o más reciente.

Geometría virtualizada con Nanite:

- Todas las versiones recientes de Windows 10 (1909.1350 en adelante) y Windows 11 con soporte para DirectX 12 Agility SDK. DirectX 12 (con Shader Model 6.6 atomics), o Vulkan (VK_KHR_shader_atomic_int64). Controladores gráficos más recientes.

Mapas virtuales de sombras:

- Mismos requisitos de software que Nanite.

Temporal Super Resolution:

- Cualquier tarjeta gráfica que soporte Shader Model 5.

4.1.6. OBJETIVOS DE RENDER

Si bien el medio final es un video, y no una aplicación interactiva, pierde sentido emplear un modelo de trabajar con tecnologías a tiempo real si no se va a poder disfrutar de una respuesta inmediata en pantalla mientras se lleva a cabo el desarrollo. Por eso mismo, es adecuado establecer unos requisitos mínimos para este entorno:

- Mínimo: 24-25 FPS (fotogramas por segundo) en el puerto de vista del editor, que equivale al empleado tradicionalmente en animación/cine. La resolución del render final se establece en 4K (3840 x 2160) pero será menor en el puerto de vista para dejar espacio a las demás ventanas.
- Óptimo: 45-60 FPS en el puerto de vista del editor, con resolución 4K.

Epic Games en la primera demo estableció un objetivo de 30 FPS en consola para Lumen y Nanite. Por lo que el equipo empleado para este proyecto tiene la potencia suficiente para alcanzar ese objetivo [41]. No obstante, no debería ser suficiente en para poder lograr el objetivo de 60 FPS, pero se va a intentar.

4.2. DESARROLLO DE LAS ESCENAS

En esta sección se recopila el planteamiento seguido en las diferentes fases con notaciones sobre la diferencia que supone integrar geometría virtualizada e iluminación global en el flujo de trabajo.

4.2.1. RECOPILACIÓN DE REFERENCIAS Y PLANTEAMIENTO DE LA ESCENA

Con todo lo estudiado en el punto 4.1.4, se llega a la conclusión de que la mejor forma de poner a prueba Nanite y Lumen es construyendo una escena de interior, poco iluminada, con una luz principal que tenga mucha fuerza y luces auxiliares para destacar los pequeños relieves donde se pierdan. Estas luces pueden cambiar de intensidad (simulando una llama) o la luz principal puede moverse o ser ocluida por otros elementos, de manera que cambia la iluminación de la sala.

Una descripción bastante incompatible con lo que se puede conseguir normalmente en unos gráficos a tiempo real, pero es lo que se busca, conseguir lo que normalmente no se podría. Para asegurar la compatibilidad con Lumen y Nanite, se emplean objetos arquitectónicos grandes y con muchos relieves. Una buena demostración de estos elementos son las ruinas, ya que tienen muchas rocas y piedras con un alto nivel de relieve y rugosidad. Además, los elementos arquitectónicos pueden tener ornamentaciones densas para aumentar el nivel de detalle.

Para este proyecto se rescata un objeto creado anteriormente para Unreal Engine 4 y así llevar a cabo una comparativa entre las diferentes formas de trabajar.

Cabe destacar que ciertos modelos de especial relevancia se esculpen en alta resolución primero, antes de convertirlos en una versión optimizada para el motor en pocos polígonos. Con Nanite se puede utilizar la versión original y sacar partido a esos detalles.

Figura 40: Abajo a la derecha, el modelo original con millones de polígonos. Arriba y a la izquierda, la versión de pocos polígonos para Unreal Engine 4.

Nótese como en la Figura 40 se puede apreciar que todos los ornamentos de la cara frontal quedan reducidos a un plano con mapa de normales. Estos detalles están introducidos adrede dentro de la silueta lisa del modelo. Para que no se rompa la ilusión de relieve, la cámara debe apuntar de frente y sin mucho acercamiento.

Teniendo en mente la descripción inicial y el objeto de la Figura 40 que se va a utilizar, se forma una pizarra con referencias de piezas de arquitectura y objetos ornamentados de estilo tradicional chino. Se explora la idea de unas ruinas ocultas.

Figura 41: Pizarra de referencias de elementos arquitectónicos.

Así mismo, se monta otra pizarra con las referencias de iluminación y cámara. En este proyecto la luz y los relieven tienen mucha importancia y se busca resaltarlos.

Figura 42: Pizarra de referencias de iluminación

Como ambientación, al disponer del modelo de espada de la Figura 40, se apuesta por insinuar un tipo de escena como en Indiana Jones: un tesoro en el centro del escenario con una iluminación que le aporta todo el protagonismo de la toma.

4.2.2. DESARROLLO DE UN CONCEPTO

El primer paso, una vez fijado unas referencias claras, es realizar una composición aproximada de lo que sería el entorno y el plano por elaborar. Para ello, se apuesta por una lluvia de ideas en 3D, en lugar de un boceto 2D. Se crean elementos arquitectónicos de prueba empleando formas simples. Se utilizan mapas de desplazamiento para extruir detalles y comprobar cómo funcionan con la iluminación. También, se emplea la biblioteca de MegaScans

para importar objetos escaneados que ayuden a completar la maqueta. Este método tiene sus ventajas y sus desventajas:

- Partir de una imagen 2D conceptual, que está hecha por un artista especializado en ello, ayuda a iterar rápidamente sobre una composición potente. Ayuda a tener una idea más rápida de la estética final. Sin embargo, no se consigue una iluminación ni un render preciso.
- Empezar directamente sobre 3D aporta la ventaja de mover ágilmente los objetos por la escena manteniendo una perspectiva fiable y una iluminación también veraz. Tiene la contrapartida de ser un proceso más lento y de requerir pasar a 2D si se necesita abocetar una idea antes de modelarla.

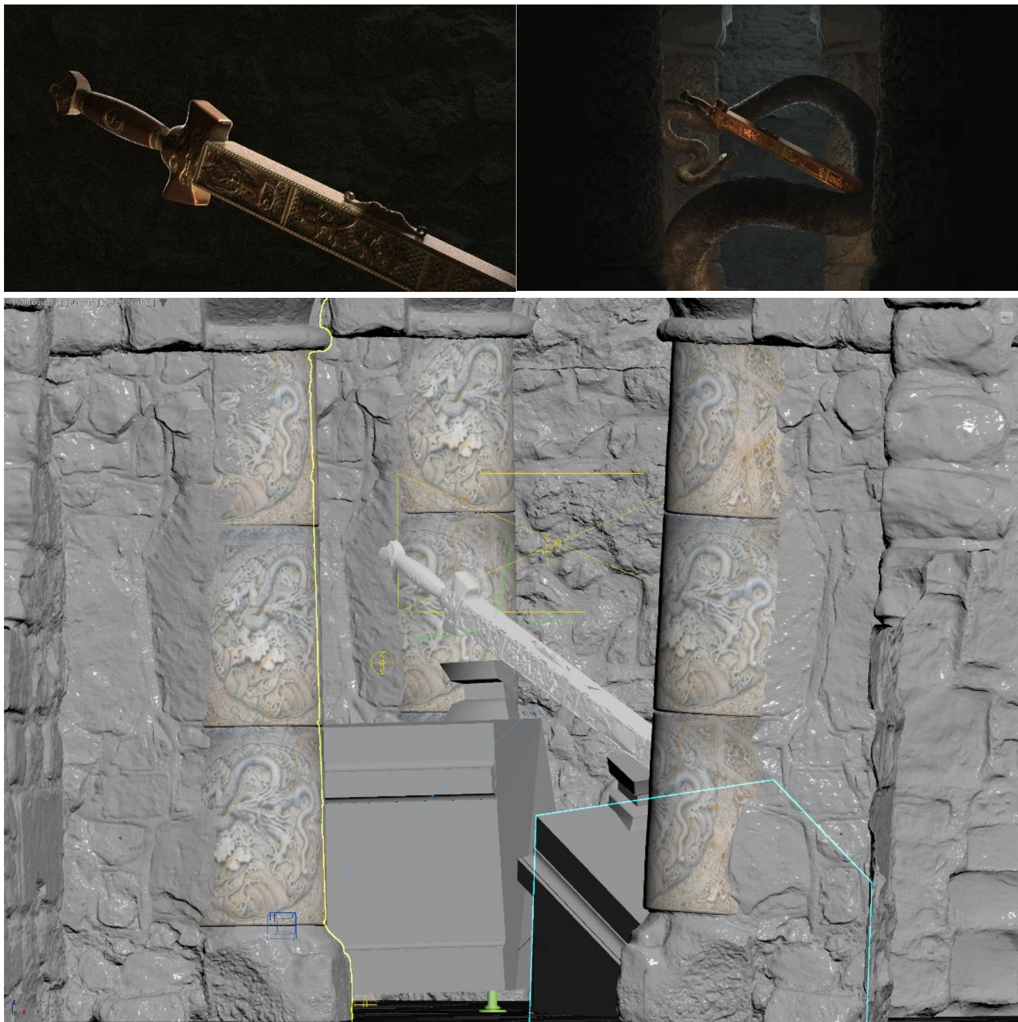


Figura 43: Lluvia de ideas en la creación del concepto.

En este caso, primero se investiga la sensación que se quiere dar con la espada interactuando con la iluminación. Posteriormente se trabaja en la composición de la escena sin texturizar nada. Lo importante es la información de la luz y las formas. Se busca una escena donde la riqueza no sea en los colores si no en las formas y los gradientes que hace la luz al chocar con el relieve.

Figura 44: Concepto final.

4.2.3. GUIÓN GRÁFICO Y CÁMARA

Aunque este no es un proyecto necesariamente de animación, sí que es relevante planear la animación de la cámara.

Por la naturaleza del concepto, lo más sencillo es un acercamiento o alejamiento de la cámara, que se sitúa de frente a la espada. Para ello se anima en 3DS Max y en Unreal y se comprueba su funcionamiento.

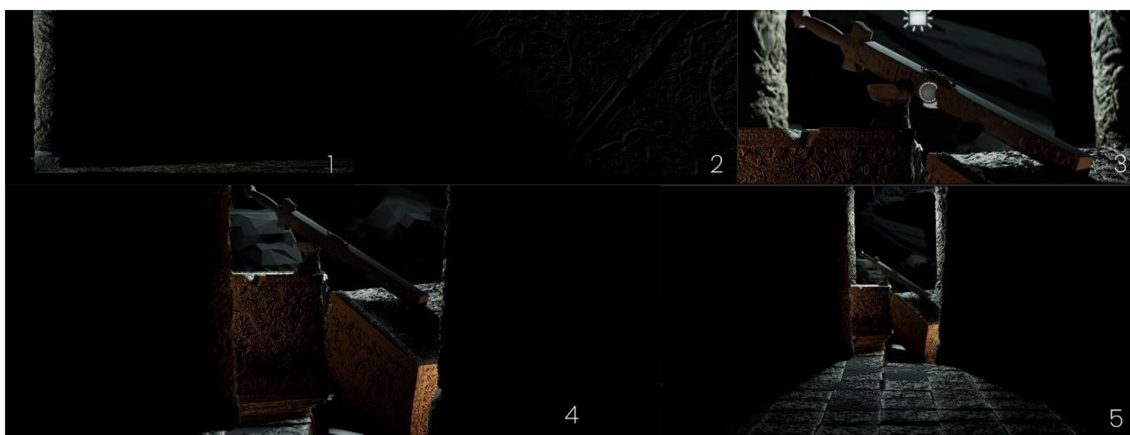


Figura 45: Propuesta de diversos planos

En plena producción se considera la posibilidad de tener distintos planos enfocando en diversos objetos del escenario como se observa en la Figura 45, pero la idea se descarta. Trabajar en un único plano asegura una mejor calidad. Finalmente se apuesta por un alejamiento de la cámara, como el plano 4 de la Figura 45.

Es necesario destacar la decisión de emplear cámaras anamórficas para este trabajo. Es un elemento interesante si se quiere intentar dar una sensación más cinematográfica, además están empezando a darse soporte desde la versión 5.1. [42] Con una profundidad de campo que tiene un efecto *bokeh* [43] estirado verticalmente como en las cámaras anamórficas y el recorte de la imagen para mantener las proporciones deseadas. Se trabaja en una proporción de 2.39 (CinemaScope o Panavision), de las más empleadas en cine.

Figura 46: Demostración de la profundidad de campo en la implementación de cámara anamórfica de Unreal Engine 5. [42]

Sin embargo, no tiene los destellos característicos, pero si fueran necesarios se pueden implementar mediante texturas o en postproducción.

4.2.4. MODELADO

El proceso de modelado ocurre a lo largo del proceso de preproducción y producción, puesto que se va iterando. Se han empleado diversas metodologías, la espada se ha hecho completamente en ZBrush, dejando 3DS Max solamente para definir las coordenadas de UV (*Unwrapping*).

Para los elementos arquitectónicos en escena, el proceso ha girado en torno a los **mapas de desplazamiento**: Se define la forma básica del objeto en 3DS Max . Luego se generan los mapas y se aplica sobre el modelo usando VRayDisplacementMod (aplica la extrusión del mapa de desplazamiento en ejecución, sin modificar la geometría original). De esta manera, se puede comprobar cómo queda en el render. Posteriormente se utiliza ese mapa en ZBrush para aplicar la extrusión de manera permanente y después acabar de esculpir el modelo.

El objetivo al modelar los objetos en este ejercicio es intentar trabajar con millones de polígonos para pasarlos al sistema de Nanite. Algunos modelos, que normalmente no superarían los 1000 polígonos en un videojuego, han alcanzado números entre 3 a 40 millones de polígonos. La razón es que la información de relieve está toda en geometría.

- **Metodología tradicional:** Crear mínimo dos modelos, uno en **alta resolución y otro en baja resolución**, posteriormente crear los **LODs** del modelo en baja resolución si el motor no los crea automáticamente. Crear las UVs en el modelo en baja resolución.

- **Metodología con Nanite:** Crear **un modelo en alta resolución** que tenga un sistema de subdivisiones dinámicas (se explica más en el apartado 5.1). Exportar el nivel máximo de resolución para el motor, pero utilizar temporalmente las versiones con menos subdivisiones para trabajar con fluidez al crear las UVs o texturizar.

4.2.5. TEXTURIZADO/MATERIALES

Esta fase se ha llevado a cabo en preproducción para planear los mapas de desplazamiento y componer el concepto inicial. Posteriormente, en producción para definir los materiales de los objetos definitivos.

Existen muchas maneras de producir mapas de desplazamiento. Se pueden hacer desde cero o partiendo de información ya existente. Esta última es la más adecuada para asegurar el mayor realismo, sacrificando la flexibilidad de poder modificarlo a gusto. Idealmente para este proyecto se llevaría a cabo de forma similar al de las grandes producciones: se organizaría un viaje a China, se recorrerían distintas localizaciones con arquitectura antigua y se realizaría un proceso de fotogrametría para extraer la topología y los materiales para su empleo en el escenario. Como esto no es posible, se ha desarrollado un método con menor coste.

Se parte de bancos de imágenes de pago o gratuitos y se consiguen esculturas antiguas con una topología interesante. Si la información de luz es ideal, hay funciones en Substance Sampler y Substance Designer que permiten extraer información de la superficie de ese objeto (relieve, color…). Esta información es poco precisa y está llena de errores. Pero se puede procesar para mejorar esa precisión y partir de una buena base con la que esculpir a mano la figura. Ahorra tiempo y es efectivo cuando los relieves que se necesitan son formas pequeñas y con poca extrusión.

Figura 47: A la izquierda, artefacto de la dinastía Ming (1368-1644) expuesto en el museo de arte de Cleveland, creative commons. A la derecha, la reconstrucción 3D de esa superficie ornamentada.

Para obtener precisión en el mapa de desplazamiento, se fusiona con el mapa de normales, que saca mejor los detalles pequeños del relieve. Este proceso se automatiza (Figura 48).

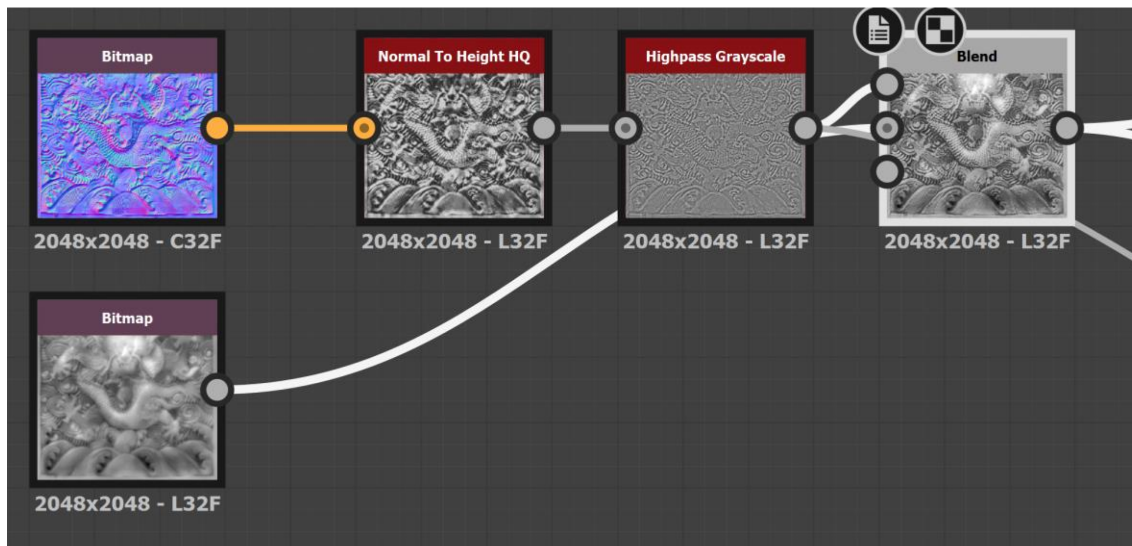


Figura 48: Pequeña función mediante nodos en Substance Designer para extraer detalle de grano fino del mapa de normales y fusionarlo en el desplazamiento.

En fase de producción, esta fase consiste en crear las texturas que definirán las imágenes finales y su implementación en el motor.

Como se pretende que todos los elementos arquitectónicos estén compuestos por los mismos materiales. Se ha optado por automatizar el proceso de estas texturas, se realiza un mapeado de texturas para obtener información del relieve y utilizarlos en el generador de texturas.

El proceso de mapeado de texturas se convierte en algo opcional:

- **Metodología tradicional**: Diseñar un modelo de alta resolución y otro en baja resolución. Proyectar uno sobre otro para extraer los mapas de normales, curvatura, etc.
- **Metodología con Nanite**: Diseñar el modelo en alta resolución, opcionalmente proyectarlo sobre sí mismo.

Quixel Mixer puede leer el relieve del objeto 3D en tiempo de ejecución sin necesidad de texturas, por eso el proceso se vuelve opcional. Pero el mapeado de texturas en este trabajo ha llevado **menos de 3-5 minutos** por objeto y sin ningún fallo. Al no usar dos objetos diferentes no hay fallos en la proyección. Conviene utilizarlo temporalmente.

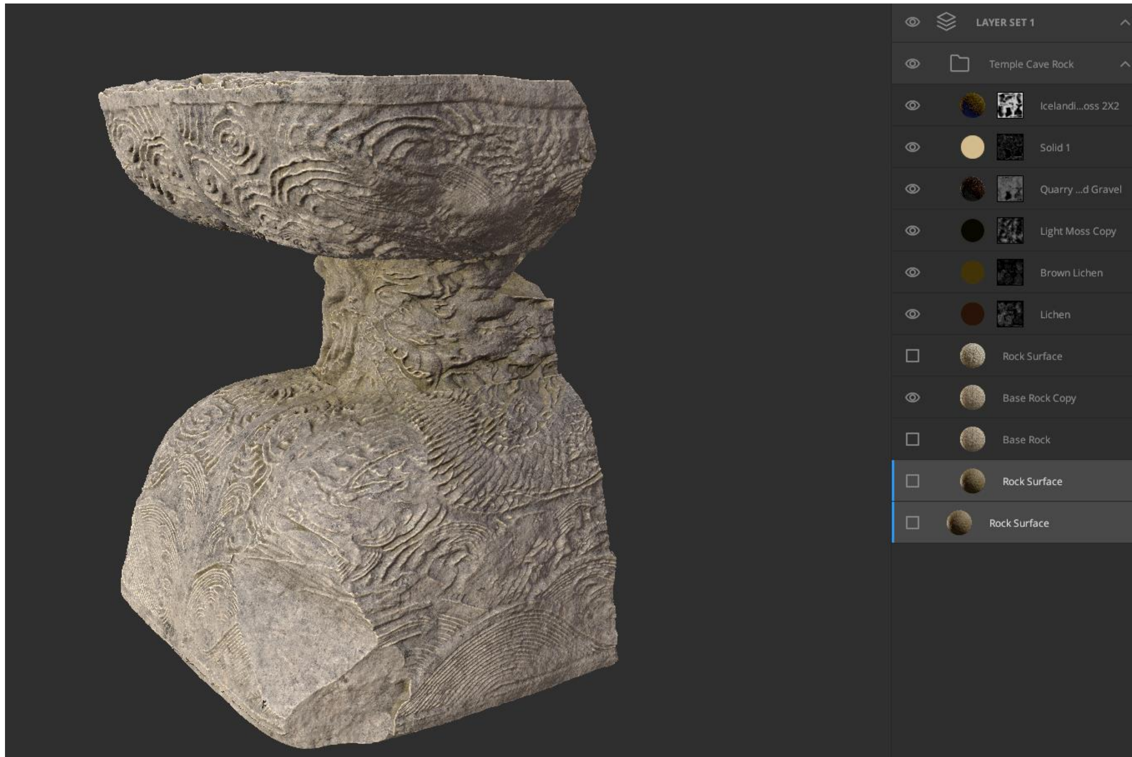


Figura 49: Generador de materiales. Lee las propiedades del relieve y aplica diferentes materiales y propiedades a las zonas cóncavas, convexas, más ocluidas, más expuestas...

4.2.6. ILUMINACIÓN

Trabajar con la iluminación utilizando lumen no difiere demasiado del render offline. Se parte creando las mismas luces que se usaron para la imagen conceptual en 3DS Max con Vray y se empieza a iterar a medida que la escena evoluciona. Por temas de optimización, en Unreal Engine las luces están delimitadas por un radio de actuación, de manera que sólo se aplican en el rango limitado que se indique.

Metodología tradicional: Poner luces estáticas, mixtas o dinámicas. Las estáticas y mixtas (llamadas "estacionarias" en este motor) requieren de un mapeado de luces, para lo que los modelos en escena deberían estar preparados con su espacio de coordenadas dedicado. Realizar versiones en baja calidad para trabajar con fluidez. Utilizar luces dinámicas para iluminar objetos que se mueven, como los personajes. Construir el escenario siguiendo los patrones de optimización adecuados para su correcto funcionamiento.

Metodología con Lumen: Poner luces dinámicas a gusto. Crear la escena siguiendo los patrones de optimización adecuados para su correcto funcionamiento.

4.2.7. ANIMACIÓN Y EFECTOS

La animación en este proyecto es básica:

- Luces que se van a animar para aprovechar que son dinámicas y comprobar así su funcionamiento.
- Rocas que caen del techo sobre el escenario, afectando así a las sombras dinámicas. Para ello, se realiza una simulación en 3DS Max con el plug-in TyFlow. Se generan varias instancias de piedra y se lanzan sobra una copia en baja resolución del escenario que se trae de Unreal Engine a 3DS Max. Una vez terminada la simulación, se importa en 3DS Max en formato Alembic [44].

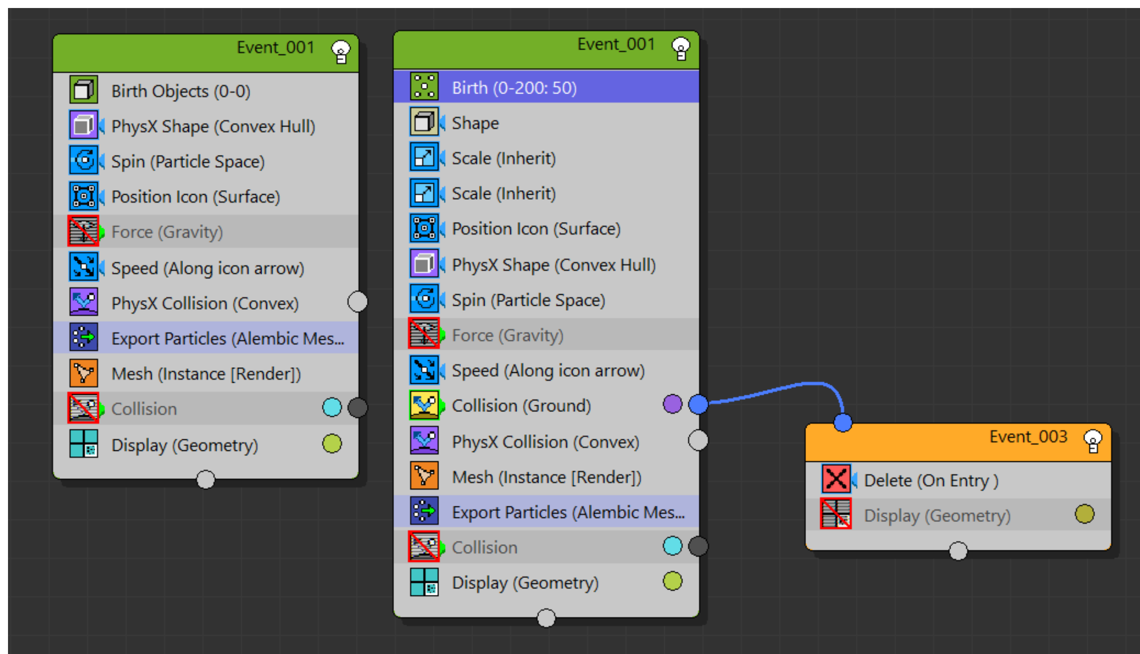


Figura 50: Sistema de partículas en TyFlow para la generación de rocas que chocan precipitándose en el escenario.

- Efectos de niebla mediante tarjetas: Se consiguen videos en blanco y negro de efectos de niebla para postproducción y se proyectan sobre planos que se colocan en el escenario. Este proceso se puede hacer en postproducción en programas externos como After Effects, pero de esta manera se consigue una perspectiva sencilla y correcta. No es una solución ideal para el rendimiento, se va a usar porque se puede activar antes de lanzar el render offline final. Adicionalmente tiene el resultado de animación más realista (al partir de humo/niebla de verdad).

4.2.8. RENDER

Para exportar el vídeo final se utiliza la herramienta Movie Render Queue porque asegura la mejor calidad, permite exportar en formato EXR (16 bits) y añadir varios pases de render para su composición. En específico, se busca el pase de profundidad, que consiste en una

escala de grises de la profundidad del escenario captado por la cámara. Es útil para ajustar los efectos de atmósfera y para poder enmascarar ciertos efectos en la escena.

Es necesario realizar pruebas con el *antialiasing* para conseguir el mejor resultado sin que los tiempos de render se salgan de presupuesto. Se ha trabajado en tiempo real durante el proceso de creación, pero si el objetivo final es un vídeo, se puede sacrificar la optimización ajustando los valores para mejor calidad en el video final. Si el objetivo es una cinemática a tiempo real, es necesario mantener una optimización máxima en todo momento.

4.2.9. COMPOSICIÓN Y MAQUETACIÓN

Se trabaja en After Effects para componer el video final, es necesario trabajar en el espacio de color adecuado (utilizando el conversor que trae el software es suficiente). Se utiliza el pase de profundidad para los efectos de atmósfera y se realizan animaciones finales cuando son necesarias. En este caso se opta por realizar un pequeño temblor en la cámara.

4.3. ITERANDO SOBRE LA IDEA ORIGINAL

A lo largo del proyecto se han realizado varias iteraciones, cambios de idea que o bien mejoran el resultado o bien acortan el tiempo de producción.

El cambio más importante sobre la idea original ha sido reestructurar levemente la disposición de los objetos en escena.

El concepto original se ve en Unreal Engine como se muestra en la figura 51:

Figura 51: Concepto original

Se ha detectado que al ser una iluminación y composición tan sutil hay poco que se aprecie en la escena. Por ello se ha procedido a abrir el marco y revelar más escenario.

Figura 52: Revisión de la composición.

En la figura 52 la iluminación aún no está perfeccionada pero se observan más elementos en comparación con la figura 51.

4.4. EVALUACIÓN DEL SISTEMA

A continuación, se va a evaluar el rendimiento final de la aplicación y se va a comparar entre las distintas técnicas a disposición del artista.

Contador	Software RT (ms)	Hardware RT (ms)	HWRT+RT Shadows VM under Budget (ms)	HWRT+RT Shadows VM over Budget (ms)
Total	14.855	14.53	16.39	62.69
LumenScreenProbeGather	2.46	2.37	1.91	30.60
Lights	1.91	1.87	1.90	5.91
TemporalSuperResolution	1.93	1.91	1.27	1.29
Shadow Depths	0.84	0.82	0.03	0.03
Slate UI	1.24	1.23	0.83	0.82
LumenSceneLighting	1.10	1.19	1.14	12.46
NaniteBasePass	0.52	0.51	2.63	2.64
Nanite VisBuffer	0.69	0.68	1.47	1.52
LumenReflections	0.68	0.80	0.99	3.29
DepthOfField	0.47	0.46	0.32	0.34
RenderDeferredLighting	0.41	0.42	0.41	0.42
VolumetricFog	0.35	0.35	0.27	0.27
Nanite Readback	0.34	0.22	0.28	0.25

Basepass	0.06	0.07	0.04	0.04
LumenSceneUpdate	0.21	0.22	0.23	0.24
Postprocessing	0.23	0.16	0.23	0.14
Prepass	0.01	0.01	0.01	0.01
Translucent Lighting	0.15	0.15	0.10	0.10
Shadows Denoiser	-	-	1.53	1.52

En el Anexo B se muestran las gráficas completas con los datos de cada columna.

Los resultados de la gráfica apuntan a un **rendimiento óptimo en el editor en el caso de Software RT y Hardware RT**. Trabajar por encima de los 65 FPS proporciona una fluidez ideal. El problema está al sustituir los mapas de sombras virtuales (sistema de sombras que va de la mano de Nanite) con sombras por Ray-Tracing. A pesar de ser unas sombras un poco más fieles, necesita realizar el ray-tracing contra la geometría densa de Nanite, no contra los mesh distance fields. El resultado es la saturación de la memoria de video, en esos casos, se ven los tiempos de render de la última columna que impiden una interacción fluida con la aplicación.

Según D. Wright, el método de Lumen por Hardware RT otorga un mejor resultado visual que Software RT a cambio de un peor rendimiento, pero **no se ha apreciado un mayor impacto**. Al contrario, se aprecia un rendimiento ligeramente superior.

El uso de las **sombras por ray tracing**, a pesar del tiempo de render total más alto, se aprecia unos tiempos de render más bajos en ciertas funciones. El mayor impacto se debe a la **interacción con Nanite y a que tiene que aplicar técnicas de denoiser**: necesita eliminar el ruido producido al renderizar las sombras mediante ray tracing.

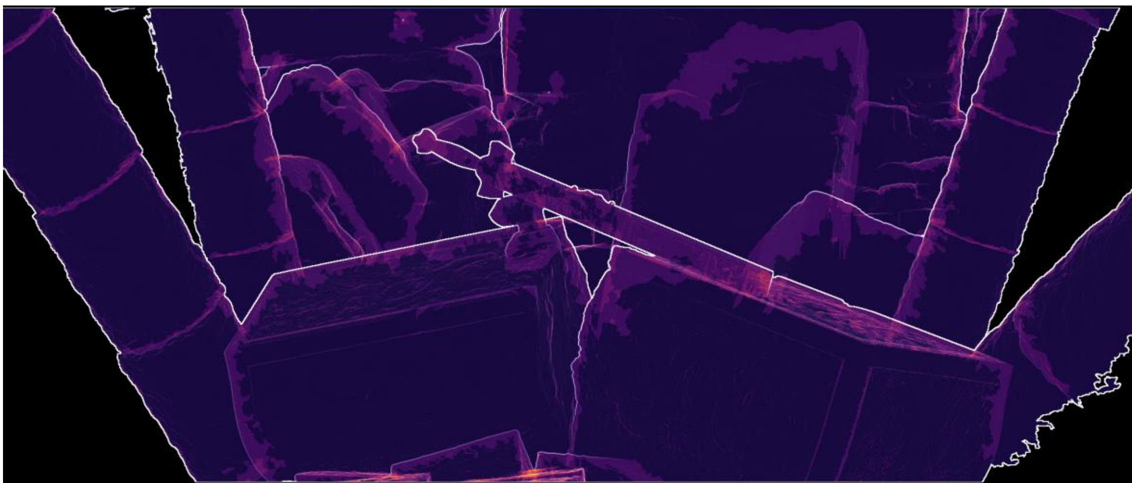


Figura 53: Modo de visualización Overdraw. Los valores claros indican superposición de geometría.

Como se puede observar en la figura anterior, los **niveles de superposición** de geometría se han mantenido **muy bajos** en la escena. Uno de los factores más importantes en la optimización de las escenas.

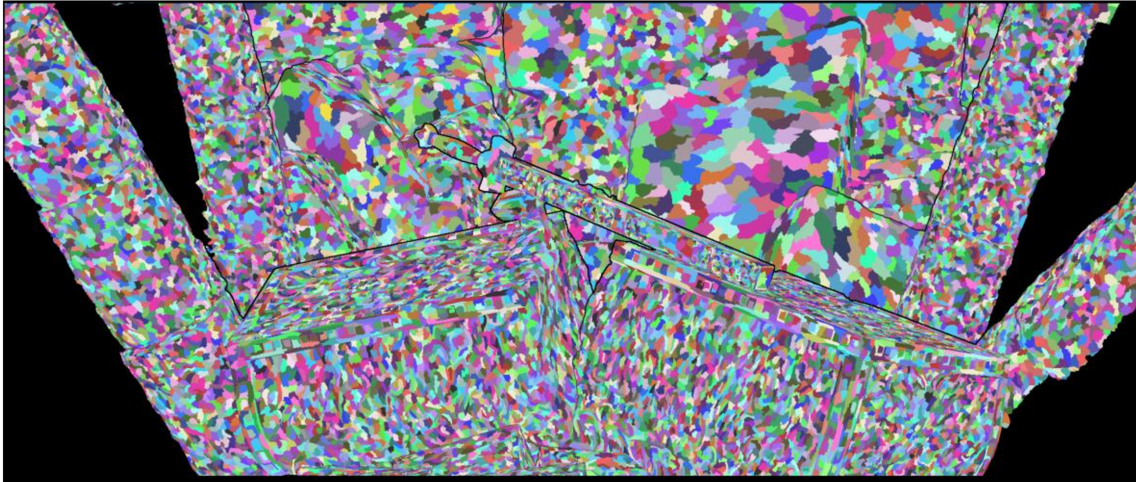


Figura 54: Modo de visualización Clusters. Se recuerda que 1 clúster = 128 triángulos

Las mallas se auto-dividen en clústeres de tamaño acorde a la distancia de la cámara, sugiriendo un funcionamiento adecuado.

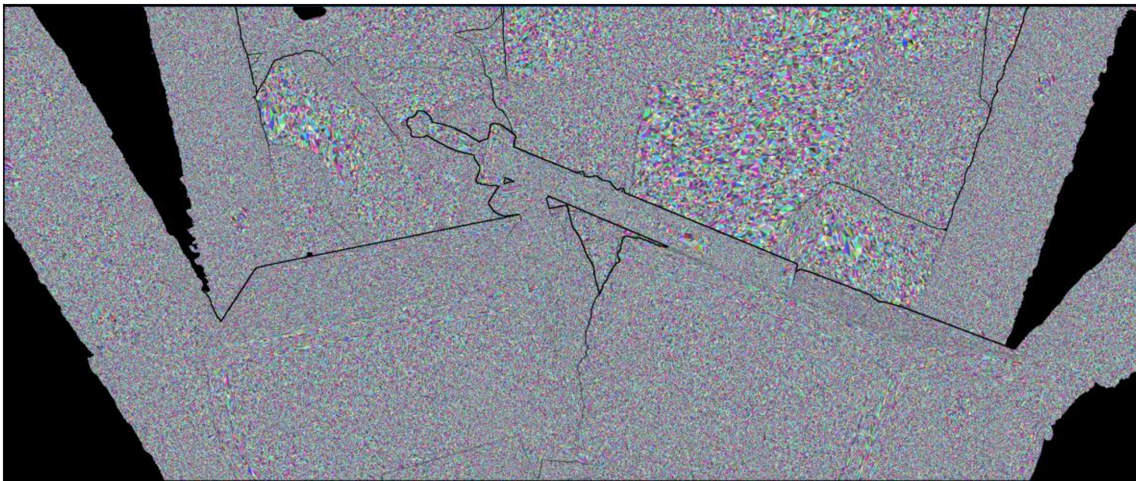


Figura 55: Modo de visualización Triangles

Finalmente, en la figura 55 se puede apreciar como resultado final la nube de polígonos en escena conseguido gracias a Nanite.

V. CONCLUSIONES

A continuación, se expone el acabado final de este proyecto junto a una reflexión de los resultados.

Figura 56: Imagen final.

En base al trabajo realizado se puede confirmar que la tecnología de **Nanite y Lumen suponen una clara mejoría en la cadena de producción** de entornos 3D. De ambos, Lumen es el que ahorra más tiempo y aporta más dinamismo a la manera de trabajar. Nanite destaca más por ofrecer mejores resultados visuales.

Sin embargo, aunque Unreal Engine ahora pueda soportar geometría densa, el resto de los programas (a excepción de ZBrush) **no están preparados** para trabajar mallas tan densas de manera nativa y supone una pérdida de tiempo e incompatibilidad si no se tiene la experiencia necesaria (véase 5.1).

Lumen ofrece resultados visuales muy aceptables, mejorando lo que se espera en un videojuego. Sin embargo, **no es lo mismo que el Path Tracing**, y los acabados siguen manteniendo ese **aspecto de videojuego** que es inherente al motor. Lo que consigue Lumen no es una potencia gráfica igual que en el mundo del cine y la animación, supone:

- **Acercar** la manera de trabajar en el cine/animación a un motor a tiempo real. Al igual que un acabado gráfico más cercano, que no igual. Hay proyectos en los que es suficiente.
- Para la industria del videojuego, poder crear cinemáticas de gran calidad sin abandonar el motor gráfico, contratando empresas externas (al poder trabajar con el mismo software) y pudiendo reutilizar el material del mismo videojuego. Especialmente con los objetos de alta resolución de Nanite. Ahorrando así dinero y tiempo.

- Dar más facilidades para la creación de efectos especiales en cámara, teniendo escenarios de Unreal Engine en lugar de un croma.
- Abrir las puertas a la industria de diseño de producto, que necesita de dinamismo en la iluminación y mucho detalle.
- Dar más herramientas a la industria de la info-arquitectura para generar resultados animados y superficies detalladas.

Lumen es una solución muy eficaz, al menos hasta que el hardware tenga la potencia para soportar el path tracing a tiempo real de verdad (actualmente se experimenta con su uso en videojuegos antiguos o pequeñas aplicaciones) o aparezca otro método que haga lo mismo de manera más eficiente.

En lo referente a esta aplicación, **no se esperaba conseguir un rendimiento de hardware RT igual o menor al de software**. Es posible que se deba por tener en cuenta los criterios de optimización desde antes de crear el proyecto: La escena tenía poco overdraw. Se ha planteado de manera que todo lo que no se ve en pantalla no se incluye en la escena, borrando elementos innecesarios como el cielo, muros o suelo cuando la luz no debería alcanzar esas zonas. Es una escena pequeña.

La tecnología que mezcla Lumen con **sombras por ray tracing no ha dado buenos resultados** de rendimiento en este caso, pero no es del todo incompatible con este proyecto: se puede utilizar única y exclusivamente para lanzar el render final. Su utilidad parece ser proyectos en los que **no se necesiten superficies complejas**: como proyectos de info-arquitectura o estéticas estilizadas, ya que así se puede descartar nanite u optimizar memoria en otros apartados como las texturas.

5.1. PROBLEMAS ENCONTRADOS

A lo largo de la producción de este proyecto han surgido diversos problemas que se deben reconocer para evitarlos en el futuro o para sopesarlos antes de decidir si Unreal Engine es una solución correcta para un proyecto.

La geometría adaptada a Nanite no es soportada por la mayoría del software.

Intentar ponerle coordenadas de textura a un objeto de 40 millones de polígonos, como es la pieza de altar del escenario, ha llevado a perder una jornada de trabajo junto con intentar texturizarlo. El hardware utilizado tiene 64 GB de RAM y aun teniendo esta capacidad, la memoria se llena al intentar importar un modelo a un programa, el cual acababa colapsando. A veces Unreal Engine tampoco es capaz de importar con estabilidad los objetos que no tiene problemas en procesar a tiempo real.

Esto se soluciona siendo estrictos a la hora de crear un pipeline. Para este caso se ha definido una manera de trabajar basado en subdivisiones dentro de ZBrush, pero es aplicable a otros programas. Consiste en lo siguiente:

- Crear el modelo A en alta resolución. Límite: número de GB de RAM del ordenador multiplicado por un millón. En este caso 64 millones de polígonos.
- Duplicar el modelo, sería el B, ejecutar Dynamesh y ZRemesher para reestructurar la geometría en un bloque contiguo de poca resolución.
- Proyectar el modelo A en el modelo B.
- Subdivir.
- Proyectar el modelo A en el modelo B y repetir hasta no apreciar una diferencia o haber alcanzado la mitad del límite de 64 millones, 32 millones en este caso.
- Ahora se puede utilizar este modelo B en cualquiera de sus subdivisiones. En el primer nivel tendrá pocos polígonos y se puede usar para ponerle las UVs. En un nivel medio se puede usar para texturizar. El nivel superior se usa para exportar.

5.1.1. Problemas con niebla y media textura

En el apartado 4.2.7 se menciona componer tarjetas de niebla en el escenario. El resultado fue bueno, pero tuvo un impacto decisivo en el rendimiento. Resulta que el motor no maneja bien las texturas con secuencias de imágenes, al menos si se usan cámaras cinemáticas. La solución ha sido desactivar ese recurso por completo, eligiendo si aplicarlo antes de lanzar el render o en postproducción en After Effects, donde no tiene coste alguno.

5.1.2. Problemas de antialiasing

El aliasing siempre ha sido un problema en la historia de los gráficos por ordendor, no es nada nuevo. Pero en este caso ha supuesto tener que reorganizar las luces en escena. Véase la Figura 51, la espada tiene unos reflejos especulares blancos en la parte superior que provienen de reflejar directamente una luz que tiene detrás. Este recurso se ha empleado en render offline y en V-Ray no suele dar problemas. Pero esto en Unreal Engine da problemas de muestreo y bordes de sierra, que luego coge las funciones de crear el blur de la profundidad de campo y crea destellos molestos que no son consistentes entre frame y frame, por lo que la imagen se vuelve inutilizable.

Figura 57: Se observan los círulos en el borde del modelo que no corresponden con nada.

5.1.3. Nanite y Path Tracing/Ray Tracing

Tanto el Path Tracer integrado en Unreal Engine como las sombras por Ray Tracing no solían ser compatibles con Nanite, y es porque utilizan la versión de la malla de más baja resolución de cada objeto con Nanite. Por lo que las sombras no concuerdan y resulta en el aspecto roto de la figura 58.

Figura 58: Escena con sombras por ray tracing

Los desarrolladores han permitido arreglarlo provisionalmente mediante la instrucción *r.raytracing.nanite.Mode 1* que hace que muestree la resolución adecuada. Sin embargo, muestrear millones de polígonos tiene un coste muy alto, por ello en el apartado 4.4 se aprecian los problemas de estabilidad. Esto al final resulta ser una herramienta para lanzar el render final o para tarjetas gráficas mucho más potentes que con la que se trabaja.

Figura 59: La misma escena empleando el comando de Nanite

5.1.4. Los detalles se perdían en la oscuridad

Los grabados no son tenidos en cuenta por la iluminación, las zonas oscuras obtenían un color negro puro que no tenía sentido al utilizar luz global. Se llega a la conclusión de que la escena de Lumen le falta resolución. La solución fue alterar los valores por defecto de Lumen Scene Detail de 1 a 2, más de 2 no se detecta ningún cambio.

Figura 60: Comparativa aplicando diferentes valores a la escena de Lumen, a la izquierda se aprecian los ornamentos del altar en la sombra.

5.2. TRABAJO FUTURO

Este proyecto ha sido muy interesante y tiene potencial para darle un acabado mejor ahora que se tiene más experiencia. A continuación, se exponen cuáles serían los siguientes pasos para mejorarlo.

- Realizar un escenario 360º. Este entorno está diseñado para funcionar con un tiro de cámara. Sin embargo, sería interesante crear un entorno completo para poder realizar diversos planos o poderse utilizar en efectos de in-camera VFX (véase la Figura 1).
- Realizar al menos 3 a 5 planos distintos para crear una pequeña escena.
- Trabajar con UDIMs [45]: La ventaja de trabajar con Nanite es poder alcanzar resoluciones en la geometría impensables, pero en este trabajo las texturas no han tenido resolución suficiente para estar a la par con la geometría. Esto es debido a que siempre se ha trabajado con un solo espacio de coordenadas UV (0-1). Implementando UDIMs se trata a ese espacio 0-1 de UVs como una baldosa, pudiendo tener diversas baldosas de UVs (1-2. 2-3…) logrando una mayor resolución.

ANEXO A: REFERENCIAS

Referencias

[1] B. Pohl, «Virtual production with Unreal Engine: a new era of filmmaking,» *Unreal Engine,* 2018.

[2] F. G. Matas, «¿Cuándo llegan los primeros juegos con Unreal Engine 5?,» *Vandal,* 3 Abril 2023.

[3] E. Tabellion, «Ray Tracing vs. Point-Based GI for Animated Films,» de *SIGGRAPH*, 2010.

[4] C. S., Radiative transfer, Courier Dover Publications, 1960.

[5] C. Barré-Brisebois, «Finding Next-Gen – Part I – The Need For Robust (and Fast) Global Illumination in Games,» [En línea]. Available: https://colinbarrebrisebois.com/2015/11/06/finding-next-gen-part-i-the-need-for-robust-and-fast-global-illumination-in-games/.

[6] G. McTaggart, «Half-Life 2 Source Shading / Radiosity Normal Mapping,» 2004.

[7] P. Devebec, «HDRI and Image-based Lighting,» de *SIGGRAPH*, 2003.

[8] R. Ramamoorthi y P. Hanrahan, «An Efficient Representation for Irradiance Environment Maps,» de *SIGGRAPH*, 2001.

[9] N. Tatarchuk, «Irradiance Volumes for Games,» de *GDC*, 2005.

[10] J. Mortensen, «Awesome Realtime GI on Desktops and Consoles,» Unity Technologies, 5 Noviembre 2015. [En línea]. Available: https://blog.unity.com/games/awesome-realtime-gi-on-desktops-and-consoles.

[11] S. McAuley, «Rendering The World of Farcry 4,» de *GDC*, 2015.

[12] C. y. S. M. Dachsbacher, «Reflective Shadow Maps,» de *SIGGRAPH*, 2005.

[13] A. Kaplanyan, «Light Propagation Volumes in CryEngine 3,» de *SIGGRAPH*, 2009.

[14] T. Ritschel, T. Grosch y H.-P. Seidel, «SSDO: Approximating Dynamic Global Illumination in Image Space,» de *SIGGRAPH*, 2009.

[15] C. Crassin, F. Neyret, M. Sainz, S. Green y E. Eisemann, «Interactive Indirect Illumination Using Voxel Cone Tracing,» de *Pacific Graphics*, 2011.

[16] M. McGuire, Interviewee, *Ray Tracing From the 1980's to Today An Interview with Morgan McGuire, NVIDIA.* [Entrevista]. 19 Marzo 2019.

[17] T. Whitted, «Turner Whitted: Untitled (Ray Traced Spheres),» 1982. [En línea]. Available: https://digitalartarchive.siggraph.org/artwork/turner-whitted-untitled-ray-traced-spheres/.

[18] R. L. Cook, T. Porter y L. Carpenter, «Distributed Ray Tracing,» de *SIGGRAPH*, 1984.

[19] J. Kajiya, «The Rendering Equation,» de *SIGGRAPH*, Dallas, 1986.

[20] S. Chandrasekhar, Radiative transfer, 1960.

[21] P. H. Christensen y W. Jarosz, «The Path to Path-Traced Movies,» de *Foundation and Trends in Computer Graphics and Vision*, 2016, pp. 103-175.

[22] J. Arvo y D. Kirk, «Particle transport and image synthesis,» de *SIGGRAPTH*, 1990.

[23] Epic Games, «Understanding Lightmapping in Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/understanding-lightmapping-in-unreal-engine/.

[24] Epic Games, «Unreal Engine Documentation: Lumen technical details in Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/single-layer-water-shading-model-in-unreal-engine/.

[25] Epic Games, «Unreal Engine Documentation: Mesh Distance Fields,» [En línea]. Available: https://docs.unrealengine.com/5.2/en-US/mesh-distance-fields-in-unreal-engine/.

[26] D. Wright, K. Narkowicz y P. Kelly, «Lumen: Real-time Global Illumination in Unreal Engine 5,» de *SIGGRAPH*, 2022.

[27] Epic Games, «Unreal Engine 5 Documentation: Nanite Virtualized geometry in Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/. [Último acceso: 2023].

[28] P. Kelly, Y. O'Donnell, K. t. Elst, J. Cañada y E. Hart, «Ray tracing in Fortnite,» de *Ray Tracing Gems II*, 2021.

[29] R. L. Cook, L. C. Carpenter y E. E. Catmull, «The Reyes image rendering architecture,» de *SIGGRAPH*, 1987.

[30] M. Sattlecker y M. Steinberger, «Reyes Rendering on the GPU».

[31] A. Keller, L. Fascione, M. Fajardo, P. Christensen, J. Hanika, C. Eisenacher y G. Nichols, «The pathtracing revolution in the movie industry,» de *SIGGRAPH*, 2015.

[32] Unity, «Optimizing draw calls,» [En línea]. Available: https://docs.unity3d.com/Manual/optimizing-draw-calls.html.

[33] Learn OpenGL, «Deferred Shading,» [En línea]. Available: https://learnopengl.com/Advanced-Lighting/Deferred-Shading.

[34] B. Karis, R. Stubbe y G. Wihlidal, «Nanite: A deep dive,» de *SIGGRAPH*, 2021.

[35] Rocket Jump: Film School, «https://www.youtube.com/watch?v=wZStU4RRGYY,» 10 Noviembre 2016. [En línea]. Available: https://www.youtube.com/watch?v=wZStU4RRGYY.

[36] A23D, «Difference between Normal Bump and Displacement Maps,» [En línea]. Available: https://www.a23d.co/blog/difference-between-normal-bump-and-displacement-maps/.

[37] «Unreal Engine Documentation: World Position Offset Material Functions,» [En línea]. Available: https://docs.unrealengine.com/5.2/en-US/world-position-offset-material-functions-in-unreal-engine/.

[38] P. Davis, «B-Splines and Geometric Design,» *SIAM News,* 1996.

[39] Epic Games, «Unreal Engine Documentation: Visibility and Occlusion Culling,» [En línea]. Available: https://docs.unrealengine.com/5.2/en-US/visibility-and-occlusion-culling-in-unreal-engine/.

[40] Epic Games, «Unreal Engine 5 Documentation: Hardware and Software Specifications,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/hardware-and-software-specifications-for-unreal-engine/#performancenotes.

[41] GinBlog82, «Youtube: Unreal Engine 5 - Valley Of The Ancient (Early Access) - 4K RTX 3070,» [En línea]. Available: https://www.youtube.com/watch?v=2NWn0EsAc9E.

[42] Epic Games, «Unreal Engine 5.1 Release Notes,» [En línea]. Available: https://docs.unrealengine.com/5.1/en-US/unreal-engine-5.1-release-notes/.

[43] H. M. Merklinger, «A Technical View of Boke,» [En línea]. Available: https://luminous-landscape.com/bokeh/.

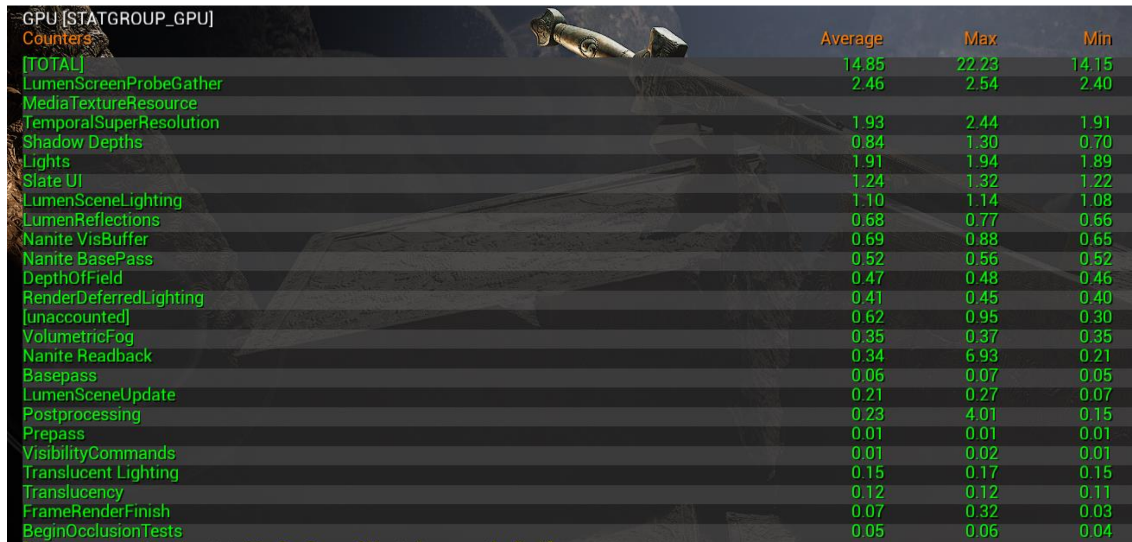
[44] Epic Games, «Unreal Engine Documentation: Alembic File Importer,» [En línea]. Available: https://docs.unrealengine.com/5.2/en-US/alembic-file-importer-in-unreal-engine/.

[45] «Blender documentation: UDIMs,» [En línea]. Available: https://docs.blender.org/manual/en/latest/modeling/meshes/uv/workflows/udims.html.

[47] Unity, «Understanding Frustum,» [En línea]. Available: https://docs.unity3d.com/es/530/Manual/UnderstandingFrustum.html.

ANEXO B: Estadísticas de render

A continuación, se exponen las estadísticas completas de los diferentes modos de render.



GPU [STATGROUP_GPU]

Counters	Average	Max	Min
[TOTAL]	14.85	22.23	14.15
LumenScreenProbeGather	2.46	2.54	2.40
MediaTextureResource			
TemporalSuperResolution	1.93	2.44	1.91
Shadow Depths	0.84	1.30	0.70
Lights	1.91	1.94	1.89
Slate UI	1.24	1.32	1.22
LumenSceneLighting	1.10	1.14	1.08
LumenReflections	0.68	0.77	0.66
Nanite VisBuffer	0.69	0.88	0.65
Nanite BasePass	0.52	0.56	0.52
DepthOfField	0.47	0.48	0.46
RenderDeferredLighting	0.41	0.45	0.40
[unaccounted]	0.62	0.95	0.30
VolumetricFog	0.35	0.37	0.35
Nanite Readback	0.34	6.93	0.21
Basepass	0.06	0.07	0.05
LumenSceneUpdate	0.21	0.27	0.07
Postprocessing	0.23	4.01	0.15
Prepass	0.01	0.01	0.01
VisibilityCommands	0.01	0.02	0.01
Translucent Lighting	0.15	0.17	0.15
Translucency	0.12	0.12	0.11
FrameRenderFinish	0.07	0.32	0.03
BeginOcclusionTests	0.05	0.06	0.04

Figura 61: Estadísticas utilizando software ray tracing.

GPU [STATGROUP_GPU]

Counters	Average	Max	Min
[TOTAL]	14.53	19.53	14.18
LumenScreenProbeGather	2.37	2.47	2.34
TemporalSuperResolution	1.91	2.45	1.88
MediaTextureResource			
Shadow Depths	0.82	0.94	0.69
Lights	1.87	1.94	1.85
Slate UI	1.23	1.29	1.21
LumenSceneLighting	1.19	1.30	1.16
LumenReflections	0.80	1.86	0.74
Nanite VisBuffer	0.68	0.74	0.65
Nanite BasePass	0.51	0.57	0.50
DepthOfField	0.46	0.48	0.46
RenderDeferredLighting	0.42	0.45	0.41
[unaccounted]	0.41	4.88	0.31
VolumetricFog	0.35	0.39	0.34
Nanite Readback	0.22	0.26	0.21
Basepass	0.07	0.63	0.05
LumenSceneUpdate	0.22	0.26	0.17
Postprocessing	0.16	0.19	0.15
Prepass	0.01	0.01	0.01
Translucent Lighting	0.15	0.17	0.15
VisibilityCommands	0.02	0.02	0.01
FrameRenderFinish	0.07	0.09	0.03
Translucency	0.12	0.13	0.11
BeginOcclusionTests	0.05	0.07	0.04

Figura 62: Estadísticas utilizando hardware ray tracing

GPU [STATGROUP_GPU]

Counters	Average	Max	Min
[TOTAL]	16.39	22.10	16.06
LumenScreenProbeGather	1.91	2.00	1.87
TemporalSuperResolution	1.27	1.29	1.26
Lights	1.90	1.98	1.87
MediaTextureResource			
Shadow Depths	0.03	0.05	0.03
Slate UI	0.83	0.88	0.81
LumenSceneLighting	1.14	1.22	1.08
Nanite BasePass	2.63	3.00	2.55
Nanite VisBuffer	1.47	1.62	1.38
LumenReflections	0.99	1.60	0.94
DepthOfField	0.32	0.36	0.31
[unaccounted]	0.27	0.30	0.22
RenderDeferredLighting	0.41	0.44	0.39
VolumetricFog	0.27	0.28	0.26
Nanite Readback	0.28	0.36	0.24
LumenSceneUpdate	0.23	0.28	0.19
Postprocessing	0.23	5.67	0.12
Basepass	0.04	0.05	0.04
ShadowsDenoiser	1.53	1.61	1.52
Translucent Lighting	0.10	0.11	0.10
Prepass	0.01	0.01	0.01
Translucency	0.11	0.11	0.11
FrameRenderFinish	0.06	0.79	0.04
VisibilityCommands	0.01	0.02	0.01

[28 more stats. Use the stats.MaxPerGroup CVar to increase the limit]

Figura 63: Estadísticas utilizando hardware ray tracing y ray traced shadows

GPU [STATGROUP_GPU]

Counters	Average	Max	Min
[TOTAL]	62.69	64.94	60.67
LumenScreenProbeGather	30.60	31.60	29.81
Lights	5.91	8.66	5.76
TemporalSuperResolution	1.29	1.68	1.27
MediaTextureResource			
Shadow Depths	0.03	0.03	0.03
Slate UI	0.82	0.85	0.80
LumenSceneLighting	12.46	13.83	10.62
Nanite BasePass	2.64	2.75	2.62
Nanite VisBuffer	1.52	1.72	1.36
LumenReflections	3.29	3.37	3.19
DepthOfField	0.34	0.35	0.33
[unaccounted]	0.28	0.29	0.27
RenderDeferredLighting	0.42	0.44	0.41
VolumetricFog	0.27	0.28	0.27
Nanite Readback	0.25	0.27	0.24
ShadowsDenoiser	1.52	1.54	1.51
LumenSceneUpdate	0.24	0.27	0.21
Postprocessing	0.14	0.15	0.13
Basepass	0.04	0.04	0.04
Translucent Lighting	0.10	0.10	0.10
Prepass	0.01	0.01	0.01
Translucency	0.10	0.11	0.10
FrameRenderFinish	0.05	0.17	0.05
VisibilityCommands	0.01	0.02	0.01

[28 more stats. Use the stats.MaxPerGroup CVar to increase the limit]

Figura 64: Estadísticas utilizando hardware ray tracing y ray traced shadows cuando la memoria virtual se satura.