Creación de entornos virtuales utilizando Unreal Engine 4

Autor: Pau Fernández Guardia

Director TFG: Sergi Grau

Titulació: Grau en Multimèdia

Data convocatoria: 11/09/2016

Resumen

Creación de un entorno virtual utilizando Unreal Engine 4, teniendo en cuenta los flujos de trabajo basados en el *Physically Based Rendering.* Además de optimizar tanto el modelado como la representación posterior in-game para conseguir una visualización en tiempo real fluida.

Palabras Clave

Unreal Engine 4, 3D, Real time, Workflow, PBR, Game Optimization.

Tabla de contenido

Glosario

UE4: Unreal Engine 4
PBR: *Physically Based Rendering*.
MRE: *Metalness* / Roughness / *Extratexture*.
GI: *Global Illumination*.
Drawcalls: Elementos que se muestran en pantalla.
FBX: Formato para exportar objetos 3D a otros programas (Propietario de Autodesk)
Subsurface Scattering: Mecanismo de la luz en el cual la luz penetra en una superficie, interacciona con el material y sale desviada.
UV: Es el proceso de "estampado" de una textura 2D en un modelo 3D.
UWV: Coordenadas de la textura.
Tessellation: Generar más geometría en las caras de un modelo.

Introducción

La industria audiovisual siempre ha intentado trasladarnos a nuevos lugares y entornos donde poder transmitir experiencias y sentimientos, desde un decorado real de una serie a un entorno medieval generado por ordenador. Esta capacidad de imaginar y generar nuevos entornos es una de las maneras que tenemos de enfrentarnos a situaciones que no serían posibles sin estos recursos.

Con cada avance tecnológico que se produce, la industria también lo incorpora a sus producciones. Fruto de ello hoy en día tenemos motores gráficos que son capaces de mostrar en tiempo real escenarios y situaciones de gran detalle y realismo. Además, estamos empezando a desarrollar una nueva etapa gracias a la realidad virtual.

Objetivos

El objetivo principal del trabajo es realizar un entorno ficticio en 3D con el que se pueda interaccionar a tiempo real utilizando en este caso Unreal Engine 4.

Algunos de los retos que me he encontrado han sido:

- Unreal Engine 4 es un motor gráfico muy potente y por ello tiene cierta complejidad. Al contrario de otros motores, este está enfocado a la industria profesional y su flujo de trabajo está adaptado a ella.
- Actualmente se está pasado del antiguo modelo de *shaders* basados en mapas especulares y reflexión a un modelo físicamente correcto llamado *Physically Based Rendering* (PBR). Por ello nuevos flujos de trabajo se tienen que aplicar y entender.
- El renderizado en tiempo real significa que hay que poder correr 25 imágenes cada segundo (como mínimo). Esto implica que cada elemento generado tiene que estar optimizado para no disminuir esta tasa.
- Gran cantidad de trabajo debido a que he tenido que realizar todas las tareas que conlleva generar un entorno.
- Errores frutos de inexperiencia en muchos conceptos.

Referentes

La creación de entornos ha estado presente desde que se podían crear imágenes por ordenador, tanto en videojuegos como en la industria cinematográfica. La diferencia es que ahora con la tecnología presente somos capaces de obtener resultado muy reales e incluso foto realísticos en tiempo real.

Muchos artistas comparten sus flujos de trabajo y técnicas con la comunidad, por lo que gracias a estas comunidades y artículos he podido entender cómo se realizan algunos elementos o entornos y aplicarlos en mi proyecto.

Motores Gráficos

Componentes

Programa principal

Donde se define la lógica del juego.

Motor de renderizado

Es el componente que se encarga del renderizado de los elementos gráficos. Se utilizan librerías de bajo nivel (DirectX, OpenGL...) y hacen uso de la GPU para el procesado.

Motor de audio

Es el componente que calcula todo lo relacionado con el sonido. Distancias, frecuencias, pos procesado…

Motor de físicas

Añade la capa de físicas en el juego, desde un simple character *RagDoll* a destrucciones en tiempo real.

Inteligencia artificial

Es el componente que se encarga de dar inteligencia a los elementos con interacción del juego.

Unreal Engine 4

Desarrollado por Epic Games, fue uno de los primeros motores gráficos orientados a los juegos FPS (*First Person Shooters*) y de los que permitió a los *modders* crear sus propias versiones.
A lo largo de los años nuevas versiones han ido apareciendo y con ellas nuevas tecnologías en todos los componentes de motor.

Actualmente se encuentra en la versión 4.12, entre sus principales características podríamos destacar, su interfaz y la calidad gráfica que puede llegar a ofrecer. Además, dispone de amplia documentación y aceptación por parte de la empresas y comunidades.

Unity 5.0

Unity fue lanzado en 2005 y su principal baza siempre ha sido la simplicidad de creación de contenidos y la posibilidad de exportar a distintas plataformas sin demasiada dificultad.

Otra de las características más interesantes y populares es su integración con los principales sistemas operativos de los móviles, lo que ha permitido que se afinque como la herramienta predilecta en el desarrollo de videojuegos destinados a estos dispositivos.

Su principal punto negativo es la calidad final que ofrece, si bien, con tiempo y recursos se puede llegar a conseguir resultados del mismo nivel que UE4, requiere más esfuerzo y en muchos casos gran conocimiento de programación de *shaders*. Aunque parece ser que este es uno de los aspectos que están intentando solventar, prueba de ello es el soporte PBR nativo a partir de la versión 5.0.

Source

Ha sido y es el motor de *Valve* y sus famosas franquicias como *Counter Strike* o *Half Life*, ha sido utilizado en juegos recientes como *Dota 2* o *Titanfall*. Su falta de actualizaciones más continuas y su dificultad de uso, lo relega a una segunda posición a la hora de desarrollar juegos.

Hay rumores de una nueva versión rediseñada que en parte solventará los problemas del motor original, pero mientras tanto, sigue siendo tosca de utilizar.

CryEngine

CryEngine es el motor desarrollado por *Crytek* y utilizado en las franquicias *FarCry* y *Crysis*. Fue uno de los primeros motores que propiciaron el salto de calidad, con nuevas tecnologías en tiempo real como la Iluminación Global. Fue un referente para comprobar la potencia de los ordenadores de sobremesa.

Actualmente sigue siendo uno de los motores que mejor aspecto visual ofrecen y prueba de ello son los nuevos títulos que se están desarrollando en él, como por ejemplo Star Citizen.

Como aspecto negativo, destacaría su elevada curva de aprendizaje para gente sin experiencia y su falta de documentación clara y ordenada.

PBR

Definición

Podríamos definir el término PBR (*Physically Based Rendering*) como un método en el que calculamos como la luz interacciona físicamente con un material a partir de datos reales. Al contrario de intuir unos valores aproximados como veníamos haciendo hasta ahora.

Aunque es un término que ahora está de moda ahora es un concepto que lleva aplicándose a la imagen generada por ordenador (CGI) desde hace muchos años ya, la principal diferencia es que los motores gráficos de hoy lo utilizan en tiempo real y no durante el render como hacen los programas de renderizado.

Ventajas

- Al utilizar materiales físicamente correctos podemos estar seguros que se verán bien independientemente de las condiciones de luz del entorno.
- Elimina el tener que adivinar los atributos de un material, es decir, los niveles de reflectancia que utilizaremos para cierto material los obtendremos a partir de datos reales.
- Al ser un sistema estandarizado evita errores de interpretación entre diferentes artistas de un equipo.
- Que sean físicamente correctos no significa que no se pueda estilizar el diseño y/o tono de nuestro proyecto. Es perfectamente posible llegar a obtener un estilo *cartoon*.

Desventajas

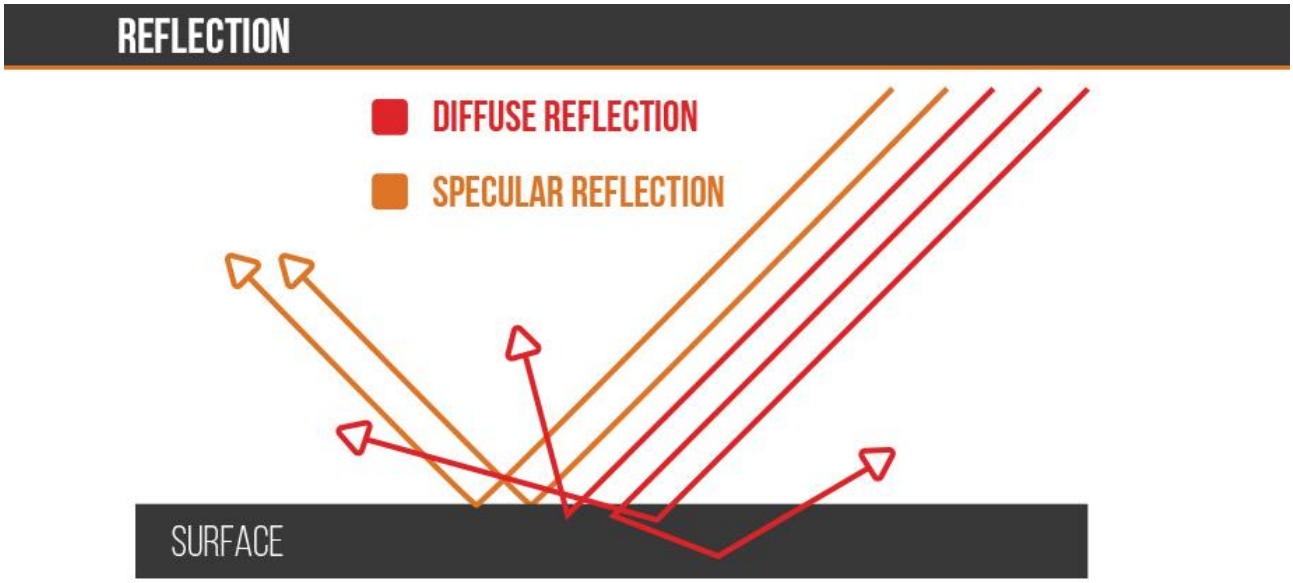
- Utilizar este sistema en vez del tradicional *Specular/Glossines* supone cambiar ligeramente el flujo de trabajo que se venía haciendo hasta ahora.
- Si tenemos modelos utilizando la metodología anterior habrá que hacer el esfuerzo de convertirlo al nuevo sistema.
- Dependiendo de la resolución y densidad de la textura, podemos observar un pequeño tinte blanco en los bordes del mapa de *roughness* si no lo arreglamos antes.

Terminología

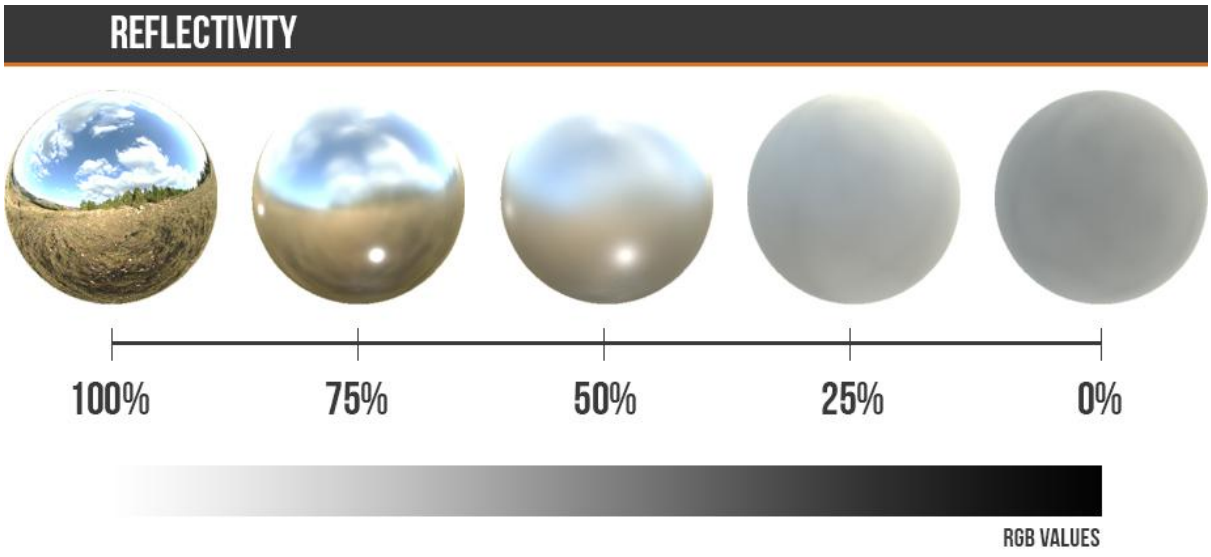
Reflexión difusa y especular

La reflexión especular y difusa son uno de los aspectos físicos básicos de la luz. En el caso de la reflexión especular se produce cuando la luz incidente rebota sobre una superficie totalmente pulida. Sin embargo, casi todas las superficies tienen irregularidades y por lo tanto al rebotar la luz no seguirá una trayectoria perpendicular.

La reflexión difusa es luz refractada que al cambiar de medio y entrar en el material, rebota múltiples veces dentro hasta que al final es refractada de nuevo afuera con un nuevo ángulo de salida.

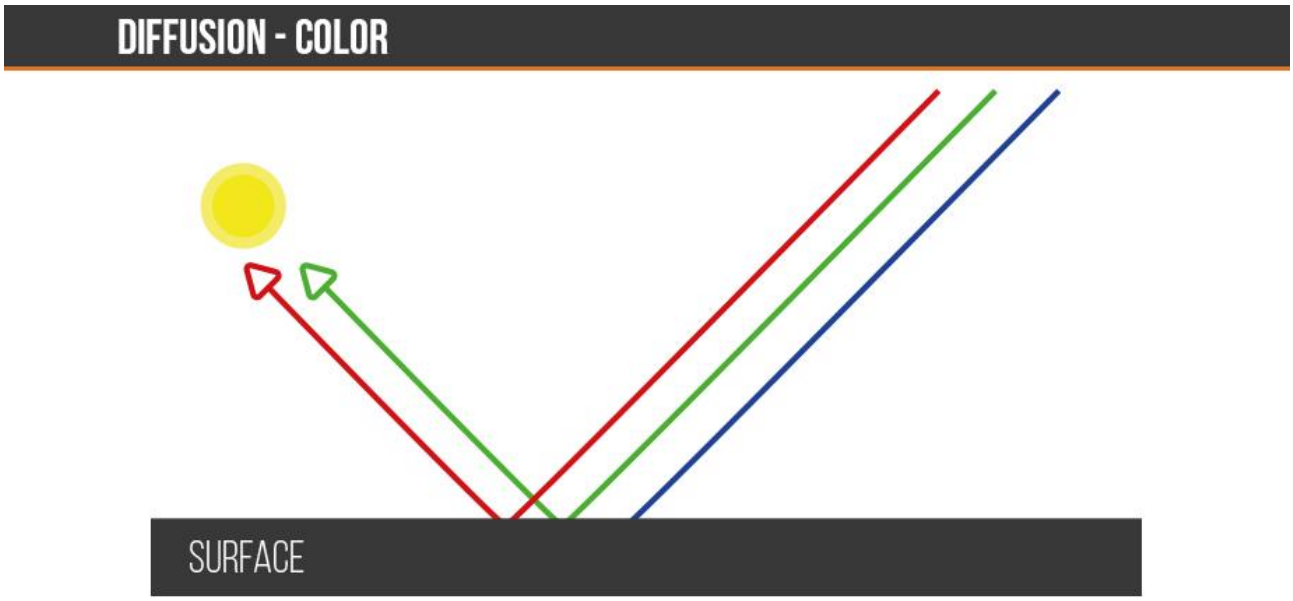


1Diferencia física entre reflexión difusa y reflexión especular



2 Visualización gráfica de la relación RGB con la reflectividad

En un caso perfecto, podríamos decir que la totalidad de la luz emitida es la reflejada, pero en la práctica esto no es así, parte de esta luz emitida es absorbida por el material, normalmente en forma de calor o en rebotes internos (*SubSurface Scattering*).



3 Gráfico representativo de la obtención del color

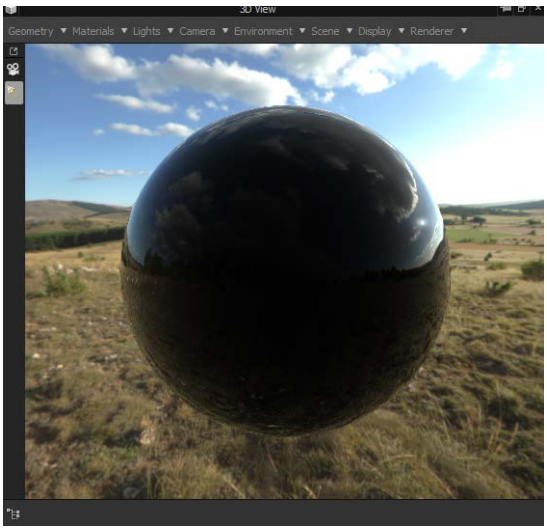
Cada material tiene una longitud de onda que refleja o absorbe, debido a esto, podemos ver los colores que tiene un objeto, de hecho, los colores son el reflejo de la luz que es no es absorbida por el material. Por el ejemplo, en el gráfico anterior podemos ver como el material solo refleja los colores verde y rojo, por lo tanto, al visualizar el objeto, el color resultante (luz no absorbida) será el amarillo (mezcla de verde y rojo).

En el caso de los motores gráficos, este color reflejado se simplifica en un campo de color denominado *Albedo o Diffuse Color.*

Efecto Fresnel

El efecto Fresnel dicta que la cantidad de luz que ves reflejada de una superficie, depende del ángulo de visionado. Cuanto más cerca se esté de la tangente de la luz incidente (90º), más cantidad de luz se verá reflejada. Unreal engine gestiona este aspecto automáticamente, aunque para algunos casos muy específicos podemos llegar a modificarlo.

Con superficies pulidas y con un ángulo de visión de 90º respeto el rayo incidente, el reflejo será especular reflejando casi el 100%.



Cuando la luz incide perpendicularmente contra una superficie, hay una cantidad de luz especular reflejada, esta cantidad es referida como F0 (Fresnel 0), la cantidad que es refractada en el material la denominamos 1-F0.

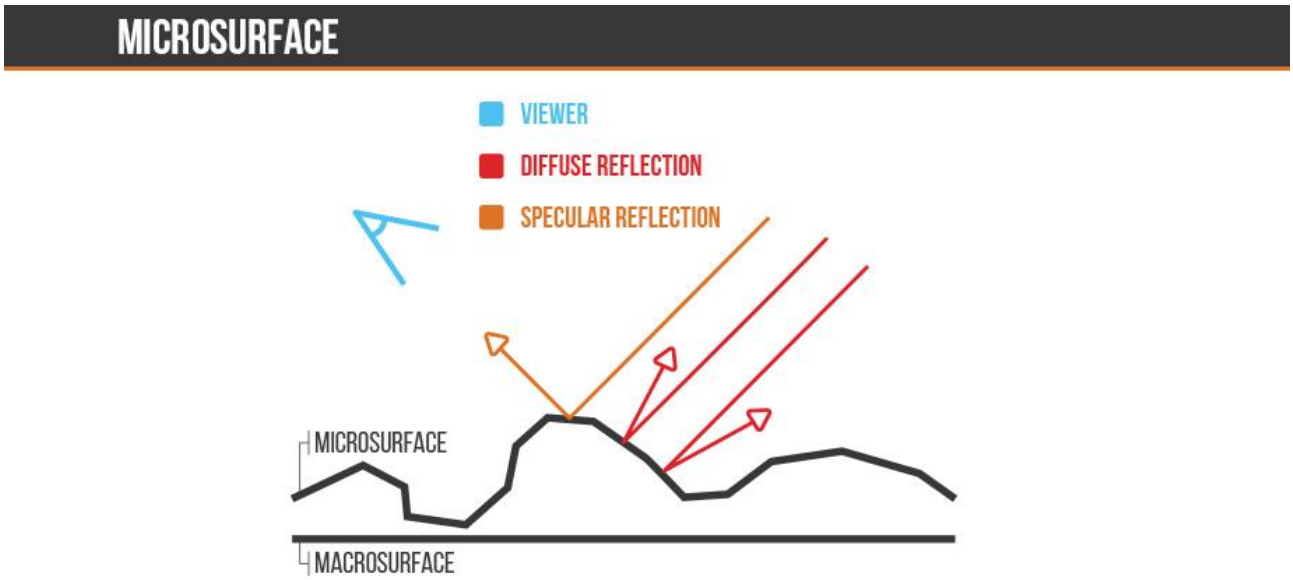
Para los materiales dieléctricos (no metálicos) suele oscilar entre 2% - 5% (0º) y en los materiales conductores (metálicos) entre 50% y 100% (0º). En ambos casos en ángulos de 90º (F0) reflejaran el 100%.

4 Efecto fresnel en un objeto completamente reflectivo

Microsuperficie

Que se produzca reflexión especular o difusa en la teoría depende de las irregularidades de la superficie. Aunque en la práctica este efecto es menos visible ya que la dispersión de la luz ocurre dentro del material.

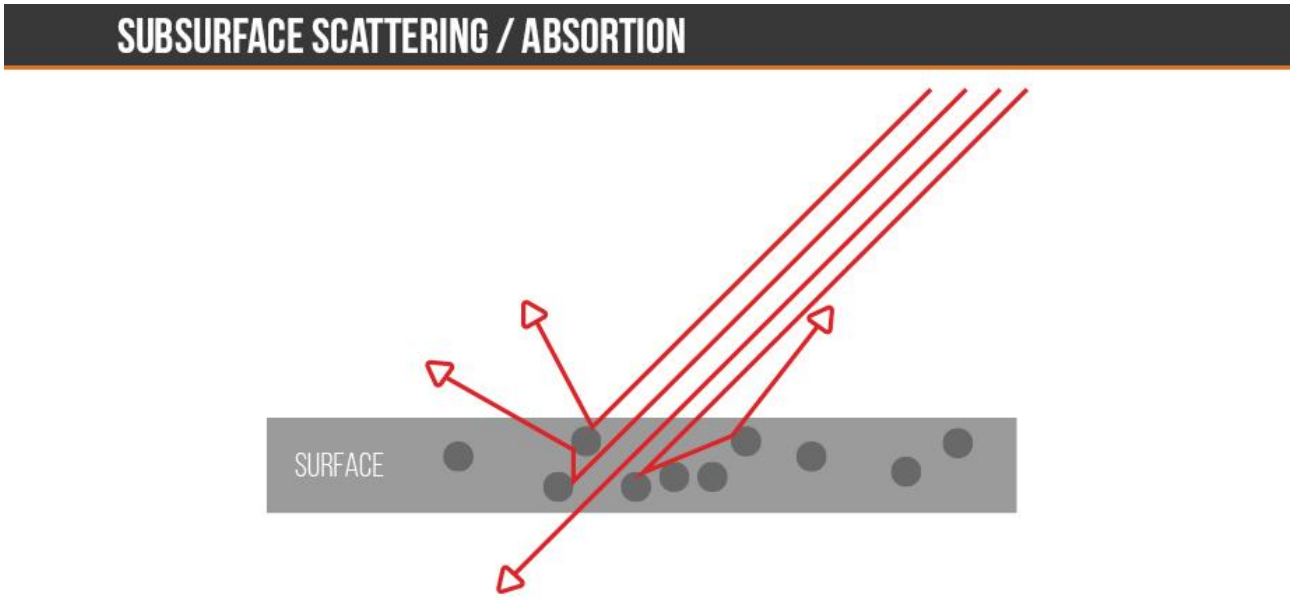
En los motores gráficos, esta superficie recibe el nombre de *Roughness, Glossiness o Microsurface.*



5 Representación física de las microsuperficies

Absorción y dispersión

En algunos casos como en la piel, cera o cristal, la luz puede ser absorbida o dispersada. En la absorción, la luz es transformada en forma de calor y no se refleja, en cambio en la dispersión, la luz es desviada al entrar en el material y si es lo suficientemente delgado, penetrara y podremos observar el reverso. Un ejemplo claro de esto es nuestra mano, si la colocamos en contra del sol, podemos observar como la luz penetra dentro. En cambio, en otras partes del cuerpo donde la piel no es tan delgada, no podremos observar este fenómeno.



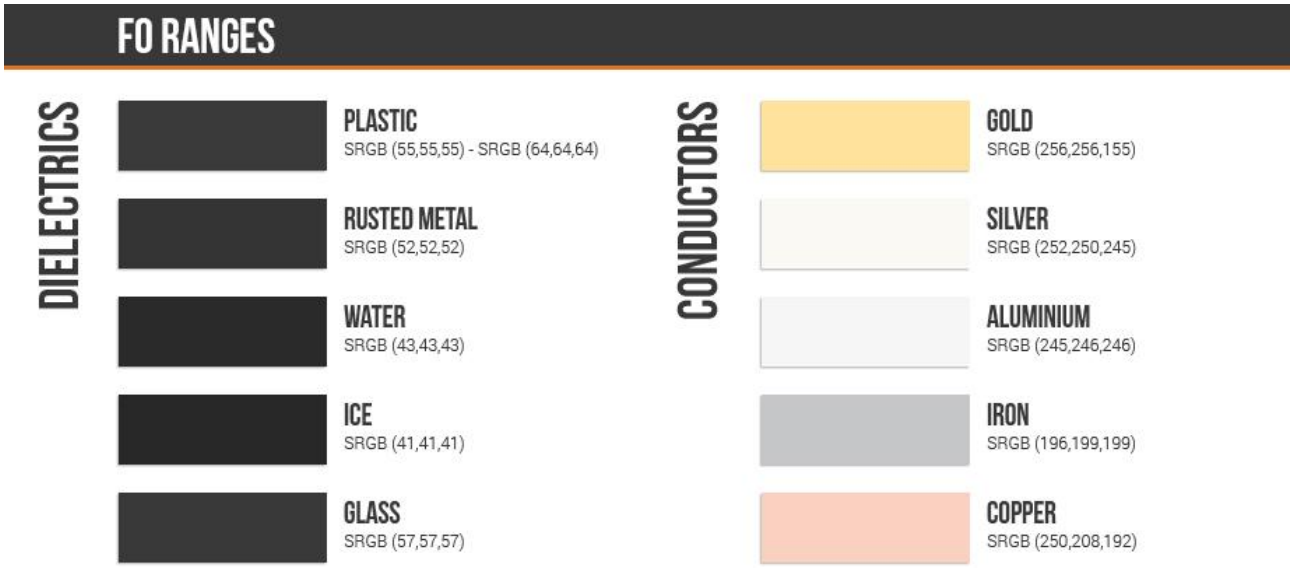
6 Representación física del Subsurface Scattering

Conservación de la energía

Este término se refiere a que la cantidad de luz reflejada por un objeto nunca podrá ser mayor a la incidente, ya que de lo contrario nos estaríamos inventando datos de la fuente lumínica.
Por suerte, esta limitación física / teórica está implícita en el motor gráfico y por lo tanto no tendremos que preocuparnos por si se cumple.

Materiales Dieléctricos y Conductores

En aras de la simplicidad podemos agrupar los materiales en metálicos y no-metálicos, siguiendo este esquema conseguiremos abarcar la gran mayoría de materiales junto a sus propiedades físicas. En algunos casos como en los semimetales tendremos que aplicar otras técnicas.



7 Gráfica con los valores RGB de distintos materiales conductores y dieléctricos.

Metales

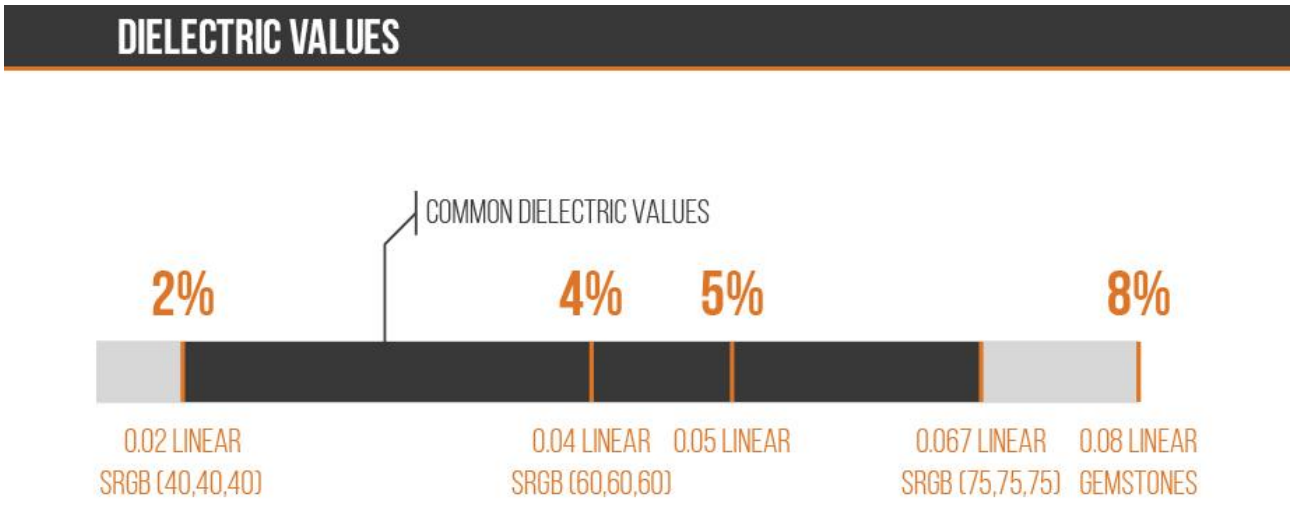
Los materiales metálicos destacan por su alta reflectividad, entre un 70 - 100% y por la capacidad de absorber toda la luz refractada en ellos. Normalmente el color difuso de los metales es negro, pero en algunos materiales como el oro o el cobre la reflexión tiene un tinte. En estos casos “colorearemos” el metal utilizando el mapa de color base (difuso) además de indicarle la reflectancia.

En el caso de los metales pintados, trataremos la pintura como dieléctrico.

Material	BaseColor (R, G, B) (Lineal)
Iron	(0.560, 0.570, 0.580)
Silver	(0.972, 0.960, 0.915)
Aluminum	(0.913, 0.921, 0.925)
Gold	(1.000, 0.766, 0.336)
Copper	(0.955, 0.637, 0.538)
Chromium	(0.550, 0.556, 0.554)
Nickel	(0.660, 0.609, 0.526)
Titanium	(0.542, 0.497, 0.449)
Cobalt	(0.662, 0.655, 0.634)

No-Metales

Los dieléctricos reflejan mucha menos luz que los metales, tienen color difuso (albedo color) y su reflectividad oscila entre 2% - 5% (40 - 75 sRGB) (0.017 - 0.067 Lineal). A excepción de las gemas que oscila sobre el 8 % (85 sRGB) (0.08 Lineal).



8 Gráfica con los valores porcentuales de los materiales dieléctricos.

Material	BaseColor Intensity (Lineal)
Charcoal	0.02
Fresh asphalt	0.02
Worn asphalt	0.08

Bare soil	0.13
Green grass	0.21
Desert sand	0.36
Fresh concrete	0.51
Ocean Ice	0.56
Fresh snow	0.81

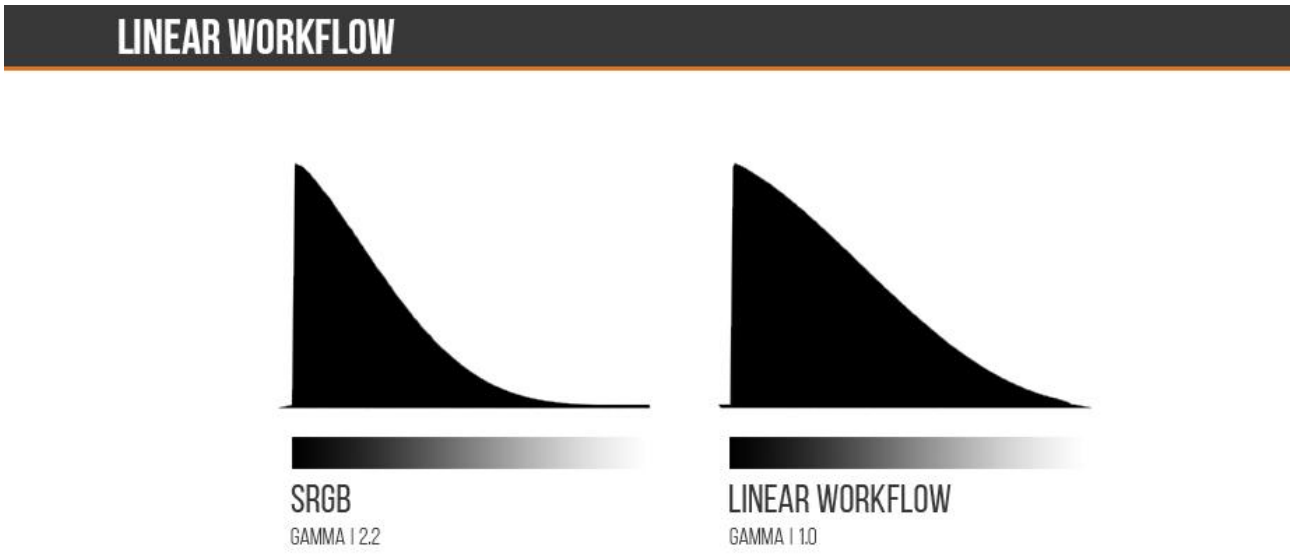
Workflow

Flujo de trabajo lineal

Trabajar con un espacio lineal permite trabajar con las mismas condiciones sin importar en qué parte del proceso se esté, además es imprescindible si queremos que los que generemos se vea igual a lo que tenemos en mente.

En el flujo de trabajo lineal, la gamma es 1.0, pero para que se vea correctamente en un soporte visual, se le aplica una corrección de gamma. Cuándo aplicar o no el flujo de trabajo lineal depende del mapa y su finalidad. Para simplificar este proceso podemos suponer que si es un mapa que se va a ver directamente en la pantalla (Color base o diffuse) tendremos que trabajar en el espacio sRGB, en cambio, si es un mapa utilizado para obtener características de la superficie (*Roughness, Metallic, Normal, Thickness, Emission*...), estos deberán ser trabajados de forma lineal.

En el siguiente render lateral podemos observar que sucede si aplicamos un simple gradiente como mapa de desplazamiento.
En sistema sRGB debido a la corrección de gamma que se hace, el gradiente en vez de ser progresivo, va modificándose a lo largo del camino generando la típica curva de corrección de gamma, en cambio, si usamos el sistema lineal, efectivamente obtenemos el resultado progresivo que esperaríamos.



9 Representación gráfica de la curva generada según el flujo de trabajo que utilicemos.

LINEAL	*Roughness, Metallic, Normal, Thickness, Emission*
sRGB	*Base Color, Diffuse*

Concept Art

Conceptualizar directamente una escena es una tarea que ni siquiera los grandes pintores pueden llevar a cabo, por ello el uso del arte conceptual es casi imprescindible en la mayoría de trabajos. No únicamente desde un punto de vista artístico, sino también desde el apartado técnico.

Modelar, texturizar, optimizar son procesos que requieren mucho tiempo y recursos, si no se planifican adecuadamente pueden llegar a colapsar un proyecto. Al usar arte conceptual este problema queda medianamente solventado y nos permite centrar esfuerzos en lo realmente útil y necesario.

El uso de arte conceptual es una tarea que normalmente se realiza en preproducción ya que requiere menos esfuerzo y permite definir desde muy temprano muchos elementos del proyecto como el tono, la ambientación, el estilo…

Modelado 3D

Cuando modelamos un objeto en 3D tenemos que tener en cuenta el medio en el que se va a visualizar, no es lo mismo modelar para entornos que tendrán más elementos a parte del espacio en si (IA, Físicas...), que para entornos en los que no hay más computación detrás y por lo tanto podemos destinar más recursos. Estas limitaciones vienen dadas por el hardware y en menor medida el software del que disponemos.

El renderizado en tiempo real es posible gracias a una serie de métodos que permite optimizar la cantidad de elementos que se muestran por pantalla. Uno de los métodos consiste en intentar tener la mínima cantidad de triángulos (tris) por objeto y suplir esta falta de detalles usando otras técnicas como pueden ser los mapas de normales.

Low Poly

Como su nombre indica, es el modelo de bajo poligonaje que importamos al motor gráfico y sobre el que le aplicaremos todos los mapas generados previamente. Si nuestro modelo está destinado a ser animado, también tenemos que tener en cuenta las deformaciones que sufrirá y adaptar la malla a estas animaciones.

Es importante que el mapeado (UV) del modelo esté bien realizado, de lo contrario cuando apliquemos los mapas generados, no se verán de manera correcta.

High Poly

Es el modelo de alto poligonaje y del que extraemos varios mapas. No lo usaremos nunca en producción ya que sería inviable si lo que queremos es una experiencia fluida.

Normalmente no es necesario mapear correctamente el modelo ya que su única función será la de volcar sus detalles al modelo *low poly*.

Retopología

Este es el proceso por el cual transformamos la malla *high poly* en una *low poly*, es necesario y obligatorio si queremos que el modelo se pueda usar posteriormente en nuestro motor gráfico. Aunque dependiendo de su uso final, podemos llegar a ser algo más laxos y automatizar este proceso sin tener consecuencias después.

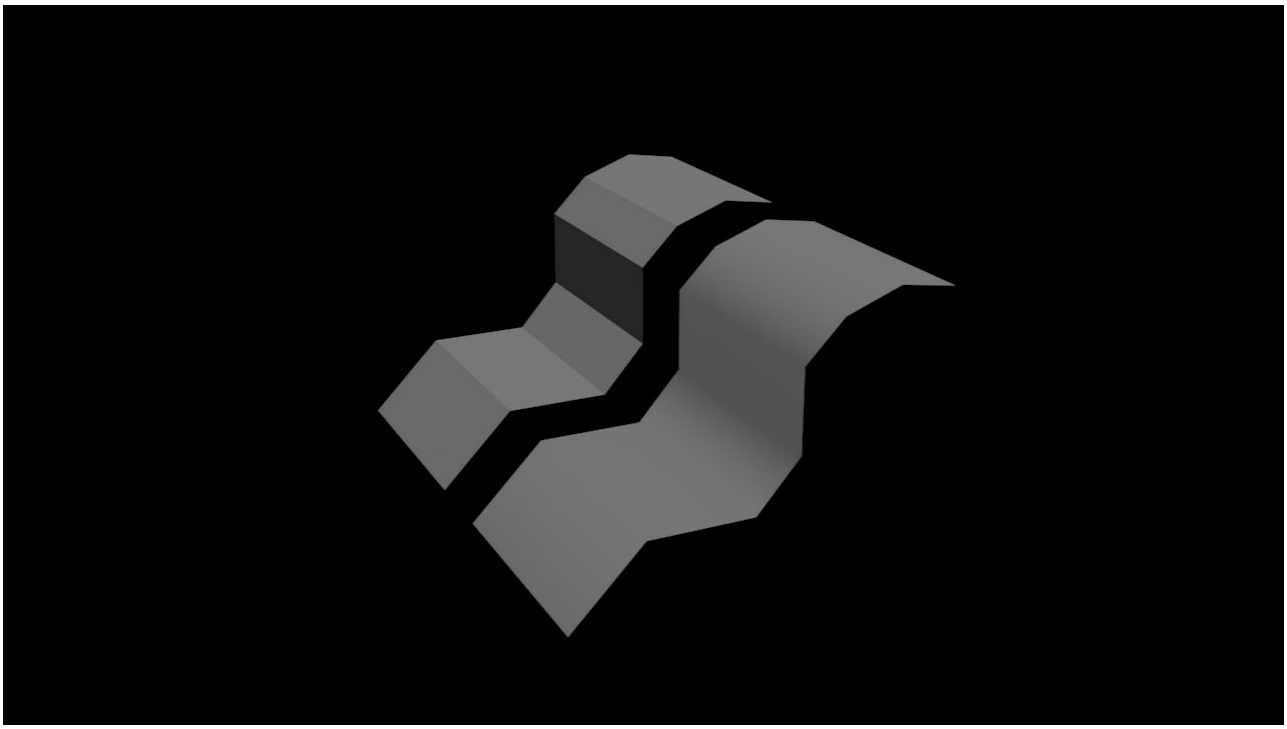
El procedimiento es simplemente volver a modelar nuestro objeto, pero intentando seguir nuestro modelo high poly con el menor poligonaje, tenemos que tener en cuenta que hay que ir adaptando la densidad de malla según el detalle original, ya que, si bien después irá aplicado el mapa de normales, si disponemos de muy poca densidad el resultado final no será satisfactorio.

Además, si el modelo está destinado a ser animado, tendremos que tener en cuenta en qué zonas añadimos detalles, ya que si nos encontramos con poca geometría seguramente surjan deformidades no deseadas o extrañas en el proceso de *rigging*.

Smoothing Groups

Los grupos de suavizado es una herramienta que nos permite definir la *suavidad* con las que se pasa de un plano a otro. Es una manera de optimizar el modelo ya que en vez de definir la suavidad añadiendo más geometría, definimos la relación entre caras y el motor gráfico ya se encarga de mostrar el resultado suavizado correspondiente.

En la siguiente captura podemos observar que, mientras el que está más al fondo no tiene grupos de suavizados asignados y se ven las distintas caras (facetado), el que está más en frente, al asignarle un único grupo de suavizado a todas las caras, la transición entre ellas es mucho menos notable y da sensación de uniformidad con la misma geometría que el otro modelo.

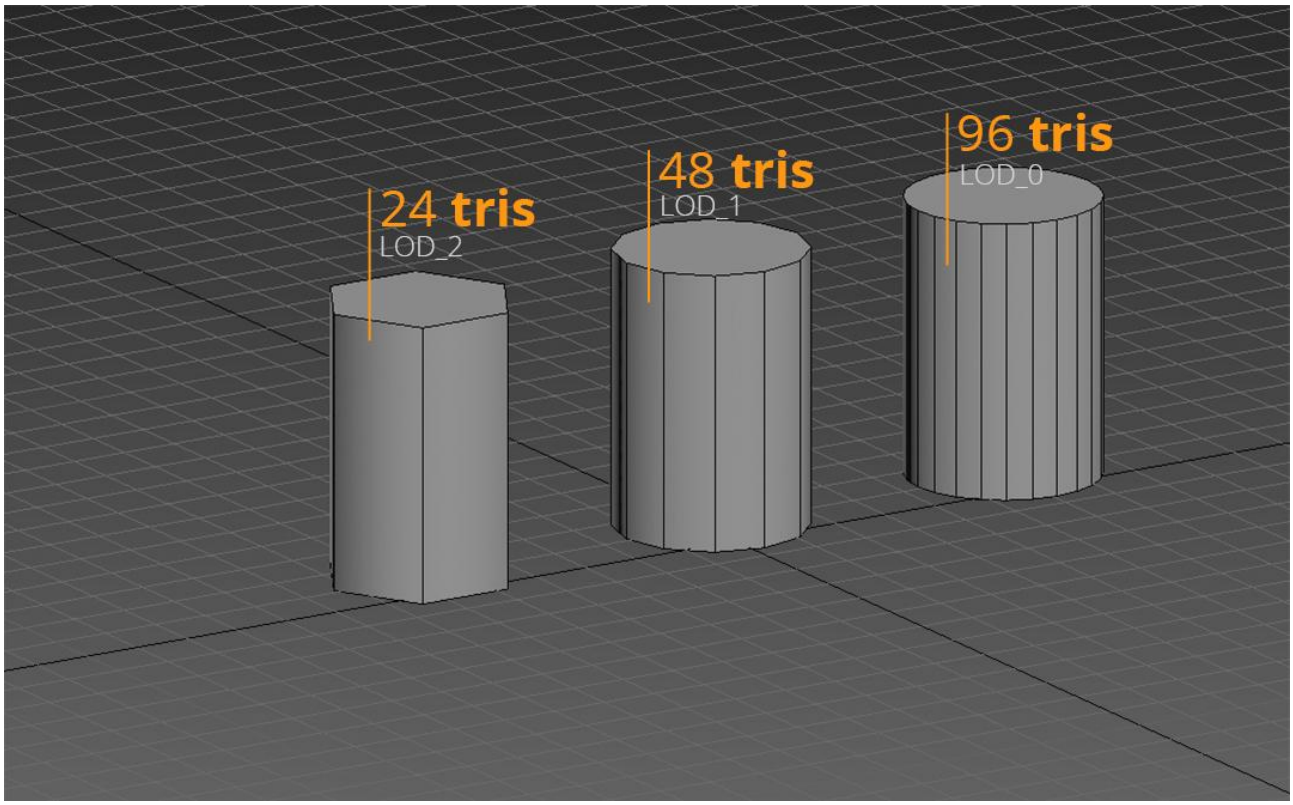


10 Representación visual de la aplicación de grupos de suavizado en un modelo

LOD

LOD viene del inglés *Level Of Detail* que vendría siendo niveles de detalle en español. Se usa principalmente para cargar diferentes modelos de mayor o menor poligonaje dependiendo de su distancia de visionado. Si el modelo está lejos de nuestra visión, podemos permitirnos cargar un modelo reducido que reducirá la carga de la escena sin que conlleve una pérdida de calidad visible por el usuario. En cambio, al irnos acercarnos, gradualmente se irá reemplazando por los diferentes modelos hasta que lleguemos al nivel LOD 0 que cargará el modelo con mayor resolución.

Es una buena técnica para optimizar el uso de recursos en la escena, como único inconveniente es que dependiendo de lo gradual que sea el cambio y de cómo lo gestione el motor gráfico, el salto entre lod's puede llegar a notarse por el usuario.



11 Modelo con diferentes niveles de detalle (LOD's) aplicados

Texturizado

Base Color (sRGB)

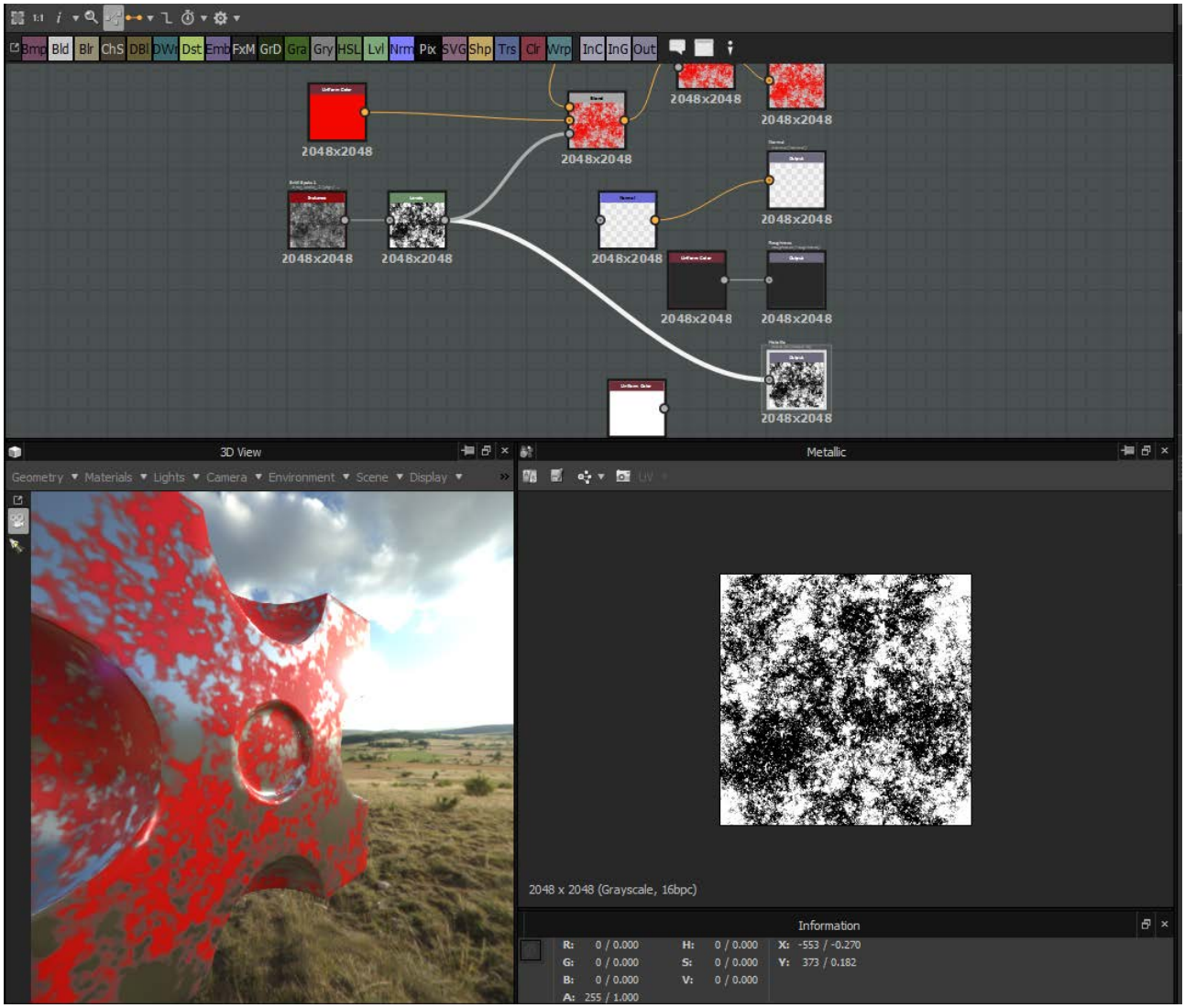
Es el mapa que básicamente contiene la textura del objeto sin ninguna característica física, es decir no debe incluir detalles como sombras, *ambient occlusion*, brillos etc. (si es que buscamos un resultado realista). También será donde pintaremos el color del metal y su reflectividad.

Por norma general, en los dieléctricos los colores oscuros no bajarán de 30 - 50 sRGB y los colores claros no superarán de 240 sRGB. En cambio, la reflectancia de los metales (recordar que no tienen un color difuso propiamente dicho) oscila entre 180 - 255 sRGB

Metallic Map (Lineal)

Dependiendo de si queremos que el material sea metálico o no, dibujaremos en este mapa las partes metálicas con blanco y las no metálicas con negro. Los intermedios de gris no suelen dar resultados esperados y por lo tanto es recomendable definir los materiales como puramente metálicos o puramente dieléctricos.

Normalmente si queremos jugar solamente con el *metallic map*, los valores están definidos entre 235 sRGB y 255 SRGB, en el caso que sea necesario definir una transición mayor (<235 sRGB) tenemos que tener en cuenta que entonces también deberemos modificar la reflectancia definida en el mapa de color base.



12 Utilización de las máscaras procedurales en el canal Metallic.

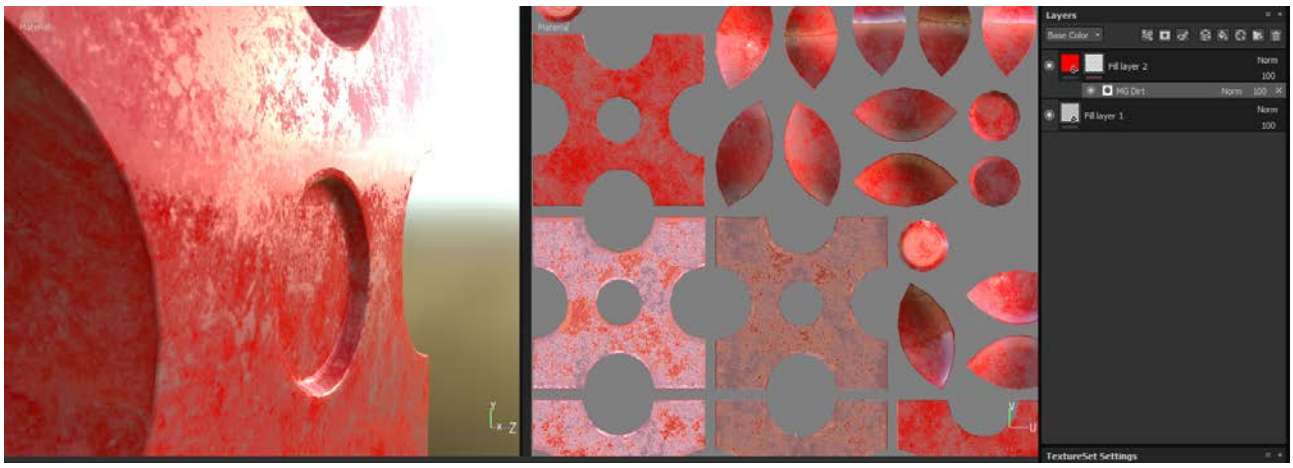
Algunas características del metal como la corrosión, tienen que ser consideradas como un material dieléctrico, por lo tanto, en el mapa metálico lo pintamos en negro. Lo mismo se aplica en los metales que tienen alguna capa de pintura.

Roughness Map (Lineal)

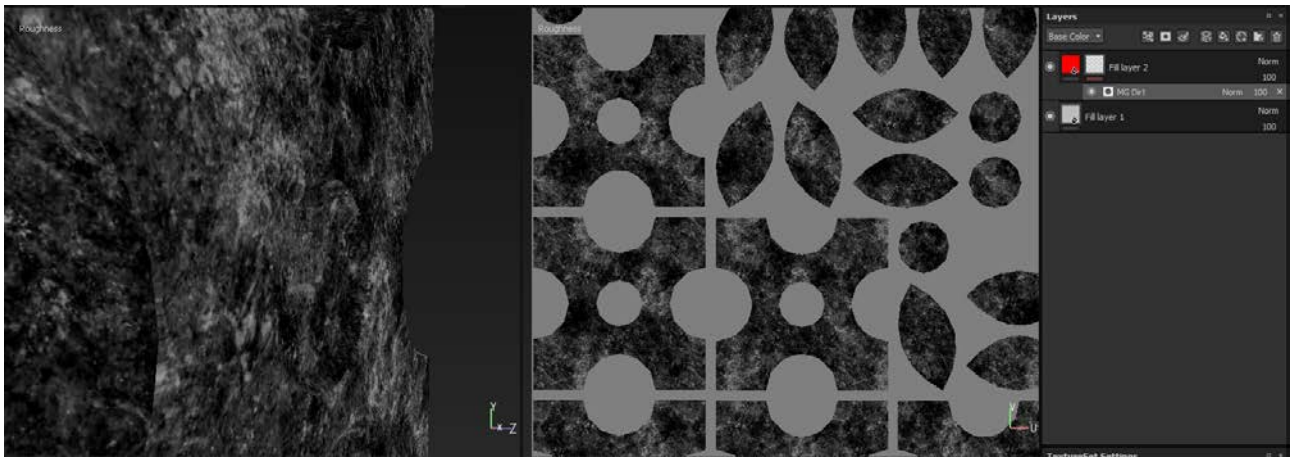
Cuando queremos definir lo pulido que esté un material, lo hacemos mediante el uso del *roughness map*, en él podemos pintar de blanco las partes opacas y las negras las partes pulidas. Las partes pulidas tendrán reflexiones mucho más brillantes y definidas que las opacas. Es el contrario del specular map que veníamos utilizando hasta ahora.

Es uno de los mapas más útiles en términos artísticos ya que nos permiten mostrar la relación del material con el entorno.

Para tener una primera idea de cómo pulida está una superficie, podemos fijarnos en el mapa de normales, ya que este nos muestra el relieve del material y por lo tanto posibles zonas de desgaste, daño etc…



13 Utilización de máscaras en el canal roughness junto al canal difuso



14 Detalle del canal roughness

A veces, puede producirse un pequeño tinte en los bordes del mapa. Esto se produce por el contraste del mapa roughness con el del color base. Se acentúa sobre todo en los cambios de color bruscos. Para minimizar este error, tenemos que asegurarnos que tenemos suficiente resolución en la textura y que las UV's tienen una densidad acorde al objeto.

Ambient Occlusion

El mapa de oclusión define cuánta luz es accesible en una determinada zona de la superficie. La principal diferencia con los otros mapas es que no afecta a las reflexiones sino solamente al mapa difuso. Por ello no debemos incrustarlo en uno de los anteriores mapas, crearemos un mapa específico para él y así poder controlar su contribución.

Podemos crearlo tanto imprimiendo la información de un *high poly* o generarlo a partir del mapa de normales.

Height Map

Normalmente se usa para generar desplazamiento en conjunto con el *parallax mapping*, esto genera un relieve con más profundidad que el *normal/bump map*, lo que dota de mayor realismo al material.

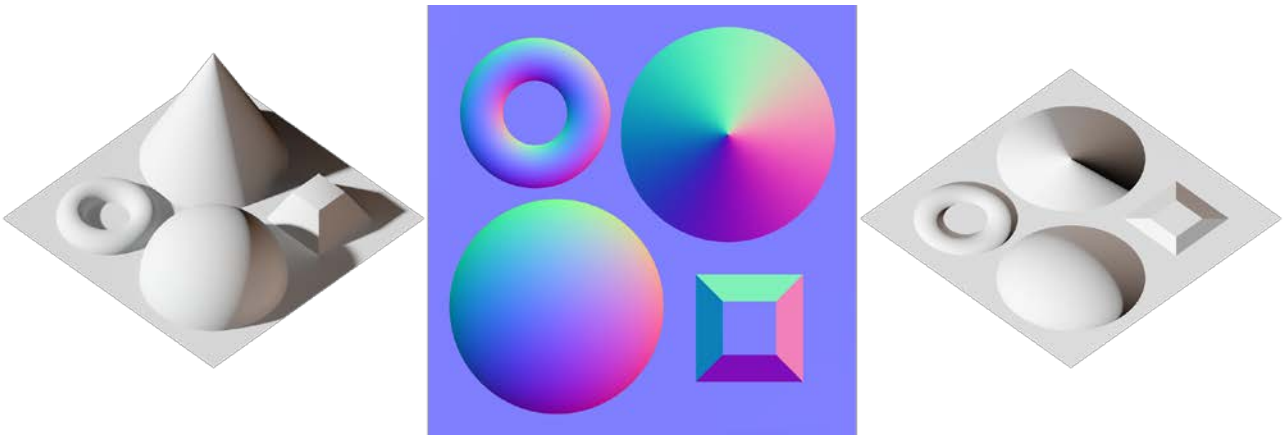
Al igual que el *ambient occlusion*, podemos generarlo o a través del *high poly* o usando el mapa de normales.

Normal Map (Lineal)

Para la realización de los mapas de normales necesitamos un modelo altamente detallado y el otro menos detallado sobre el que "imprimimos" (*baking normal maps*) los detalles.
Es un mapa que nos permite tener mallas con poco poligonaje, pero con el detalle de las de alto poligonaje. Además, tenemos la ventaja que observaremos profundidad ya que está pensado para interactuar con las luces y su dirección.

En la actualidad con el uso de sistema de realidad virtual (VR) están surgiendo problemas con este tipo de mapas ya que el efecto con el que *engañamos* al usuario no funciona. Como alternativa usaremos el *parallax mapping* a costa de necesitar más recursos.

En la siguiente ilustración podemos observar cómo hemos grabado el detalle la imagen de la izquierda en el plano de la derecha con el consiguiente ahorro de poligonaje.

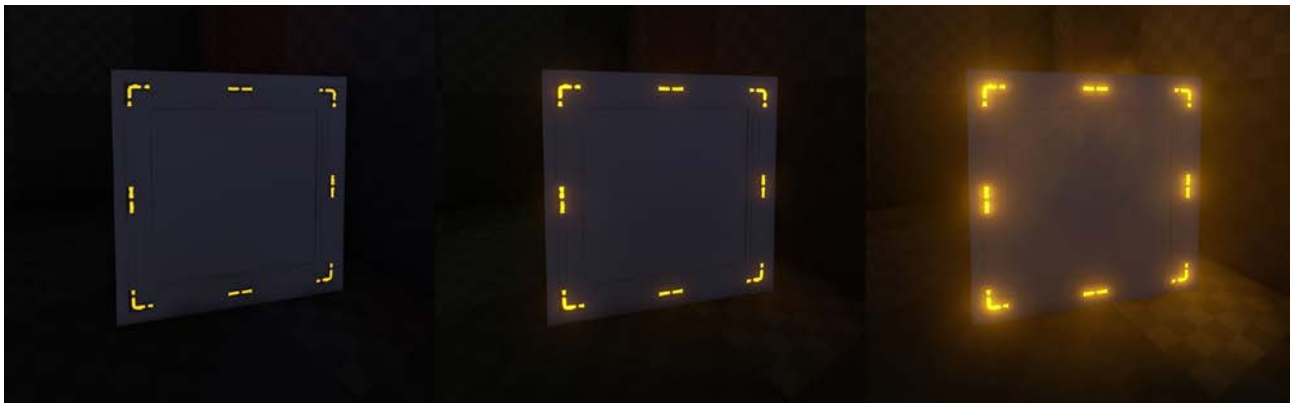


15 Generación de un mapa de normales en base a una geometría ya existente. Wikipedia

Emission Map

Es un mapa que únicamente tiene la función de definir las zonas que emitirán iluminación, en antiguas versiones del motor, estos mapas por si solos no tenían la opción de emitir luz más allá del efecto visual en la cámara. Sin embargo, en las recientes versiones podemos permitir la posibilidad de que el propio mapa sea el que genere la luz como si fuese un foco.

En la siguiente imagen podemos observar los tres estados, el primero define el mapa de emisión, el segundo el mapa de emisión, pero sin generar luz físicamente y el tercero con la opción de generar luz como elemento independiente.



16 Visualización del Emission map- Polycount Website

Texel Density

Cuando se texturiza un objeto, querremos tener en cuenta la relación entre tamaño del objeto / tamaño textura. De nada servirá tener una taza con una textura de 4096 x 4096 si al final esta tendrá un tamaño de no más de 10 cm, y, al contrario, no queremos una textura de una montaña con una resolución de 512 x 512 ya que lo más seguro que se verá desenfocada o pixelada. Por lo tanto, tenemos que tener en cuenta este tipo de relaciones para mantener uniformidad en todos los assets creados.

Una primera referencia desde la que partir podría ser la siguiente:

1 unit = 1 cm
Texel Density: 8 pixels x unit

- 8 units = 8 cm = 64x64
- 16 units =16 cm = 128x128
- 32 units = 32 cm = 256x256
- 64 units = 64 cm = 512x512
- 128 units =128 cm = 1024x1024
- 256 units = 256 cm = 2048x2048
- 512 units = 512 cm = 4096x4096

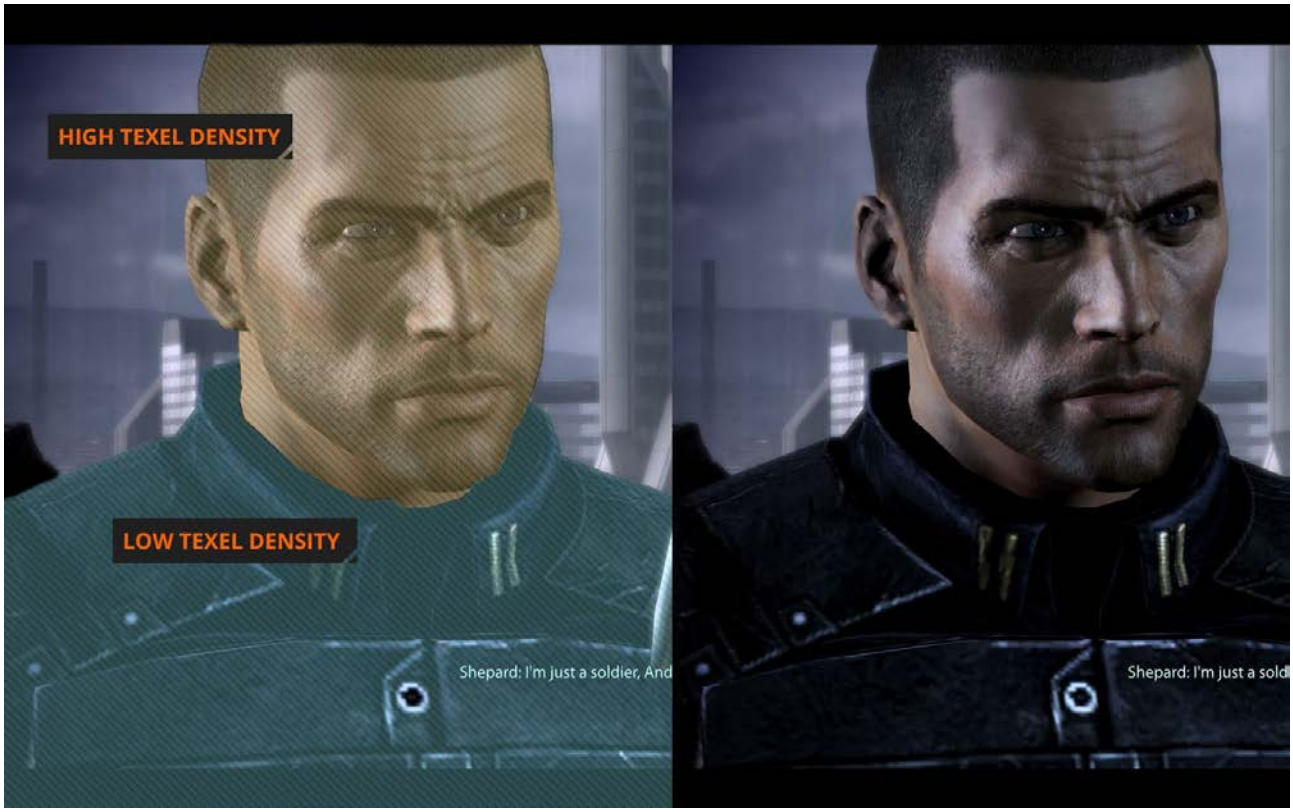
Dicho esto, hay que mencionar que, si bien sigue siendo utilizado bajo ciertas circunstancias, hoy en día, unreal engine 4 escala automáticamente las texturas dependiendo del *texel density* que necesite, es decir, si creamos todos los assets con una resolución de 4096 x 4096, UE4 cuando

detecte que dicho asset tendrá un tamaño pequeño, automáticamente escalará la textura a una resolución más pequeña.

De qué resolución partir es uno de los aspectos que dependen del tipo de juego que hagamos y las necesidades y limitaciones que tengamos. No es lo mismo trabajar con texturas 4k (aunque se reescalen) que partir ya con texturas 2k.

En algunos casos, cuando realizamos este proceso, tenemos que tener en cuenta que partes del modelo sacrificaremos en favor de otras. Por ejemplo, si modelamos un arma que se verá en primera persona, podemos dar prioridad a las texturas que irán en la parte visible (mirilla, cargador, gatillo) y quitarles espacio a aquellas que estén fuera de la vista

En la siguiente captura del juego Mass Effect, podemos observar cómo se priorizan algunas partes de modelo sobre otras, en este caso, observamos que la cara tiene mayor densidad de texels que el traje. Es una decisión que casi seguro será visible por los usuarios, pero en ciertos casos artísticamente será necesaria.



17 Ejemplo de uso del Texel Density en el juego Mass Effect

Light Mapping

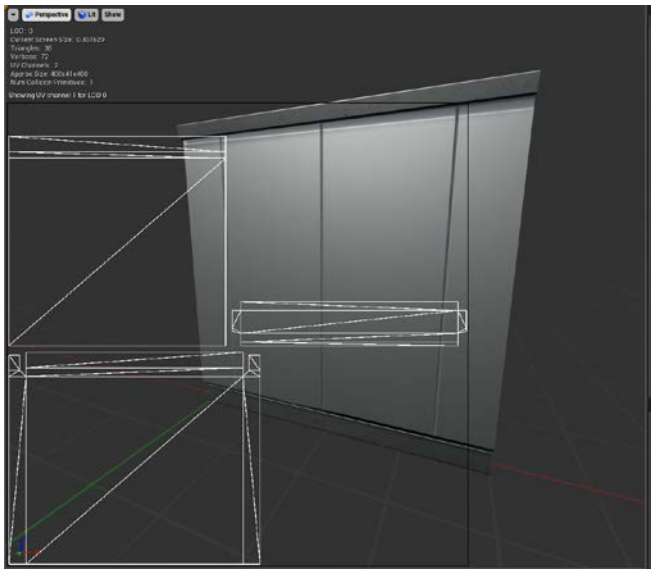
El mapa de luces (*LightMap*) es donde se guarda la información lumínica del objeto, es decir, las sombras, luces, colores que adopta al ser iluminado por una fuente lumínica, etc... Debido a esto, es importante entender su funcionamiento ya que de ello depende la visualización correcta o no del objeto. A diferencia del mapa de texturas, el mapa de luces es bastante más pequeño y requiere un espacio mayor entre celdas (*UV Shells*) para evitar sobreimpresiones (*bleedings*).

La utilización de este tipo de mapas permite ahorrar recursos de procesamiento, ya que, al estar pre calculado, durante la ejecución el motor de render se limitará a mostrar el mapa sin necesidad de calcular nada más. Otra de las ventajas es la posibilidad de usar técnicas como la iluminación global y dotar así de mayor realismo a la escena.

A la hora de crear los mapas de luces, existen dos opciones, la primera sería dejar que el propio motor gráfico lo genere automáticamente, o, por el contrario, crearlo manualmente en un segundo canal UV dentro de nuestro programa de modelado. Dependiendo de la complejidad del objeto, la opción automática puede ser suficiente, aunque para otros casos es recomendable hacerlo manualmente y evitar así problemas como el *bleeding* o la incorrecta iluminación de los bordes o caras del objeto.

Uno de los requerimientos que se tienen que seguir es el de evitar que los uv shells se solapen ya que provocará errores en la interpretación de la información lumínica. También tendremos que dejar un margen mínimo de 2 px entre celdas para evitar el mismo fallo.

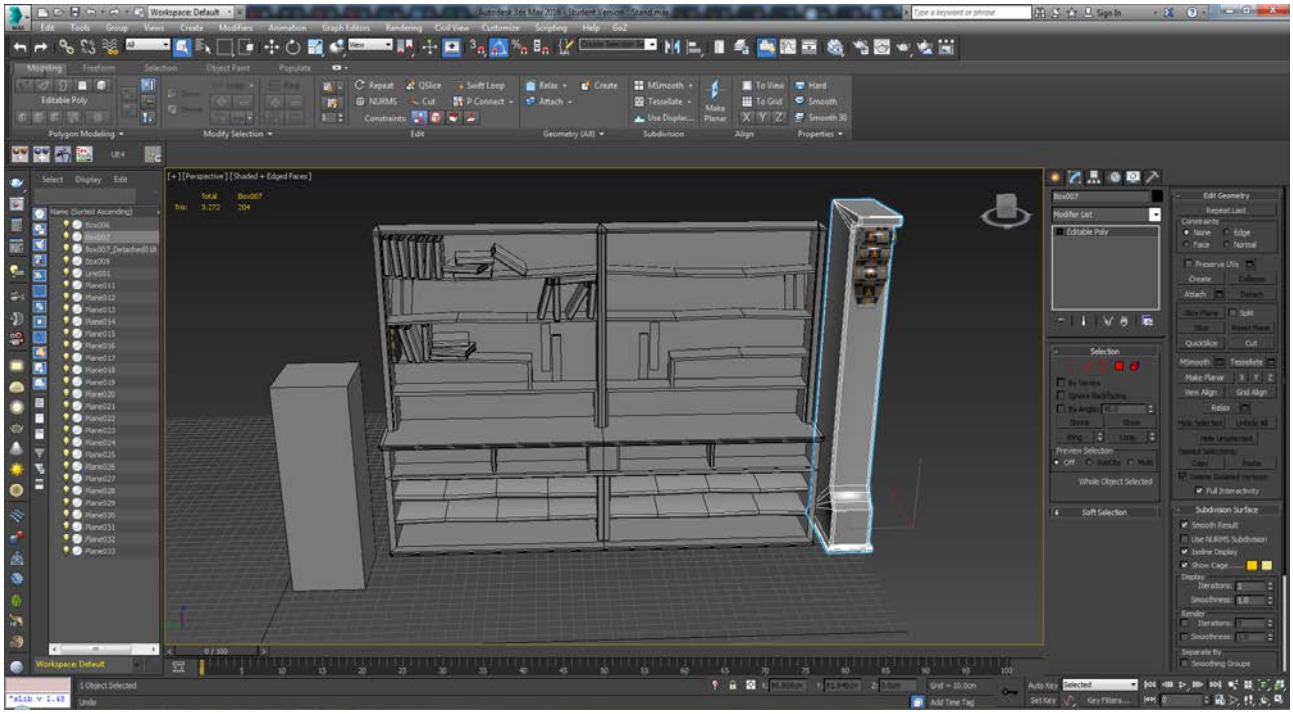
18 Aspecto del mapa de luces dentro de UE4.



19 Segundo mapa UV en un modelo.

3ds Max

3ds Max es un programa de modelado/texturizado/renderizado 3d desarrollado por Autodesk. Tiene una gran aceptación por la comunidad y el ámbito profesional, por lo que hay gran cantidad de plugins y scripts que facilitan muchas tareas.



20 Captura de la interfaz del programa 3dsMax.

Flujo de trabajo

Hay dos formas principales de trabajar, generando inicialmente la versión *low poly* o importarla de otro programa y hacer la retopología en 3ds max.

Generar low poly:

- Modelamos el objeto low poly en 3ds max.
- Hacemos el unwrap del modelo
- Exportamos el modelo a Zbrush y esculpimos la versión *high poly*.
- Importamos el modelo *high poly* y lo optimizamos para volverlo a exportar a Xnormal

Importar high poly:

- Importamos el modelo high poly de Zbrush o similar.
- Hacemos la retopología en 3ds max.
- Unwrapeamos el modelo *low poly*.
- Exportamos las dos versiones a Xnormal.

Que flujo utilizar dependerá de nuestras necesidades y el modelo. En ocasiones incluso podemos combinar hasta un punto ambas metodologías. Por ejemplo, podemos crear un modelo base (sin el detalle incluso de la versión low poly) en 3ds max para luego importarlo a Zbrush para darle el detalle que queramos. A partir de ahí seguimos con flujo de trabajo de importar *high poly*.

Zbrush

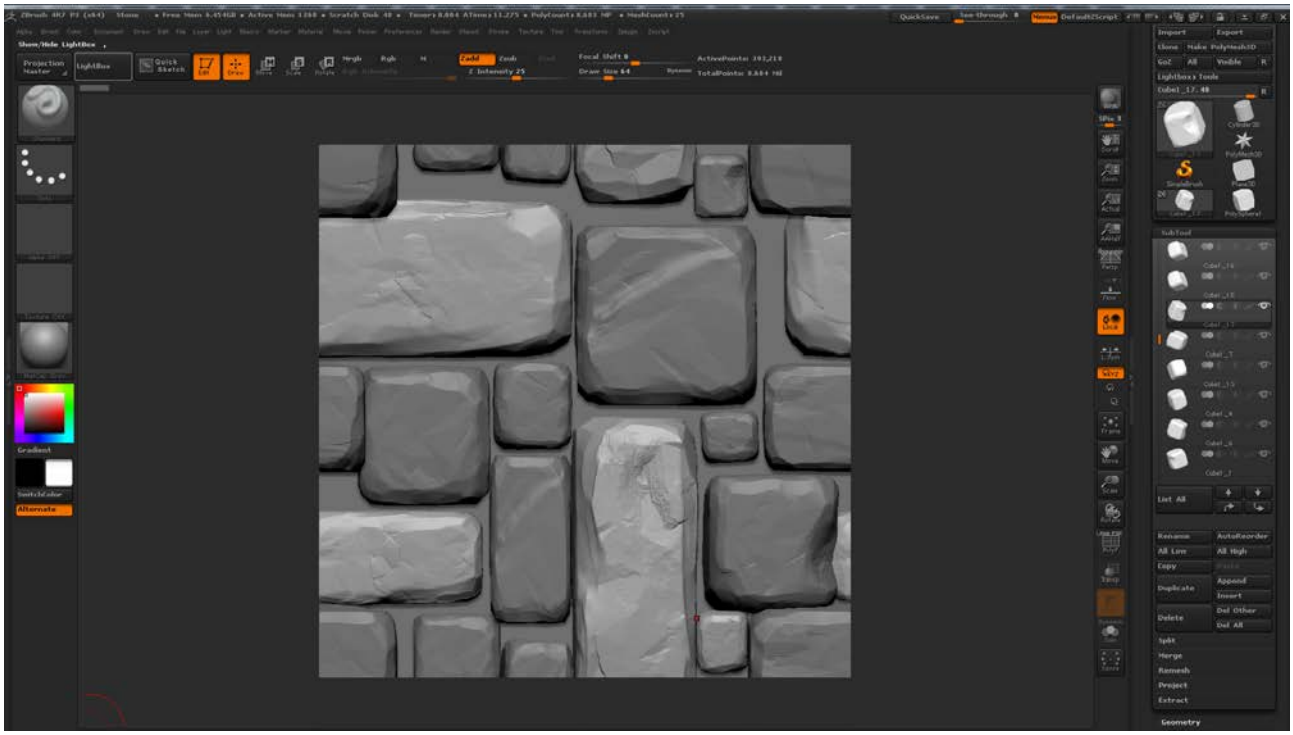
Uno de los principales programas que he utilizado para generar geometrías detalladas ha sido Zbrush.

Es un programa que permite trabajar con millones de polígonos sin apenas pérdida de rendimiento. Aunque inicialmente se pensó como herramienta de dibujo con ayudas 3D, en los últimos años se ha posicionado como el programa para esculpir por excelencia. Autodesk dispone de su propio software llamado Mudbox, pero actualmente carece de algunas de las herramientas imprescindibles que si dispone Zbrush.

Flujo de trabajo

- Importamos o creamos la primitiva 3D.
- Subdividimos la primitiva o modelo hasta conseguir una malla con el detalle mínimo necesario para esculpir. A medida que necesitemos más detalle iremos incrementando la subdivisión.
- Para esculpir nuestro modelo tendremos a la disposición gran variedad de pinceles, mapas de opacidad y herramientas para crear nuestros detalles.
- Una vez estemos contentos con el resultado dependiendo de nuestras intenciones podemos o bien exportar directamente la malla de alto poligonaje a nuestro programa de generación de normales preferido, o por el contrario reducir el poligonaje usando Zdecimator y generar las normales directamente en zbrush.
- Si optamos por generarlas en Zbrush, al final obtendremos un modelo low poly y su respectivo mapa de normales, a continuación, ya estaríamos listos para el siguiente paso, texturizar.

En la siguiente captura se muestra el resultado de esculpir y tilear (sin bordes ni costuras) unas piedras que posteriormente utilizaré para el suelo de mi escenario.



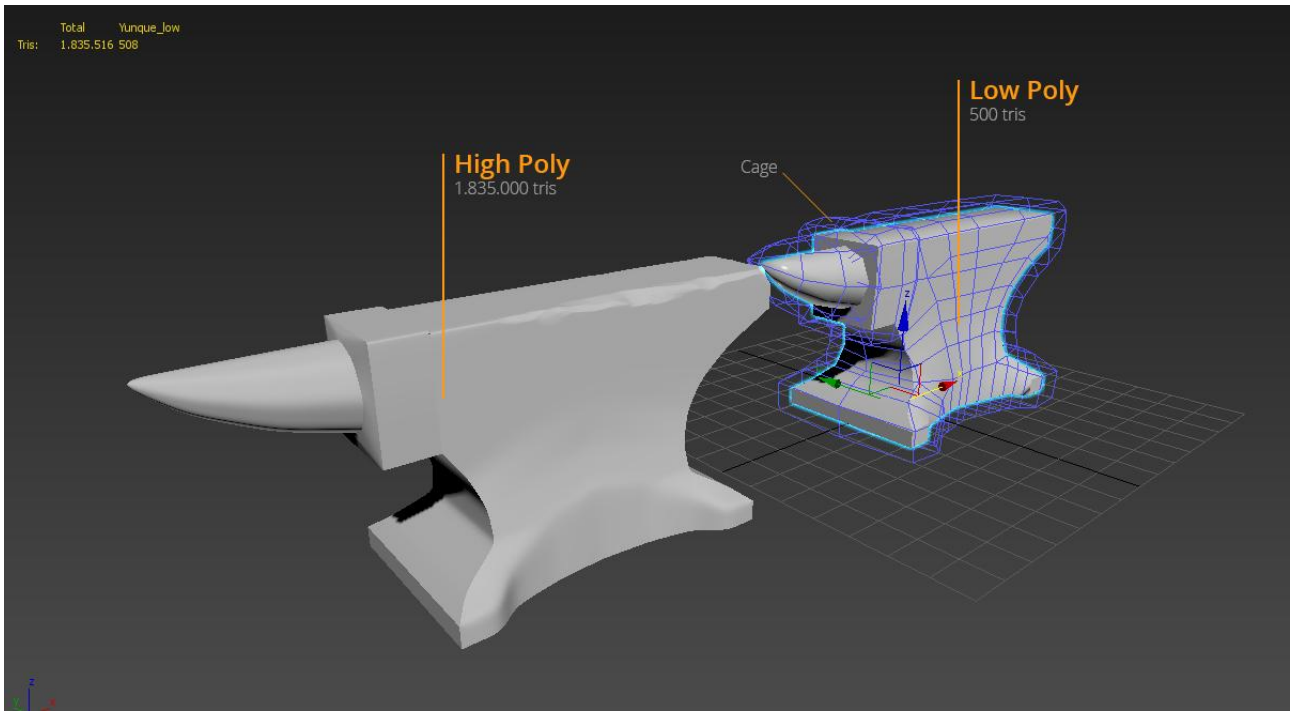
21 Captura del programa Zbrush

Xnormal

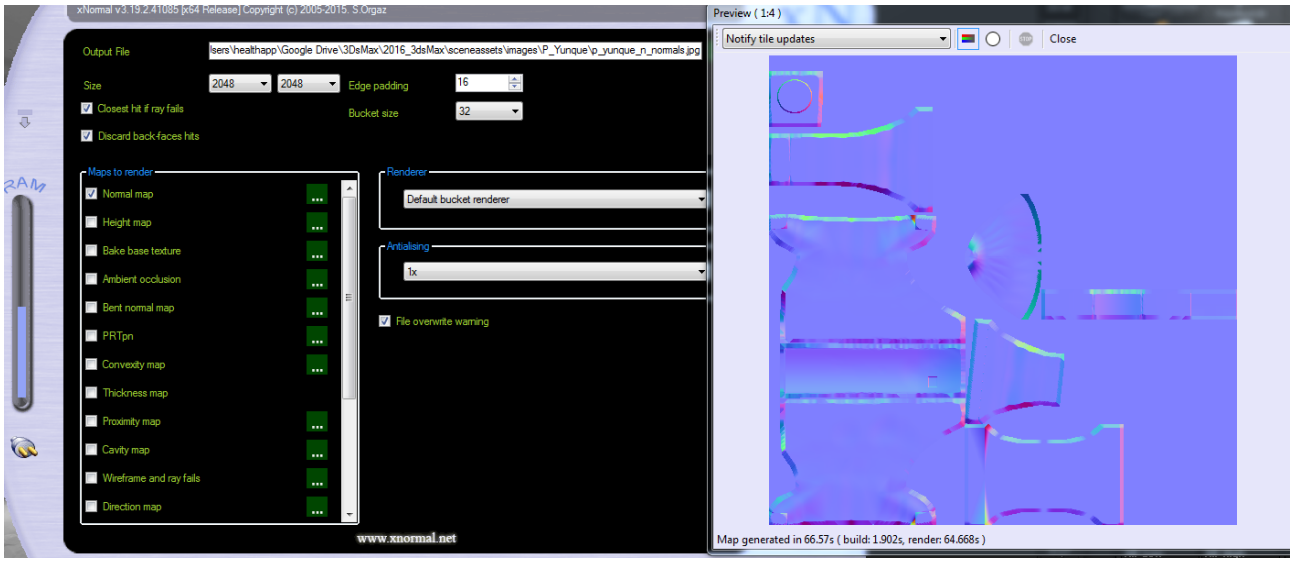
Para elementos más complicados en los que haga la retopo en 3dsMax normalmente Xnormal es la opción que utilizo para generar las normales. La ventaja de usar este programa es la calidad que ofrece sin la necesidad de complicadas configuraciones.

Flujo de trabajo

- Importamos tanto el modelo low poly como el high poly en 3ds Max.
- Al modelo low poly le añadimos el modificador Projection, que básicamente como indica el nombre, permite que el modelo high poly se proyecte sobre el modelo low poly.
- Para delimitar dónde empieza y termina los bordes de los modelos, utilizamos una *cage*.
- Exportamos a Xnormal.
- En Xnormal definimos las dos mallas *low* y *high*.
- Generamos los mapas que queramos.



22 Comparación del modelo Low Poly y High Poly



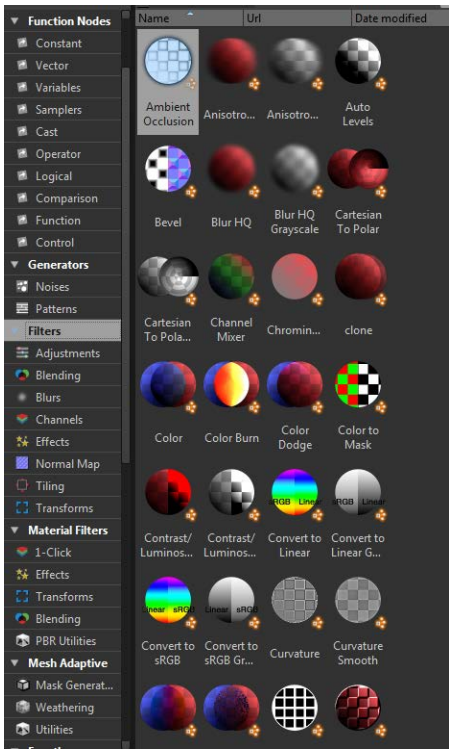
23 Interfaz del programa xNormal y ejemplo de generación del mapa de normales en base a un modelo High Poly

Substance Designer 2

Substance Designer 2 es un programa desarrollado por Allegorithmic que permite generar mapas procedurales a partir de nodos. Permite ir iterando nuestras texturas hasta encontrar el resultado que deseamos y mostrarlo casi sin esfuerzo directamente en el UE4 gracias a los plugins que incorpora.

Nodos

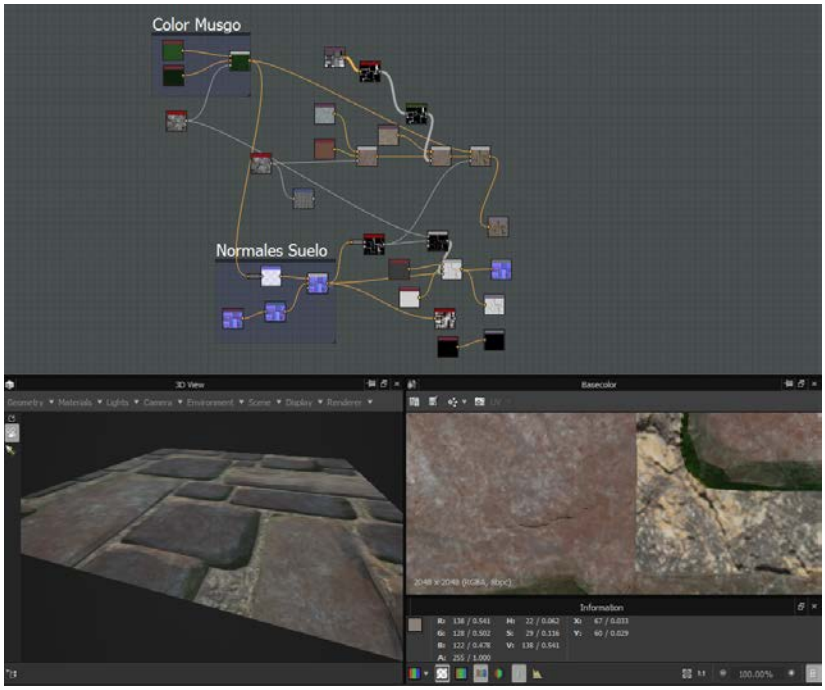
Este es el sistema en el cual se basa el programa, disponemos de multitud de nodos con sus funciones específicas, desde simples niveles a generador de rayones en base al mapa de curvatura.



24 Nodos del programa Substance Designer

Ejemplo

En la siguiente captura podemos ver como he generado el mapa del suelo a partir del modelo importado.



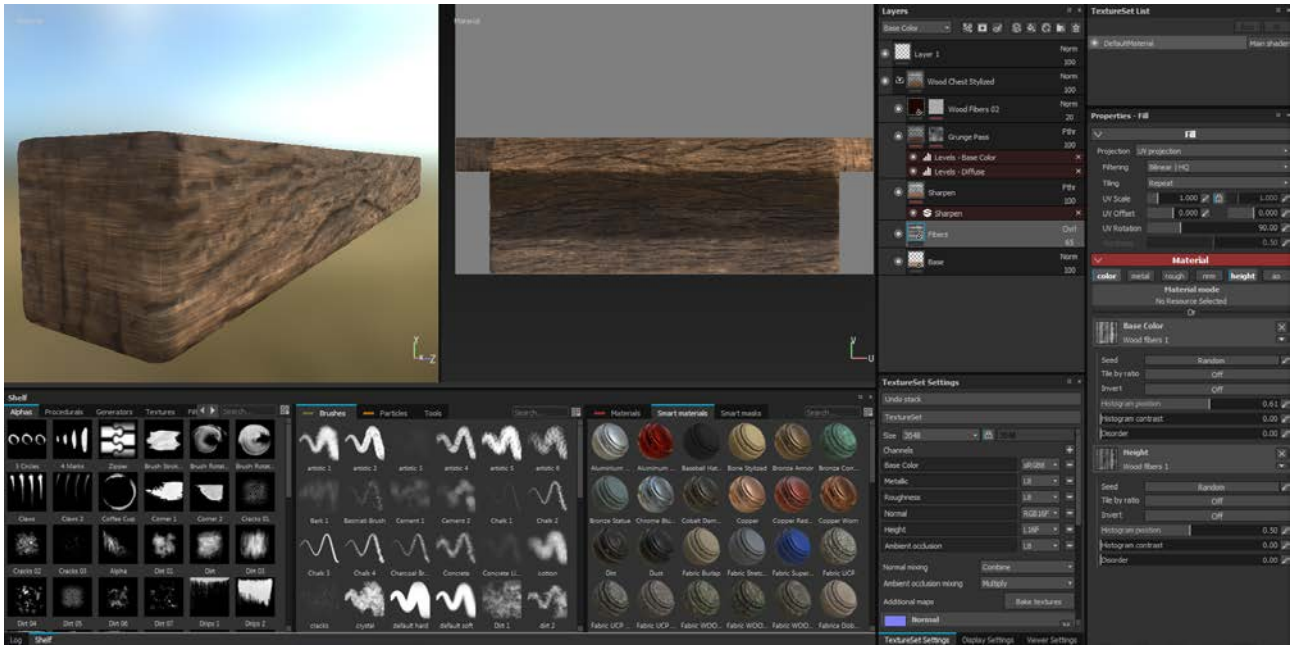
25 Captura de los nodos generados para crear la textura.

Substance Painter 2

Substance Painter 2 al igual que el Designer, está desarrollado por Allegorithmic y permite pintar directamente sobre el modelo 3D que importemos, además, al ser de la misma compañía dispone de gran compatibilidad con su programa hermano, permitiendo así generar texturas en uno y definir en qué zonas se mostrarán en el segundo.

Flujo de trabajo

- Importar modelo *low poly* al programa.
- Importar/generar los mapas de normales.
- Generar los demás mapas (Oclusión, curvatura, *world space tangent*...).
- Definir materiales para los diferentes elementos/zonas del modelo. Si hemos creado los materiales en Substance Designer, podríamos importarlos también.
- Exportar texturas.

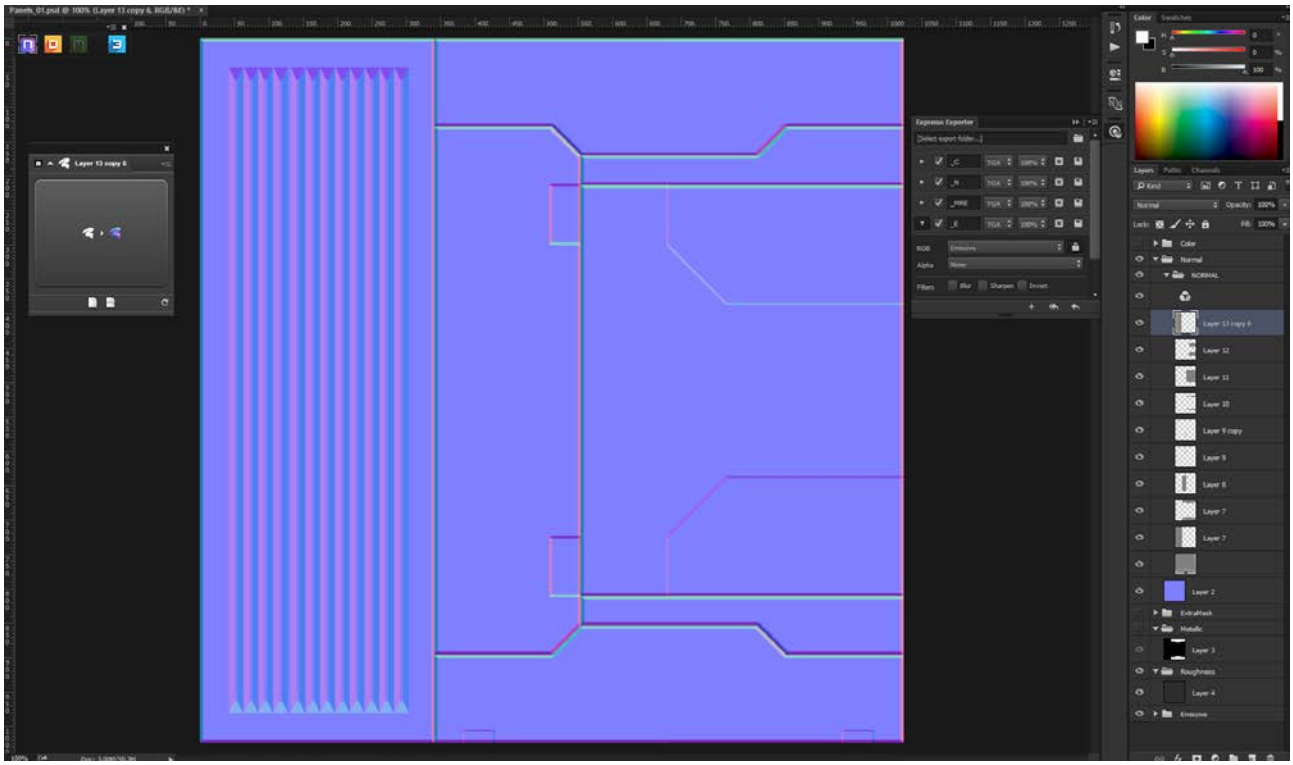


26 Interfaz del Substance Painter 2

Quixel Suite

Para el manejo y creación de normales he utilizado NDO (Quixel), que es un conjunto de herramientas para generar estos mapas utilizando Photoshop.

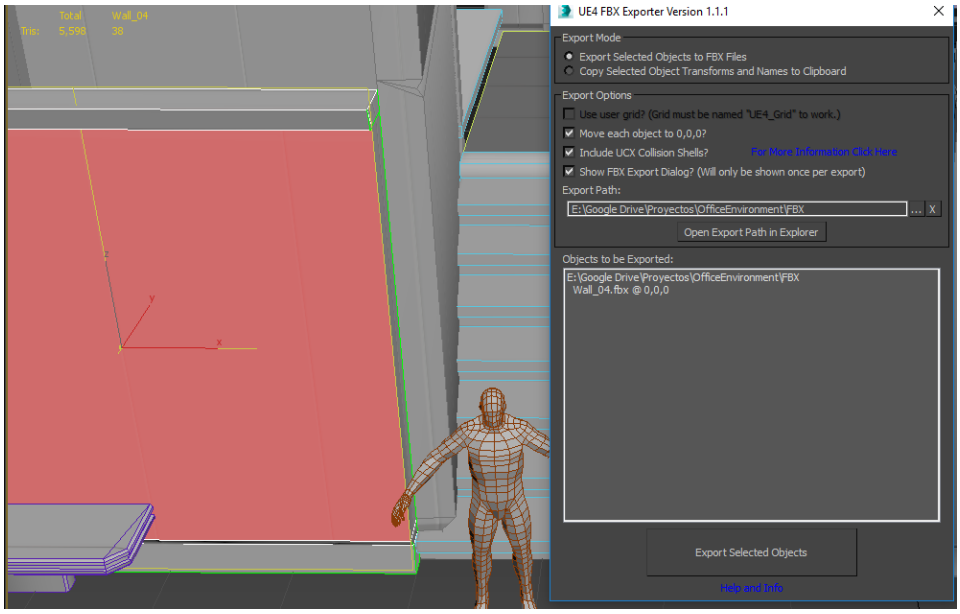
Realmente es un programa muy útil ya que, dependiendo de la complejidad del modelo, nos puede llegar a bastar simplemente con NDO.



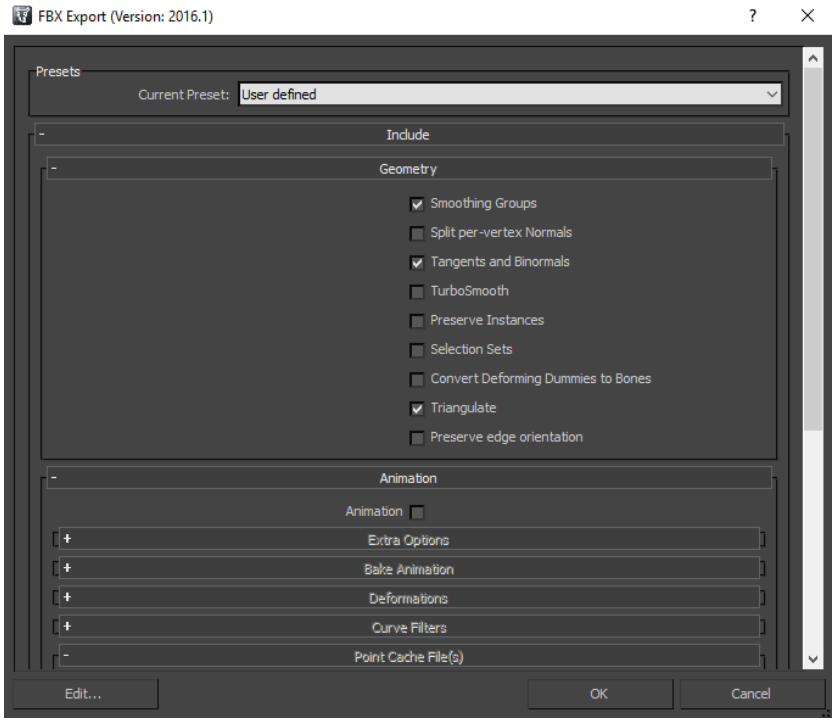
27 NDO en Photoshop

Importar a UE4

Importar modelos a UE4 es un proceso bastante fluido e intuitivo. Simplemente tenemos que seguir unos pasos específicos para automatizar el proceso.



28 Script para exportar modelos de 3DSMax a UE4

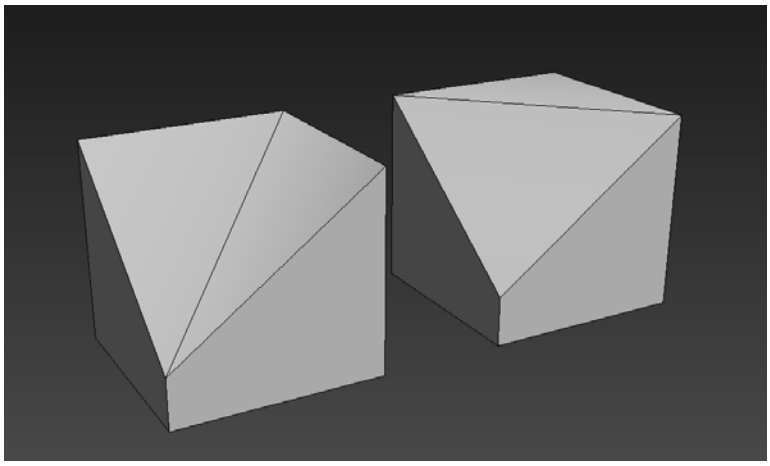


29 Configuración del formato FBX

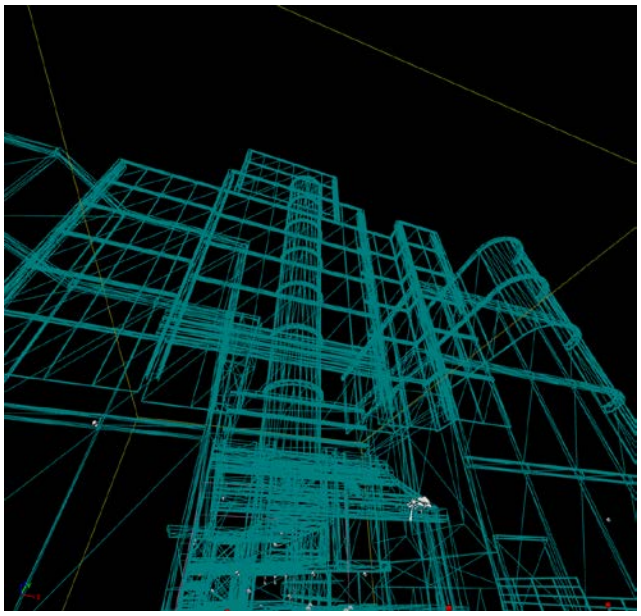
Triangulación

Todos los modelos que exportamos al final serán triangulados por lo tanto habrá que tener en cuenta de qué manera se realiza y reajustarlo si es necesario.

Como podemos ver, una cara de cuatro lados se puede triangular de dos maneras, dando cada una a una forma diferente.



30 Ejemplo de diferente triangulación en un mismo modelo.



31 Triangulación de los modelos dentro de UE4

Collision Box

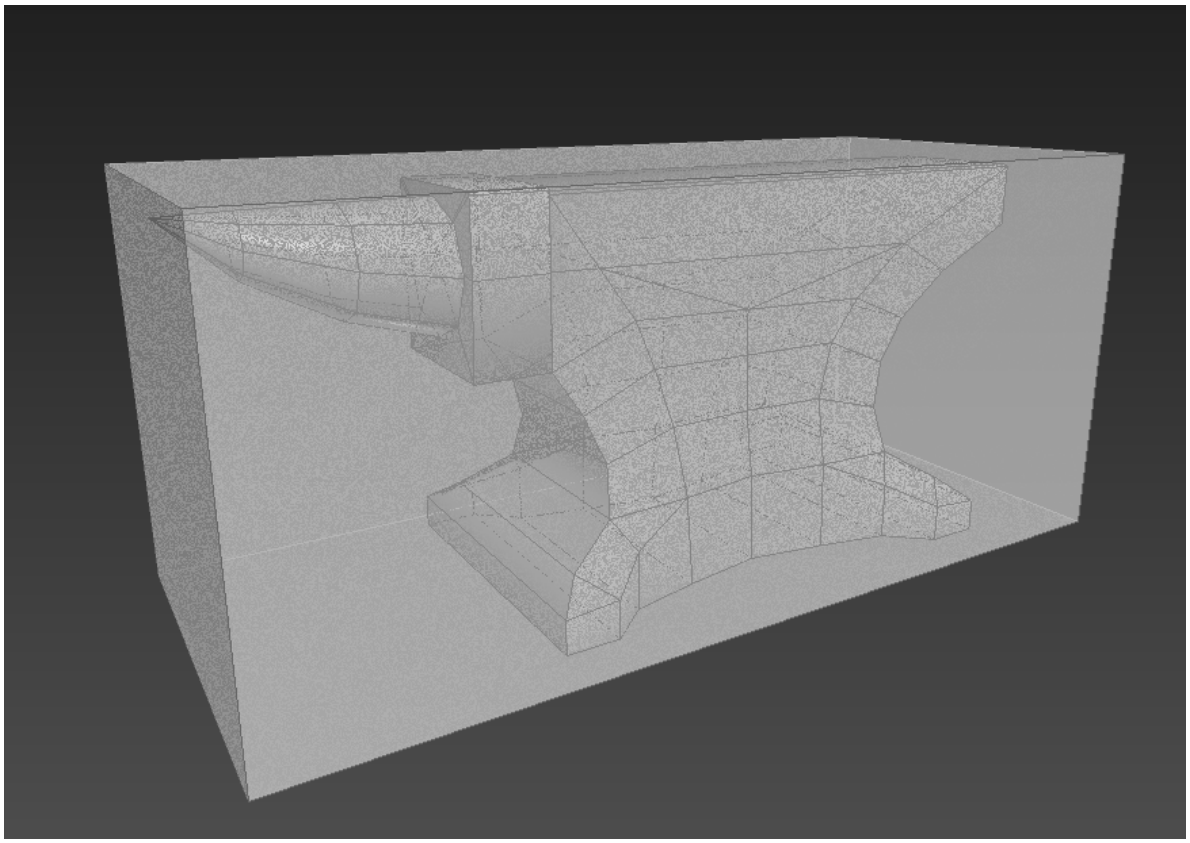
Las colisiones en UE4 son un elemento importante del desarrollo del nivel, permite definir y limitar el espacio de interacción. Tenemos dos maneras de realizar este proceso, o bien realizarlo dentro de UE4 o, por el contrario, y es lo recomendable, definirlo manualmente dentro de la aplicación de modelado 3D que utilicemos. De esta manera nos aseguraremos que el resultado estará optimizado y evitaremos mallas innecesariamente complicadas.

El *naming* que deberemos seguir según el tipo de *collision box* que queramos será el siguiente:

UBX_[MeshName]_##
Lo utilizaremos cuando nuestra *collision box* sea una caja o un prisma rectangular, es importante no modificar los vértices o de lo contrario no funcionará.

USP_[MeshName]_##
Lo utilizaremos cuando nuestra *collision box* sea una esfera, como en el caso anterior no podremos modificar los vértices.

UCX_[MeshName]_##
En el caso que necesitemos mover algún vértice o simplemente las formas primitivas no nos sirvan, deberemos utilizar esta nomenclatura. Ya que, aunque tendrá un coste computacional algo más elevado (imperceptible en la mayoría de casos), podremos adaptar mejor el *collision box*.



32 Collision box de un modelo.

Sockets

En alguna ocasión nos podemos encontrar con la necesidad de pegar algún elemento en alguna parte de la malla, esto se puede lograr mediante la utilización de sockets, que son simplemente unos puntos de coordenadas definidos previamente. Un ejemplo de su uso sería cuando movemos un arma a la mano del personaje.
La nomenclatura que se tiene que seguir es la siguiente:

SOCKET_[MeshName]_##

Formato

El formato habitual para importar y exportar objetos es el FBX ya que nos permite incluir todo lo anterior mencionado sin prácticamente ninguna acción del usuario.

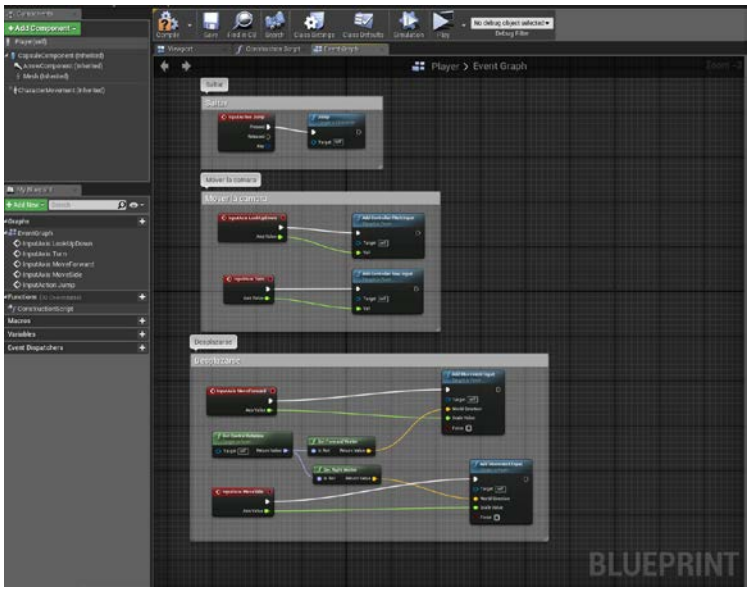
Unreal Engine 4

Historia

Desarrollado en 1998 con el juego *Unreal* tuvo como principales bazas: un motor de renderizado, detección de colisiones, inteligencia artificial (IA), *networking*, *scripting* entre otras. A lo largo de los años nuevas versiones con más potencia y mejoras han ido apareciendo hasta la actual versión UE4. Una de las razones del porqué ha tenido éxito es por su facilidad para generar mods, permitiendo así a la comunidad de modders generar más contenido e incluso nuevas franquicias.

Blueprints

Uno de los cambios más significativos es la introducción de los *Blueprints (anteriormente Kismet)* y la eliminación del *UnrealScript.* Haciendo un resumen rápido, sería una programación basada en nodos y funciones predefinidas que, interconectadas entre sí, permiten realizar gran variedad de acciones, llegando a estar a la par con el código directamente escrito en cuanto a funcionalidad, aunque con el principal inconveniente de tener menor rendimiento (hasta 10 veces menos rendimiento).



33 Ejemplo de Blueprint para controlar la cámara del jugador dentro del juego.

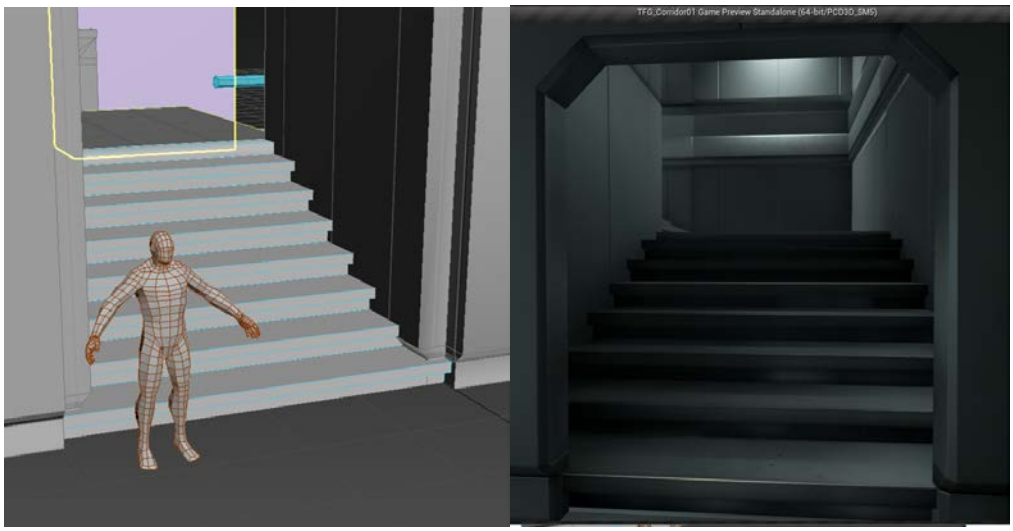
Esto a priori puede parecer una seria limitación, pero la facilidad para generar funcionalidades sin necesidad de tener conocimiento de programación permite prototipar e incluso desarrollar aplicaciones y juegos en relativamente poco tiempo.

Igualmente, si se desea, es posible programar en C++ cualquier función que se requiera, incluso editar el motor gráfico mismo ya que es de código abierto.

Escala

Un aspecto que puede parecer poco relevante en un principio es la escala que se percibe al ir montando el nivel. Lo que puede parecer correcto en 3dsMax, al importarlo y testearlo en UE4 nos damos cuenta que no percibimos la escala de igual manera. Fue uno de los errores que más tardé en darme cuenta y que por suerte tiene solución, aunque la tarea de escalar es bastante tediosa.

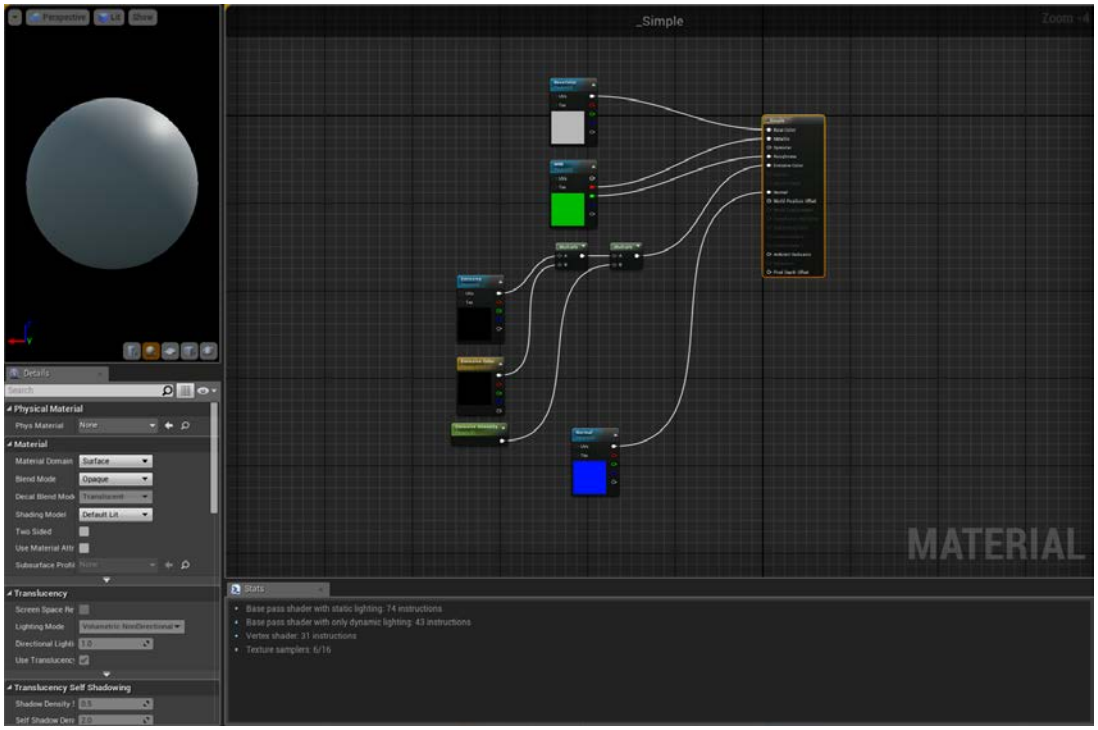
Por ello el ir testeando continuamente el nivel a ras de suelo (Personaje / Cámara / Recorrido) es una práctica muy recomendable y casi obligatoria, ya que nos permitirá darnos cuenta de errores en los primeros estados de desarrollo.



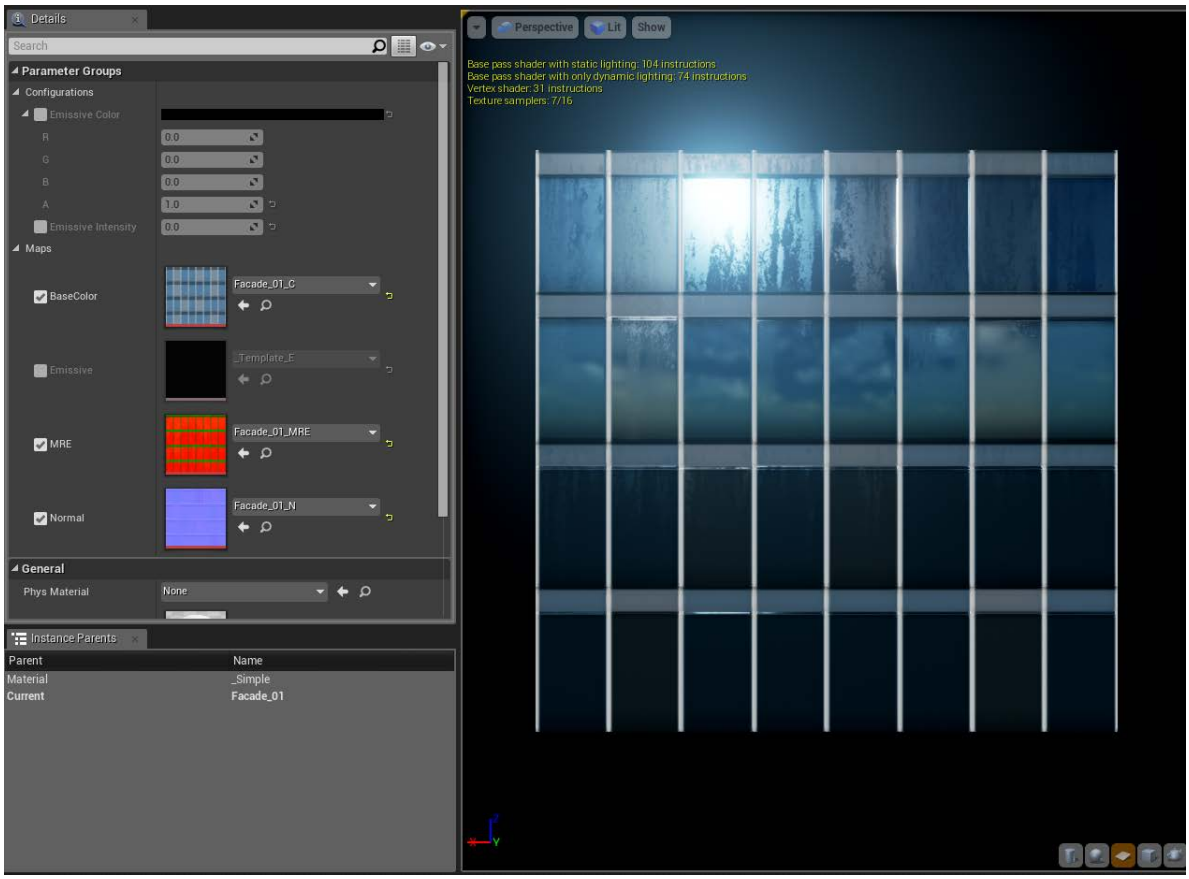
34 Representación del mismo modelo y misma escala en 3dsMax y Unreal Engine 4

Material Editor

UE4 trae consigo un potente editor de materiales basado en nodos, en el podremos crear casi todos los materiales que necesitemos e inclusive programar los shaders si es así como lo deseamos.



35 Editor de materiales de UE4



36 Material Instance

Estructura básica de una textura

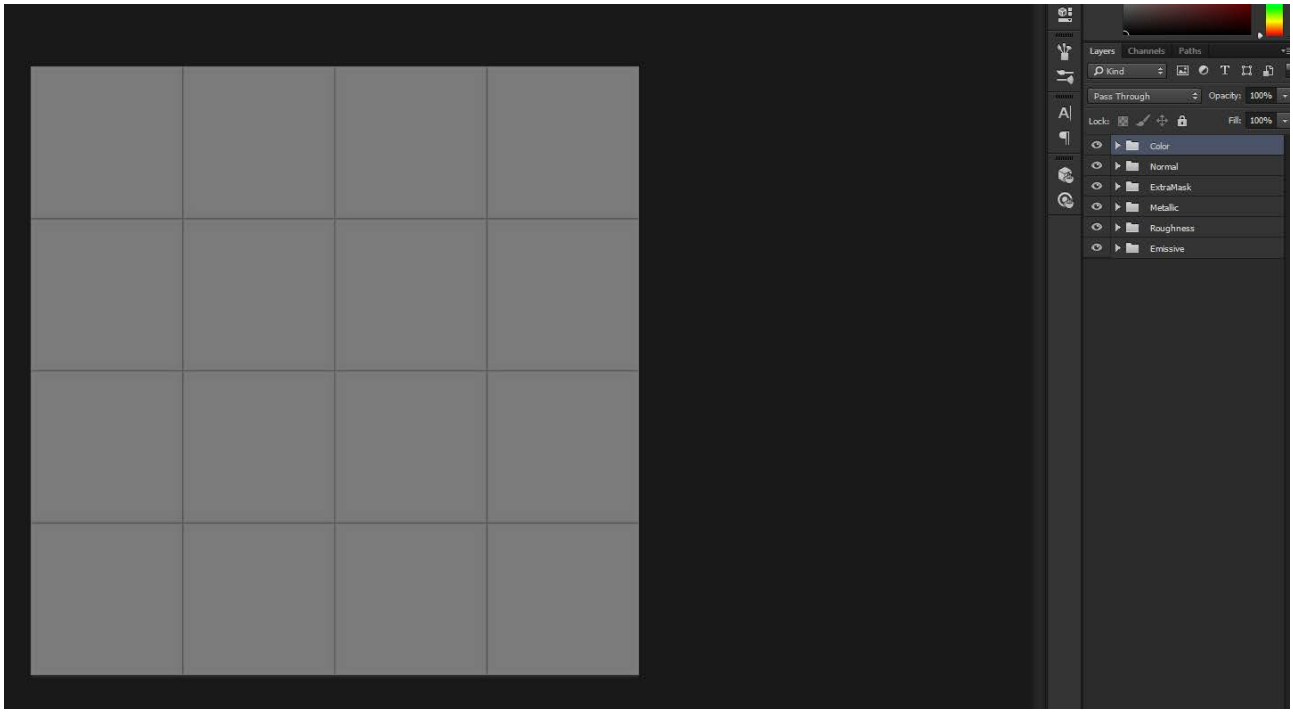
Cuando creamos una textura para UE4 tenemos que tener en cuenta que hay que tratar de optimizarla al máximo, ya que cada nueva textura es un nuevo *drawcall* que mandamos al motor gráfico. Por ello trataremos de combinar canales en la medida de lo posible.

La estructura básica que sigo para crear una textura es:

- Crear un archivo Photoshop con la resolución deseada pero que sea potencias de 2 (128x128, 256x256, 512x512, 1024x1024 ...) ya que si seguimos estos patrones después el motor gráfico es capaz de comprimir las texturas.
- Crear carpetas con los diferentes mapas que necesitaremos
    - Base Color
    - Normal
    - Metallic
    - Roughness

- ExtraMask
- Emissive

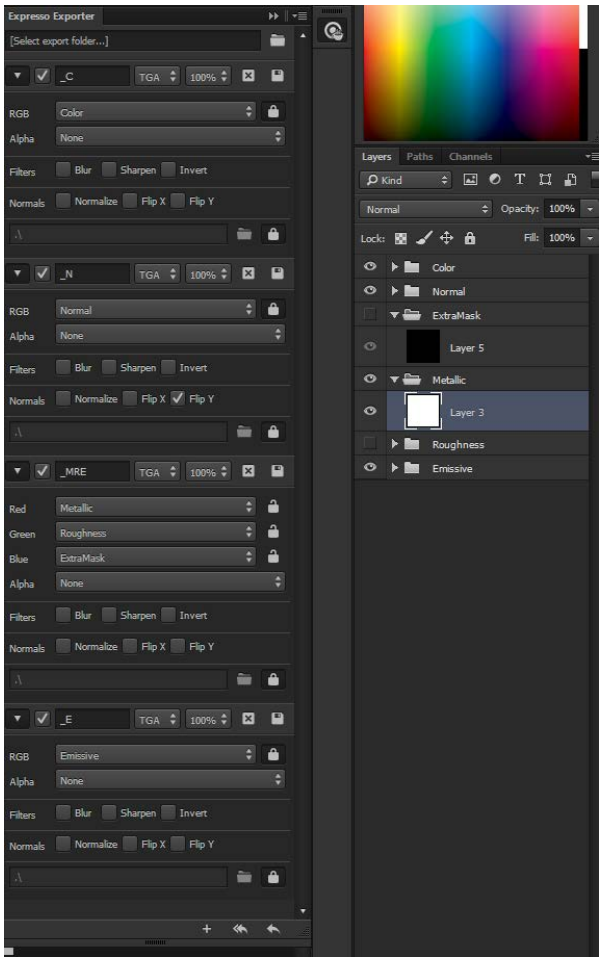
Una vez lo básico esté realizado, dependiendo de si vamos a crear la textura en base a un modelo o libre, iremos llenando las carpetas con la información adecuada.



37 Creación de la textura del suelo en Adobe Photoshop CC

Una vez completado este paso, entra en juego un plugin muy útil de Photoshop llamado *Expresso!* (http://minifloppy.it/expresso/), que nos permite realizar la combinación de canales y exportación de los mapas de una manera muy sencilla y rápida. De esta manera el proceso de ir iterando y modificando las texturas nos lleva no más de un par de clics.

Cabe decir, pero, que al ser un plugin que lleva cierto tiempo sin actualizarse, algunas veces no es tan estable como me gustaría que fuese.

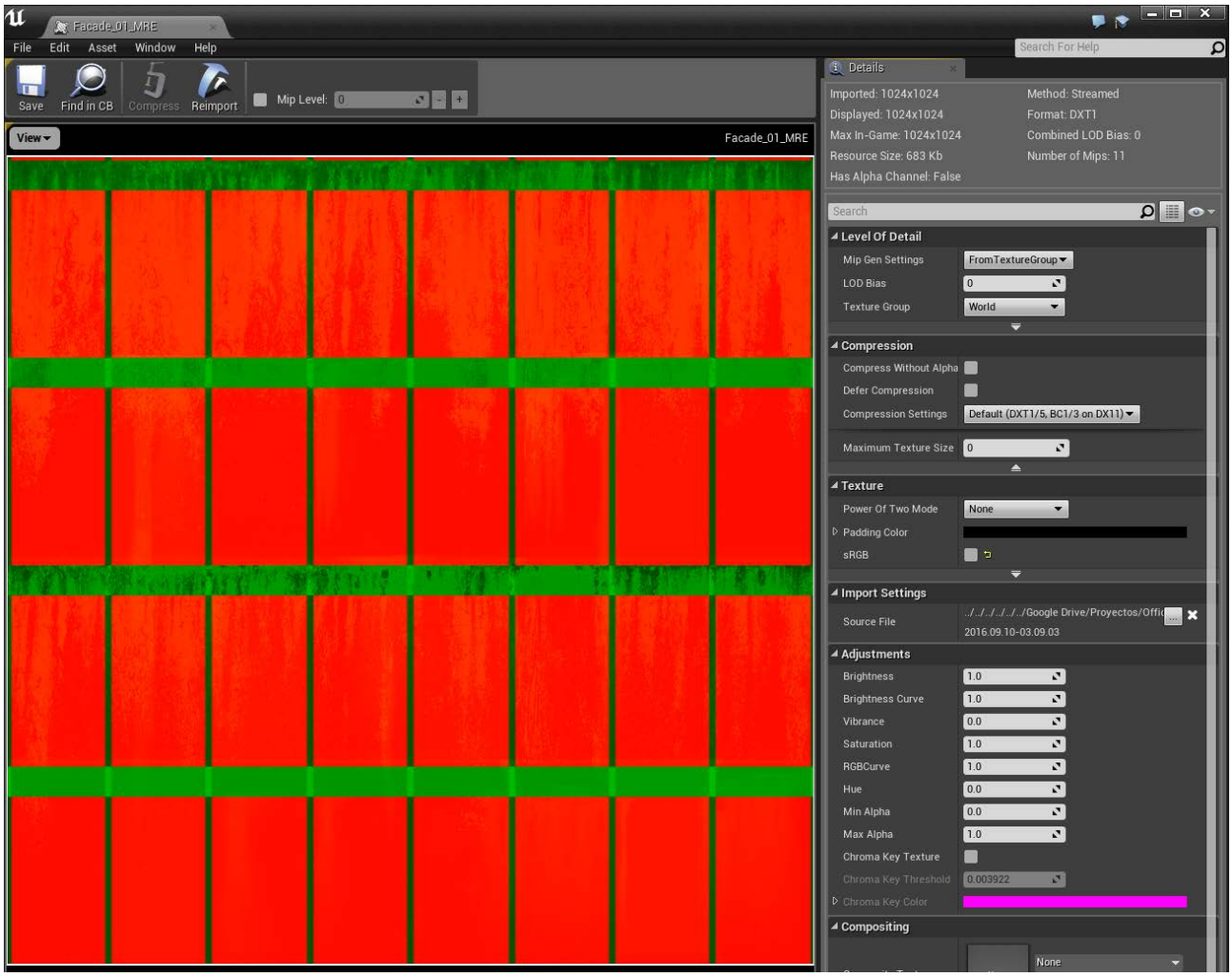


38 Interfaz del plugin Expresso!

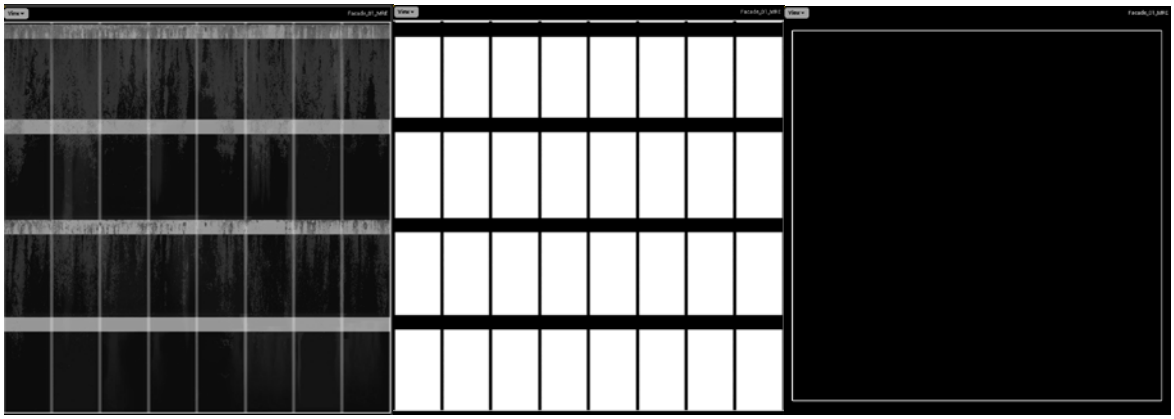
Optimización de las texturas

Una manera de minimizar la cantidad de memoria alojada para un modelo es combinar varias texturas en una. Esto es posible gracias a que los mapas de *Roughness* y *Metallic* no tienen componente de color (incluso podríamos incluir el mapa de *emission* si definimos el color dentro del material en UE4) por lo tanto podemos guardar cada uno en una de las componentes RGBA de la textura para posteriormente obtener solamente el canal que nos interese en UE4.
Dicho de otras palabras si el mapa es sin color podemos guardar cuatro texturas en un solo archivo y solicitar el canal dependiendo de la necesidad después.

Además de lo anteriormente dicho, también tenemos que generar las texturas con una resolución que sea en base dos. Ya que para el motor gráfico le es mucho más fácil procesar y manejar texturas con estas resoluciones.



39 Texturas combinadas



40 Diferentes canales (RGB de izq. a der) de una misma textura

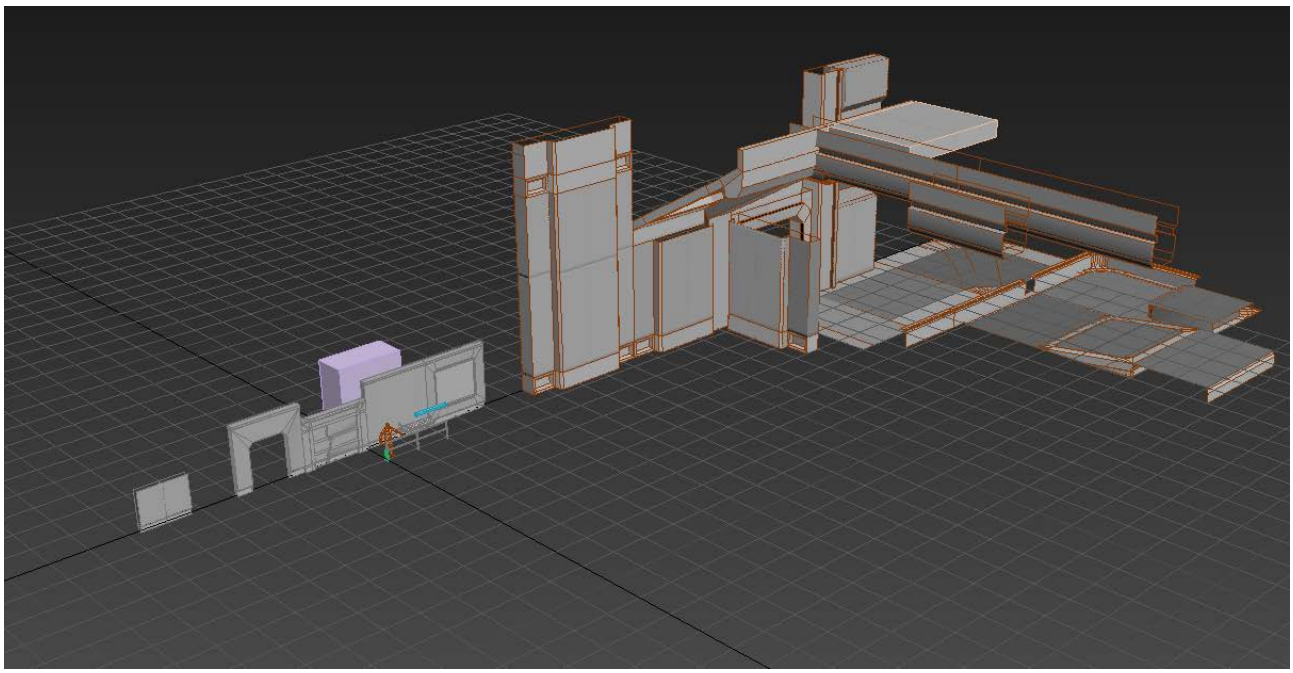
41 Ejemplo de reutilización de texturas

Modularidad

Trabajar con elementos modulares significa con modelos que siguen una escala acorde a una rejilla previamente definida. La principal ventaja de seguir esta metodología es la facilidad para generar nuevos niveles sin necesidad de ir colocando mm a mm cada elemento. Sabemos que al ir arrastrando modelos al motor se van ajustar sin dejar espacios entre ellos. Además, al estar perfectamente alineados evitamos otros problemas típicos de la mala distribución como el *Z Fighting* (Dos caras en la misma posición producen errores de visualización al no definir cual tiene que mostrarse).

Para poder realizar un modelo modular tenemos que definir un tamaño de rejilla (*Grid*) acorde con el motor y guiarnos por ella en todo momento. De lo contrario al montar el nivel dentro de UE4 no nos coincidirán entre ellos y obtendremos huecos indeseados que afectarán negativamente cuando se realice el render la iluminación global.

En la imagen inferior podemos ver algunos de los elementos estructurales utilizados. Como podemos ver todos están definidos por la rejilla.



42 Elementos modulares.

43 Elementos modulares aplicados a una escena.

Sound Editor

Al igual que en cine una buena banda sonora define una película, en un entorno el sonido es una parte también fundamental de él. Unreal por suerte gestiona muy bien este aspecto y nos ofrece gran variedad de ayudas.

Lighting

El apartado de iluminación es, sin duda, uno de los más importantes del motor. Poder generar escenarios realistas es posible si hay un equilibrio entre buenas texturas y buena iluminación. Además, es un elemento clave en la ambientación del nivel, simplemente con el color ya podemos definir qué tipo de zona es.

Unreal, hace uso de técnicas de renderizado a tiempo real y de renderizado previo. En otras palabras, es capaz renderizar previamente cierta iluminación y a la vez dejar que ciertas luces sean dinámicas. Esto se traduce en una mayor calidad y realismo de la iluminación.

Como veremos a continuación, esto también conlleva que sigamos unas pautas y limitaciones.

Static Light

Las luces estáticas, como su nombre indica, son luces que no pueden cambiar una vez se han calculado en el mapa de luces. Los objetos que tienen movimiento, por lo tanto, no son afectados por estas. Como ventaja podemos destacar que, al no ser generadas en tiempo real, son las más eficientes computacionalmente hablando.

Stationary Light

Son luces que, aunque no se pueden mover, sí que son capaces de cambiar algunas de sus características como la intensidad o el color. Dicho esto, también comentar que solo afecta a la luz directa, la luz indirecta (proveniente de rebotes) seguirá siendo pre procesada previamente en el mapa de luces.

Es la luz qué mejor calidad ofrece sin un coste computacional demasiado elevado. Aunque como principal limitación destacaría su máximo de cinco luces estacionarias superpuestas.

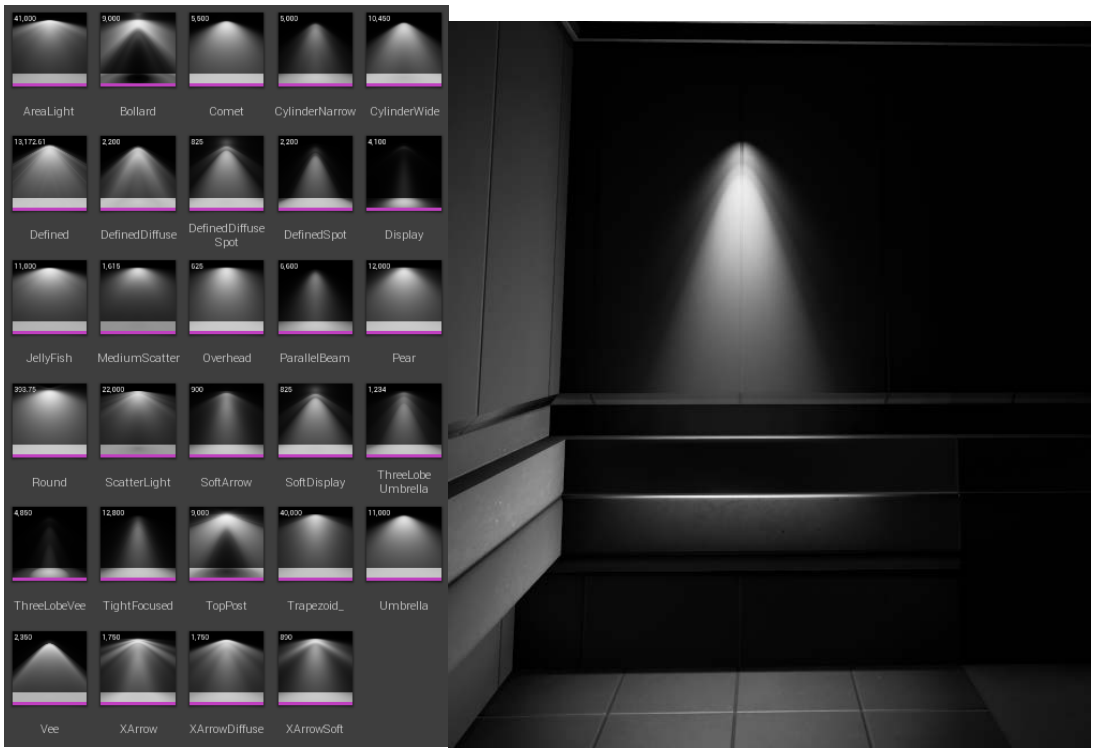
Movable Light

Este tipo de luces permiten que la fuente de luz se desplace y además cambie sus características. Al utilizar el sombreado dinámico en toda la escena, es muy costosa computacionalmente y se debería reservar para ciertos objetos como linternas, luces de coches etc.

Otra desventaja que tiene es que de momento no soporta iluminación indirecta.

IES

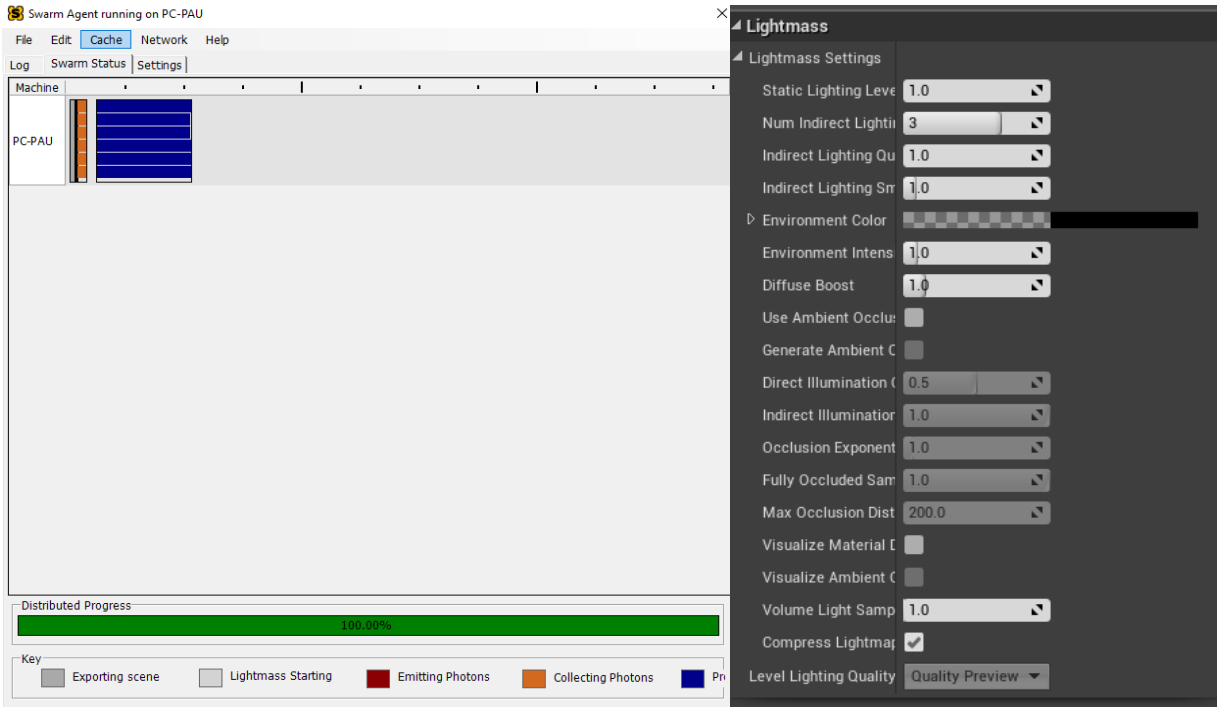
Unreal permite importar archivos fotométricos con los valores y detalles de luces reales. Esta funcionalidad otorga de gran realismo a distribución de estas.



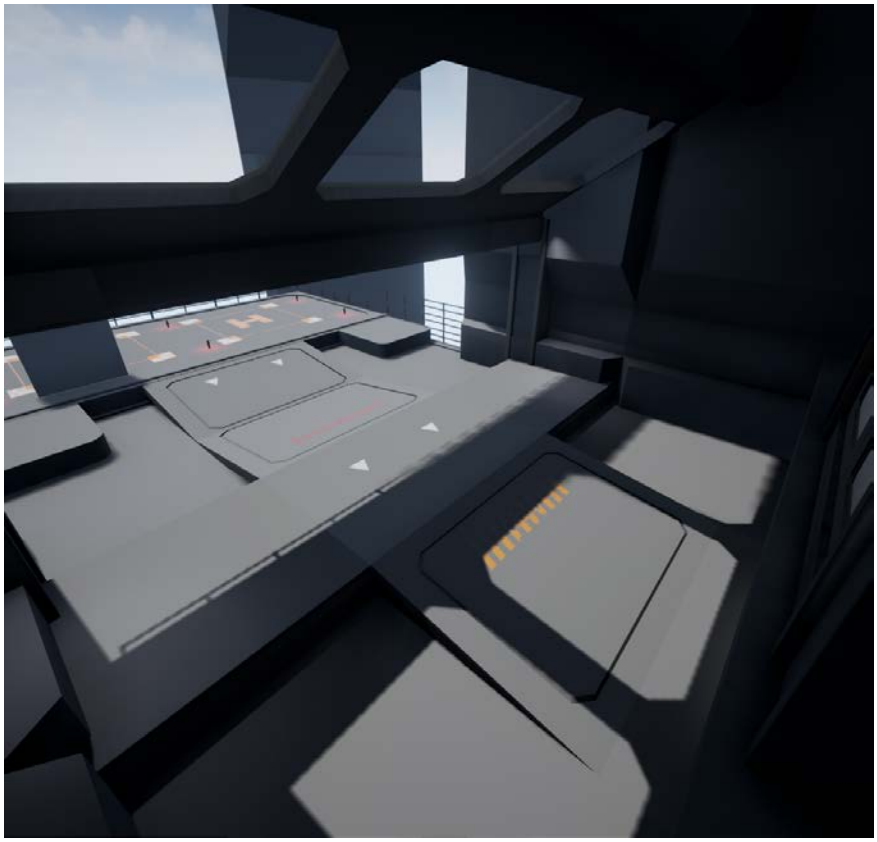
44 Conjunto de archivos IES (Izq.) y luz con ies aplicado (Der.)

GI

UE4 se beneficia del cálculo de iluminación global para calcular rebotes y comportamiento de la luz. Su principal ventaja es que es calculado previamente y por lo tanto no tiene coste computacional



45 Gestor de render Swarm (Izq.) Configuración de la iluminación global (Der.)

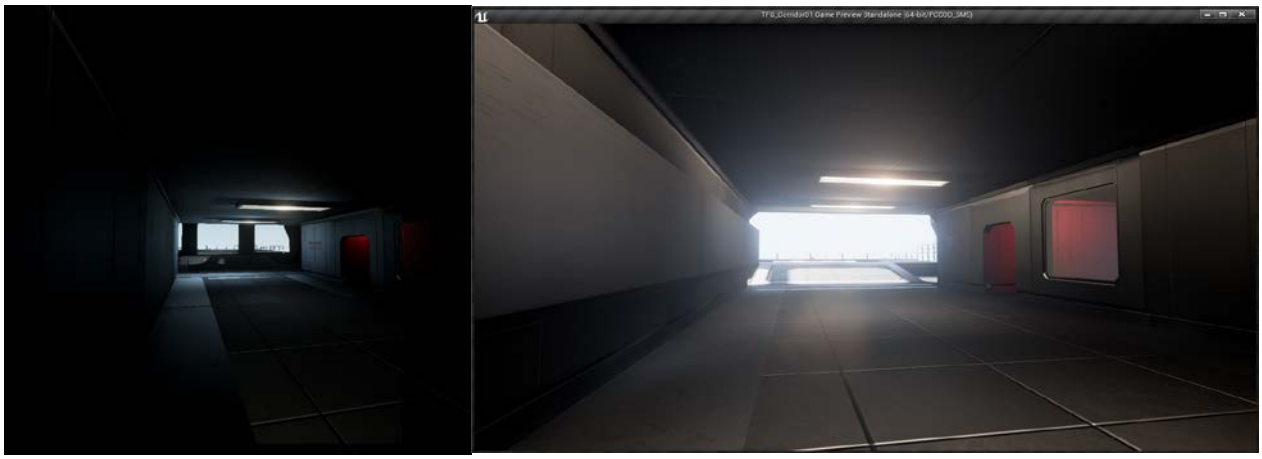


46 Visualización de la iluminación global únicamente

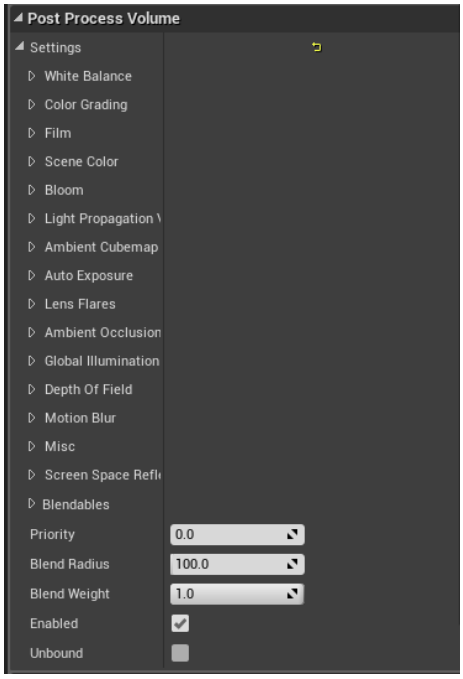
Pos procesado

Una vez tengamos gran parte del nivel montado, viene un parte bastante crucial para darle el detalle que marcará el tono y sentido de la escena, el pos procesado.

Por suerte UE4 dispone de varias herramientas que permiten modificar y crear estos efectos finales de manera muy fácil y visual.



47 Diferentes tipos de pos procesado

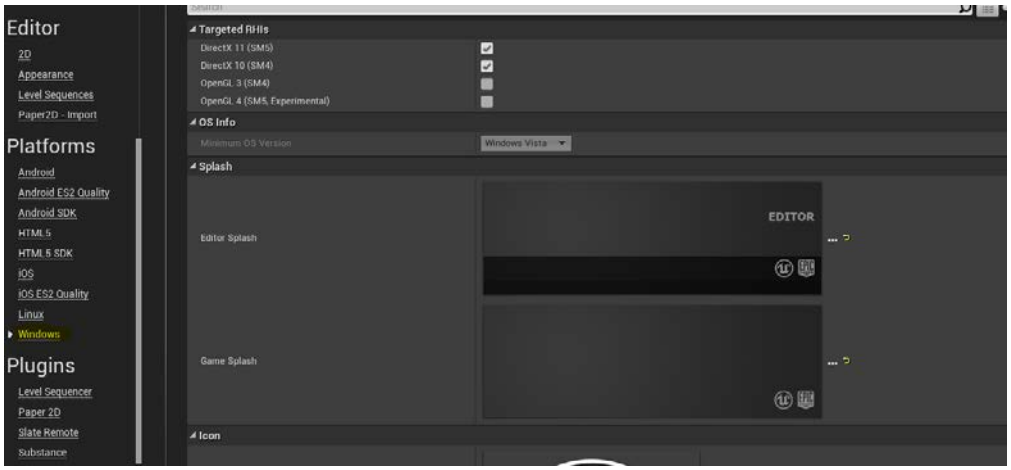


48 Configuraciones del pos procesado

Deployment

UE4 permite exportar el proyecto a las principales plataformas disponibles (consolas incluidas) de manera bastante rápida y fácil. Lo único que tenemos que tener en cuenta es que, para exportar a móviles, hay modificar ciertos parámetros.

Para exportar en consolas es necesario tener licencias de desarrollador proporcionadas por las compañías distribuidoras (Sony / Microsoft).



49 Configuración de plataformas disponibles para exportar

Conclusiones

Sin duda ha sido un proyecto que me ha permitido entender el funcionamiento y el flujo de trabajo que se utiliza hoy en día al crear elementos destinados a ser reproducidos en tiempo real, además de todo lo que ello conlleva como la optimización o el uso de la metodología basada en PBR.

El principal problema que me he encontrado es la extensa cantidad de información que se debe conocer y dominar para poder ir avanzando satisfactoriamente. En alguna fase del proyecto me he encontrado estancado debido a no saber cómo mejorar el resultado que hasta entonces tenía. Pero gracias en parte a toda la información disponible hoy en día en internet, he ido ajustándome a los procesos que se utilizan profesionalmente.

Bibliografía

allegorithmic,. (2016). *PBR Guide*. Retrieved 05 January 2016, from https://www.allegorithmic.com/pbr-guide

Marmoset,. (2014). *PBR In Practice*. Retrieved 11 February 2016, from http://www.marmoset.co/toolbag/learn/pbr-practice

Docs.unrealengine.com,. (2016). *Physically Based Materials | Unreal Engine*. Retrieved 11 February 2016, from https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/PhysicallyBased/index.html#burley

Marmoset,. (2014). *PBR Theory*. Retrieved 11 February 2016, from https://www.marmoset.co/toolbag/learn/pbr-theory

Solid-angle.blogspot.se,. (2014). *Solid Angle: BioShock Infinite Lighting*. Retrieved 12 February 2016, from http://solid-angle.blogspot.se/2014/03/bioshock-infinite-lighting.html

Wiki.polycount.com,. (2016). *Texturing - polycount*. Retrieved 12 February 2016, from http://wiki.polycount.com/wiki/Texturing

Galuzin, A. (2016). *UE4: Guide to Player Scale and World Architecture Dimensions*.*Worldofleveldesign.com*. Retrieved 12 February 2016, from http://www.worldofleveldesign.com/categories/ue4/ue4-guide-to-scale-dimensions.php

80.lv,. (2016). *Studying Lighting & Materials in Unreal Engine 4* . Retrieved 12 February 2016, from http://80.lv/articles/studying-lighting-materials-in-unreal-engine-4/

Galuzin, A. (2016). *UDK: Lightmap UV Layout Techniques and How to Create a Second UV Channel in Maya*.*Worldofleveldesign.com*. Retrieved 12 February 2016, from http://www.worldofleveldesign.com/categories/udk/udk-lightmaps-02-uv-techniques-and-how-to-create-second-uv-channel-in-maya.php

Physically Based Rendering Encyclopedia. (2016). *Google Docs*. Retrieved 10 May 2016, from https://docs.google.com/document/d/1Fb9_KgCo0noxROKN4iT8ntTbx913e-t4Wc2nMRWPzNk/edit

Texel Density for game art by Tim Diaz (page 1 of 2) / 3ds Max, Unreal Dev Kit, Project Overview, Texturingtutorial from 3dtotal.com. (2016). *3dtotal.com*. Retrieved 15 May 2016, from http://www.3dtotal.com/tutorial/1813-texel-density-for-game-art-3ds-max-unreal-dev-kit-by-tim-diaz-udk-bricks-tutorial-environment

Texel density "standars" in UE4 ?. (2009).*polycount*. Retrieved 10 June 2016, from http://polycount.com/discussion/157875/texel-density-standars-in-ue4

Consistent Texel density vs. Importance/Space efficiency?. (2013). polycount. Retrieved 10 June 2016, from http://polycount.com/discussion/118678/consistent-texel-density-vs-importance-space-efficiency/p1

Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck. (2016). Ericchadwick.com. Retrieved 10 June 2016, from http://www.ericchadwick.com/examples/provost/byf1.html

How often are 4096 textures used in next-gen gaming.. (2013). polycount. Retrieved 10 June 2016, from http://polycount.com/discussion/134911/how-often-are-4096-textures-used-in-next-gen-gaming/p1

Unwrapping UVs for Lightmaps . (2016). Docs.unrealengine.com. Retrieved 10 June 2016, from https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/LightmapUnwrapping/

Lightmap Creation for Unreal Engine 4. (2016). YouTube. Retrieved 10 June 2016, from https://www.youtube.com/watch?v=joKnmShAdJE

Galuzin, A. (2016). UDK: Lightmapping Basics and 18 Important Principles for Creating and Using Lightmaps for UDK. Worldofleveldesign.com. Retrieved 10 June 2016, from http://www.worldofleveldesign.com/categories/udk/udk-lightmaps-01-basics-and-important-principles-for-creating-using-lightmaps.php

FBX Static Mesh Pipeline. (2016). Docs.unrealengine.com. Retrieved 13 June 2016, from https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/StaticMeshes/

FBX Import Options Reference. (2016). Docs.unrealengine.com. Retrieved 13 June 2016, from https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/ImportOptions/index.html

Substance Painter. (2016). allegorithmic. Retrieved 14 June 2016, from https://www.allegorithmic.com/products/substance-designer

Normal mapping. (2016). Wikipedia. Retrieved 14 June 2016, from https://en.wikipedia.org/wiki/Normal_mapping

Art Rules for VR environments?. (2005). polycount. Retrieved 14 June 2016, from http://polycount.com/discussion/138023/art-rules-for-vr-environments/p1

Use the Emissive Material Input. (2016). Docs.unrealengine.com. Retrieved 14 June 2016, from https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/HowTo/EmissiveGlow/

Ambient Occlusion and Normal map bake using Xnormal - CRYENGINE Manual - Documentation. (2016). Docs.cryengine.com. Retrieved 14 June 2016, from http://docs.cryengine.com/display/SDKDOC2/Ambient+Occlusion+and+Normal+map+bake+using+Xnormal

Materials . (2016). Docs.unrealengine.com. Retrieved 15 June 2016, from https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/

Physically Based Rendering Encyclopedia. (2016). Google Docs. Retrieved 17 June 2016, from https://docs.google.com/document/d/1Fb9_KgCo0noxROKN4iT8ntTbx913e-t4Wc2nMRWPzNk

Filmic Games » Linear-Space Lighting (i.e. Gamma). (2016). Filmicgames.com. Retrieved 17 June 2016, from http://filmicgames.com/archives/299

Feeding a physically based shading model. (2011). Sébastien Lagarde. Retrieved 17 June 2016, from https://seblagarde.wordpress.com/2011/08/17/feeding-a-physical-based-lighting-mode/

Audio and Sound. (2016). Docs.unrealengine.com. Retrieved 10 August 2016, from https://docs.unrealengine.com/latest/INT/Engine/Audio/

Releasing Your Project. (2016).*Docs.unrealengine.com*. Retrieved 10 August 2016, from https://docs.unrealengine.com/latest/INT/Engine/Deployment/Releasing/

Static or Stationary?. (2016). Forums.unrealengine.com. Retrieved 14 August 2016, from https://forums.unrealengine.com/showthread.php?2931-Static-or-Stationary

LightingTroubleshootingGuide - Epic Wiki. (2016). Wiki.unrealengine.com. Retrieved 15 August 2016, from https://wiki.unrealengine.com/LightingTroubleshootingGuide