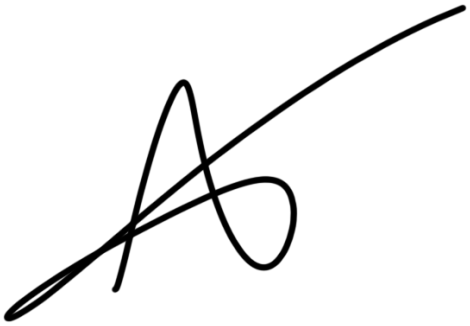
Universitat de Barcelona
Escola de Noves Tecnologies Interactives
Grau en Continguts Digitals Interactius

ALONE

Arnau Aguilar Emanuel

Treball Final de Grau
Curs 2020-2021

Arnau Aguilar



Signatura de l'alumne

Ben Kleber

Signatura del tutor

Índice

Abstract

There are several fields where we've achieved our objectives.

Rendering
We've made a renderer that merges the most important features of HDRP (High Definition Render Pipeline) with the performance of URP (Universal Render Pipeline) and compatibility with low end consoles like Nintendo Switch

Animation
We ended up with a fully controllable player that moves without any animations or animators and also it's fully configurable

Physics
At the end of this project we had a modular system for multiple types of gravity shapes that allowed the designers to build a non linear world. Also the system is performant enough to run on low end consoles.

Keywords

Rendering, Inverse Kinematics, Physics, Puzzles, Alone, Shaders, C#, Unity

Resum

Per a aquest TFG teniem com a objectiu aconseguir tenir un joc de puzles narratiu que fos capaç de correr a un bon *framerate* com a mínim en l'*average* pc publicat per steam de fa 5 anys. Hem hagut de desenvolupar diverses tecnologies desde 0 per a que l'equip pogués assolir aquest objectiu tenint en compte les limitacions de personal que hi ha en un equip indi.
Aquestes tecnologies són:

- Un *render pipeline custom* per a Unity que combini el millor de HDRP i URP
- Un sistema de físiques dinàmic, fàcil d'utilitzar i que permeti als dissenyadors centrarse en el level design.
- Un sistema d'animació que no requereixi d'animadors

Al acabar aquest treball hem aconseguir la gran majoria dels objectius proposats, pero sí que és cert que el sistema de IKs (*Inverse Kinematics*) es podria millorar per que fos més modular i reacciones millor a les malles complexes.
Per últim, com a línies de futur creiem que sería interessant centrar-se més en els IKs per assolir el que ens ha faltat, però també seguir expandint el render pipeline per a que es mantingui al dia amb els requeriments de les noves tecnologies de renderitzat.

Resumen

Para este TFG tenemos como objetivo conseguir tener un juego de puzles narrativo que fuera capaz de correr a un buen *framerate* como mínimo al pc medio publicado por Steam de hace cinco años. Hemos tenido que desarrollar diversas tecnologías desde cero para que el equipo pudiera alcanzar este objetivo teniendo en cuenta las limitaciones de personal que hay en un equipo indie.
Estas tecnologías son:

- Un *render pipeline custom* para Unity que junte lo mejor de HDRP y URP
- Un sistema de físicas dinámico, fácil de utilizar y que permita a los diseñadores centrarse en el level design.
- Un sistema de animación que no requiera de animadores.

Al acabar este trabajo hemos conseguido la gran mayoría de objetivos propuestos, pero sí que es cierto que el sistema de IKs (*Inverse kinematics*) se podría mejorar para que fuera más modular y reaccionara mejor a mallas complejas.
Por último, como líneas de futuro creemos que sería interesante centrarse más en los IKs para conseguir lo que nos ha faltado, pero también seguir expandiendo el render pipeline para que se mantenga al día con los requerimientos de las nuevas tecnologías de renderizado.

Summary

For this TFG our objective was to have a narrative puzzle game that was able to run at a good framerate at least on the average pc published by Steam five years ago. We have got to develop various technologies from scratch so that the team could achieve that objective, taking into account the limitations of staff that an indie team has.
Thos technologies are:

- A custom render pipeline for Unity that gets the best from HDRP and URP.
- A dynamic physics system, easy to use and that allows designers to focus on level design.
- An animation system that does not require animators.

At the end of this project we have achieved mostly all of the proposed objectives, although it's true that the IKs (Inverse Kinematics) could be improved to be more modular and to react better to complex meshes.
Lastly, as future work, we think that it would be interesting to focus more on the IKs to achieve what we lacked, but also keep expanding the render pipeline so it keeps updated with the requirements of the new rendering technologies.

Introducción

Para este TFG desarrollaremos Alone, un videojuego de puzles relacionados con la gravedad con un fuerte peso narrativo para PC.

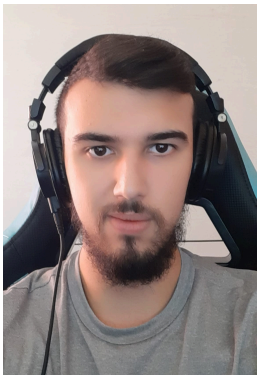
En esta experiencia, el jugador deberá resolver puzles activando y desactivando distintas gravedades que se encuentre en el entorno, asimismo, deberá encontrar el camino a través de estas gravedades para avanzar y descubrir más sobre qué pasó aquí. En este viaje descubrirá cómo la civilización vivió un post-apocalipsis recluidos dentro de una fábrica de robots, y cómo se las ingeniaron para sobrevivir durante muchos años. Podremos ver un estilo artístico oscuro, realista y futurista, aunque desgastado por el tiempo. También encontraremos elementos rudimentarios hechos con piezas futuristas. Nuestro personaje es el único robot que quedó dentro de las instalaciones cuando sucedió todo, y ha estuvo conviviendo con la humanidad hasta que esta se extinguió, aunque tras quedar congelado durante tanto tiempo, no tiene ningún recuerdo. El juego transcurrirá en el post-post-apocalipsis, donde nuestro robot recorre la fábrica totalmente sólo, después de que la humanidad no consiguiera resurgir del post-apocalipsis.

Aunque esta sea una propuesta muy tradicional, un juego de puzles narrativo, las mecánicas principales y nuestro público objetivo presenta diversos retos tanto técnicos como de diseño que deberemos resolver. En este documento se podrá encontrar un estudio en profundidad sobre cómo desarrollaremos un *renderer* que

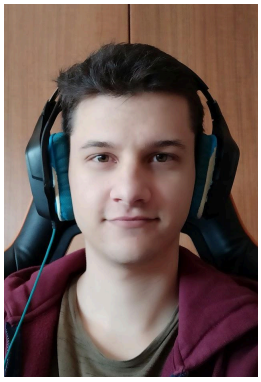
nos permita llegar a dispositivos de hace más de 5 años y correr a 60 fps o más. También se podrá encontrar todo el trabajo que será necesario para desarrollar todo el sistema de gravedades interactivas y cambiantes, además de todos los sistemas necesarios para integrar todo esto en Unity y con el sistema de IKs que controlará de forma procedural todo el personaje. Pero este trabajo no serán solo retos técnicos, nuestro juego presenta un nuevo tipo de paradigma, nuestros diseñadores tendrán que tener en cuenta un entorno que se puede recorrer en todas direcciones por todas las superficies. Por tanto en este TFG podréis encontrar todos los detalles y el aprendizaje recogido.

El equipo lo formamos:

Marc Solis
Designer &
Programmer



Ares Fernandez
Designer &
Programmer



Ignasi Pelayo
CTO

Arnau Aguilar
Render Programmer &
Engine Programmer

Pol Sanchez
Artist

Jordi Soler
Artist

Objetivos

Como se ha mencionado en el apartado anterior desarrollaremos un juego de plataformas, con puzles de gravedad y creemos imprescindible que lo puedan correr una gran variedad de ordenadores, por lo tanto necesitaremos cumplir los siguientes objetivos:

- [Grupo] Realizar una vertical slice de 30 minutos de duración.
- [Grupo] Realizar un juego de puzles y gravedades.
- [Grupo] Desarrollar una vertical slice con 3 niveles.
- [Grupo] Que el juego corra a 60FPS o más en el *Average PC* de steam de hace 5 años.

Además de forma individual, me encargare de mantener el engine al nivel que necesite el equipo tanto en gráficos como funcionalidad, para ello deberé cumplir:

- [Individual] Desarrollar un render pipeline custom con features de HDRP y URP con la performance de URP.
- [Individual] Que el personaje esté animado al completo por un sistema de IKs (*Inverse Kinematics*) personalizado.
- [Individual, Arnau] Construir un sistema de multi-escenas con carga asíncrona para evitar pantallas de carga y simular un mundo sin “límites”.
- [Individual, Arnau] Diseñar y desarrollar shaders complejos para que los artistas bistan los niveles (nieblas, triplanares, translucencias, etc...).

Estado del arte y diseño de la solución

Renderizado

Estado actual

Actualmente en Unity existen tres opciones para renderizar. La primera es usar el renderer default, que durante los últimos años ha caído en desuso porque su calidad de renderizado y el rendimiento que es capaz de conseguir se ven eclipsadas por las que se pueden conseguir con las otras dos opciones, siendo estas el *universal render pipeline* (URP) y el *high definition render pipeline* (HDRP).

El primero de los dos (URP) es un *renderer* simple con un consumo medio, muy utilizado para desarrollos móvil, para consolas y para PC si se define como objetivo máquinas con poca potencia. El otro es HDRP, un render de muy alta calidad pero muy demandante en cuanto a rendimiento, que actualmente se usa para desarrollos de muchísima calidad, orientados a PCs de alta gama o directamente al cine, animación o incluso CGI.

Referencias

- Approximating translucency for a fast, cheap, and convincing subsurface scattering, (Barré-Brisebois, 2011)[1].
- Unity Scriptable Render Pipeline Tutorials, (Flick, 2018)[2]
- VALORANT Shaders and gameplay clarity, (Riot Games Technology, 2020)[3]
- Technical and Visual Analysis of Overwatch, (Bermanseder, 2019)[4]

Solución

Ninguno de las dos opciones actuales (URP o HDRP) cubre completamente nuestras necesidades, que se encuentran justo entre las dos opciones: necesitamos resultados realistas pero a un coste bajo, así que decidimos que desarrollar un renderer custom que cogiera lo que necesitara de los dos existentes y descartara lo que no, sería la mejor opción. De esta forma, tenemos por una parte el conocimiento absoluto del renderer y por lo tanto podemos optimizar muchísimo más y mejor, y por otra parte podemos decidir que necesitamos, por lo que el renderer solo calculara lo estrictamente imprescindible para nuestro título.

No hay mucha documentación de cómo desarrollar un render pipeline, por suerte, Jasper Flick[2] tiene los únicos tutoriales sobre cómo hacer esto, lo que nos ha sido de extrema utilidad. En ellos enseña a desarrollar un pipeline que concuerda en muchos casos con lo que necesitamos: un renderer realista y bonito, con poco coste computacional. Aun así, hemos tenido que modificar el renderer en muchos puntos para adaptarlo a nuestras necesidades concretas, lo que nos ha permitido optimizar

aún más en varios apartados. Lo primero que decidimos fue obviar las funcionalidades de URP y HDRP que no necesitábamos, siendo estas:

- Casi todos los post procesados de HDRP
- El *stencils buffer* de HDRP
- La luz volumétrica de HDRP
- PerObjectMaterialProperties
- Reflejos realistas
- Soporte para múltiples cámaras (pantalla partida)
- Escala de renderizado dinámica.

Como todo esto nos deja mucho margen para actuar, añadimos ciertos pasos (textura de profundidad con máscaras y soporte nativo para *thickness maps*) a este pipeline para tener nieblas a muy bajo coste y efectos de translucencia como el del hielo que de otra forma hubieran sido imposibles.

Además incluimos features de HDRP que actualmente no se pueden conseguir en URP, como soporte para sombras en tiempo real, hasta lo que nosotros denominamos dieciséis unidades de sombra, donde una *point light* consume seis unidades, una *spotlight* consume una unidad y una *directional light* consume una unidad también, solo que esta está separada en otro *buffer* para permitirnos tener más espacio para las luces adicionales.

El *shader Lit* es el *shader* básico de este *renderer*. Aquí es donde se encuentran la mayoría de las diferencias, pues gran parte de este es distinto al propuesto por Flick[2] ya que nosotros no damos soporte a *detail maps*, lo que nos deja un canal libre en el *MODS maps* que usamos para integrar *emission maps* dinámicos que simulan movimiento.
Por último creímos imprescindible integrar este *shader* con muchas de las herramientas que unity integra, como *pro builder* o *poly brush*, para permitir a nuestros artistas pintar directamente en el *vertex color* de la malla para hacer uso de algunos *shaders* como los *triplanar*, o los *terrain*.

Inverse Kinematics (IKs)

Estado Actual
Actualmente Unity no provee ningún componente para usar IKs en su motor, por lo que nos hemos visto obligados a desarrollar el sistema desde cero. Aun así este es un tema muy investigado por lo que pudimos ver que la solución más extendida era seguir el algoritmo llamado FABRIK, ya que es extremadamente robusto y se puede modificar para tener *constraints.*

Pero tener un *solver* es solo el primer paso, si lo que queremos es tener un personaje completamente animado de forma procedural, no hay mucha información al respecto.
En estos casos, la solución más habitual es usar un sistema de *positions/desiredPositions/RestingPositions*. Estos sistemas solo se acostumbran a usar para enemigos con formas arácnidas o robóticas, ya que no es suficientemente avanzado como para animar un humanoide completo con una calidad suficiente. Esta es una de las limitaciones que tuvimos que tener presentes a la hora de desarrollar el mundo del juego, de tal forma que el protagonista pudiera ser un robot.

Referencias

- Unity PROCEDURAL ANIMATION tutorial (10 steps), (Codeer, 2020)[6]
- FABRIK: A fast, iterative solver for the Inverse Kinematics problem. (Aristidou, 2011)[7]
- Forward And Backward Reaching Inverse Kinematics. (Aristidou, 2011)[8]
- Climbing a moving robot - Unity game mini devlog. (Codeer, 2020b).[13]

Solución
Las animaciones del personaje son uno de los apartados más delicados de este estudio ya que hay muy pocos ejemplos de personajes principales animados de forma procedural, así que uno de los pasos más importantes fue diseñar un personaje que encajara con este tipo de movimiento. Un robot cuadrúpedo fue la mejor decisión, ya que de esta forma solo hay que animar cuatro patas, la cabeza y algunos periféricos como antenas o luces para dotarlo de personalidad.
El sistema de animado de las patas es muy tradicional en el sentido de que sigue la convención estándar para mover una pierna: hay una posición de reposo y cuando la pata está demasiado lejos de esta, se mueve hacia ella. Aunque el sistema sea el tradicional, hemos tenido que innovar mucho para poder hacer esto en nuestro juego manteniendo el objetivo de un alto rendimiento en equipos antiguos ya que encontrar los puntos de superficie para toda la geometría de la escena generalmente es muy caro debido a que normalmente requiere de muchos *raycasts*. Como no podemos permitirnos malgastar milisegundos, desarrollamos un sistema que *bakea* la geometría en cuadrantes para tenerla toda organizada en árboles de fácil y rápido acceso. Este sistema requirió que usaramos un sistema de *GUIDs* y *GUID references* ya que trabajamos en multi escena para agilizar el *workflow* y poder cargar escenas de forma dinámica y aditiva, así que no podríamos referenciar geometría entre escenas a no ser por este sistema.

El resto del sistema es bastante más sencillo. La animación de la cabeza requiere de cierta complejidad en cuanto a los cálculos de rotación para sincronizarlo con las gravedades cambiantes y además debe permitir cierta flexibilidad a los diseñadores y artistas para poder tocar las curvas de interpolación que usaremos para animarlas desde código y conseguir así un *gamefeel* acorde a la visión del proyecto.

Diseño de la Mecánica Principal

Estado Actual

Alone es un juego de puzles relacionados con la gravedad y con una narrativa centrada tras un apocalipsis. Actualmente existen dos juegos que tratan de forma excelente estos temas: Por un lado Mario Galaxy[12], con sus puzles plataformeros de gravedades. Y por otro lado Portal[11], con unos puzles excelentes y una narrativa post apocalíptica que se desarrolla en salas, donde además el jugador puede jugar caminos extras y descubrir sobre la historia a través de su entorno.

Estos dos juegos representan todo lo que queremos conseguir con cada pieza de el nuestro, por eso consideramos que son la referencia más excelente para nosotros.

Referencias

- Mario Galaxy. (Nintendo, 2007)[12]
- Portal. (Valve, 2007)[11]

Solución

Si bien es cierto que estos juegos son excelentes a su manera, nosotros pretendemos juntar estos dos conceptos y mejorarlos en algunos aspectos. Para ello pretendemos diferir de Mario Galaxy[12] en diversos apartados, el primero es que el jugador tiene control de su entorno y de las gravedades que hay en él. Mario Galaxy[12] usa las gravedades para darle un giro a todo su plataformeo, pero nunca forman parte del enigma que el usuario tiene que resolver para progresar, en cambio nuestras gravedades se pueden activar y desactivar, y pasan a formar parte de las herramientas que tiene el jugador a su disposición para poder resolver los puzles. El segundo apartado es que la cámara es completamente distinta a la de Mario. En el exitoso juego de Nintendo, la cámara es estática en gran parte del juego y se alinea con los ejes del mundo, es decir, si el jugador está en el techo de una sala el jugador verá a Mario boca abajo.

Figura 1. Gameplay de Mario Galaxy

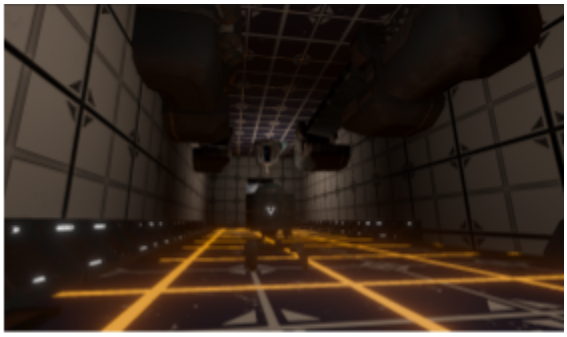


Figura 2. Captura de pantalla de Alone con el jugador en el techo

Como se puede ver en Alone la cámara siempre está alineada con el eje de coordenadas locales del jugador, lo que resulta en el player siempre viento al PJ bien orientado y la sala en la que está rotada. Sabemos que esta aproximación es menos obvia para el jugador, ya que si está en el techo solo lo percibe si se fija en el entorno, pero creemos que es más conveniente. En Mario Galaxy[12] la cámara es susceptible de marear a los jugadores porque se crea una disonancia cognitiva entre los controles y la cámara. Esto se debe a que los controles están alineados con el personaje, lo que, dependiendo de la orientación del personaje, difiere con la cámara, que está alineada con los ejes del mundo, con lo cual, mover hacia arriba el joystick no siempre significa mover a Mario hacia arriba. Esto mismo además, dificulta muchísimo los saltos y el parkour. Nuestra perspectiva es más inmersiva y aunque puede generar desorientación si el nivel no está muy bien definido, lo podemos usar en nuestro favor para enfatizar momentos narrativos.

En cuanto a Portal[11], nuestro juego también pasa dentro de unas instalaciones, en nuestro caso una fábrica donde vivieron los últimos supervivientes del apocalipsis durante muchos muchos años. Nos inspiramos en las ambientaciones, en cómo el jugador puede descubrir más de la narrativa simplemente observando los restos de los últimos que estuvieron ahí. Además, también queremos mejorar la integración de los puzles con el mundo, en Portal[11] son puzles muy obvios porque la narrativa lo permite, pero creemos que podrían ser mucho más inmersivos si el jugador se encuentra con un entorno que podría existir y fuera funcional por sí solo.

Programación de Gravedades

Estado Actual

No hay muchos juegos que usen múltiples gravedades complejas en un mismo escenario. No obstante, hemos encontrado varias formas de reproducir esto. La primera la encontramos en Mario Galaxy[12], en este el personaje se ve atraído por la superficie más cercana a él, independientemente de la orientación que tenga. De esta forma se aplica al jugador una fuerza de gravedad en forma de aceleración hacia ese punto de la geometría. Esto permite tener gravedades complejas para cualquier tipo de superficie con la condición de que el personaje no puede ser atraído por varias gravedades a la vez.

Otra aproximación es la que sugiere J. Flick[5], que consiste en tener múltiples gravedades posicionadas en el mundo. Teniendo en cuenta la posición del jugador y la posición, área y rango de la gravedad en cuestión, se calcula si se debe aplicar una fuerza de gravedad en forma de aceleración y hacia dónde. Esto se calcula para todas las gravedades, y finalmente se suman todas las fuerzas de gravedad individuales para obtener la gravedad resultante que se debe aplicar en el jugador. Esta aproximación permite tener mucho más control sobre cómo te afecta cada parte del entorno y es más óptimo en run time, pero tiene un coste mucho más elevado en cuanto a tiempo de implementación tanto de programación como de diseño.

Referencias

- Mario Galaxy. (Nintendo, 2007)[12]
- Custom Gravity. (Flick, 2020)[5]

Solución

Hemos decidido seguir la segunda aproximación, debido a que las gravedades en nuestro juego se pueden activar y desactivar, lo cual hace que la primera aproximación no funcione en todos los casos, ya que la geometría más cercana al jugador no tiene porqué ser la que tiene la gravedad activa. Tampoco permite ser afectado por más de una gravedad a la vez, teniendo sólo un sólo origen (la gravedad más cercana). En cambio en la segunda aproximación sí que es aplicable.

Aunque el sistema sea compatible no quiere decir que esté preparado para ello, así que hemos tenido que adaptar y crear sistemas adicionales para soportar la activación y desactivación de gravedades, así como también la creación de más tipos de gravedades con formas y direcciones distintas que hemos ido necesitando para adaptarlas al nivel, ya que había un catálogo muy pequeño de formas con sus gravedades.

Gravedades soportadas en un inicio:
Planos infinitos, esferas (interior y exterior), cubos (interior y exterior)

Gravedades adicionales implementadas:
Planos finitos, cubos con activación y desactivación por caras (interior y exterior), cilindros (exterior), gravedades en forma de L (interior y exterior, una gravedad compuesta por dos planos finitos y un cilindro en el punto de unión de las gravedades de planos), una gravedad en forma de plano finito que sólo afecta si en un punto no afecta ninguna gravedad (gravedad de último recurso) y una gravedad global que sólo afecta si ninguna afecta en un punto (ni siquiera la gravedad de último recurso)

Narrativa post-post-apocalíptica

Estado Actual
Nuestra narrativa parte del género post-apocalíptico.
Haciendo un breve análisis de las tendencias socioculturales de los años 2020-2021 nos dimos cuenta de la propensión actual hacia la estética futurista y el género post apocalíptico, teniendo previstos estrenos relevantes tanto en libros, películas y videojuegos.
Así pues, es de esperar un mayor interés por obras similares durante los próximos años.
Aunque el post apocalípsis suele funcionar bien, es cierto que resultaría una buena idea buscar un factor diferencial que atrajera a la audiencia. Con esta idea en mente, surge la propuesta del post-post-apocalipsis, la cual ha explorado con mucho éxito recientemente el videojuego *Horizon Zero Dawn* (Guerrilla Games, 2018)[10]. Este juego explora el género a través del resurgimiento de una nueva sociedad que mezcla robots con naturaleza. En este entorno, abunda la vida de este estilo, con mucha naturaleza y seres robóticos, haciendo que los paisajes sean muy coloridos, con un mundo vivo y muy dinámico. Otro punto de vista del género post-post-apocalíptico lo encontramos parcialmente en Wall-E[14]. Decimos parcialmente, ya que si lo analizamos, vemos que la humanidad se encuentra en un post-apocalipsis, ya que han tenido que irse de la Tierra y están viviendo en otro planeta, no obstante, el post-apocalipsis de Wall-E sucede en la Tierra con un montón de compañeros que recogen basura. Eventualmente, todos sus compañeros se acaban estropeando, y él se queda sólo. Su post-post-apocalipsis explora el camino que hace este robot para volver a encontrarse con la sociedad.

Referencias

- Horizon Zero Dawn. (Guerrilla Games, 2018)[10]
- Wall-E (Pixar Animation Studios, 2008)[14]

Solución

Nuestro juego explora la idea del post-post-apocalipsis inspirándose en *Horizon Zero Dawn*[10] y el éxito que tuvo su propuesta narrativa. A partir de esto nos dimos cuenta que los diferentes ejemplos que encontramos del post-post-apocalipsis suelen explorar esta idea del mismo modo: Evento apocalíptico, período de incertidumbre y caos (post-apocalipsis) y posteriormente la vuelta a una nueva normalidad (post-post-apocalipsis), ya sea con el resurgir y la consolidación de una nueva civilización en el territorio,con el viaje de vuelta a la sociedad establecida...

Nuestro juego sin embargo, explora otro posible desenlace, ¿Qué pasaría si la humanidad no lo lograra? ¿Y si en vez de adaptarse y resurgir, perece y se acaba la vida tal y como la conocemos? Queremos explorar una visión más pesimista y oscura, en la cual no hay una sociedad a la que volver, ni una nueva que resurja de las cenizas. Nuestro post-post-apocalipsis indaga en la idea de una soledad absoluta, de un viaje de reencuentro con la sociedad fallido, un camino a la nada, donde la esperanza de encontrar una civilización que resurja desaparece.

De esta forma, nuestro juego muestra el mundo desde los ojos de un pequeño robot, un ser no biológico testigo del declive de la civilización hasta su extinción. Seguiremos el viaje de este robot que intenta reencontrarse con una sociedad que vivió recluida mucho tiempo en un post-apocalipsis, pero en este caso nos encontraremos con un final no tan bonito, donde el jugador tan sólo podrá contemplar el mundo desde la soledad de no quedar nadie en él.

Estudio de la competencia

Como competencia hemos elegido solamente juegos lanzados entre los años 2019 y 2020, ya que son títulos muy recientes y son los que más probabilidad tienen de competir con el nuestro. Para ello hemos analizado juegos con una estética o tema parecido y juegos con mecánicas parecidas (o ambos).

Tema y *Aesthetics*

Nombre	Propietarios	Precio	Lanzamiento	Steam	SteamSpy
UNHERD	0 .. 20,000	5,69€	Feb 15, 2021	Save 10% on UNHEARD on Steam	UNHERD
TurnTack	0 .. 20,000	12,49€	Jan 15, 2021	TurnTack on Steam	TurnTack
The Medium	50,000 .. 100,000	49,99€	Jan 2021	The Medium on Steam	The Medium

Tabla 1. Juegos competidores en tema y *aesthetics*

Mecánicas

Nombre	Propietarios	Precio	Lanzamiento	Steam	SteamSpy
Alucinod	0 .. 20,000	12,49€	Aug 30, 2019	Alucinod	Alucinod
Etherborn	0 .. 20,000	16,99€	Jul 18, 2019	Etherborn on Steam	Etherborn
Journey	100,000 .. 200,000	12,49€	Jun 11, 2020	Journey on Steam	Journey

Tabla 2. Juegos competidores en mecánicas

Propuesta de valor

Nuestra propuesta consiste en un juego que se asemeje más a leer un libro, una experiencia personal e introspectiva, con un ritmo pausado que se adapte al jugador y disfrutable aunque se disponga de sesiones de juego cortas.

Nuestro *renderer* propio, junto a los bajos requisitos técnicos y precio asequible, hará de nuestro juego un producto disfrutable para todos los bolsillos.

Mapa de posicionamiento

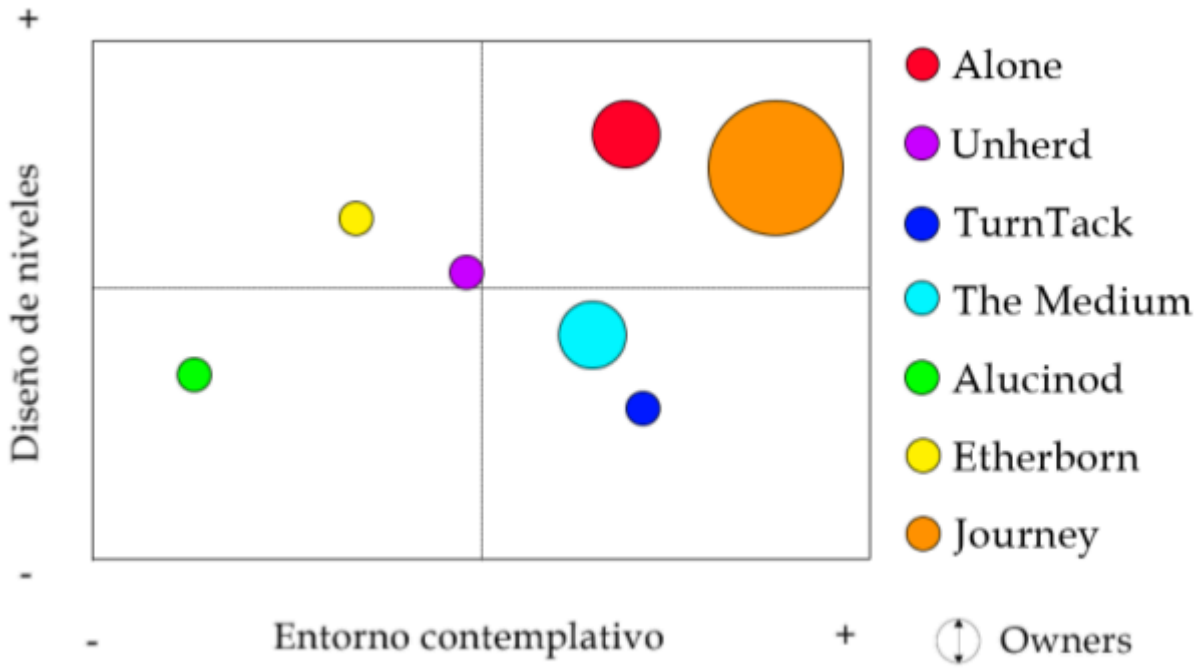


Figura 3. Mapa de posicionamiento

Hemos considerado que los puntos más importantes a tener en cuenta en los juegos de nuestro género (puzles-contemplativo) son, por un lado el diseño de niveles, para hacer los puzles atractivos y que sean un reto para el jugador, y por otro lado un entorno contemplativo, y con esto nos referimos a un entorno visualmente muy atractivo, y que el jugador disfrute con el mero hecho de contemplar el entorno.

También hemos medido los owners (usuarios que tienen el juego) ya que es un dato objetivo y con números específicos que podemos cuantificar. Lo hemos hecho así,

debido a que popularidad o conocimiento del producto son valores poco cuantificables y subjetivos.

Nuestro principal competidor por similitud de mecánicas que es Etherborn, podemos ver como se queda atrás artísticamente, ya que no tiene un entorno muy elaborado.

Por otro lado, podemos observar como Journey sí que es un buen ejemplo tanto de entorno contemplativo como de diseño de niveles. No obstante, sus mecánicas son muy diferentes, son más simples, y por ello nuestro diseño de niveles es un poco más elaborado. Su estilo visual y temática son bastante diferentes, aunque la calidad también es muy alta. No obstante, creemos que Journey aunque es competencia, es un juego con temática muy diferente y ya hace bastante tiempo que salió. Por tanto, creemos que precisamente muchos jugadores de nuestro público objetivo ya lo habrán jugado y por tanto, no sea una competencia tan directa sino más bien, que esos jugadores puedan venir a nuestro juego buscando algo similar a Journey.

Por último, creemos que tanto las mecánicas como la temática de nuestro juego Alone, son puntos fuertes e innovadores que lo diferenciarán del resto de la competencia.

Segmentación de target

Variable Geográfica
- España
- Europa
- América

Variable demográfica
- Adulto joven
- Tamaño familiar medio - alto
- Profesión estresante

Variable psicográfica
- Persona que quiere pasar un rato sola
- Persona que busca entretenimiento pasajero
- Gente con personalidad tranquila

Variable conductual

- Persona con poco tiempo
- Gente que no juega en exceso

Buyer persona

Producto

Figura 4

En esta propuesta queremos explorar la idea del post-post-apocalipsis, queremos alejarnos de la estética caótica y depresiva que predomina en el género post-apocalíptico pero manteniendo el espíritu ya que es un formato que funciona muy bien con las grandes masas. Para ello este juego habla sobre qué pasa después de que muera toda la población tras haber estado muchos años recluida y sobreviviendo a duras penas, ¿cómo renace la vida?, ¿llega siquiera a resurgir?, ¿cómo interactúa el jugador con este entorno y con los restos de los que le precedieron?

Para responder a todas estas preguntas acompañaremos a un pequeño robot que se despierta después de pasar cientos de años congelado bajo el hielo. El jugador tendrá que avanzar e investigar todo lo que vea sobre la civilización que vino antes

de él. De esta forma descubrirá que esos seres aprendieron a controlar la gravedad en su favor, lo que llevará al jugador a tener que resolver puzles relacionados con la gravedad.

Precio
14.99€
Alone no va a ser *free to play* ni contendrá micro-pagos ya que eso nos alejaría demasiado de la experiencia que busca nuestro target.
Por eso hemos seguido una estrategia basada en la competencia para elegir el precio, nuestra competencia ronda los 15/17€.
Además hemos tomado como competencia otros sectores fuera de los videojuegos, sobretodo los libros, por eso el precio ha acabado siendo 14.99€

Distribución

La distribución de nuestro juego se hará únicamente en formato online, puesto que los costes de fabricación y distribución físicas, así como los contactos necesarios para esta se escapa de nuestras capacidades, además nos obligaría a vender el producto a un precio mayor.

Así pues, siendo un juego para PC y en formato online, nos decantamos por Steam como plataforma de distribución. Aunque han aparecido en el mercado otras plataformas de distribución de videojuegos, Steam sigue siendo la que más vende, pese a las quejas de los desarrolladores por las comisiones que aplica y por la baja visibilidad de sus juegos entre la abundante oferta.

Además de ser la plataforma estrella en PC, planeamos usar Steam porque nos proporciona opciones como desarrolladores tales como las rebajas, que son una herramienta realmente útil para conseguir visibilidad en la store.

Una práctica que empieza a extenderse en las empresas indies pequeñas como la nuestra y con bastantes buenos resultados, es poner el juego de rebajas entre los períodos habituales de rebajas, en vez de durante estos. De esta forma, es mucho más factible conseguir visibilidad en la tienda, ya que en estas fechas no compites con muchos juegos para aparecer en la cartelera de rebajas especiales.

Para los tags de Steam, nos hemos basado en tags que definen bien nuestro juego, que use nuestra competencia cercana y que sean de calidad como Journey, para

que si buscan juegos parecidos a este porque les ha gustado, encuentren el nuestro. Estos tags son: Aventura, Indie, Belleza, Atmosférico y Relajante. Además, también hemos puesto algún tag específico y diferencial que no tiene la competencia pero nosotros sí, y que creemos que nos da una ventaja competitiva, como lo son Misterio, Exploración y Gravedad. Si bien nuestro juego tiene componente de Puzles, este es un tag muy sobrecargado en Steam, así que lo pondremos con menos relevancia que los demás ya que no creemos que nos aporte una gran ventaja en cuanto a posicionamiento.

Además de los tags, creemos conveniente incluir la palabra "journey" en la descripción de nuestro producto, en una frase como: "explore this mysterious world during a wonderful and peaceful journey". La descripción encaja con nuestro juego y además incluimos la palabra journey, esto lo hacemos debido a que aunque dentro de la aplicación de steam no sirva para posicionarnos, sí que nos sirve como posicionamiento en buscadores web como Google o Bing ya que la página web de steam de nuestro juego contendrá esa palabra.

También queremos poner palabras clave como "juego parecido a Journey" o "juego relajante y tranquilo" en las reviews de Steam de nuestro juego, mediante cuentas alternativas, con el mismo objetivo anterior de posicionarnos en buscadores web.

Metodología y herramientas

Metodologia
Programación y diseño
Todas las tareas de programación y diseño se regirán por la norma de los *Quality Levels.* Esta norma marca que todas las tareas pasarán por tres estados:

- QL1: Donde la tarea está prototipada, es decir, la tarea es funcional pero requerirá de pulido.
- QL2: La tarea funciona y está pulida, si el juego tuviera que salir así podría hacerlo.
- QL3: La tarea es funcional y pulida al extremo, no hay nada que creamos conveniente cambiar.

De esta forma, para realizar una tarea, primero se deberá diseñar y redactar qué partes tendrá lo que estemos haciendo, si es código realizar un UML y decidir qué funciones se expondrán al resto de programadores y si se trata de diseño redactar en el GDD todo lo relacionado. Seguidamente se implementará la tarea en QL1 y se actualizará la documentación con los cambios que hayan surgido. Los siguientes pasos serían implementar la QL2 y QL3, pero en nuestro pipeline no tiene porque estar pegadas, el momento en el que esto se realice dependerá de producción.

Herramientas:

Ide de programación
Los programadores trabajamos todos con JetBrains Rider. La primera opción que valoramos fue usar Visual Studio 2019 ya que es gratuito, pero por contraparte es muy pesado, consume muchos recursos y su funcionalidad se queda muy corta comparada con la gran variedad de features y lenguajes soportados por Rider, en especial lenguajes de programación de shaders. Estas características son esenciales para nuestro proyecto con tanto peso gráfico y además valoramos mucho que todo el equipo use las misma herramientas a ser posible.

Motor
Estudiamos diversas opciones, pero finalmente elegimos Unity por diversas razones: la primera es que como necesitamos alta rendimiento a bajo coste y el *bottleneck* de Unity en muchos casos es el renderer, que nosotros vamos a sustituir por el nuestro propio. Otros motores, como podría ser Unreal, son mucho más capaces gráficamente sin modificarlos y tienen muchas más funcionalidades pre-programadas. Esto sería un problema para nuestro desarrollo ya que

necesitamos que el *engine* sea tan liviano como sea posible y contenga solo lo estricto y necesario para que nuestro juego corra exactamente como queremos que lo haga. Este paradigma y la larga experiencia que tiene nuestro equipo con el motor hicieron que fuera la elección obvia.

Repositorio

Nuestro repositorio estará alojado en GitLab. Estuvimos debatiendonos entre GitHub y GitLab ya que son las dos opciones sin coste con las que podíamos trabajar. No valoramos ninguna más porque estos dos ya tienen todas las features que necesitamos, repositorios privados, espacio ilimitado y soporte para LFS. Al final nos decantamos por GitLab porque toda su página web desde donde controlas el repositorio y todas sus opciones son mucho más bonita, claras, entendedoras y está mejor organizado, lo que nos ayudará mucho en nuestro *workflow,* ya que prevemos muchos problemas con el repositorio, sobretodo con los artistas que nunca han trabajado con uno.

Git GUI

Trabajaremos con GitKraken para manejar el repositorio desde nuestros ordenadores. Usamos este cliente porque era el más simple, bonito y fácil de entender de todos los clientes de Git a nuestro parecer. Otras compañías ofrecen opciones más técnicas y con muchas más opciones, pèro vimos que ninguna de estas opciones son cosas que no podamos hacer por comandos de windows o desde GitLab. Por ello nos decantamos por esta opción para que fuera mucho más entendedor y amigable para los artistas y así minimizar los errores al subir los cambios al repositorio ya que pueden resultar en trabajo perdido.

Documentación

Para toda nuestra documentación usamos la Google Workspace suite. Valoramos otras opciones para los documentos de texto como Coda, donde es más fácil redactar documentos que sean más agradables a la vista y donde la información está mejor referenciada y ordenada con menos esfuerzo. El primer problema que vimos con Coda u otras herramientas es que tiene un tamaño limitado y para ampliarlo hay que pagar suscripción, además estábamos preocupados por su seguridad, mientras que con Google no tenemos esos miedos. Por último, muchas de estas herramientas no tienen soporte para presentaciones o hojas de cálculo, lo que nos obliga a tener muchas herramientas distintas, cuando en Google lo podemos tener todo centralizado.

Producción
Para la producción, las tareas y la organización de sprints, usamos Google Sheets. Nos encontramos con el mismo problema que con la documentación, necesitábamos una herramienta que nos permitiera sacar gráficos, calcular estadísticas, repartir tareas y llevar una contabilidad de las horas. Muchas herramientas como Trello o Favro solo cumplen algunas de estas necesidades, mientras que como Google sheets es altamente programable podemos adaptarlo a nuestras necesidades específicas. El único problema que teníamos con él es que otras personas del equipo podrían modificar las funciones por error, pero como dispone de un historial de versiones eso no será un gran problema.

Comunicación
Para mantener el equipo informado y facilitar vías de comunicación usaremos Discord. Usamos Discord porque es la alternativa más completa que hemos podido encontrar en el mercado, otras como Skype no tienen tanta funcionalidad, mientras que en Discord podemos crear salas de texto, grupos, salas de voz, roles, manejar permisos, moderar canales… Esta flexibilidad es esencial en estos tiempos de pandemia para mantener al equipo tan unido como sea posible. Además es la herramienta que usamos todo el equipo para nuestra vida profesional y universitaria, así que todos estamos más familiarizados y pasamos más tiempo delante de él, esto facilita mucho la celeridad en la respuesta.

Desarrollo

Render Pipeline

Introducción

Alone es un juego singleplayer contemplativo de puzles, y tiene como foco de la experiencia la inmersión, no en la acción. Es por esto que nuestros artistas querían desarrollar entornos con mucha calidad visual y para ello requerían de características como: sombras en tiempo real para todos los tipos de luces, LODs (*Level Of Details*), *proxy volumes*, *lightmaps* en modo *shadowmask* y otras propiedades que les permitieran no quedarse limitados por culpa del renderizado.
Estas necesidades chocaron con las que desde producción y programación creíamos esenciales, como por ejemplo una tasa de refresco de al menos 60 fps (*frames per second*) en como mínimo el ordenador medio de steam de hace cinco años:

- Sistema Operativo: Windows 7 (64 bit)

- RAM: 8GB
- Velocidad de la CPU (intel): 3.0 / 3.29Ghz
- Número de CPUs físicas: 4
- Tarjeta de vídeo: Nvidia Geforce GTX 1060
- VRAM: 2047 MB
- Resolución del monitor primario: 1920x1080
- Resolución multi-monitor: 3840x1080
- Porcentaje de usuarios con micrófono: 99.84 percent
- Lenguaje: Simplified Chinese
- Espacio libre en el disco duro: 100-249GB
- Espacio total en el disco duro: 1+TB
- Casco de RV: Oculus Rift

Debido al nivel de calidad exigido, cumplir los requisitos artísticos marcados resulta caro en cuanto a rendimiento, ya que para tenerlos en Unity hace falta usar HDRP (*High Definition Render Pipeline*), y para cumplir los objetivos de rendimiento necesitamos URP (*Universal Render Pipeline*).
Por esta razón decidimos crear un *render pipeline* que uniera estos dos *pipelines* existentes para quedarnos con lo que necesitamos tanto de uno como del otro. Esto no es tarea fácil, ya que algunas de las características que el proyecto requería iban a ser costosas las implementáramos como las implementáramos, como por ejemplo muchos de los efectos de post procesado, así que también tuvimos que prescindir de algunas funcionalidades extra que nos hubiera gustado incluir con tal de mantener el rendimiento deseado.

Unity custom render pipeline

Lo primero y más importante es definir qué es exactamente un *render pipeline* en Unity:

- El *render pipeline* (RP) es el conjunto de clases en C# encargadas de decidir qué, cómo y dónde se pinta por pantalla, además de con qué configuración y orden se hace. Esto es especialmente importante cuando se tiene que lidiar con transparencias, sombras o postprocesados. Aun así, un *render pipeline* no sirve de nada sin unos *shaders* que dibujen lo que él necesita, así que según nuestro punto de vista, un elemento clave del *render pipeline* son los *shaders* que lo conforman.

Para que Unity sepa qué *render pipeline* debe usar necesitamos generar un *Scriptable Render Pipeline* que podamos asignar en *ProjectSettings>Graphics>ScriptableRenderPipeline*. Por lo tanto, deberemos generar

una clase que herede de *RenderPipeline* y defina la función *Render* para que unity sea capaz de llamarla.

```
public class CustomRenderPipeline : RenderPipeline{
    protected override void Render(ScriptableRenderContext context, Camera[] cameras) {}
}
```

Código 1

Como se muestra en el código 1, Unity le pasa a la función *Render()* un *array* de cámaras porque tal y como está pensado el motor, cada cámara se renderizada por separado. Es por ello que al igual que URP y HDRP, decidimos separar el *scheduling* de qué y cómo se renderiza cada cámara en una clase diferente a la que llamamos *CameraRenderer* que se encarga de renderizar cada cámara decidiendo en qué orden se pinta cada cosa.

En este código decidimos seguir el siguiente orden de renderizado:

1. Configuración de la cámara, preparación de buffers y preparación de la ventana de escena (solo en editor)
    a. Configuración de las luces
        i. Configuración de sombras
        ii. Configuración de luces
        iii. Renderizado de sombras
    b. Configuración de post procesados
    c. Configuración de la cámara y las *render textures*
2. Dibujado de la geometría visible
    a. Pintado de la geometría opaca
    b. Pintado de la geometría transparente
3. Dibujado de los *shaders* no soportados
4. Dibujado de *gizmos*
5. Dibujado de Post Procesado
6. Dibujado de *gizmos* (algunos no deben ser afectados por el post procesado)

A continuación hablaremos de los distintos pasos, a excepción de los puntos 1 y 5, que por su complejidad tendrán un apartado entero dedicado a cada uno más adelante.
Cada paso de nuestro renderer tiene un apartado dedicado a explicar en profundidad su funcionamiento a excepción de los pasos 2, 3, 4 y 6 que explicaremos ahora:

Dibujado de la geometría visible (paso 2)

- Para diferenciar entre geometría opaca y transparente usamos las herramientas que provee Unity en su editor, siendo estas los *SortingCriteria* y los *RenderQueue* que aparecen en cada shader como se muestra en la figura 5 para que el usuario pueda hacerle saber al *render pipeline* cuándo hay que pintarlo. De esta forma le quitamos al *renderer* el coste de decidir por sí mismo qué pintar cuándo ya que cada material puede tenerlo definido él mismo.

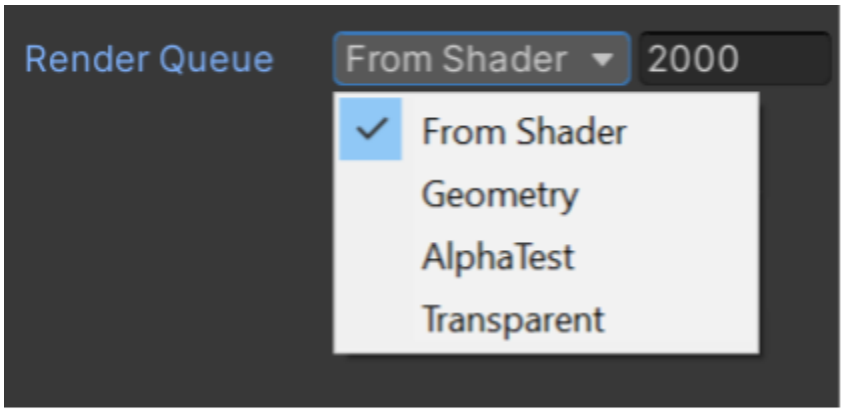


Figura 5; Ejemplo de la UI con las diferentes render queues

Dibujado de *shaders* no soportados (paso 3)

- El dibujado de los shaders no soportados podríamos obviarlo, ya que en realidad en las *builds* este paso lo saltamos porque no queremos que el usuario final vea partes rosas si algo estuviera roto. En editor sin embargo, pintamos todo lo que no esté soportado o no compile bien de un color rosa chillón para que los desarrolladores y artistas puedan detectar fácilmente si algo está mal. En la figura 6 se observa un ejemplo de shader no soportado.
  Esta es la misma técnica que siguie Unity con la excepción de que nosotros decidimos no pintarlos en *build* cuando Unity si.

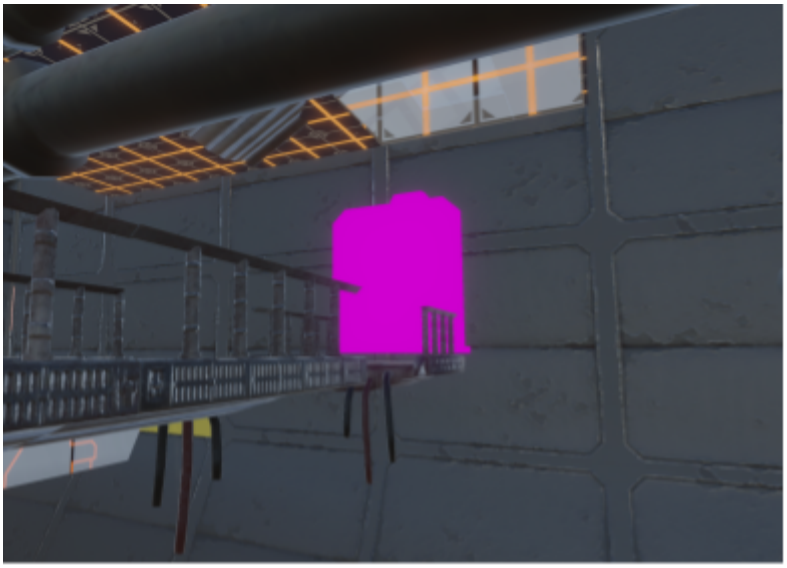


Figura 6: Visualización de missing shader

Dibujado de Gizmos y postprocesados (pasos 4 y 5)

- El dibujado de los post procesados (o post FX) decidimos partirlos en post y pre FX porque Unity provee de un *enum* llamado *GizmoSubset* que contiene estas dos variaciones. En la documentación oficial de Unity no hay explicación alguna de por qué esto es así, pero nos parece un detalle muy útil para nuestros programadores de herramientas ya que en Unity puedes escribir códigos con *gizmos* personalizados y poder elegir si se pintan antes o después de los post procesados puede ser muy útil, sobretodo teniendo en cuenta que tenemos post procesados como *bloom* o *blur* que podrían distorsionar los *gizmos*.

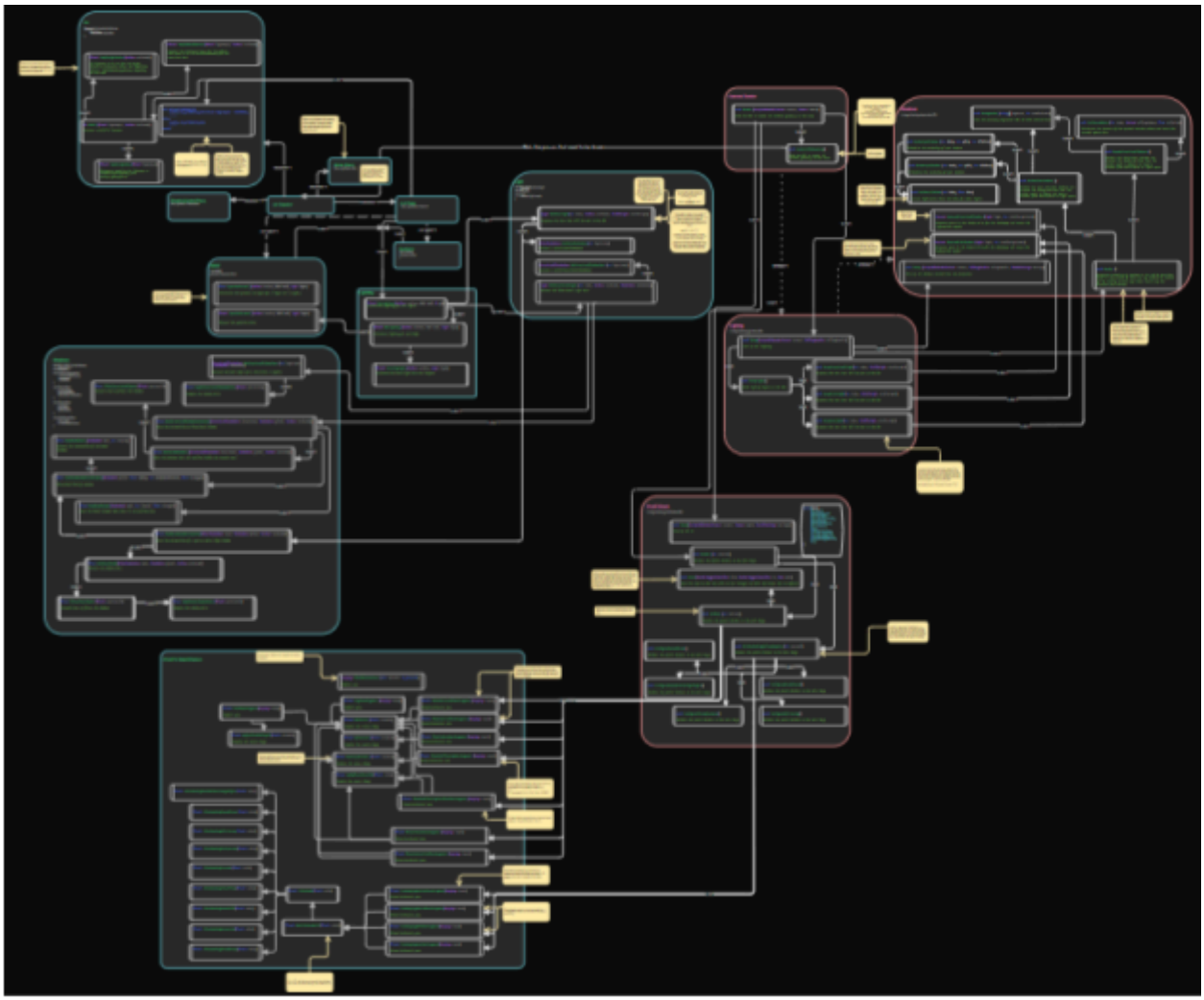


Figura 7: Esquema de todas las clases del renderer

Antes de entrar en los detalles de nuestro *renderer* hay ciertos conceptos que tenemos que aclarar:
Primeramente, la figura 7 es un diagrama que representa el flujo de llamadas y las responsabilidades de nuestro *renderer.* Creímos que un UML no era muy práctico para mostrar un *render pipeline* entero y además consideramos que no ayuda a entender la lógica que sigue el algoritmo de forma clara.

La segunda cosa que hay que aclarar es por qué hay tantas clases en el sistema y por qué muchas de ellas están duplicadas en la *Graphics Processing Unit* (GPU) y la *Central Processing Unit* (CPU). La razón de esto es que seguimos una filosofía de diseño muy marcada para desarrollar el *renderer*: toda la funcionalidad que requiere de más de una o dos funciones se convierte en su propia clase que toma toda la responsabilidad de esa área, por ejemplo existe la clase *Lighting* que se encarga de las luces, pero esta instancia una clase *Shadows* ya que las sombras son una parte suficientemente compleja como para encapsularla y desacoplarla.
Este funcionamiento nos facilita mucho adaptar el *renderer* a cualquier escenario ya que si hay que modificar por ejemplo la sombras, es tan sencillo como montar una clase que implemente las mismas funciones que Shadows para que *Lighting* pueda llamarla en vez de sustituir el renderer entero.
Esto nos ha sido extremadamente útil para cosas como llevar el juego a consolas, ya que para cumplir con las especificaciones concretas de cada plataforma habría que expandir el renderer muchísimo, cuando con este sistema nos vale con implementar la versión "_Switch" de las pocas clases que necesitan modificaciones para lanzar el juego en Switch.

Lighting

La iluminación es el factor clave de nuestro *render pipeline*. Necesitábamos poder iluminar las escenas de forma realista pero a la vez barata. Esto requirió decidir qué queríamos implementar y qué tenía que quedarse fuera, por tanto decidimos que soportaríamos sólo tres tipos de luces: *direccional lights*, *point light* y *spot lights.* A pesar de esto, como queremos que sean realistas y se sientan vivas, a diferencia de URP nosotros decidimos soportar iluminación en tiempo real para los tres tipos de luces.
Pero aún había que decidir cómo calcularíamos estas luces. La respuesta a esta pregunta la encontramos en URP, que implementa una versión de BRDF (*Bidirectional Reflectance Distribution Function*) que es bastante realista, pero a su vez muy barata computacionalmente; el caso ideal para nuestro proyecto.
Por último, como queremos realismo a poco coste, no podemos permitirnos que los objetos dinámicos se vean muy distintos a los estáticos para mantener el realismo, pero tampoco puede ser todo dinámico para mantener el rendimiento deseado, así que la solución es *bakear* la iluminación para los objetos que puedan ser estáticos y soportar las siguientes features para los que deban ser dinámicos: *LightProbes*, *LightProxyVolumes*, *ShadowMask*, *OclussionProbes* y *OcclusionProbeProxyVolume*

Así pues desglosaremos este apartado en los tipos de luz que soportamos:

Luz direccional

La figura 8 muestra un esquema de las clases que tienen un papel en renderizar las luces direccionales.

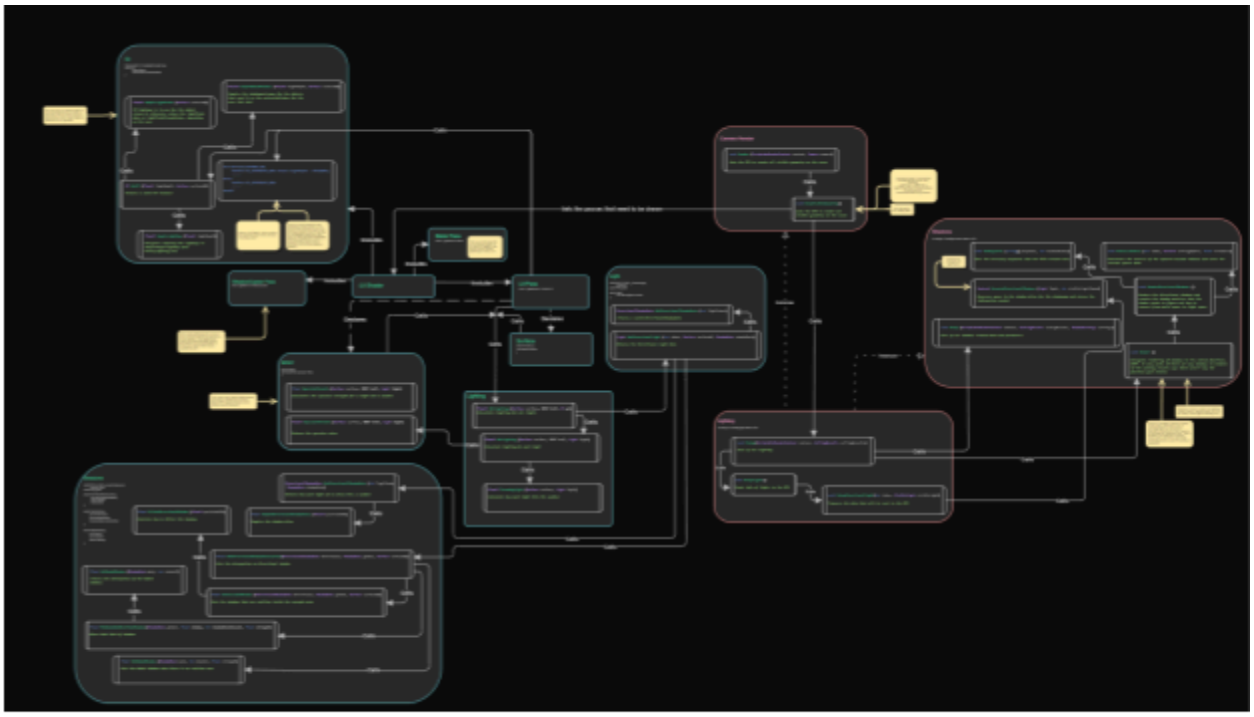


Figura 8: Esquema de las clases involucradas en las luces direccionales. Azul: shaders en hlsl, Rojo, clases en C#

Nuestro algoritmo empieza en la clase *CameraRender*, que tiene una función *Render()* que se llamará para cada una de las cámaras que hay en la escena. Esta llamada la hace el *asset* de *render pipeline* de Unity y es el punto en el que Unity nos cede el control a nosotros para renderizar todo lo necesario. Así mismo, la función *Render()* acaba llamando a *DrawVisibleGeometry()*, que es la función que realmente prepara todo lo necesario y le pide a la GPU que renderize todo lo que necesitemos.

Todo esto lo hace en dos pasos:
Primero para avisar a la gráfica de qué y cómo tiene que pintarlo, tenemos que definir ciertas banderas para los shaders. Las banderas que definimos nosotros se encuentran en el código 2:

```
PerObjectData.ReflectionProbes | PerObjectData.LightData |
PerObjectData.LightMaps | PerObjectData.ShadowMask |
PerObjectData.LightProbes | PerObjectData.OcclusionProbe |
PerObjectData.OcclusionProbeProxyVolume | PerObjectData.LightIndices
```

Código 2

Que la gráfica sepa cómo tiene que pintar las cosas no es suficiente para que las renderize, es por eso que en Unity se genera un contexto a partir de cada cámara que contiene la información necesaria y al que se le va añadiendo toda la información que generamos en el renderer.
Aclarado esto, el segundo paso para pintar la geometría consiste en llamar a las funciones *context.DrawRenderer*(*CullingResults*, *drawingSettings*, *filteringSettings*) y *context.DrawSkybox(camera)* para completar la escena. En este caso el orden es muy importante ya que si lo último que pintamos es el *skybox* taparemos todo lo que ya hemos pintado que no escriba en la textura de profundidad (*DepthTexture);* es decir, taparemos todos los elementos transparentes, que son aquellos que no escriben en la textura de profundidad. Así pues, para evitar esto el orden que seguimos es el estandard en la industria:

1. *DrawRenderes* con configuración para geometría opaca
2. *DrawSkybox*
3. *DrawRenderer* con configuración para geometría transparente

Después de estas llamadas el *renderer* ya renderiza, pero aún no tenemos ni sombras ni iluminación; solo pintamos siluetas sólidas. Para ayudar a entender esta diferenciación, la figura 9 muestra cómo se vería nuestro robot con tan solo estos pasos (la silueta verde de la izquierda) en comparación con cómo se ve realmente en nuestro renderer (el robot de la derecha).

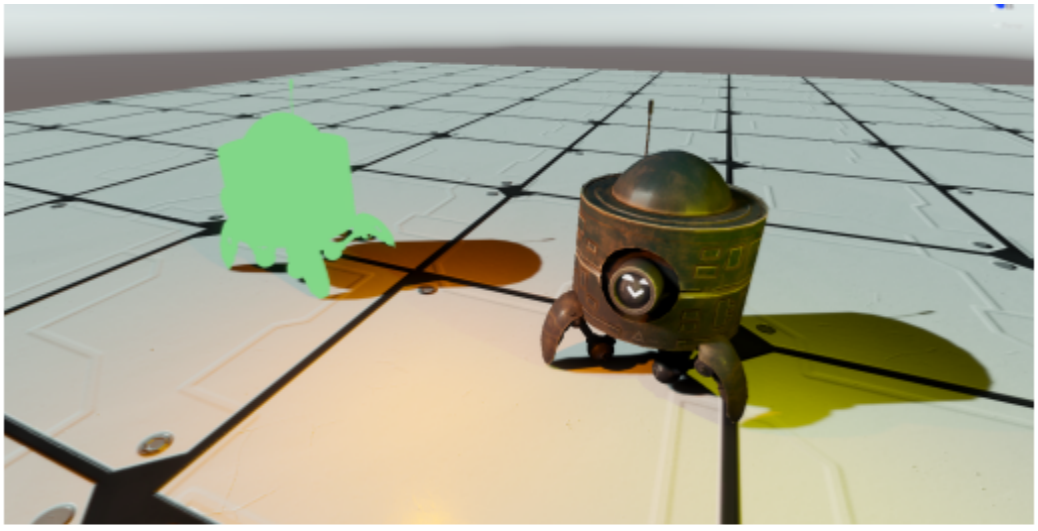


Figura 9: Derecha, robot con un shader Unlit. Izquierda: mismo robot con un shader Lit

Para pintar la escena de una forma más entendible y realista (como el robot de la derecha) necesitamos muchas clases y operaciones para preparar todos los datos necesarios para que los *shaders* puedan percibir el entorno y pintarse acorde con

ello, ya que en cada *shader* de forma nativa sólo tenemos la información relativa a la malla que vamos a pintar, no tenemos información de la escena, sombras, resoluciones, lightmaps... (es por esto que el robot de la izquierda no es más que una silueta). Toda esta información tendremos que prepararla, empaquetarla y enviarla a la GPU para poder usarla y conseguir el acabado realista.

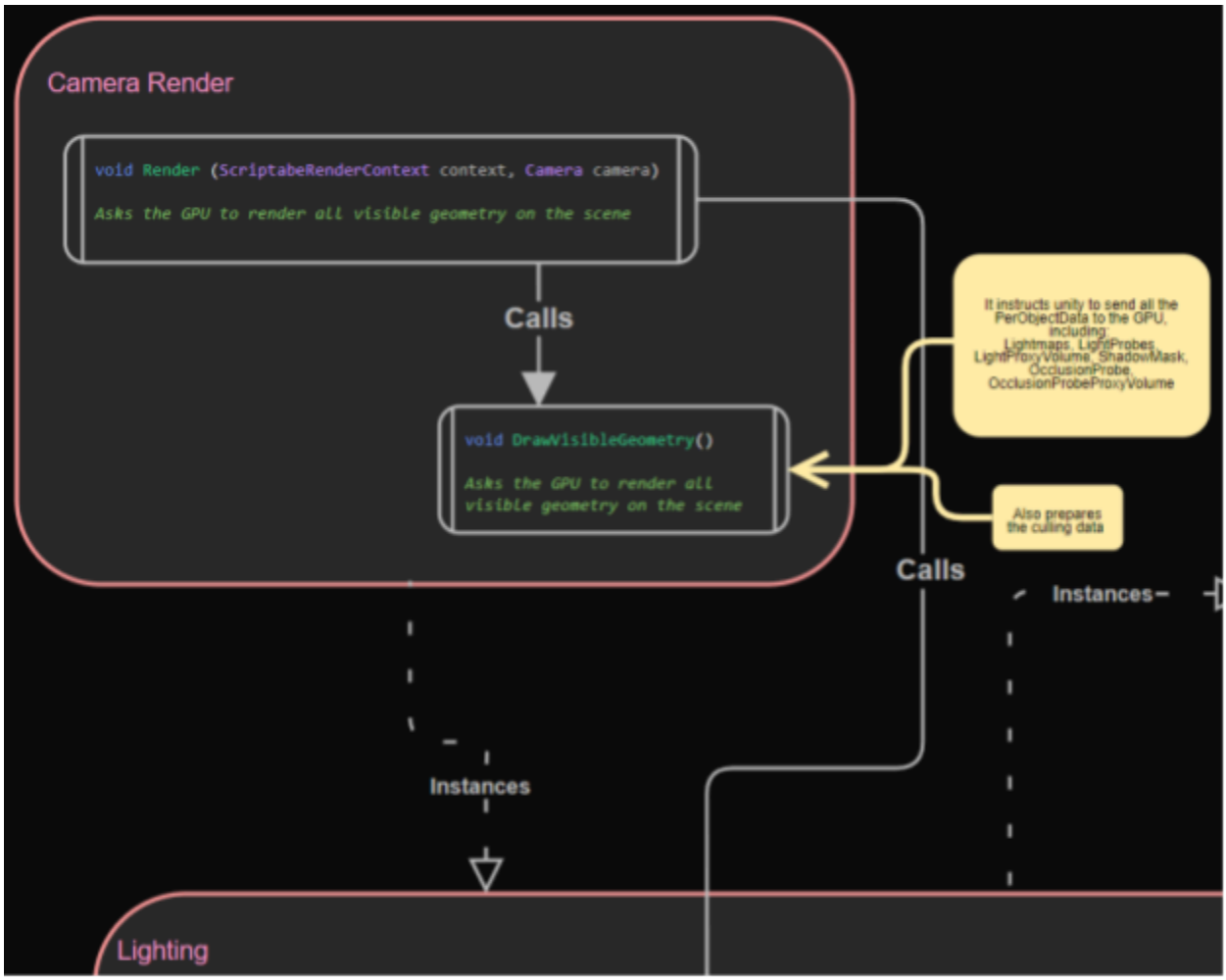
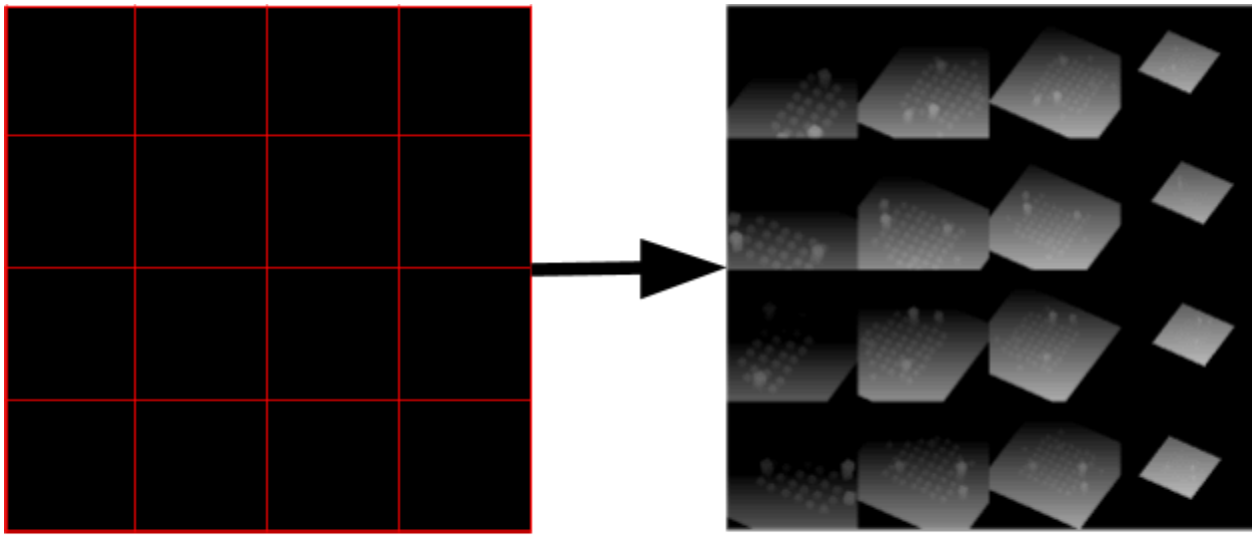


Figura 10: Clase de C# Camera render con sus funciones relevantes

Para ello lo primero que hace nuestro *renderer* es llamar a la función *Setup()* de la clase *Lighting*, que es la encargada de preparar y enviar a la GPU toda la información relacionada con la iluminación. Necesitamos también la función *CameraRender.DrawVisibleGeometry()* encargada de preparar los *CullingResults.* Esta función encapsula la información relacionada con qué objetos son visibles y qué objetos no, permitiéndonos así dar soporte a *OclusionCulling*, una característica clave de Unity y de la mayoría de *renderers* que permite abaratar en gran medida el renderizado.

La clase *Lighting* debe dejar preparada la información tanto de las luces como de las sombras, que son los dos componentes que definen la iluminación.

En este punto, como solo nos preocupamos de renderizar las luces direccionales, preparar las luces es uno de los pasos más sencillos: la función *Setup()* delega el trabajo a *SetupLights()*, que a su vez delega este trabajo a *SetupDirectionalLight()*. Esto puede parecer demasiado rebuscado, ya que técnicamente sólo la función *SetupDirectionalLight()* está calculando algo, pero este tipo de estructura nos permite tener un bajo acoplamiento y una alta cohesión del código, ya que cuando tengamos que preparar la información de los demás tipos de luces que no son direccionales en el futuro, podrán tener su propia función que *SetupLights()* llamará. De esta manera, añadir más tipos de luces supone un cambio mínimo en el código agilizando el trabajo y asegurando la flexibilidad y mantenimiento del código a la hora de añadir o quitar tipos de luces, ya que todo está encapsulado por responsabilidades.
Cabe decir que la encapsulación no es perfecta porque como podemos ver en la figura 7, donde se muestra todo el esquema de clases de C#, ya que la función *SetupDirectionalLight()* tiene que llamar a *ReserveDirectionalShadows()*, que reserva espacio en la textura donde pintaremos nuestro *shadowmap* para cada una de las luces y sus cascadas.
Aunque esto pudiera parecer responsabilidad de *Shadows*, aquí no estamos generando ninguna textura ni datos de las sombras perse, solo creamos los datos que definirán dónde estará cada uno de los *tiles* en la textura ya que en este punto del código tenemos la información de las luces, datos que no son necesarios para la clase *Shadows,* y por tanto no debemos delegar esta función a la clase *shadows*, sinó que es ReserveDirectionalShadows*()* quién debe ocuparse como dicta el "principio de experto en información" de la programación orientada a objetos (OOP). Además esta información debe estar ya preparada para el paso de generar los datos de las sombras.



Ejemplo de reserva de espacio en el shadowmap

Ejemplo de Shadowmap final, con todas las cascadas renderizadas

Figura 11

Con esto tenemos todos los datos necesarios para empezar a trabajar en las sombras.

Para ello *Lighting.Setup()* llama a *Shadows.Setup(),* que prepara todos los datos y seguidamente a *Shadows.Render()*, que renderiza las sombras llamando a *RenderDirectionalShadow()* y *SetCascadeData()*.

Tanto para las sombras como para las luces usamos siempre funciones como *RenderDirectionalShadows()* y no *RenderShadows()* porque el *renderer* también tiene que soportar luces y sombras no direccionales que explicaremos más adelante.

Separar cada tipo de renderizado en su propia función tiene varios propósitos: mantener la legibilidad del código, la alta cohesión y el bajo acoplamiento. De esta forma, se consigue un código mucho más flexible permitiéndonos en un futuro modificar alguna parte del código sin afectar al resto.

Otra observación a señalar es que, si seguimos el flujo inverso de nuestro *renderer* que acabamos de presentar, *shadows.Render()* se llama desde *Lighting.Setup()*, no en el paso de *render*. Esto puede parecer extraño, pero la razón de esto es que nuestro *renderer* es un *forward renderer y* por tanto los *lightmaps* con la información de profundidad tienen que estar previamente renderizados y preparados para el paso de renderizar la geometría, por lo tanto las sombras son parte de los datos que hay que preparar en *Setup()* antes de pintar la geometría.

Por último es interesante mencionar que *RenderDirectionalShadows()* solo renderiza las luces, pero crea las matrices de transformación de espacio de mundo a espacio de luz a partir de la función *SetCascadeData()*. De esta forma resulta muy sencillo sustituir el sistema de cascadas de la iluminación para mejorar su rendimiento.

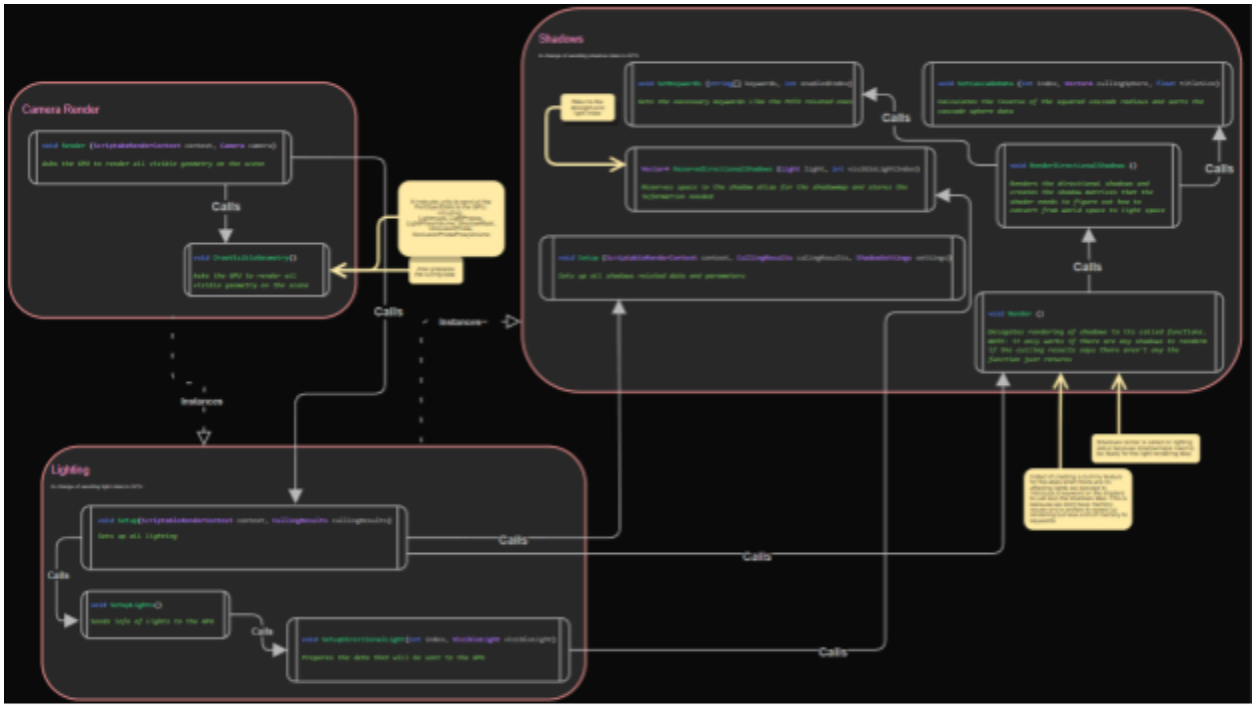


Figura 12: Clases de C# involucradas en las luces direccionales (entre otros)

En este momento, si omitimos los otros tipos de luces y algunas funcionalidades que veremos más adelante, llegamos al final del *callStack.* En este punto ya hemos hecho todo el trabajo necesario en la CPU y es el turno de la GPU. Durante todo este proceso hemos estado preparando datos y enviándolos a la GPU para que los *shades* en la gráfica los puedan usar. Para entender la función de la GPU seguiremos el ejemplo de nuestro *shader* principal, *Lit shader*, el cual es muy versátil y con la interfaz casi calcada al de URP.

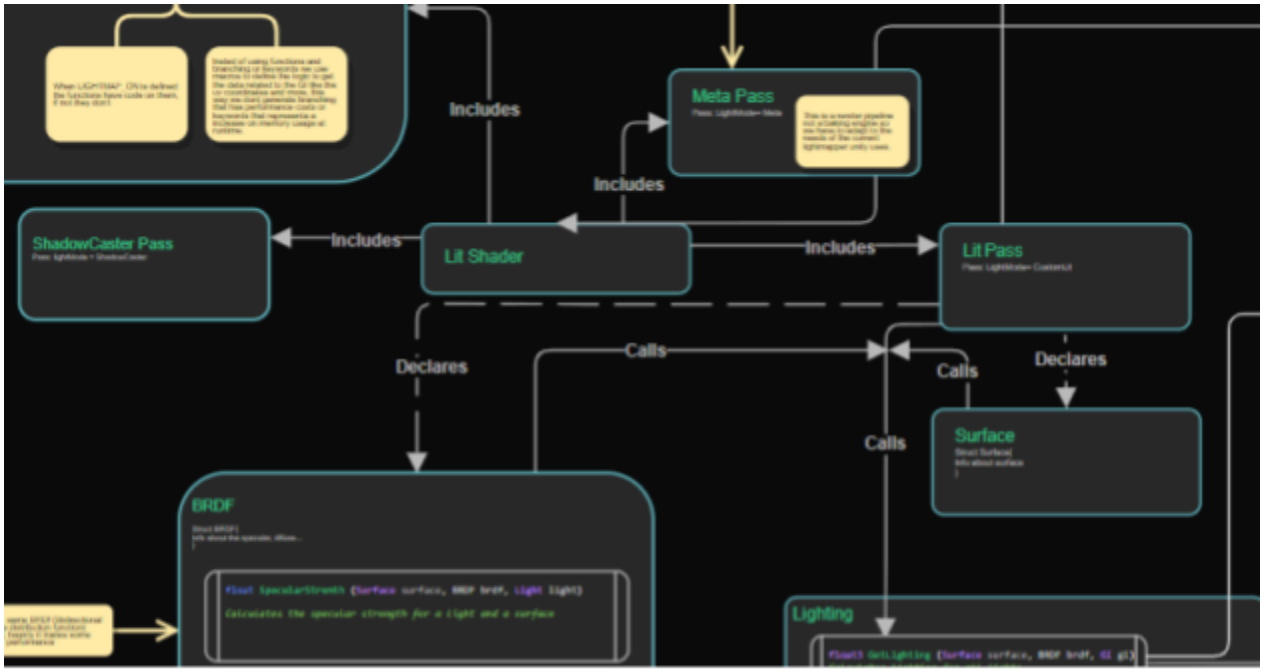


Figura 13: Lighting.hlsl

Hasta el momento, nuestro *shader* consta de tres pasadas: *ShadowCasterPass, LitPass* y *MetaPass*.

ShadowCasterPass

La primera pasada es *ShadowCasterPass*. Como su nombre indica, es la encargada de pintar la profundidad de cada fragmento del objeto en la textura de profundidad. Esta es una pasada muy simple: ya hemos preparado las matrices de transformación, tanto de espacio de mundo a espacio de luz para posicionar la cámara virtual, como las matrices de cascadas para saber exactamente en qué superficie de la textura pintar.

Es importante hacer esta pasada primero como ya explicamos en *shadows* para que la información esté en el *lightmap* cuando lleguemos a la pasada de *LitPass*.

LitPass

En el *LitPass* iluminaremos el objeto y lo texturizaremos. Como en este apartado solo nos interesa la iluminación, trataremos solo eso y dejaremos el texturizado y la lógica del *shader* para futuros apartados.

Así pues, para iluminar un objeto dividiremos las funcionalidades y responsabilidades en los *shaders* de la misma forma que hemos hecho en las clases de C#, pero como todos los shaders están escritos en HLSL no necesitamos declarar distintas clases; podemos escribir la funcionalidad en ficheros diferentes e incluirlo unos ficheros en otros. De esta forma si necesitáramos substituir alguna funcionalidad en HLSL sería más sencillo aún, porque no haría falta modificar la declaración de la clase a modificar en las clases que la usan, con cambiar el fichero en la ruta del *include* o cambiar la propia ruta todo seguiría funcionando correctamente.

El primer paso para iluminar algo con nuestras librerías es llamar a la función *GetLighting()*, que podemos encontrar en el fichero *Lighting*. Esta función se encarga de controlar la iluminación. Para llamar a esta función el *shader* primero debe pedir y montar cierta información necesaria: la primera es la información sobre la superficie (*Surface).* Esta es una estructura de datos que hemos definido para encapsular la información necesaria para iluminar un fragmento.

En el código 3 podemos observar la estructura en cuestión.

```
#ifndef CUSTOM_SURFACE_INCLUDED
#define CUSTOM_SURFACE_INCLUDED

struct Surface {
  float3 position;
  float3 normal;
  float3 interpolatedNormal;
  float3 viewDirection;
  float depth;
  float3 color;
  float alpha;
  float metallic;
  float occlusion;
  float smoothness;
  float fresnelStrength;
  float dither;
  uint renderingLayerMask;
};

#endif
```

Código 3

Lo siguiente que necesita preparar es otra estructura llamada *BRDF*. Como ya hemos mencionado anteriormente, usando este algoritmo se consigue un acabado bastante realista a un coste muy bajo. Esta estructura encapsula todos los datos necesarios para calcular la luz siguiendo este algoritmo y tiene el aspecto mostrado en el código 4:

```
struct BRDF {
  float3 diffuse;
  float3 specular;
  float roughness;
  float perceptualRoughness;
  float fresnel;
};
```

Código 4

Estos datos se pueden extraer de la estructura *Surface*, pero debido a su complejidad de cálculo decidimos preparar un seguido de funciones encargadas de hacer todos estos cálculos en BRDF.hlsl y simplemente llamando a *GetBRDF* con una surface nos retorna una estructura BRDF preparada. Todas estas funciones podéis encontrarlas en el anexo X.

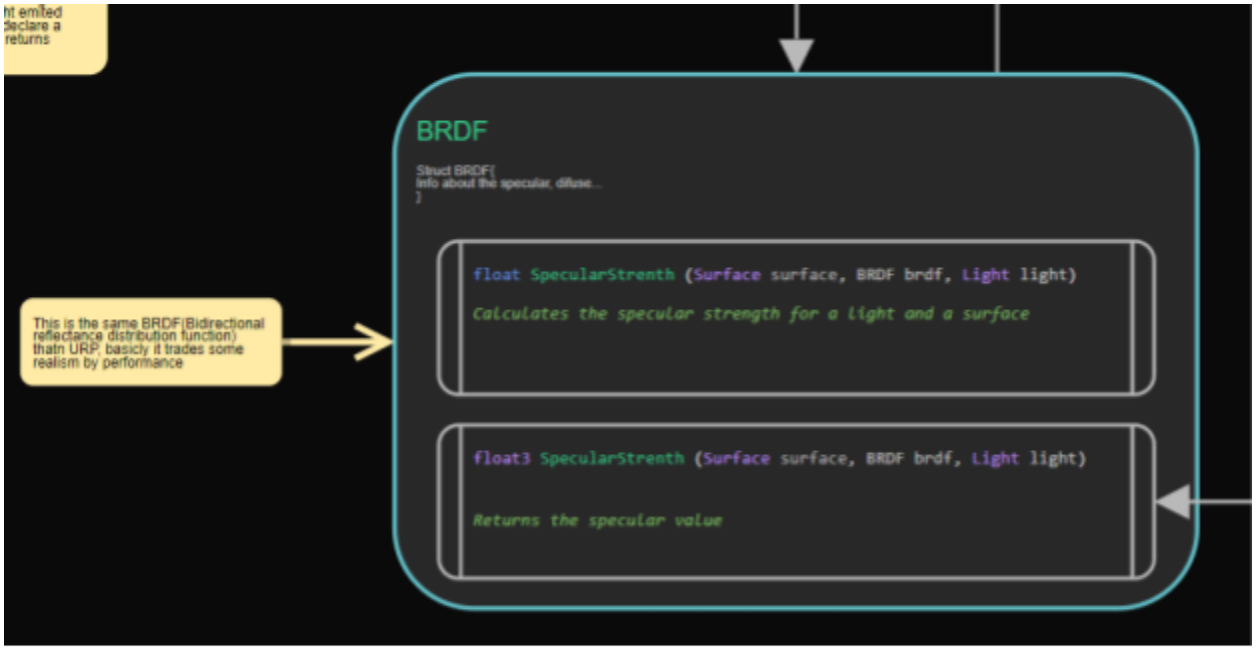


Figura 14: BRDF.hlsl con sus funciones

Es necesario una última estructura antes de poder llamar a la función GetLighting*()*. Esta estructura es GI, diminutivo de *GlobalIllumination*, y engloba todo lo necesario para leer y hacer cálculos relacionados con los *lightmaps*, por ello también tenemos un fichero GI.hlsl que engloba un seguido de funciones para realizar los cálculos pertinentes.
Para preparar estos datos seguiremos un patrón similar al usado para BRDF, tendremos una función *GetGI*, que a su vez llama a las funciones:

- *SampleLightmap()*: encargada del muestreo del *lightmap* haciendo uso de *EntityLighting.hls*l de Unity
- *SampleLightProbe()*, que se encarga del muestreo de los *lightprobes* y los *proxy volumes*. Esta función solo se llama si el objeto no tiene *lightmaps* definidos, y por tanto al no tener *lightmaps* solo podemos usar la información de los *lightprobes* para la luz espectacular ya que no requiere información a cerca de la dirección de la luz, la iluminación especular proviene de rebotes en toda la escena.
- *SampleBakedShadows()*: se encarga del muestreo de las sombras del *shadowmap*.

Como se puede observar en el código 5, en vez de separarlo en funciones, en este caso usamos un seguido de macros. Esto es así, porque se trata de funcionalidades muy básicas y fáciles de leer, como por ejemplo conseguir

las UVs de *lightmap*, por tanto al usar macros en vez de funciones nos ahorramos hacer *branching* o *keywords* para los casos en que hay o no hay GI.

```
#if defined(LIGHTMAP_ON)
  #define GI_ATTRIBUTE_DATA float2 lightMapUV : TEXCOORD1;
  #define GI_VARYINGS_DATA float2 lightMapUV : VAR_LIGHT_MAP_UV;
  #define TRANSFER_GI_DATA(input, output) \
     output.lightMapUV = input.lightMapUV * \
     unity_LightmapST.xy + unity_LightmapST.zw;
  #define GI_FRAGMENT_DATA(input) input.lightMapUV
#else
  #define GI_ATTRIBUTE_DATA
  #define GI_VARYINGS_DATA
  #define TRANSFER_GI_DATA(input, output)
  #define GI_FRAGMENT_DATA(input) 0.0
#endif
```

Código 5

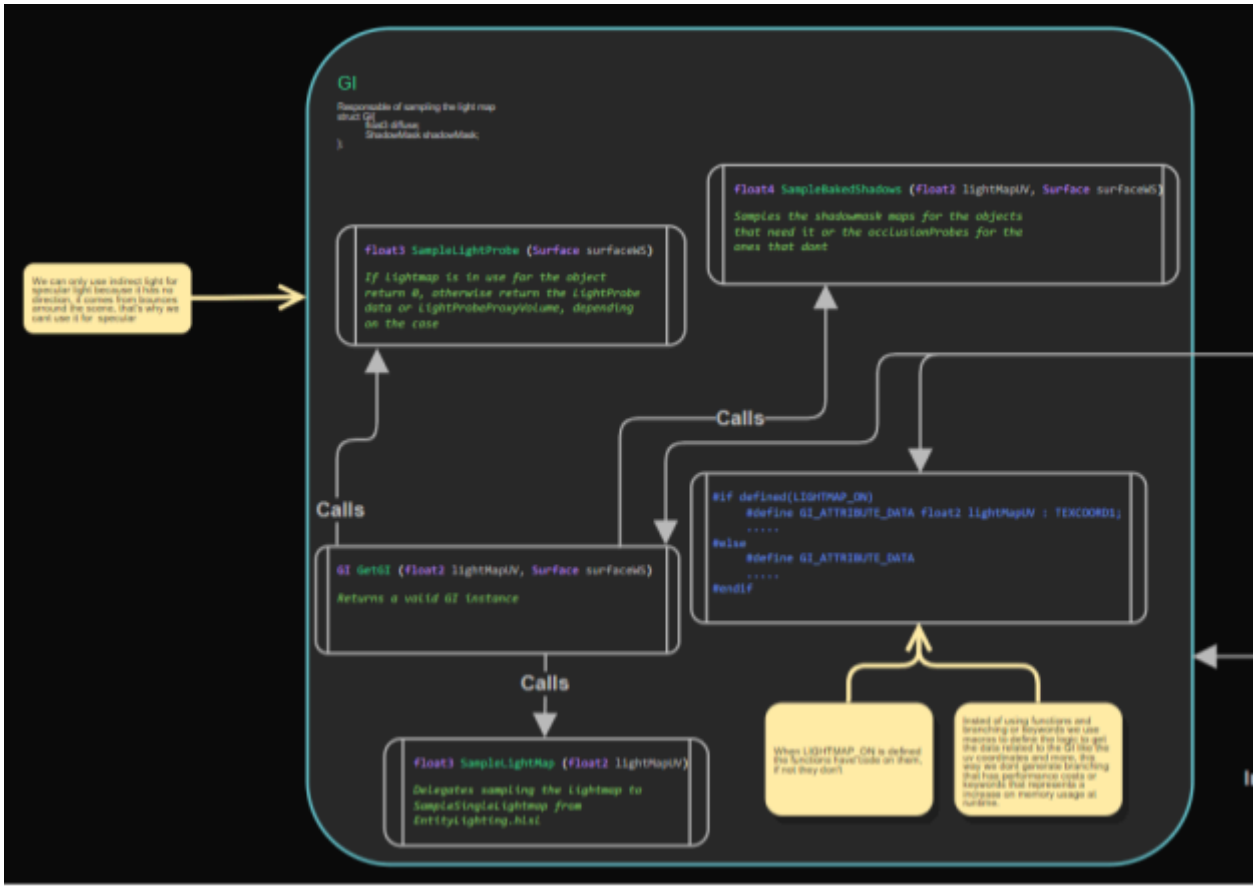


Figura 15: GI.hlsl con sus funciones i defines

Una vez construidas las estructuras de *Surface*, *BRDF*, y *GI*, llega el momento de llamar a la función *GetLighting(),* que es la encargada de devolver toda la iluminación.
Esta *a su vez* llama a otra función con el mismo nombre pero diferente firma encargada de calcular la iluminación por cada una de las luces que afectan al objeto.

De nuevo, el cálculo de las luces está separado en distintas funciones según el tipo de luz, manteniendo así el bajo acoplamiento y la alta cohesión como ya hemos explicado en repetidas ocasiones. Así pues, para el caso que nos atañe ahora, nos centraremos en la función *GetDirectionalLight()* únicamente. *GetDirectionalLight()* llama a *GetDirectionalShadowData()* para conseguir la información del *lightmap*. Para eso creamos *Shadows.hlsl*, uno de los ficheros más extensos de nuestros *shaders. Shadows.hlsl* define diferentes estructuras y las funciones encargadas de conseguir y procesar todos estos datos. En la figura 16 podemos observar las funciones que conforman a Shadows.hlsl.

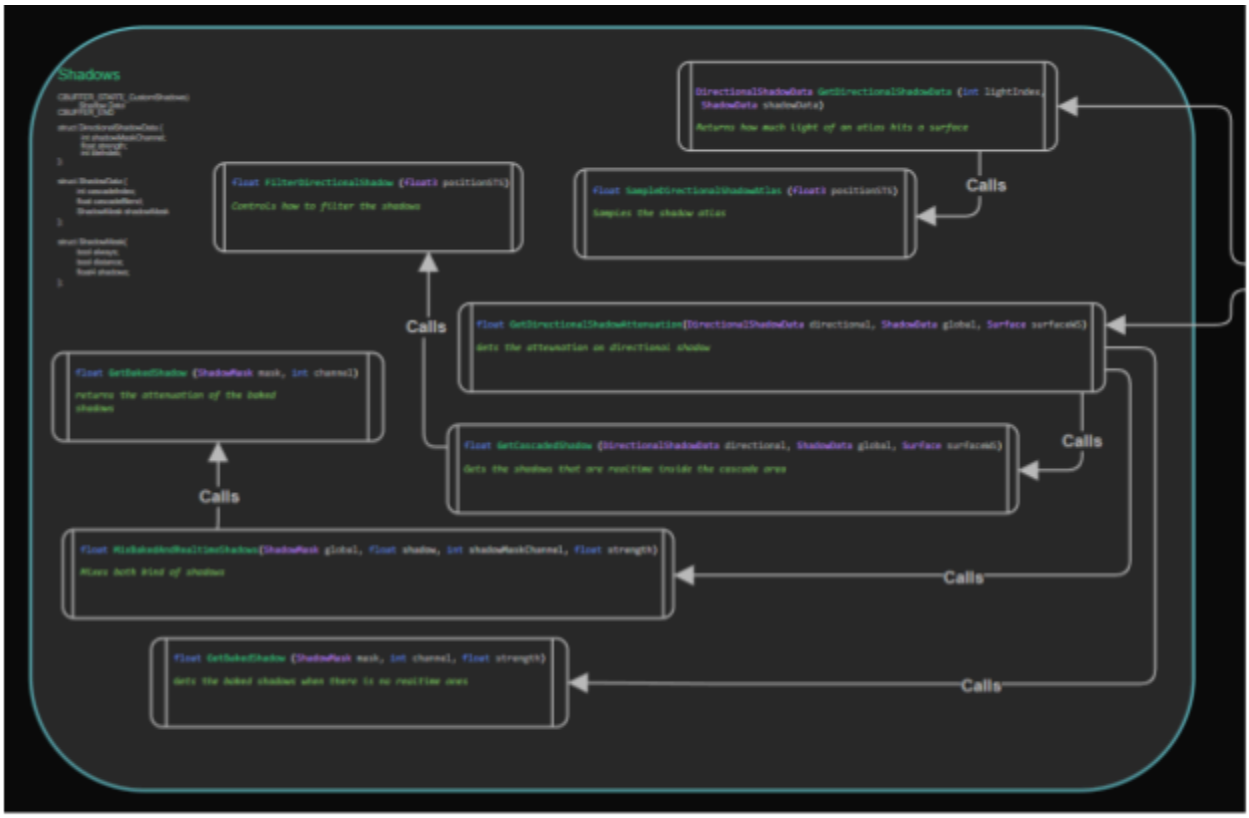


Figura 16: Shadows.hlsl con todas sus funciones, encargadas de todo lo relacionado con las sombras en la GPU

También puedes ver las estructuras que definimos en Shadows.hlsl en el código 6.

```
struct ShadowMask {
  bool always;
```

```
  bool distance;
  float4 shadows;
};
struct ShadowData {
  int cascadeIndex;
  float cascadeBlend;
  float strength;
  ShadowMask shadowMask;
};
struct DirectionalShadowData {
  float strength;
  int tileIndex;
  float normalBias;
  int shadowMaskChannel;
};
```

Código 6

Para entender el uso y la razón de las funciones repasamos el flujo de código: Primero, desde *Lighting.hlsl* se hacen dos llamadas. Una es *GetDirectionalShadowData()*, que muestrea el *shadow atlas* llamando a *SampleDirectionalShadowAtlas()* y que con esta información devuelve cuánta luz llega a la superficie según el atlas. La otra es *GetDirectionalShadowAttenuation()*, ya que necesitamos que las sombras se puedan atenuar, no sólo por razones artísticas sino también porque nuestro *renderer* soporta *shadow cascades*, por lo que para cada luz renderizamos la textura de profundidad desde diferentes distancias para tener diversas resoluciones. Esto es así ya que para fragmentos más cercanos necesitamos más detalle que para los fragmentos más lejanos. Para que los cortes no sean obvios definimos 2 formas de juntar estas cascadas: hacer un difuminado gradual o hacer un efecto de *dither* entre cascadas.
Las funciones responsables de hacer todos estos cálculos son *FilterDirectionalShadow() que* es la encargada de filtrar las cascadas, y *GetCascadedShadows()* que es llamada por *FilterDirectionalShadow()* y devuelve las sombras en tiempo real dentro de la zona de cascada.

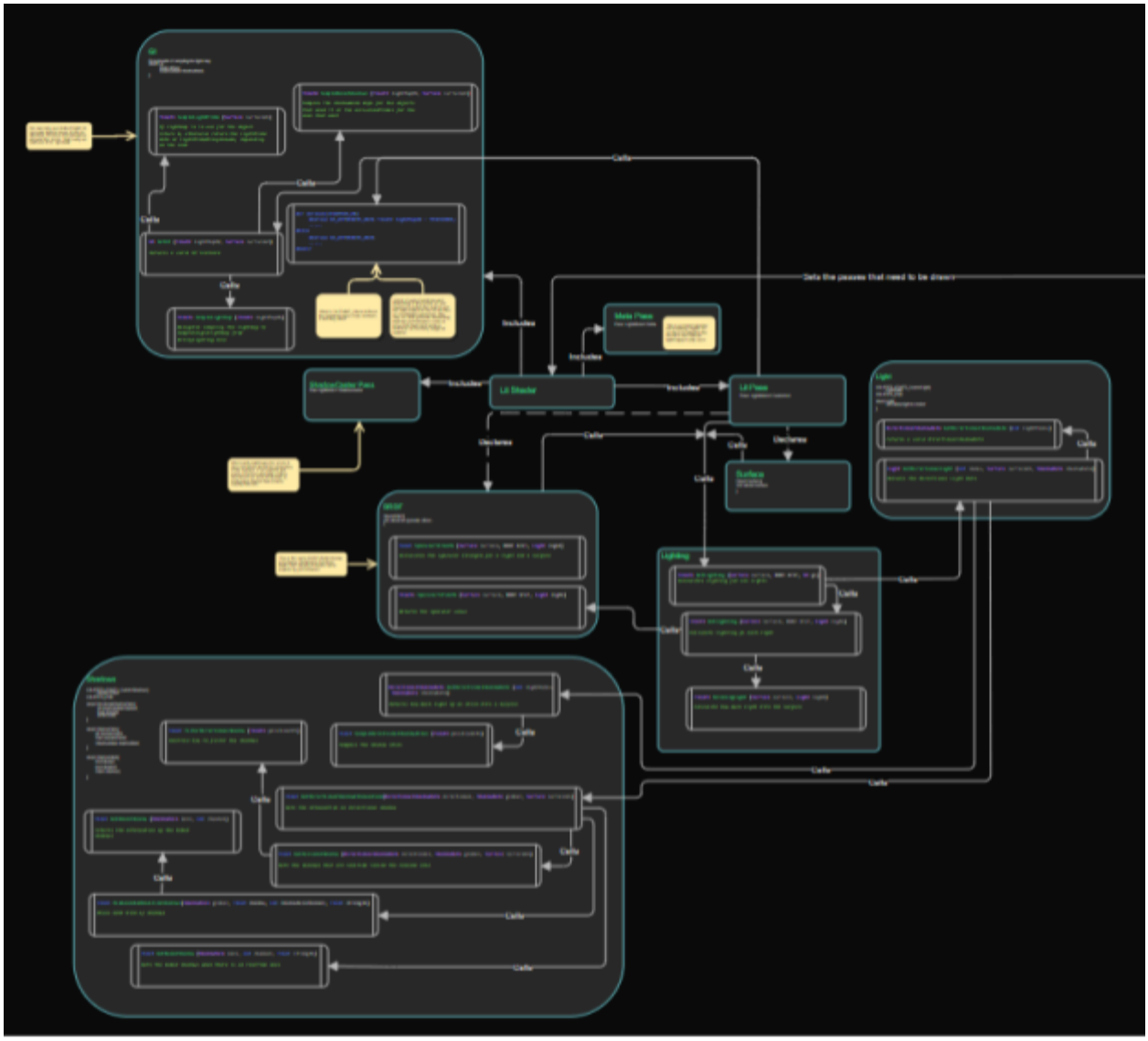


Figura 17: Todas los documentos hlsl relacionados con las luces direccionales

Con esto hemos acabado de repasar todas las clases relacionadas con iluminar con luces direccionales, pero aún nos falta una pasada por mencionar: el *MetaPass*.

MetaPass

El *MetaPass* es una pasada que es necesario definir para que el *lightmaper* de Unity sea capaz de saber qué clase de luz, con qué color y con qué forma emiten las superficies que vas a *bakear*. Esta pasada se llama durante el *lightmapping*.

En la figura 18 se muestra el resultado de esta pasada.

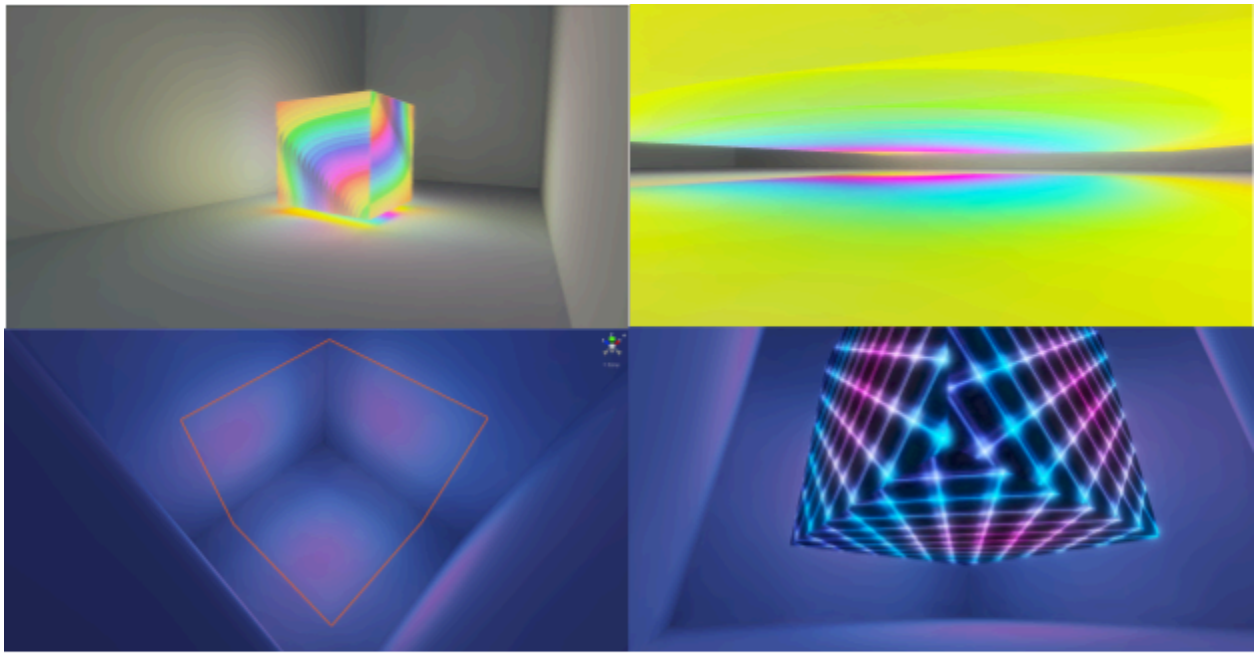


Figura 18: Lightmaps con contribución emisiva de objetos en la escena gracias al MetaPass

Con esto concluímos la iluminación con luces direccionales. En los siguientes apartados veremos cómo modificar estas bases para soportar otros tipos de luces, pero esta es la funcionalidad *core* de nuestro renderer.

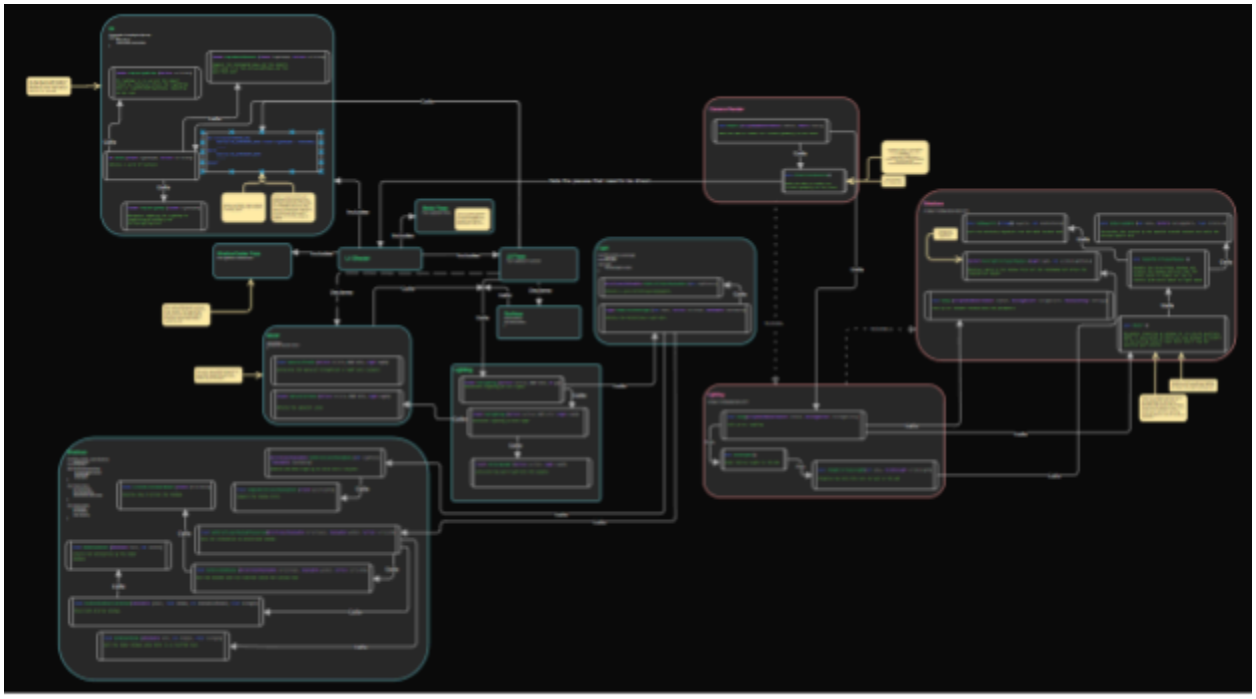


Figura 19: Esquema de clases

Otras luces

Una vez tuvimos esta base, añadir soporte para otros tipos de luces y sus sombras no representó tanto trabajo.

Lo primero fue definir qué tipos de luces vamos a soportar y si tendrán sombras en tiempo real o no. En nuestro caso implementamos *point* y *spot light*, ambas con sombras en tiempo real.
Esta decisión está motivada por dos factores: el primero es que damos soporte a este tipo de luces porque son muy útiles y además toda la interfaz para modificar sus parámetros ya está definida en Unity a excepción de un pequeño *slider* para la *spot light*.
El segundo motivo es que les damos sombras en tiempo real a diferencia que URP para poder acentuar ciertos puntos del juego.
Las *spot lights* son muy útiles para crear luces situacionales, pero para poder controlarlas bien y tener una variedad como la que vemos en la figura 20 necesitamos exponer un parámetro que existe en HDRP pero no en URP. Este parámetro es el último que se puede ver en el inspector, el que controla el ángulo interior y exterior. De esta forma los artistas pueden definir el *falloff* con el que la luz pierde intensidad:

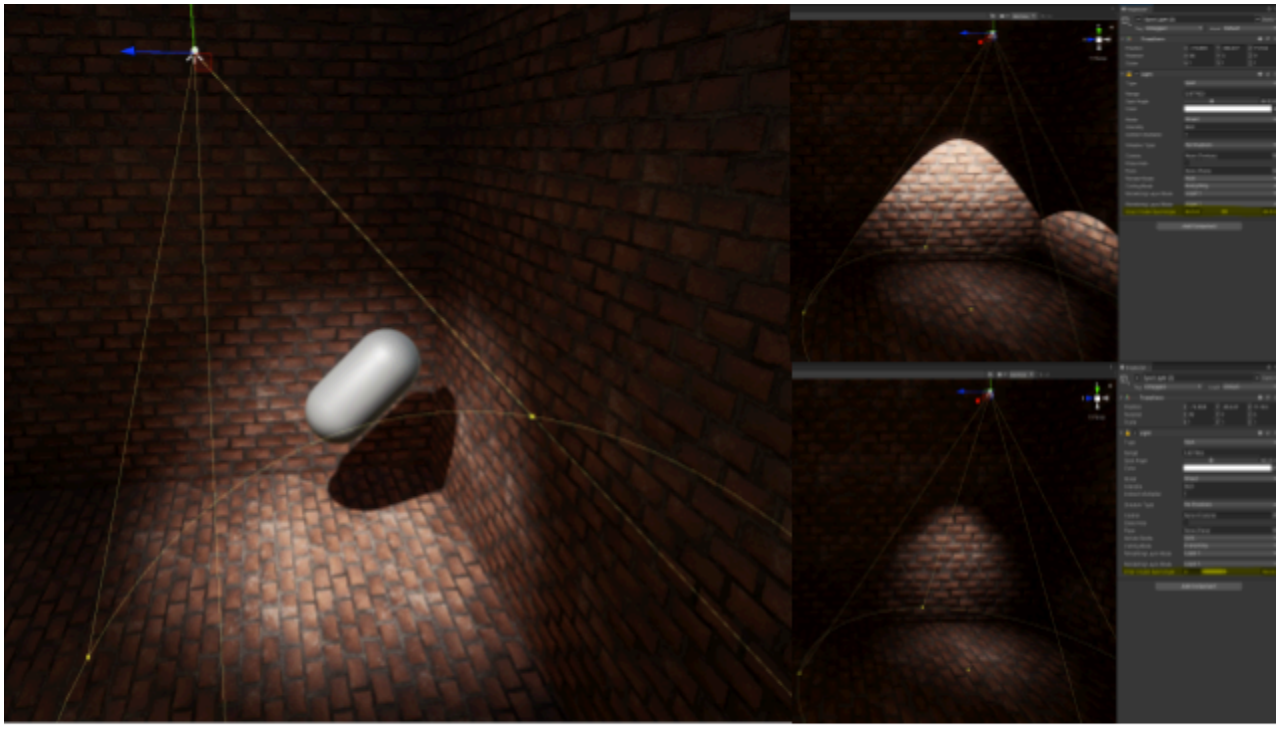


Figura 20: Comparativa con distintos valores de Falloff

En cuanto a las *point lights* se refiere, no diferimos en funcionalidad con URP a excepción de las sombras en tiempo real; el editor es el mismo que usa Unity.
Para conseguir que todo esto funcione, el renderer requiere de ciertas modificaciones: La primera será crear las funciones para cada una de las luces donde antes solo teníamos la versión *Directional.* Empecemos por *lighting.cs*:

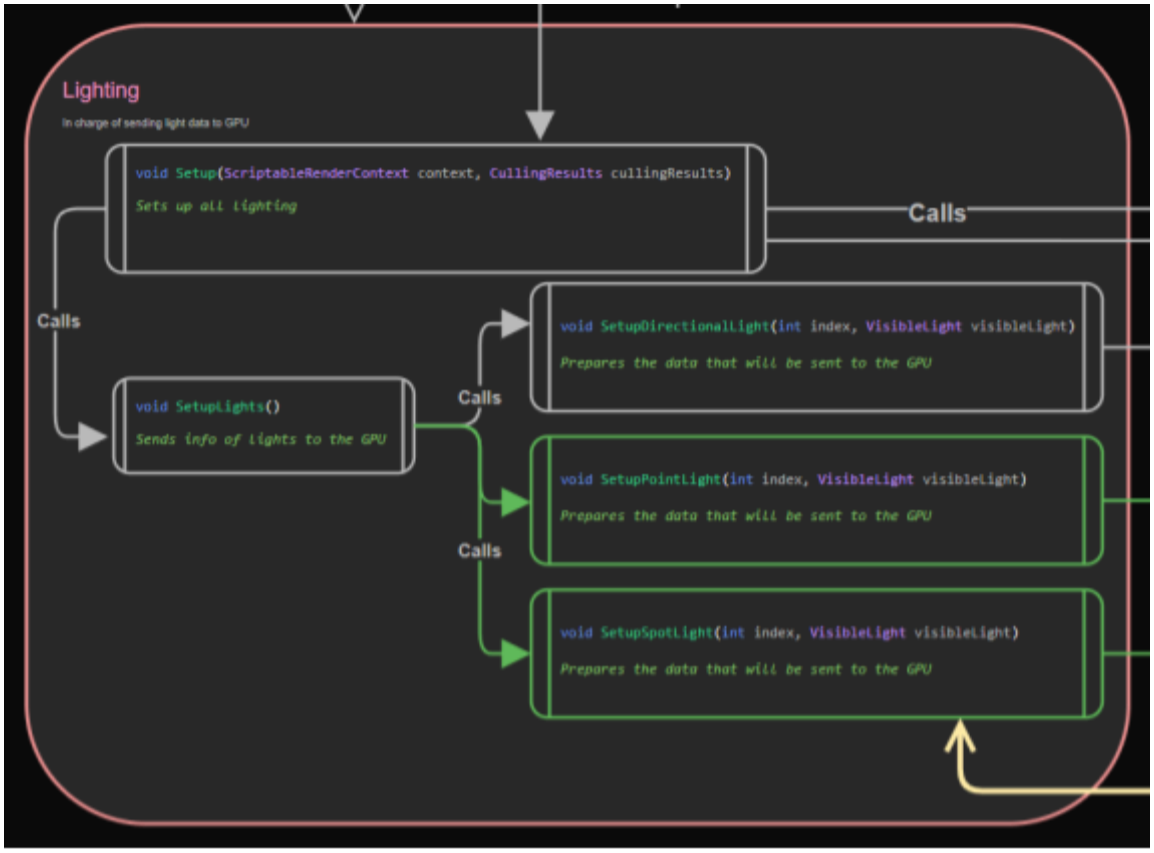


Figura 21: Esquema Lighting C#

Creamos *SetupPointLights()* y *SetupSpotLight()*, que preparan los datos para que la gráfica sepa cómo iluminar: en el caso de las *point*, información como la posición, el rango o el *falloff entre otros*; mientras que para las *spot lights* tenemos que preparar de forma adicional información sobre el ángulo interior y exterior. Para aliviar la carga de la gráfica podemos hacer parte de los cálculos aquí, antes de enviar la información a la gráfica. La fórmula que usamos es la misma que usa Unity ya que no queremos reinventar la rueda cuando esta ya es eficiente y se ve bien. La fórmula en cuestión es la siguiente:

$$saturate\left(\frac{d-cos\left(\frac{r_o}{2}\right)}{cos\left(\frac{r_i}{2}\right)-cos\left(\frac{r_o}{2}\right)}\right)^2$$

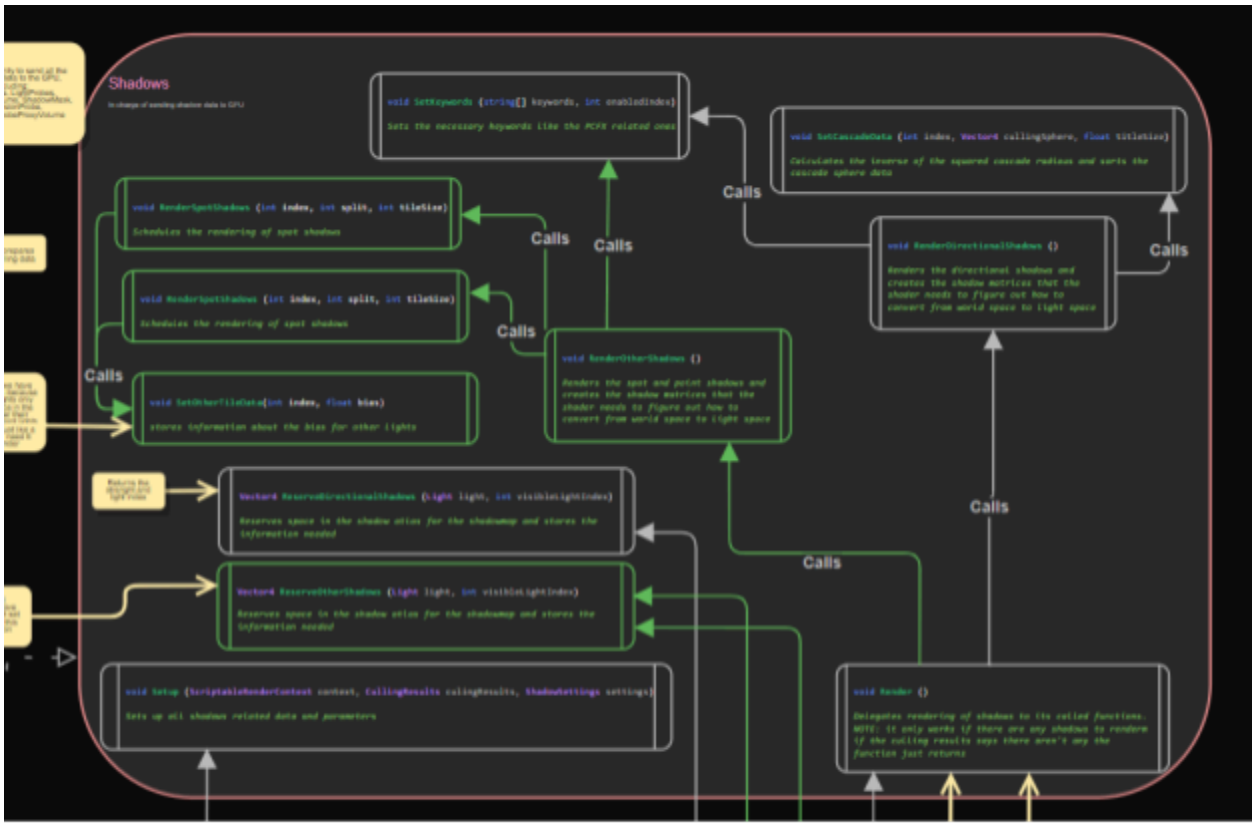


Figura 22: Esquema Shadows C#

Seguidamente llamaremos a una nueva función, *ReserveOtherShadows().* Esta es la primera vez que nos encontramos con esta distinción, ¿porque *ReserveOtherShadows()* y *ReserveDirectionalShadows()* en vez de tener una versión para cada tipo? La respuesta es sencilla: la única luz que se calcula de forma distinta es la *directional*, las demás se calculan todas del mismo modo, por tanto ni tiene sentido una distinción. Esta distinción entre las *directional* y el resto se debe a las cascadas: las luces que no son direccionales tienen un rango finito, por lo tanto no necesitan de cascadas ya que no se mueven con la vista. De ese modo aprovechamos también para generar un nuevo atlas para las luces no direccionales, así tenemos más resolución para estas, ya que las direccionales con sus cascadas ocupan mucho espacio.

Figura 23: *Shadowmap* de las luces *pointlight*

La siguiente parte a modificar es la función *Render()*. Ya no nos basta solo con llamar a *RenderDirectionalShadows()*, ahora tenemos que crear todo un camino para los nuevos tipos de luces:
Primeramente crearemos la función *RenderOtherShadows()*, que hace lo mismo que *RenderDirectionalShadows()* pero con la configuración de los nuevos tipos de luces. Esta función se encargará también de llamar a *RenderSpotShadows()* y *RenderPointShadows()*, las cuales a su vez llamarán a *SetOtherTileData()*.
Esta última función es la razón de que tengamos lo que nosotros llamamos *ShadowUnits*, y su funcionamiento es el siguiente:
Para renderizar una *spot light*, tenemos que reservar un trozo del atlas para guardar los datos de profundidad, ya que la luz tiene una dirección.

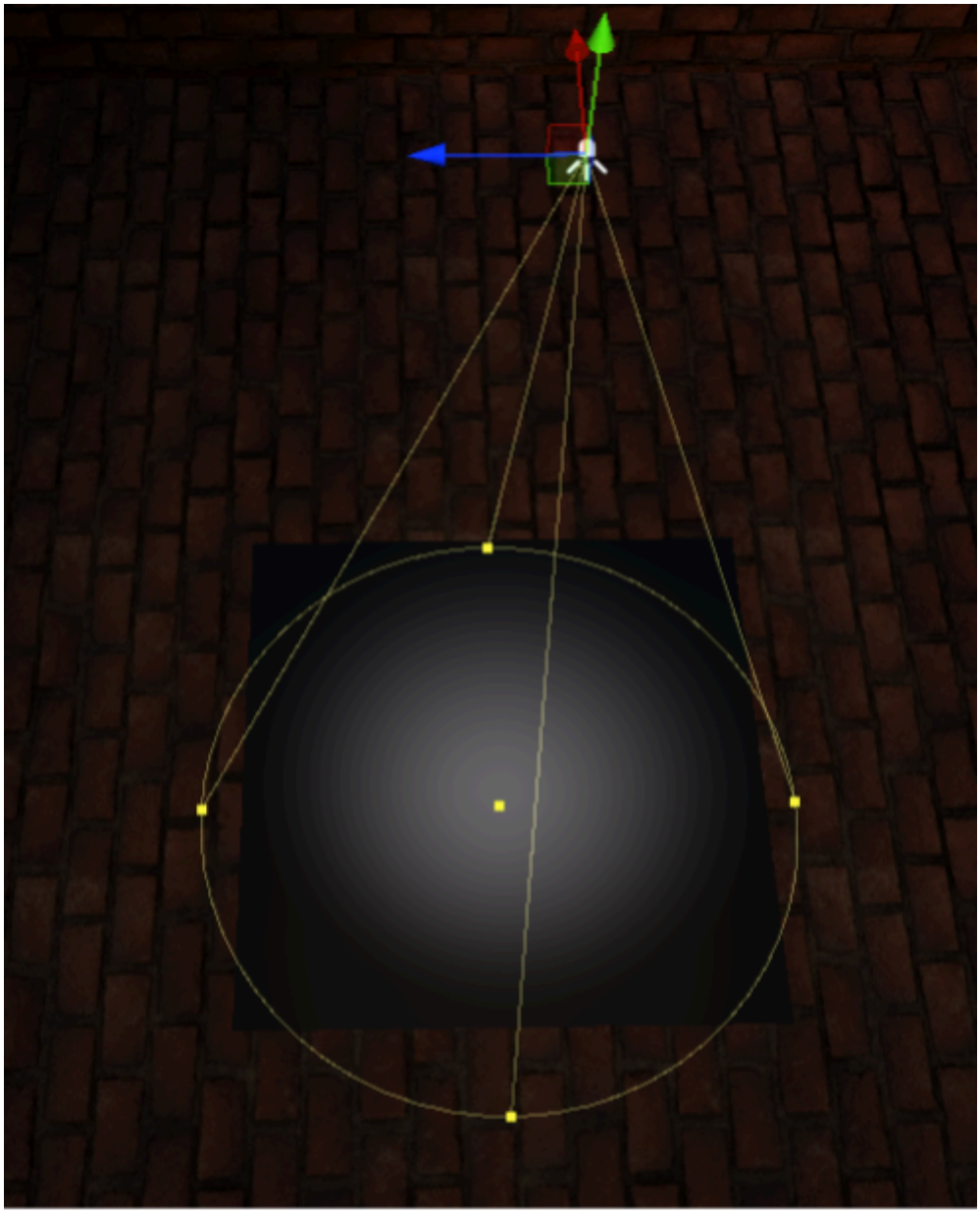


Figura 24: Representación del *shadowmap* de una spot light (1 *shadow unit)*

Por lo tanto con un *Quad* nos vale para cubrir toda la superficie iluminada. Este quad representaria un *shadowUnit*. Las point light sin embargo, al emitir luz en todas direcciones ya no podemos usar un único quad para almacenar toda su información. La solución que encontramos más factible fue usar 6 *quads* para cubrir toda su superficie:

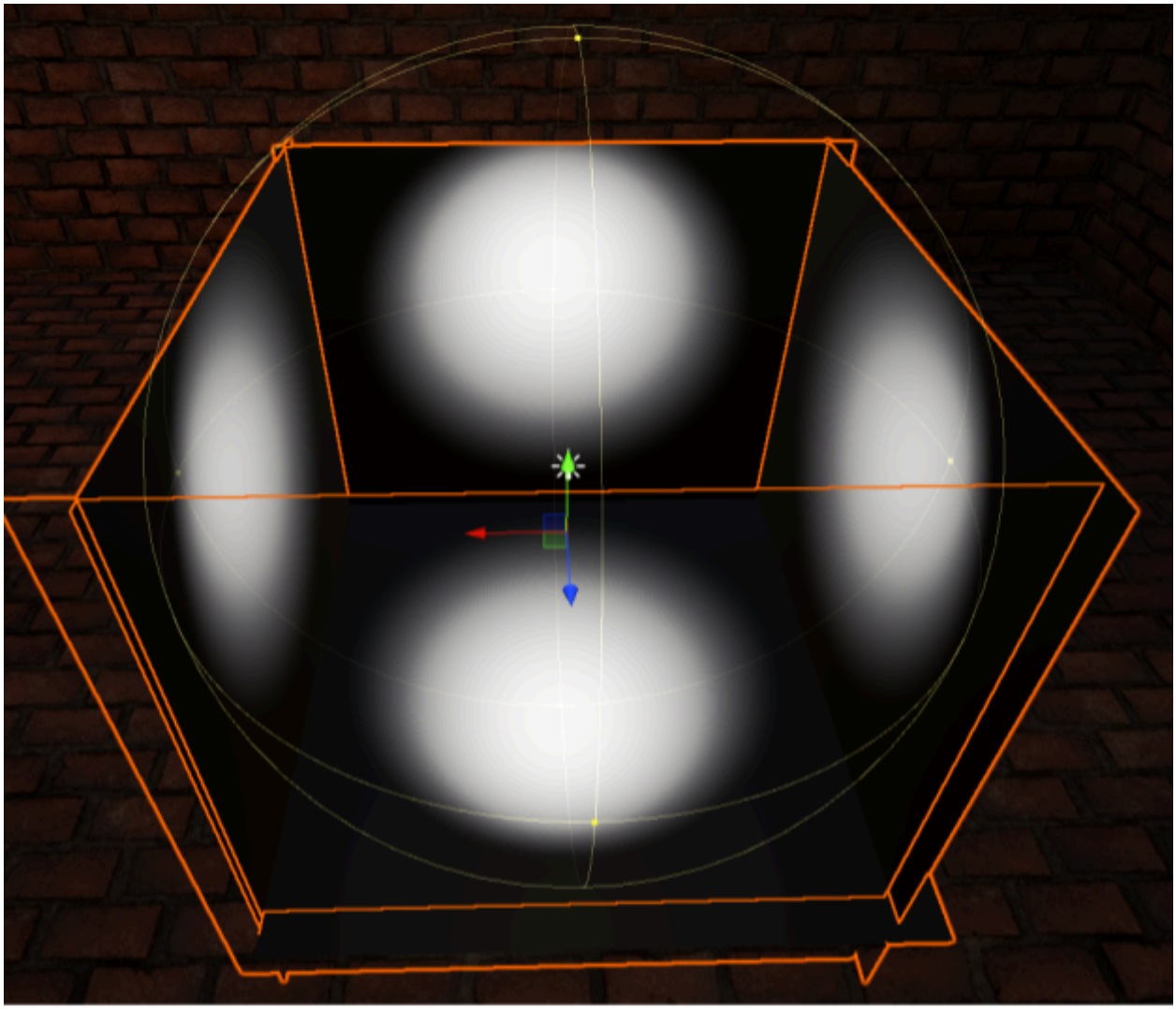


Figura 25: Representación del *shadowmap* de una point light (6 *shadow units*)

Por lo tanto en nuestro renderer las *spot lights* consumen una *shadow unit* mientras que las *pointlights* consumen 6 shadow units. Ahora bien, tenemos una limitación de 16 *shadow units* al mismo tiempo independientemente de la resolución del *shadowmap*. Esto es porque prevemos que el renderer tenga que funcionar otras plataformas a parte de PC. Si el juego fuera solo a PC podríamos haber establecido una calidad visual mayor teniendo en cuenta las especificaciones medias de un PC, pudiendo así subir la resolución del *lightmap y* tener más *shadow units* disponibles, pero por contra lo que hacemos es que si el mapa escala, las unidades escalan con el, de esta forma más resolución se traduce en sombras más definidas, no en mayor número de luces.
Esto es importante para poder usar el *renderer* en múltiples plataformas, porque si usáramos la primera aproximación, la de aumentar el número de shadow units en

vez de la resolución de estas, para consolas como la Switch, que son mucho menos potentes que un PC, al bajar el tamaño de el *lightmap* para vajar milisegundos en la GPU y que el juego funcionara correctamente en Switch, muchas luces que los artistas colocaron teniendo en cuenta 16 shadow units no se verían, ya que ahora dispondríamos de menos shadow units. Esto no solo afecta al diseño pensado para el juego al haber menos luces, sino que además provocaría desagradables artilugios visuales. Si superamos el número máximo de *shadow units*, se van a pintar por pantalla las primeras luces que lo soliciten cada frame, el orden de llamadas puede variar de frame a frame, provocando que en unos momentos se pinten unas luces, y a los pocos frames se pinten otras distintas, provocando multitud de efectos de parpadeo de las luces en la escena.
Para evitar esto preferimos la segunda aproximación: que las unidades fueran siempre las mismas y fuera la resolución de estas lo que variara según el rendimiento de la plataforma.
Con esto cubrimos todos los cambios necesarios en la CPU, pero nos faltan aún los cambios pertenecientes a los *shaders*.

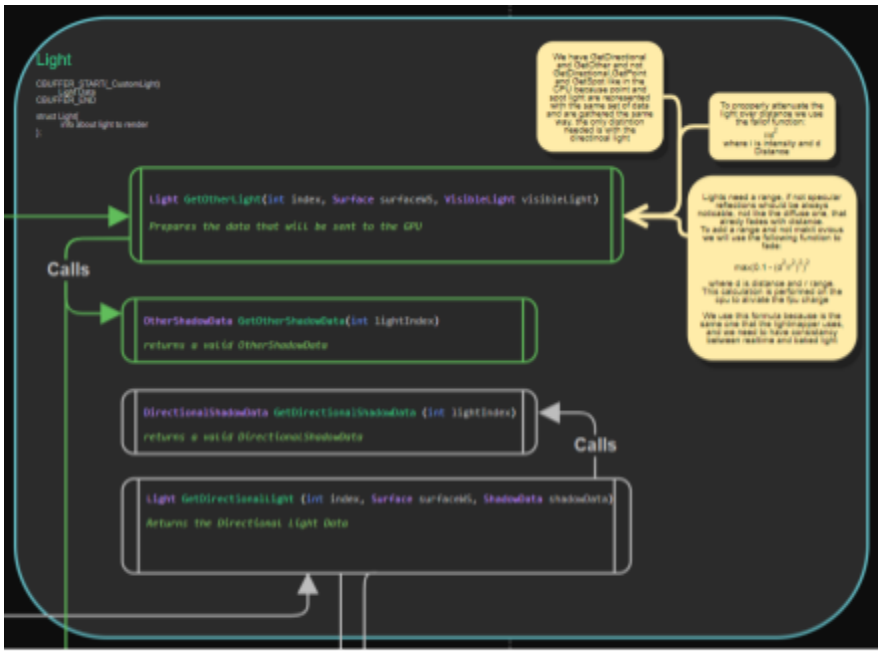


Figura 26: Esquema Light HLSL

Para empezar tenemos que modificar *Light.hlsl* para generar la función *GetOtherLight()*. Esta función además de devolver los valores de iluminación llama a las funciones pertinentes para recoger las sombras y atenuarlas de forma correcta.
Para atenuar estas luces usamos dos funciones: la primera es para atenuar la distancia, y es la siguiente:

$$\frac{i}{d^{2}}$$

Con esto conseguimos que la luz no sea infinita sino que decaiga con la distancia, pero esto no es suficiente ya que las luces también necesitan un rango, sino los reflejos especulares se pueden ver desde muchísima distancia, ya que la fórmula anterior sólo atenúa la luz difusa.
Así pues, necesitamos esta segunda fórmula para solventar eso::

$$max\left(0.1 - \left(\frac{d^{2}}{r^{2}}\right)^{2}\right)^{2}$$

Decidimos usar esta fórmula porque es la que usa el *lightmapper* de Unity ycomo queremos que la atenuación sea la misma tanto en tiempo real como en el *lightmap* nos decidimos por esta.
Por último, como la función tiene muchos elevados y divisiones preferimos calcularla en la CPU y pasarla ya calculada a la GPU, ya que estos son un tipo de cálculos nada eficientes para realizarse en la GPU.
Cabe destacar que todos estos cálculos son inútiles si el *shader* no es capaz de recoger la información de las sombras, y para eso hay que modificar *shadows.hlsl* de la manera mostrada en la figura 27:

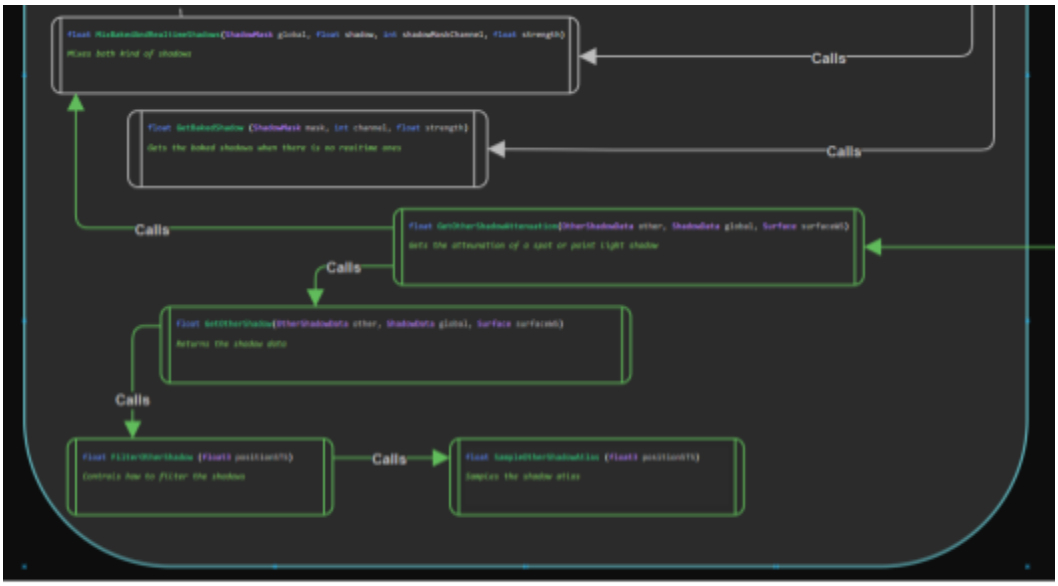


Figura 27: Funciones añadidas a Shadows.HLSL

Todas estas funciones son las versiones para *other shadows* de las que ya cubrimos en el apartado anterior, La razón es la misma que en la CPU, las podemos hacer juntas porque solo hay que realizar el muestreo de su zona de *lightmap*, sin

preocuparnos por cascadas. La única que tiene alguna peculiaridad es la *pointlight*, porque requiere juntar los seis *quads*, pero en una de las librerías de *core render pipeline* existe ya una función que hace justo eso.
Como habréis podido apreciar, no hemos modificado nada para permitir el *bake* de las sombras, y es porque no hizo falta: el *lightmapper* de Unity ya soporta el *bake* de todo tipo de luces, solo ha hecho falta leer esa información en las funciones que ya leían de *lightmap*.

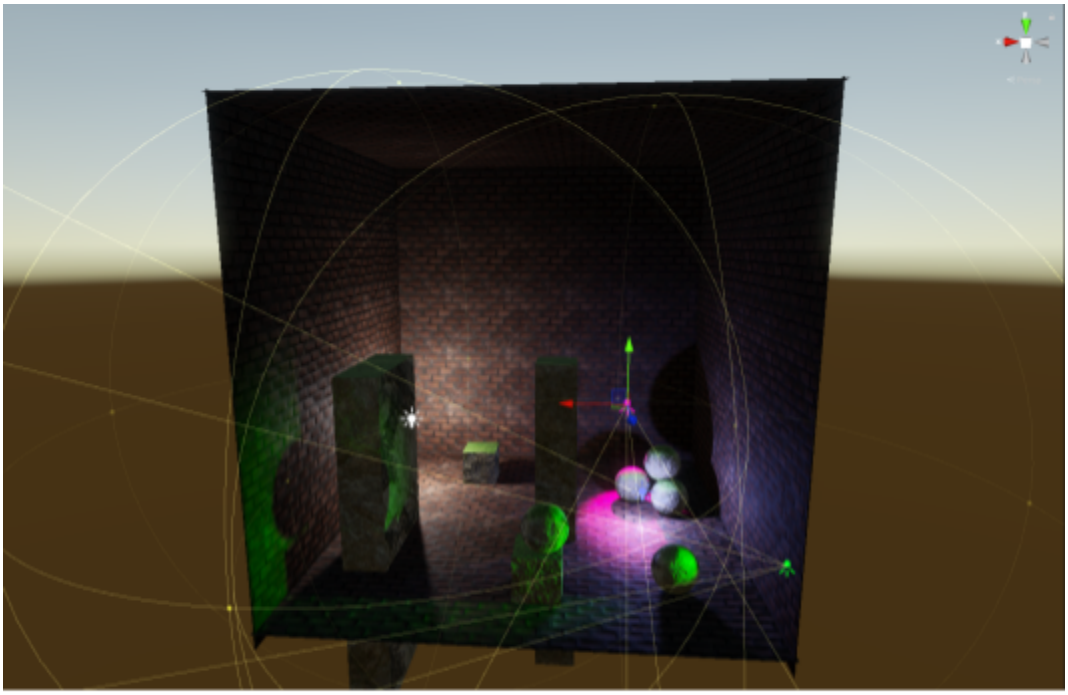


Figura 28: Distintas sombras en tiempo real

Texturing

Ahora que ya podemos iluminar objetos es hora de texturizarlos. Para esto aprovechamos que todos los *shaders* son personalizados para optimizar las texturas necesarias.
Con los artistas acordamos que los mapas que podrían necesitar son los siguientes:

- *Base map* para el color
- *Normal* map
- *Metallic* map
- *Occlusion* map
- *Smoothness* map
- *Emission* map

Nuestra primera opción fue implementar el estandard *MODS* map (*Metalic, Oclusion, Detail, Smoothness*), una textura que engloba los mapas de *Metalicness* en el canal r, *occlusion* en el g, *detail* en el b y *smoothness* en el a. Esta idea no nos acabo de gustar porque como nuestro juego es en tercera persona, la camara nunca va a estar suficientemente cerca de una superficie como para necesitar de detalle, así que aprovechamos la estructura de el MODS map para substituir el canal de *detail* por un canal de *emission*, lo que necesitamos es que los artistas introduzcan un canal en escala de grises que define que zonas emiten luz y que zonas no, y en el *shader* exponemos un parametro de color para poder decidir que color emiten.
Hacer este cambio nos permite ahorrarnos tener una segunda textura solo para guardar el mapa de emisión, el cual es fundamental en nuestro juego, ya que todas las superficies por las que el jugador puede moverse o interactuar emiten luz para mostrar que están activas.

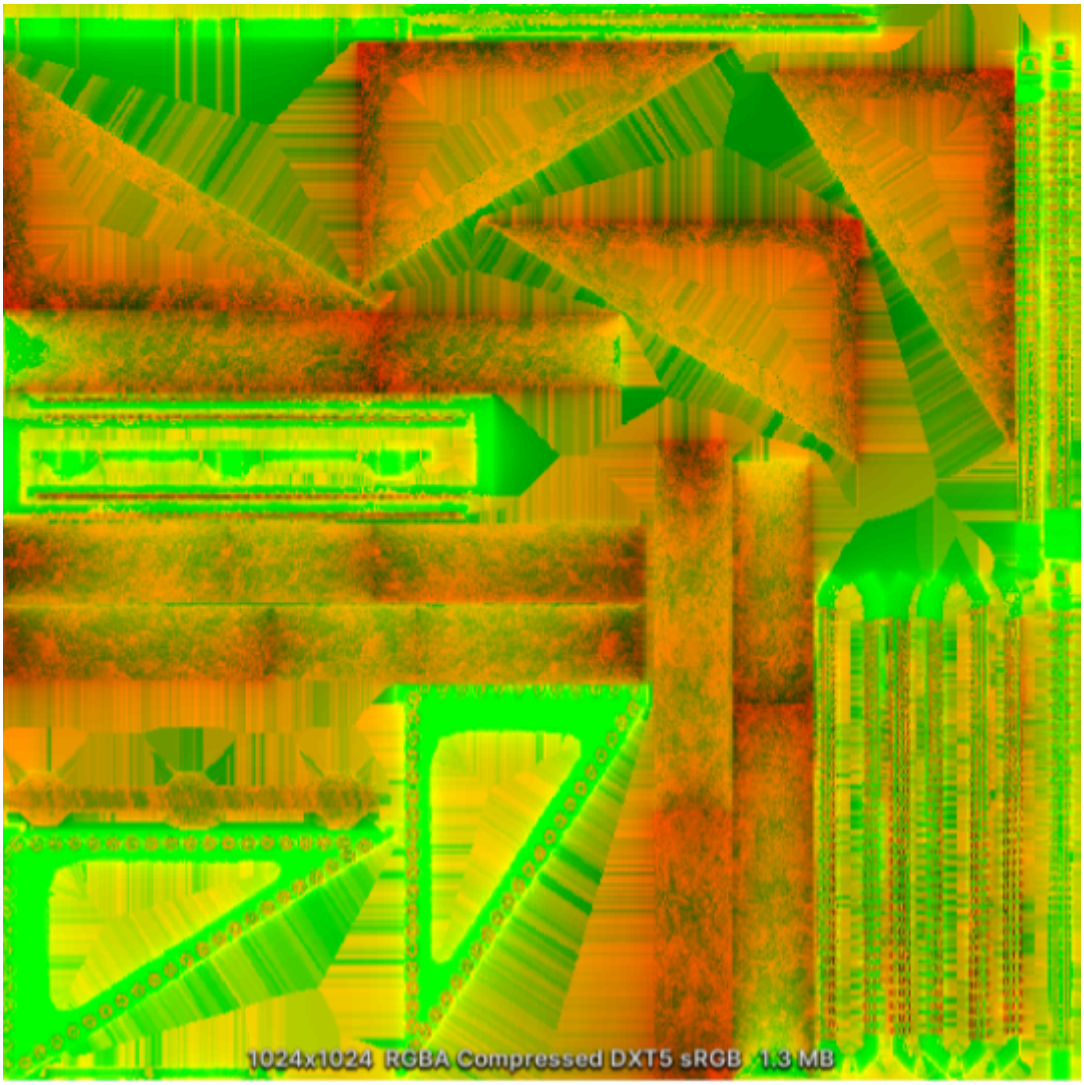


Figura 29: Custom MODS map

De esta forma acabamos pasando de 6 texturas iniciales a solo 3 por cada material que necesitemos. Esto es especialmente importante para consolas como la Nintendo Switch, ya que la memoria de esta es muy escasa y no la queremos malgastar.

Optimization

Lo primero que tenemos que conseguir para que nuestro *renderer* sea eficiente es que todos nuestros *shaders* sean compatibles con el *SRP Batcher* de Unity. Este sistema es una forma de cachear la información necesaria para cada material directamente en la GPU. Esto no reduce las llamadas a pintado (drawcalls) per se, pero las hace menos costosas, ya que es capaz de enviar la información de diversas *drawcalls* en un solo *buffer* para no tener que comunicar tanto la CPU con la GPU ganado de esa forma el tiempo que perderíamos enviando los datos cada vez.
No es necesario mucho trabajo para que un *shader* sea compatible, solo necesitamos agrupar todos los parámetros que tienen que ser cacheados bajo los siguientes buffers:

- CBUFFER_START(UnityPerMaterial)
- CBUFFER_START(UnityPerDraw)

Una vez hecho esto, los materiales con este *shader* ya se pueden bachear. Estos han sido los parámetros que hemos usado:

```
UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
  UNITY_DEFINE_INSTANCED_PROP(float4, _BaseMap_ST)
  UNITY_DEFINE_INSTANCED_PROP(float4, _DetailMap_ST)
  UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
  UNITY_DEFINE_INSTANCED_PROP(float4, _EmissionColor)
  UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
  UNITY_DEFINE_INSTANCED_PROP(float, _ZWrite)
  UNITY_DEFINE_INSTANCED_PROP(float, _Metallic)
  UNITY_DEFINE_INSTANCED_PROP(float, _Occlusion)
  UNITY_DEFINE_INSTANCED_PROP(float, _Smoothness)
  UNITY_DEFINE_INSTANCED_PROP(float, _Fresnel)
  UNITY_DEFINE_INSTANCED_PROP(float, _DetailAlbedo)
  UNITY_DEFINE_INSTANCED_PROP(float, _DetailSmoothness)
  UNITY_DEFINE_INSTANCED_PROP(float, _DetailNormalScale)
  UNITY_DEFINE_INSTANCED_PROP(float, _NormalScale)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

Código 7

```
CBUFFER_START(UnityPerDraw)
  float4x4 unity_ObjectToWorld;
  float4x4 unity_WorldToObject;
  float4 unity_LODFade;
  real4 unity_WorldTransformParams;
```

```
  float4 unity_RenderingLayer;

  real4 unity_LightData;
  real4 unity_LightIndices[2];

  float4 unity_ProbesOcclusion;

  float4 unity_SpecCube0_HDR;

  float4 unity_LightmapST;
  float4 unity_DynamicLightmapST;

  float4 unity_SHAr;
  float4 unity_SHAg;
  float4 unity_SHAb;
  float4 unity_SHBr;
  float4 unity_SHBg;
  float4 unity_SHBb;
  float4 unity_SHC;

  float4 unity_ProbeVolumeParams;
  float4x4 unity_ProbeVolumeWorldToObject;
  float4 unity_ProbeVolumeSizeInv;
  float4 unity_ProbeVolumeMin;
CBUFFER_END
```

Código 8

A priori puede parecer que este extracto de código esta mal ya que comentábamos que usaríamos el *buffer CBUFFER_START(UnityPerMaterial)* y estamos usando *UNITY_INSTANCING_BUFFER_START* y *UNITY_DEFINE_INSTANCED_PROP*. Esto es porque también queremos que los materiales se puedan instanciar con distintos parámetros para poder consolidar el número de *drawcalls*.

Para ello hay que hacer varias cosas:

La primera es usar estos nuevos *defines* en vez de los anteriores, y la segunda es que en el *fragment shader* dejarle saber que identificador le corresponde usando la siguiente macro: *UNITY_VERTEX_INPUT_INSTANCE_ID*

Esto es porque *instancing* funciona parecido a un *array*: enviamos a la GPU un grupo de parámetros distintos y cada fragment shader necesita saber a qué espacio tiene que ir a buscar la suya.

Por último, para reducir el coste computacional de pintar la escena necesitamos poder usar el sistema de *LODs* (*Level Of Detail*) de Unity. Por suerte para nosotros este sistema ya es compatible sin que tengamos que hacer ningún paso extra ya que qué malla se va a pintar se decide en los procesos de *culling* que habíamos descrito anteriormente. Aún así el cambio entre una malla y la siguiente es muy brusco y queríamos usar la opción de *crossfade* que anima la transición entre una y otra.

Por desgracia esto si que hay que implementarlo.

Para que todos los *shaders* sean compatibles creamos una función llamada *ClipLOD* que gracias al *define* #if *defined*(*LOD_FADE_CROSSFADE*) sólo actuará si esta opción

está activada, en cuyo caso llama a la función *Clip* para simplemente pintar una malla o la otra dependiendo del patrón.

Post processing

Una vez cubierta la optimización del renderizado llegamos al último paso que necesitamos cubrir para nuestro *render*, el post procesado.
Para dotar al juego de un estilo visual único y muy configurable necesitamos poder aplicar efectos a toda la imagen una vez esta ha sido completamente renderizada, de esta forma damos a los artistas mucho control para modificar cómo se ve y siente todo el entorno.

Esta es una de las áreas donde sentimos que la producción no fue óptima: como desarrollar un renderer es muy costoso tanto a nivel de esfuerzo como de tiempo, empezamos el desarrollo mucho antes de que empezara el proyecto y aún no contábamos con los artistas en el equipo. Como no pudimos acordar con ellos que post procesados necesitarían decidimos implementar todos los que hay en Unity para que la barrera de entrada a nuestro render no fuera muy alta viniendo de Unity por defecto, así que acabamos implementando:

- Color Grading
    - Shadows, Midtones, Highlights
    - Channel Mixing
    - Split Toning
    - Post Exposure
    - Contrast
    - Color Filtering
    - Hue Shifting
    - Saturation
    - White Balance
- ToneMapping
    - Reinhard
    - Neutral
    - ACES
- Bloom
    - Scatter
    - Additive

Creemos que esta es una de las áreas de mejora porque en cuanto los artistas entraron al proyecto acabaron usando:

- ACES
- Scattering bloom
- Post Exposure

Todo este tiempo podríamos haberlo usado en otros apartados que acabaron necesitando más horas de trabajo.

Una vez aclarado esto empecemos. Nuestro sistema de FX funciona como un *stack*, por eso se llama PostFXStack, como el de Unity. La razón de esto es que todos los efectos se aplican en orden uno encima del anterior.
Así pues, el orden en que llamamos los efectos no es trivial, y tuvimos que pensarnos bien en qué orden se ejecutarían.

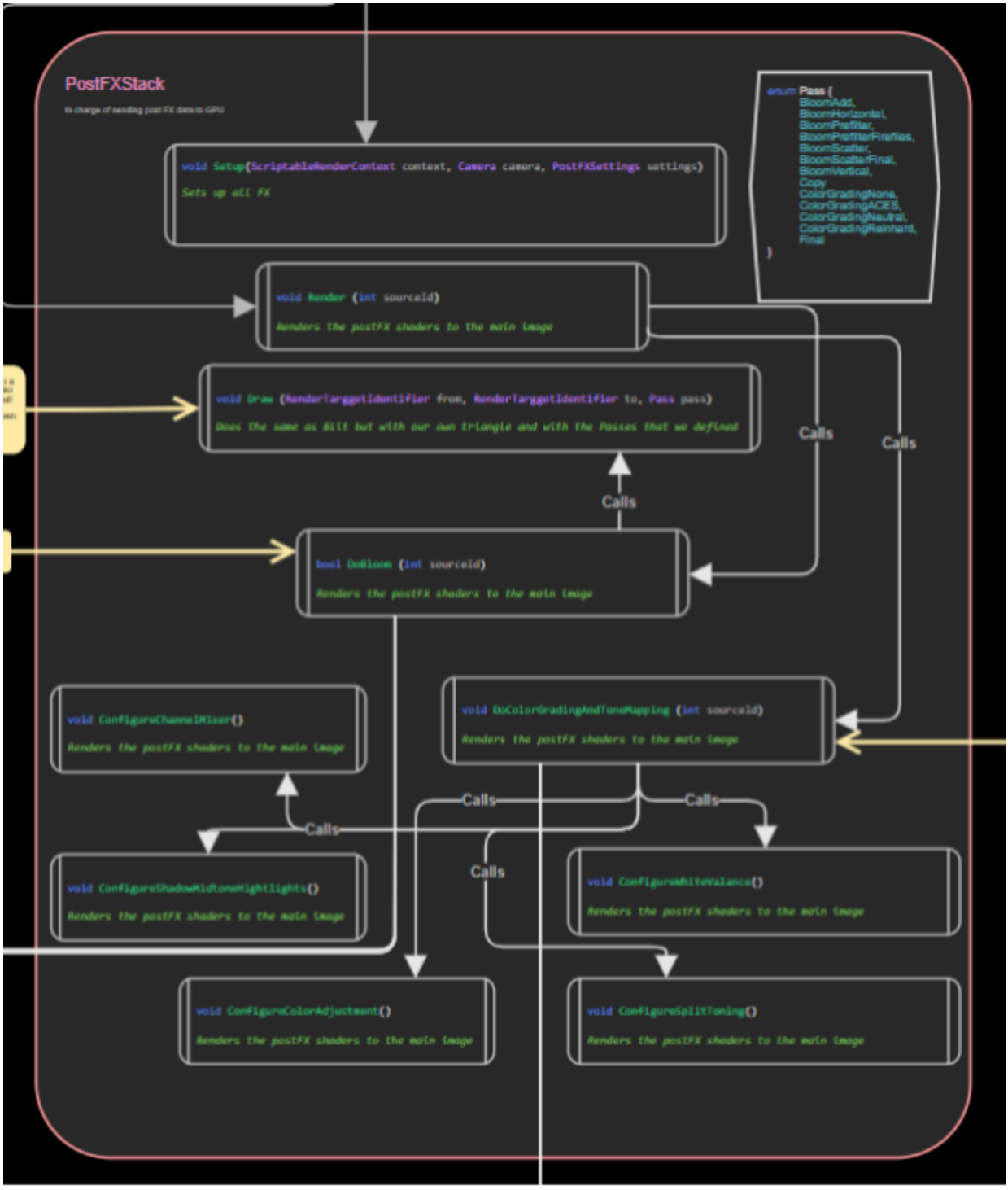


Figura 30: PostFXStack C#

Este es el diagrama de flujo de la clase de C# que controlara nuestros FX.
Empezamos llamando a *Setup()* desde la función *Render()* de *CameraRender* ya que podemos tener distintos efectos por cada cámara e incluso pueden modificarse en

tiempo de ejecución para ambientar distintas zonas. De igual manera llamamos a *PostFXStack.Render()* desde *CameraRender.Render()*.
La función *Render()* llama a solo dos funciones. Esto es porque todos nuestros efectos se pueden separar en dos subgrupos: *Bloom* y *ColorGradingAndTonneMapping*. Esta separación es muy útil porque va a ser muy simple expandir nuestro *stack* en el futuro si fuera necesario ya que todas las funciones que añadamos en el futuro tendrán que ir seguro antes de todo el tratado del color para mantener la cohesión visual de la escena y antes del *Bloom* para que si algún efecto resulta en colores muy brillantes el Bloom los haga ¨Resplandecer¨.
Empezaremos pues por la función *DoBloom()*, este efecto de post procesado es uno de los más caros ya que seguimos un algoritmo de *downSampling* y *upSampling*, lo que significa que reducimos la resolución de la imagen final repetidas veces y una vez reducida, la vamos combinando con sus versiones de más alta resolución para conseguir el efecto de *bloom,* y por tanto necesitamos generar muchas texturas intermedias para reducir y combinar el tamaño de la imagen final.

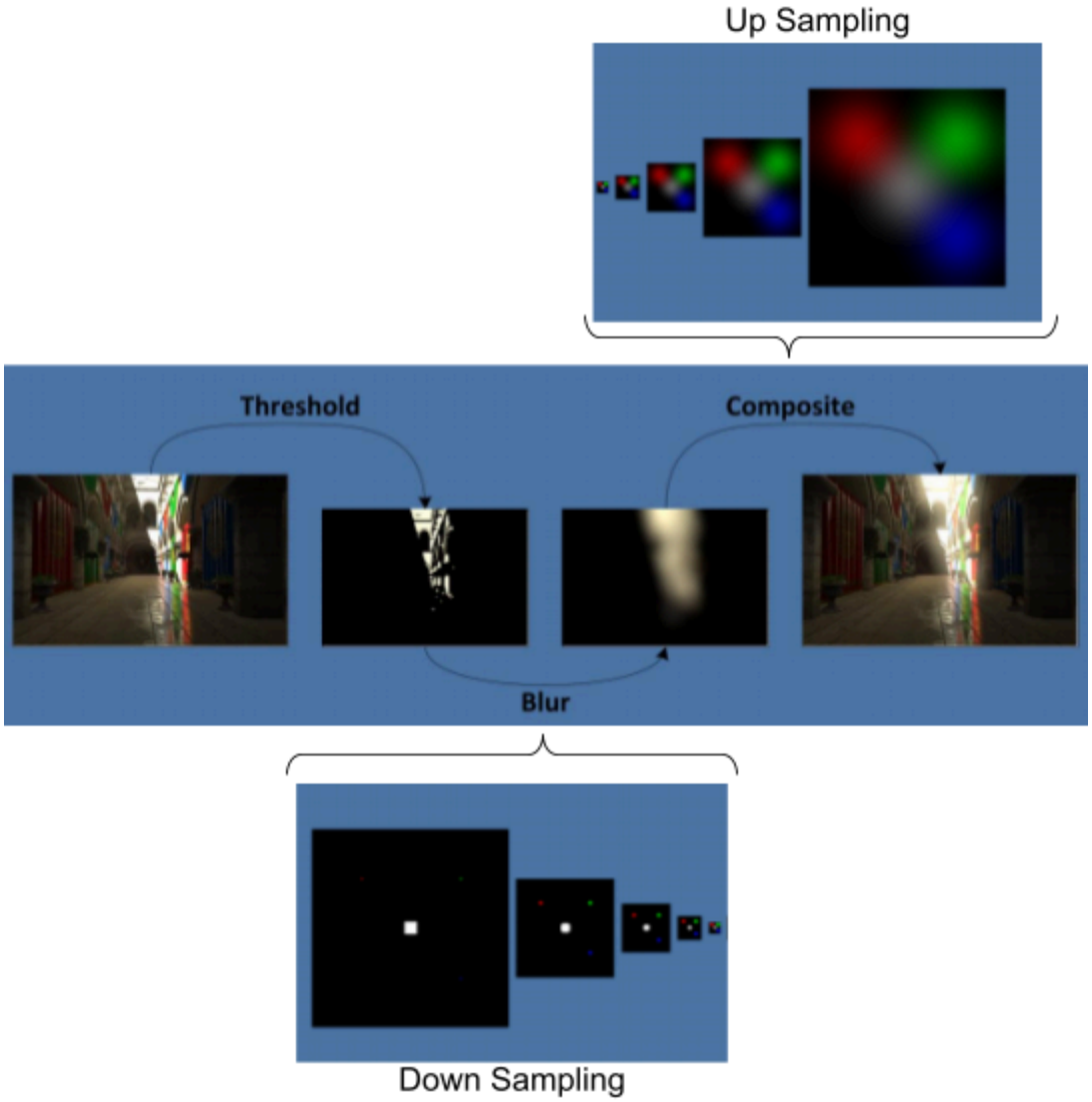


Figura 31: Pasos del efecto de *bloom*

Para poder hacer todo esto, nuestra clase de C# necesita poder pedirle a la gráfica que haga el muestreo de diversas texturas, que las combine y además que pueda hacerlo de distintas formas. Aquí es donde entra nuestro fichero .hlsl más grande hasta el momento, *PostFXStackPasses.hlsl*.

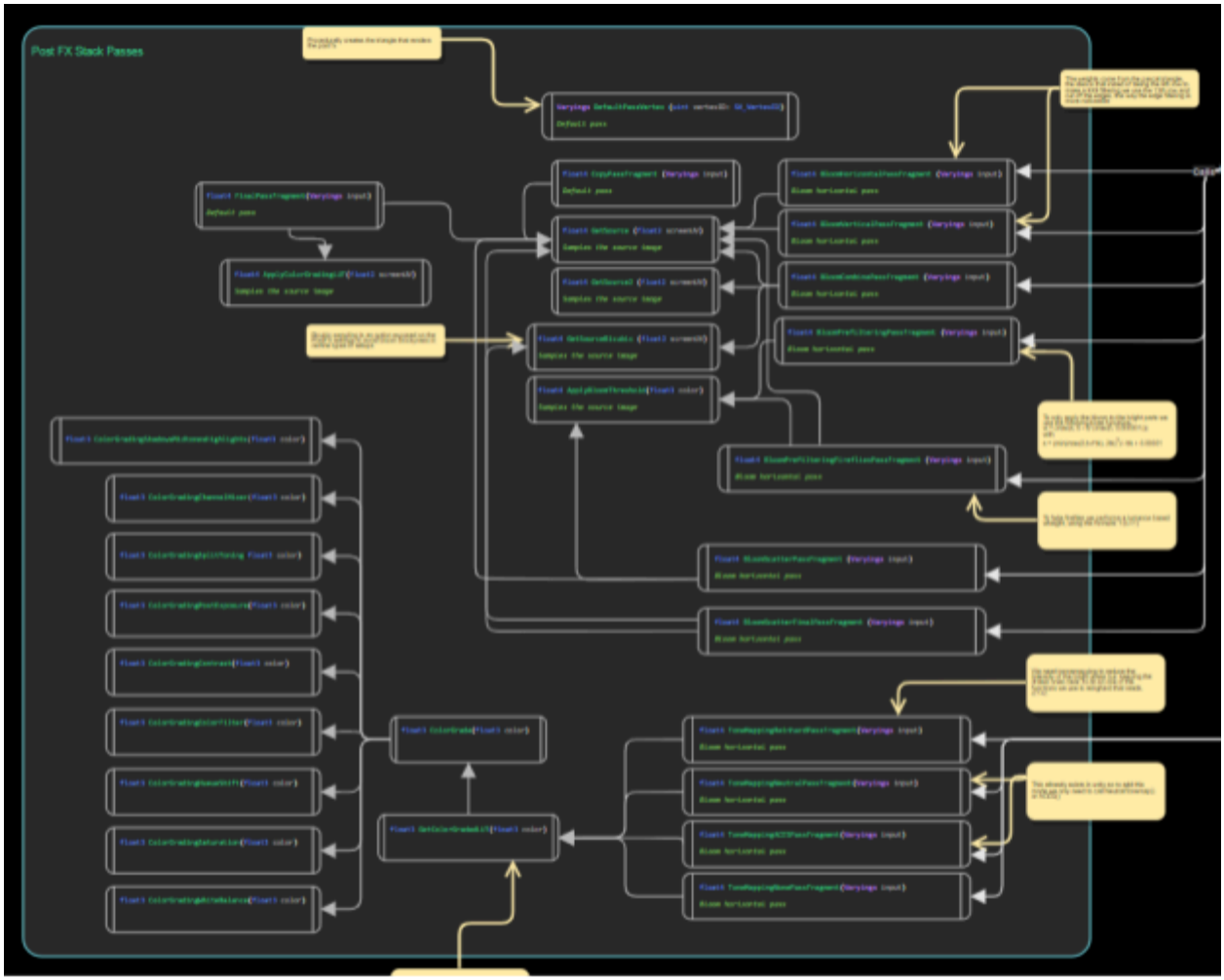


Figura 32: PostFXStack.HLSL

Puede parecer un poco anti intuitivo que necesitemos tantas funciones para el *Bloom*, pero esto es por diversas razones.

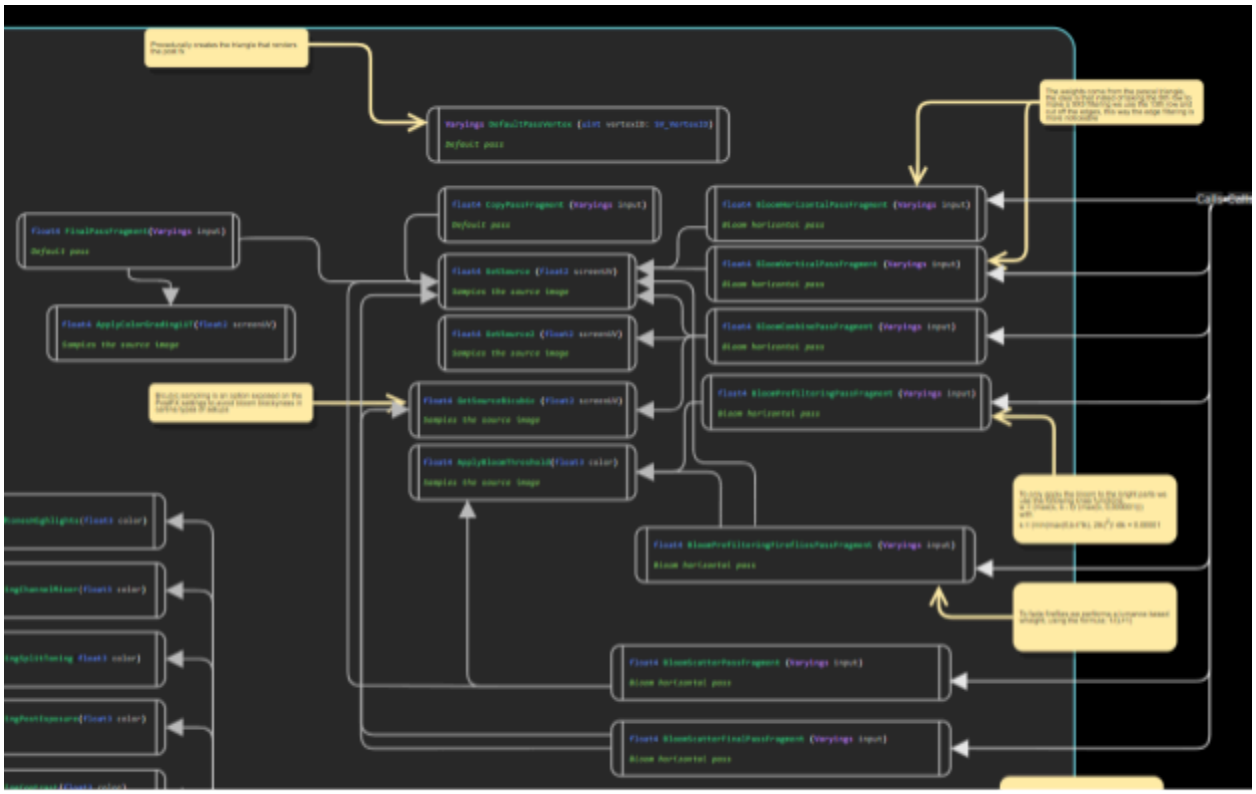


Figura 33: Funciones relacionadas al *bloom*

La primera razón es que tenemos dos tipos de *bloom*, aditivo y *scattering.* Tenemos estas dos versiones porque el modo aditivo es mucho más barato, pero menos creíble, ya que no respeta la conservación de energía, es decir, si filtramos solo las partes brillantes de la imagen, las emborronar y lo añadimos a la imagen original, estamos añadiendo luminosidad a la imagen que no estaba ahí, y así no es como funcionan las cámaras.

Por eso también implementamos *scattering*, que simplemente difumina el exceso de luz en el resto de la imagen.

Esto solo cubre unas pocas funciones del esquema, otra razón por la que tenemos tantas funciones es que el *downsampling* lo separamos en dos pasos: *bloom* horizontal y vertical. Si solo redujeramos el tamaño de la imagen para tener la versión emborronada solo conseguiríamos una imagen de baja resolución con pixeles muy grandes

Figura 34: *Downsampling* de una imagen

Partiendo el proceso en vertical y horizontal podemos hacer que el emborronado sea más suave.

Para hacer este proceso mas simple cojemos los pesos para el filtering de la pirámide de pascal, la idea es que en vez de coger los valores de la novena fila, porque queremos hacer un filtrado de 9x9, es mejor coger los de la fila 12 y simplemente eliminar los valores sobrantes, de esta manera el filtrado en los bordes se nota mucho más.

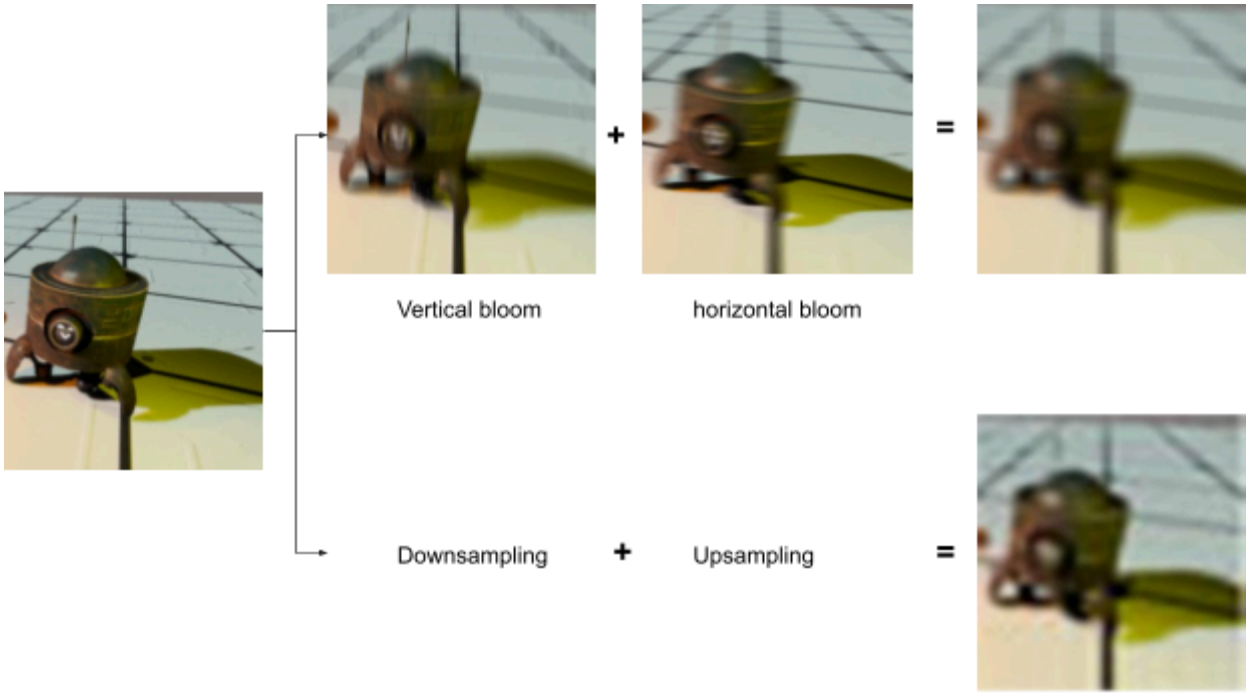


Figura 35: Emborronado por *bloom* vs *downsampling + upsampling*

Un aspecto del *blur* que no hemos comentado aún es el filtrado. Si siguiéramos los pasos que hemos comentado hasta el momento simplemente emborronaríamos la imagen. Para hacer bien el efecto tenemos que emborronar solo las partes de la imagen que sean muy brillantes y sumar esto a las versiones anteriores.
Para esto usamos la siguiente función:

$$w = \frac{max(s, b-t)}{max(b, 0.0001)} \text{ i } s = \frac{min(max(0, b-t+tk), 2tk)^2}{4tk + 0.0001}$$

Por último, como tenemos soporte para colores HDR, podemos tener fragmentos que brillan mucho pero que ocupen menos de un píxel, lo que se traduce en encontrarnos con zonas de la imagen con destellos aleatorios que aparecen y desaparecen ya que como ese fragmento ocupa menos de un píxel, hay veces en que se pinta y veces en las que no. Este artefacto visual es especialmente notable durante el movimiento de la cámara, y se conoce con el nombre de *fireflying.*

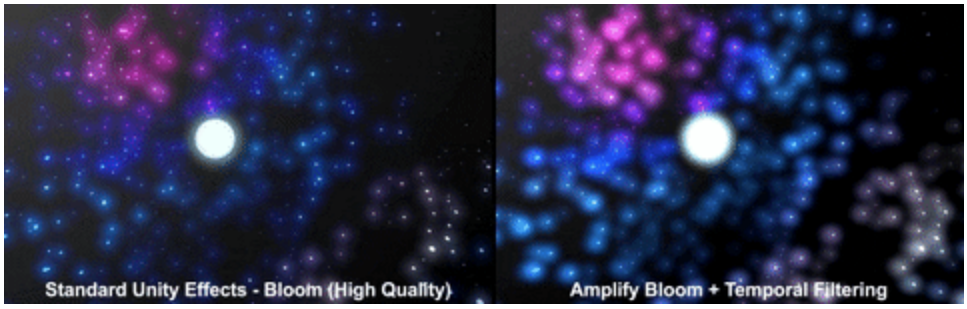


Figura 36: bloom sin filtering vs bloom con filtering

Para mitigar este efecto calculamos los pesos de cada zona basándonos en su luminancia siguiendo la fórmula:

$$\frac{1}{l+1}$$

Hemos mencionado que soportamos HDR, pero Unity no soporta pantallas HDR y todavía no son el estandard en los hogares. Así pues, aunque trabajamos con colores HDR (lo cual nos brinda una mayor precisión de colores), tenemos que introducir un paso intermedio para ajustar el rango de los colores que tenemos en pantalla para adaptarlos a los monitores LDR que son los que tenemos en la mayoría de los hogares y con las que Unity trabaja, pero cómo hacer esto?
Si ajustamos para que los colores más brillantes sean uno y los más oscuros sean cero, si por ejemplo tenemos un pequeño destello brillante con intensidad diez, el resto de colores serían diez veces más oscuros de lo habitual. Casos como este son muy importantes a tener en cuenta, porque son muy comunes ya que por ejemplo el sol tiene un valor de luminancia de $1.6 * 10^{9} cd/m^{2}$. Obviamente esto son valores físicos y nadie los usa en un juego, pero ayuda a ejemplificar el caso.
Como estamos tratando la luminosidad de píxeles en pantalla, no hay una fórmula física que resuelva esto como tal, así que hay muchas propuestas distintas para mapear los colores al rango de cero a uno que mostrará el monitor.
Empezamos implementando las fórmulas que Unity soporta de por sí, las cuales son las conocidas como Neutral y Reinhard.
Aún teniendo estas dos aproximaciones, decidimos añadir una tercera llamada ACES (Academy Color Ecoding System) que como se creó en la academia de cinematografía y está muy asociado a un estilo visual cinemático y fidedigno, y es capaz de mantener muy bien los colores oscuros aun resaltando los claros.
Esto es así porque su fórmula ofrece una mayor resolución en el mapeado a los colores oscuros que a los claros, lo cual funciona muy bien ya que el ojo humano es más sensible a los colores oscuros que a los claros, por tanto esta fórmula se

asemeja mucho más a cómo percibimos el mundo de lo que lo hacen las aproximaciones Neutral o de reinhard.

- None

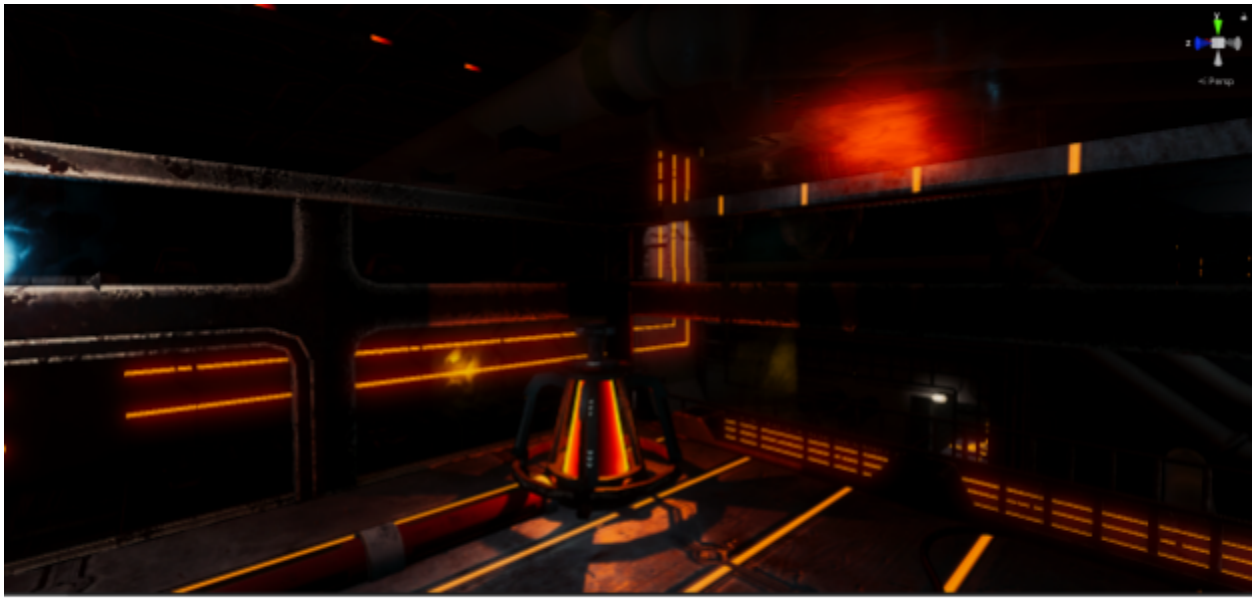


Figura 37: Imagen sin tratamiento de color

- ACES

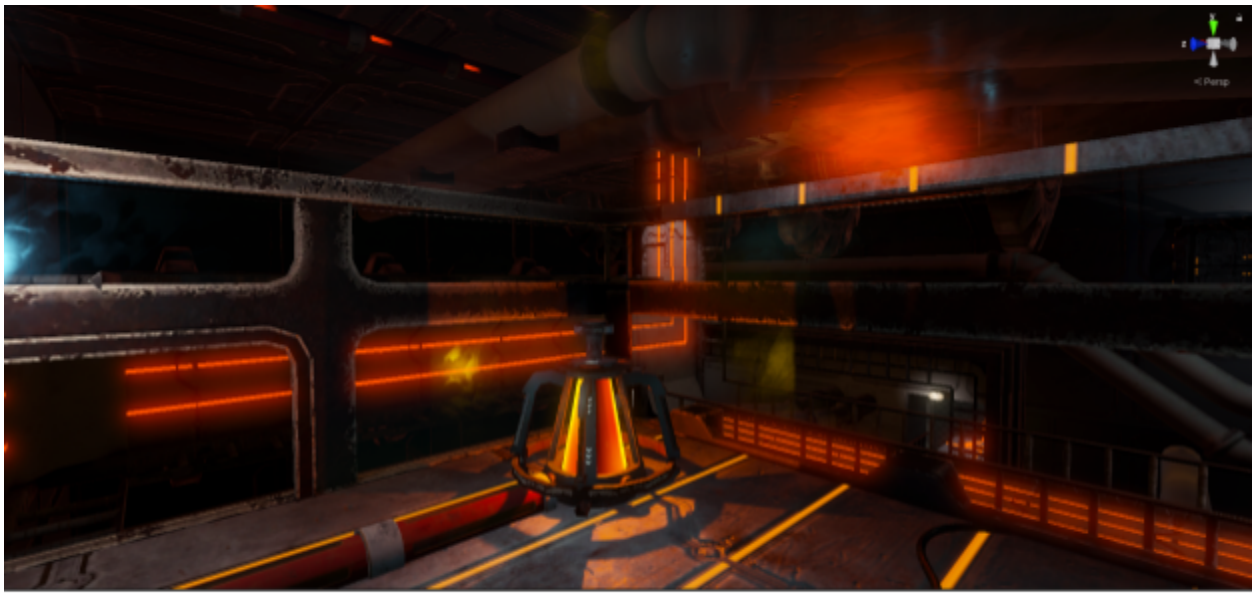


Figura 38: Imagen con tratamiento de color ACES

- Neutral

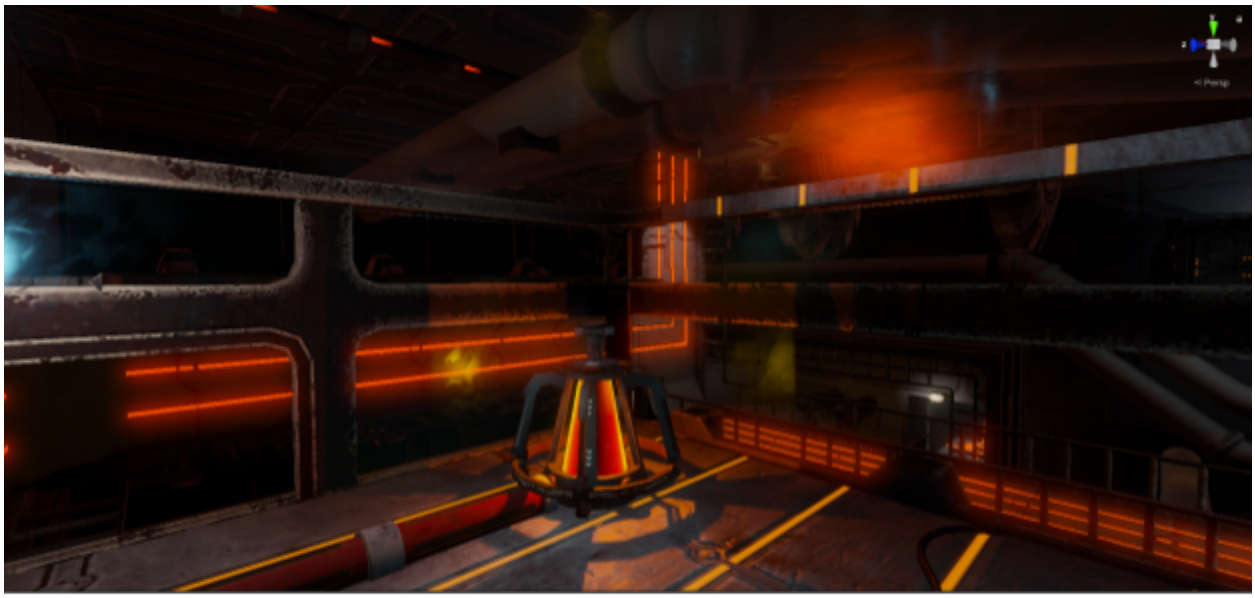


Figura 39: Imagen con tratamiento de color NEUTRAL

- Reinhard

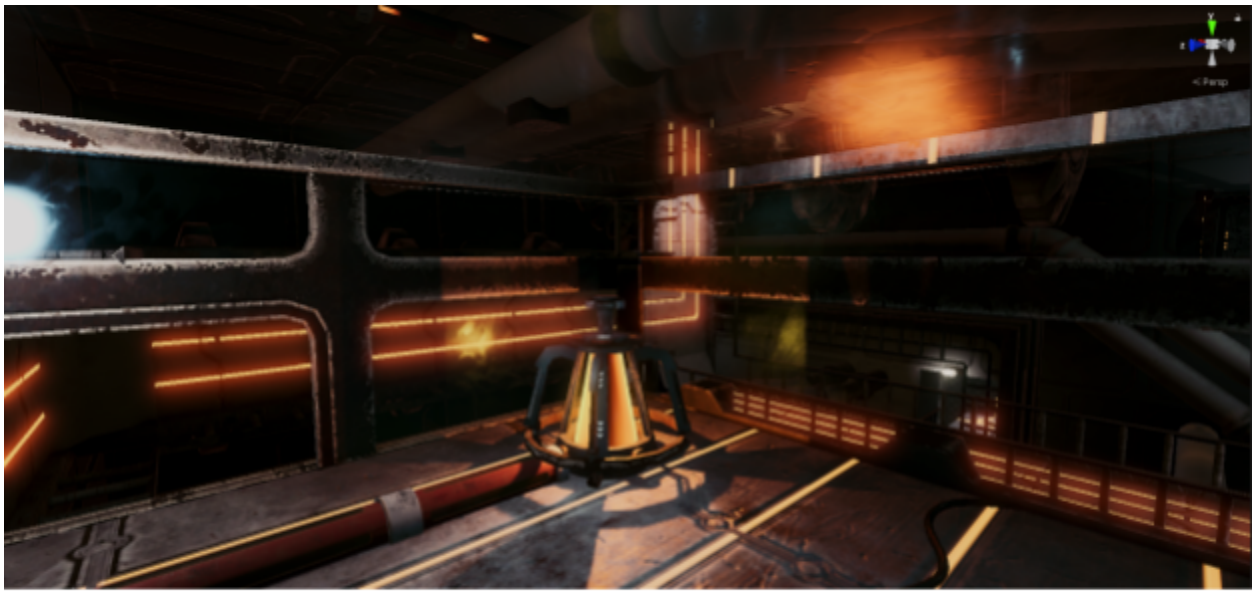


Figura 40: Imagen con tratamiento de color REINHARD

Ahora repasaremos todos los efectos de *color grading* y ajustes que soportamos. Como estos son estándares de la industria, solo mencionaremos su formulación y enseñaremos algunas imágenes de los resultados:

- Sin efectos:

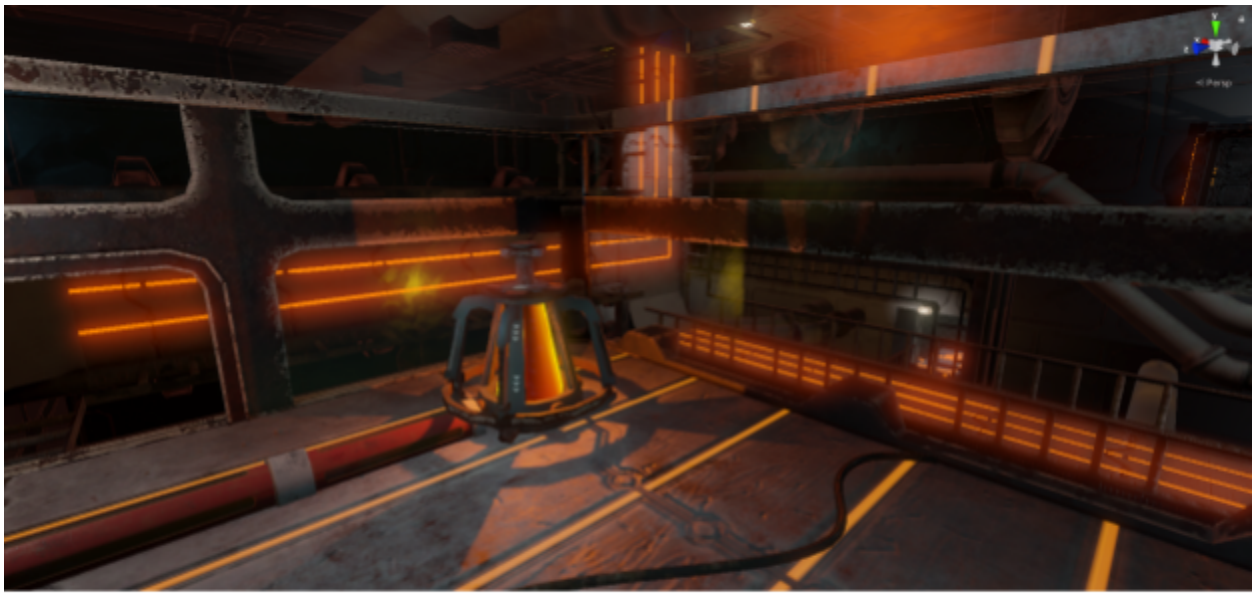


Figura 41: Imagen sin efectos

- Post Exposición: $c * 2^{k}$

Figura 42: Imagen con distinta exposición

- Contraste: $(c - ACEScccMidGray) * (k * 0.01 + 1) + ACEScccMidGray$

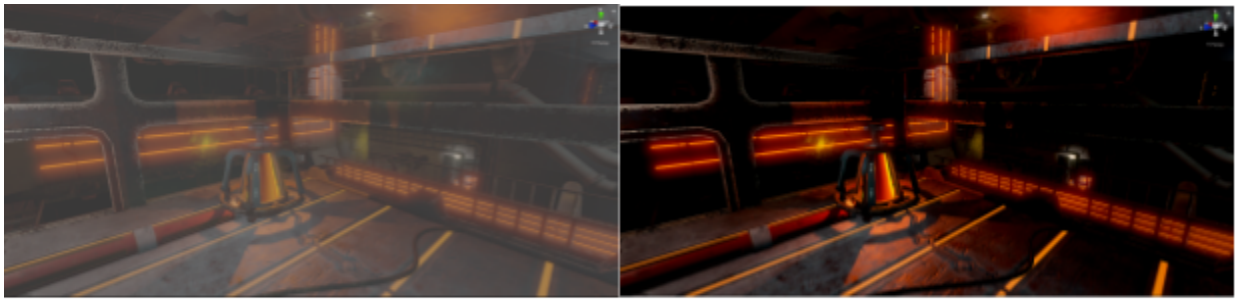


Figura 43: Imagen con diferente contraste (-50 y +50)

- Filtro de Color: $c * c_1$

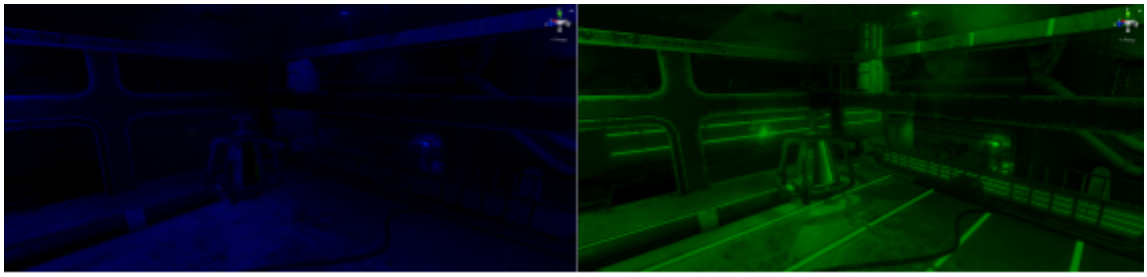


Figura 44: Imagen con filtro de color (0,0,1) y (0,1,0)

- Hue Shift: tenemos que transformar el color de RGB a HSV, añadir el shift a la H y convertir otra vez a RGB:

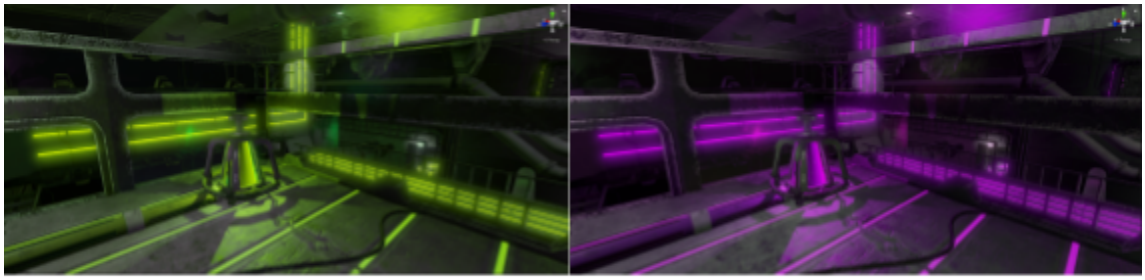


Figura 45: Imagen con HUE shift (+90 y -90)

- Saturation: $(c - l) * (k * 0.01 + 1) + l$

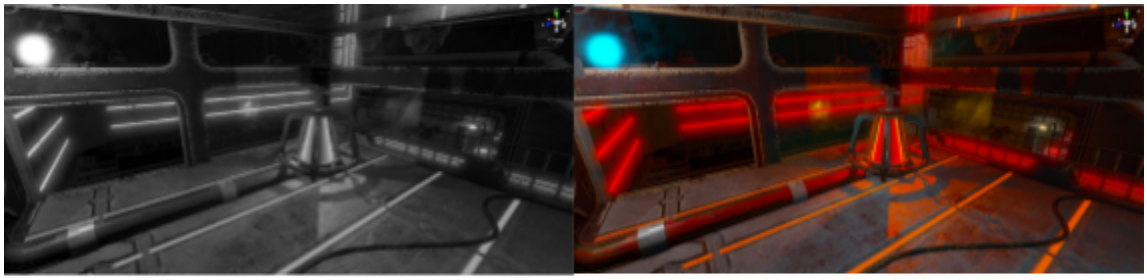


Figura 46: Imagen con distinta saturación (-100 y +100)

- White Balance: Solo tenemos que cambiar el color a LMS y multiplicar el valor

Figura 47: Imagen con diferente (temperatura -100 y 100)

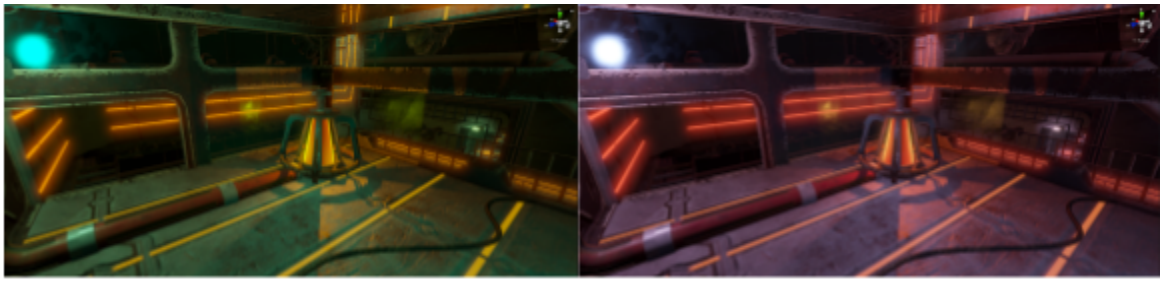


Figura 48: Imagen con diferente tinte (-100 y 100)

- Split Toning: diferencias entre sombras y los destacados con elevados y elevados negativos

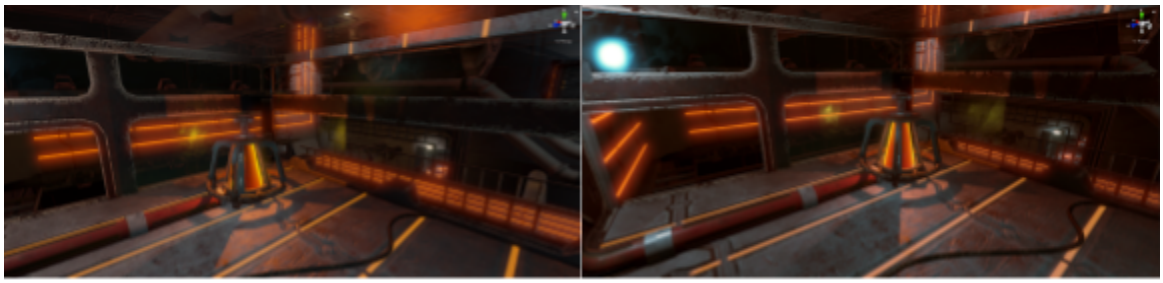


Figura 49: sin efectos y con sombras(0.5, 0.3, 0.1) y luces(0,0.6,1)

- Channel mixel: multiplicación de matriz

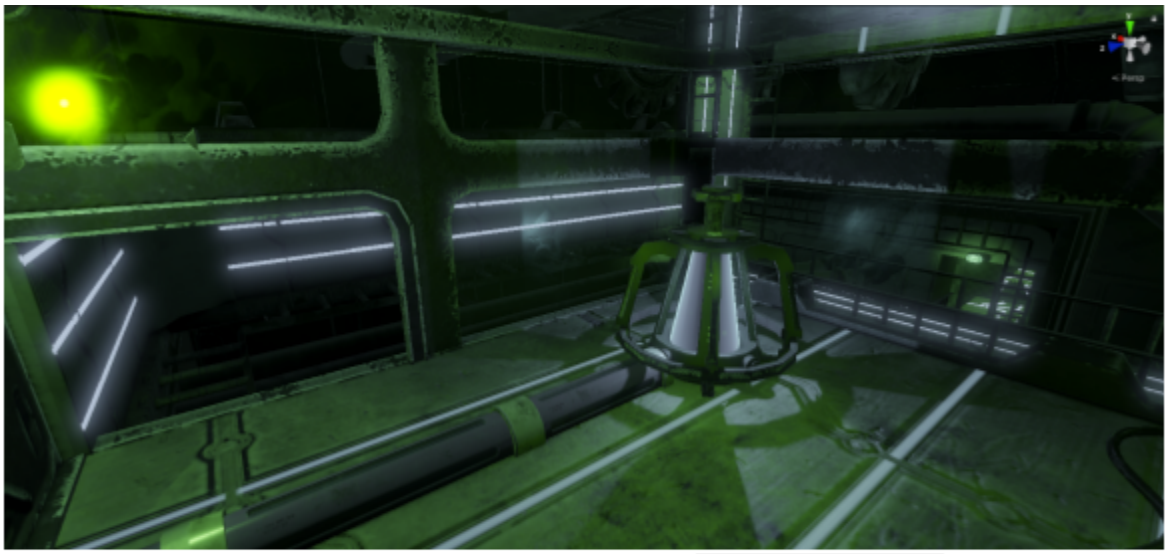


Figura 50: Channel mixer con matriz

X 0.14	Y 0.09	Z 0.11
X 0.2	Y 0.44	Z 0.6
X 0.31	Y 0.35	Z -0.45

ACESccMidGray = 0.4135884
c = color
l = luminance

Todos estos efectos son caros en tiempo de ejecución, así que calcularlos todos es muy caro. Para hacer todos estos cálculos implementamos el mismo sistema de LUT (Look Up Table) que Unity para poder codificar los colores en la textura 3d y simplemente hacer un muestreo en ella. Por suerte las librerías de Unity vienen con una función llamada *GetLutStipValue()* que hace exactamente esto, lo único que tuvimos que modificar fue cambiar de espacio Log C a lineal para soportar HDR en nuestro LUT.

El último detalle que nos queda por comentar es dónde estamos renderizando todos estos efectos. Unity usa la función Blit para copiar la imagen generando en este proceso un *Quad* del tamaño de la pantalla y copiando la imagen en él. Como esto es un proceso que se hará todos los frames decidimos usar solo un triángulo en pantalla escalándolo como se muestra en la figura 51.

Esto requiere de un poco de trabajo para reescalar las UVs, pero aunque parezca trivial vale la pena por dos razones: La primera es que reducimos la geometría a la mitad, y la segunda y más importante, es que si usamos la aproximación de Unity y pintamos un *quad*, los píxeles que caen justo entre los dos triángulos que conforman el *quad* se pintan dos veces ya que forman parte de los dos triángulos.

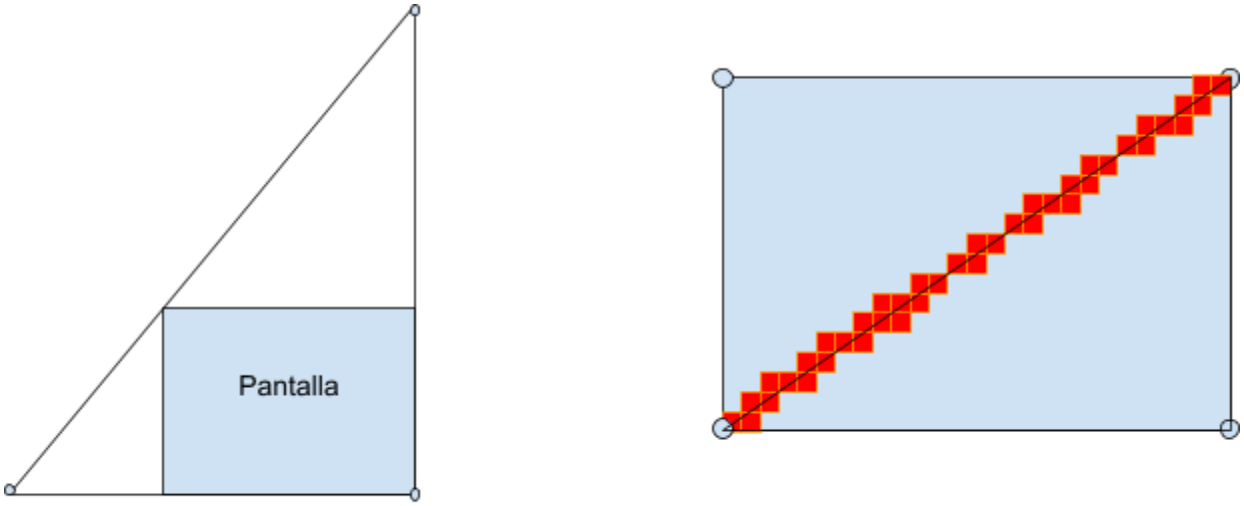


Figura 51: Renderizado custom (1 triángulo) VS renderizado estandar (1 quad)

Esto es de vital importancia porque por cada píxel se ejecutan todos los post fx, y teniendo en cuenta que las pantallas tienen millones de píxeles, usando un cuad estamos recalculando los post fx dos veces para una gran cantidad de píxeles, realizando así un trabajo inutil que reduce nuestro rendimiento si lo comparamos con la aproximación de usar un único triángulo.

Inverse kinematics

En nuestro equipo de desarrollo solo hay 2 artistas y ninguno de los dos son animadores, lo que nos plantea un problema: ¿Cómo animaremos los personajes del juego?

La solución a este problema viene del equipo de programación: como no podemos dedicar los recursos de arte de que disponemos a animar, decidimos que las animaciones que necesitemos serán procedurales usando un sistema de cinemática inversa o IK (Inverse Kinematics). Esta decisión marcará fuertemente el diseño ya que es una tarea muy grande y compleja, así que para hacerla razonable para nuestro equipo y tiempo, reducimos los elementos animados del juego al mínimo, quedando solo el personaje principal. Además, el diseño de este se hizo pensando justamente en que sería animado proceduralmente. Es por eso que decidimos crear un robot, alejándonos de seres vivos que resultan mucho menos creíbles al estar animados con algoritmos en vez de a mano.

Diseño

Para nuestro solver usaremos FABRIK, un algoritmo para implementar cinemática inversa muy versátil, con soporte para restricciones y no muy caro de ejecutar. El único problema es que la complejidad es directamente proporcional al número de nodos que conforman la estructura que intentamos mover, por lo tanto nuestro robot tendrá solo 3 nodos: 2 codos y la punta de la pata. Además decidimos que fuera un robot porque como es un sistema basado en nodos y tendremos pocos no podremos simular nada que tenga un movimiento suave como por ejemplo un tentáculo. Necesitamos movimientos mucho menos fluidos, como lo son los de un robot.

Figura 52: Imagen del personaje

El diseño del robot nos da una ventaja más, no necesitamos que sea una *skined mesh*. Este tipo de mallas se usan para todo lo que tenga huesos y animaciones, ya que permiten flexión en la superficie de la malla, dando un aspecto mucho más creíble para cuerpos blandos. El problema de estas mallas es que son mucho más caras de renderizar, así que como nuestro diseño de un robot metálico no necesitamos esa flexión y por tanto nos podemos ahorrar usar este tipo de mallas y simplemente importar las piezas y controlar su posición y rotación por código.

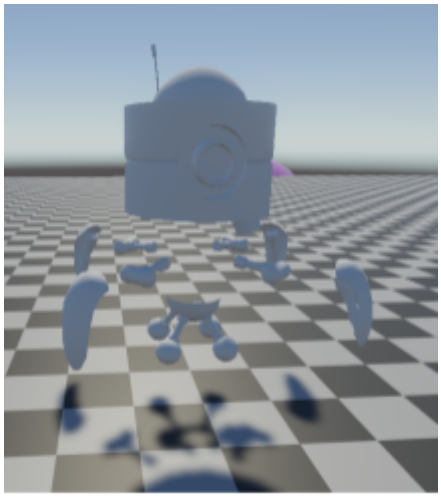


Figura 53: Piezas del personaje

Lo primero y más importante será definir qué estructura tendrá la jerarquía de nuestro robot para poder definir el esqueleto y las partes movibles.

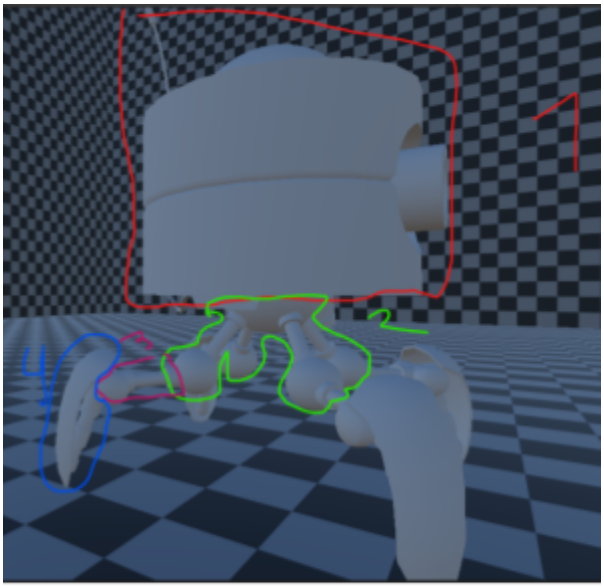


Figura 54: Partes que conforman el personaje

En la figura 53 podemos ver todas las piezas que tiene el robot y en la figura 54 cómo van a estar conectadas y qué estructuras forman.

Necesitamos mantener las piezas uno y dos (mostradas en la figura 54) separadas porque queremos poder expresar emociones a través del movimiento de la cabeza del robot, pero estas dos piezas no forman parte de la animación por IK. Únicamente las partes tres y cuatro se verán afectadas por el sistema de IK, de esta forma con la pieza dos separamos la cabeza del suelo y separamos también los puntos de unión iniciales entre sí lo suficiente como para evitar que puedan colisionar los unos con los otros y evitarnos de esta forma restricciones de colisión entre ellos, haciendo así el *solver* muchísimo más sencillo.
Como se puede ver en la figura 55 el movimiento no se siente extraño aunque solo se muevan dos partes de la pierna para caminar.

IK

Que usemos FABRIK no significa que las piernas anden de forma mágica, con este algoritmo lo que conseguimos es que la estructura que hemos definido se recoloque para simular que "persigue" o intenta "coger" el objetivo al que llamaremos *Target.* En la figura 55 el target está representado por la esfera de color rojo.

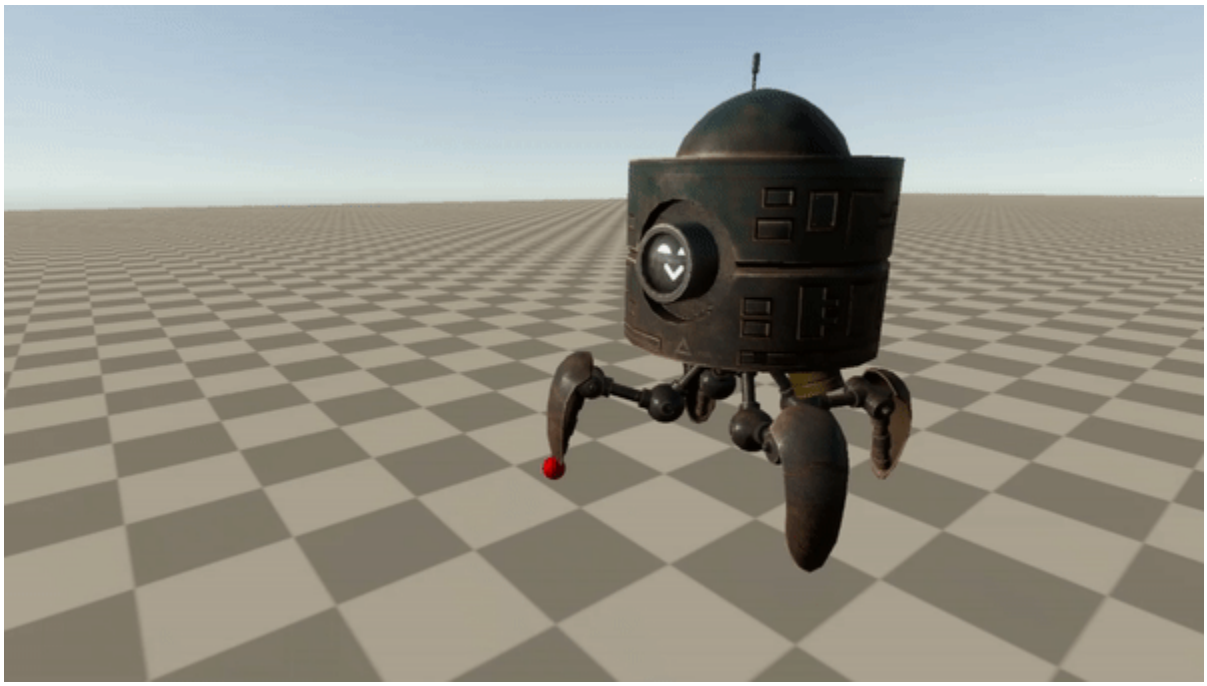


Figura 55: FABRIK en la pierna con el punto rojo como target

Cabe destacar que en el sistema que hemos implementado está desacoplado el solver de reposicionamiento (en nuestro caso FABRIK) del sistema procedural que

decide cómo, cuándo y dónde mover las piernas y demás partes del cuerpo, y a su vez todo esto está desacoplado por completo del movimiento del jugador.
En realidad el jugador mueve una esfera invisible que es la que calcula colisiones, posición y rotación y es encima de esta esfera que se monta el cuerpo del player que persigue a este controlador

Así que, ¿cómo animamos al personaje?
Como hemos mencionado, el movimiento está separado en diversas partes:

- Posicionamiento
- Piernas
- Cuerpo

Antes de empezar tenemos que aclarar unos conceptos que se repetirán a través de todas las fases del sistema:

- Cada pierna tiene un controlador FABRIK que se encarga de asegurar que esa pierna apunta a lo que nosotros llamaremos *leg target*. De esta forma, si modificamos la posición de *leg target* esa pierna se moverá con él.
- Cada pierna también tiene una *resting position*, que define la posición que tendría la pierna si estuviera en reposo. Esta *resting position* por tanto se mueve de forma local acompañando al cuerpo del robot.

Con esto aclarado podemos empezar a revisar el sistema. En la código 9 se muestra el código que se ejecuta cada frame:

```
void Update()
{

   upDirection = CustomGravity.GetUpAxis(player.position);

   SetPositionAndRotationToPlayer();
   if (!OnAir)
       MoveLegs();
   else
       MoveLegsAir();

   if (lastCheck == checkLegsFrameInterval)
   {
       CheckAllMovingLegsTargets();
       lastCheck = 0;
   }

   lastCheck++;

   lastPosition = translationTransform.position;
}
```

Código 9

Lo primero que hace nuestro código es llamar a la función *SetPositionAndRotationToPlayer()*.
Esta función hace diversas cosas, la primera ajustar la posición de nuestro robot a la posición del jugador.

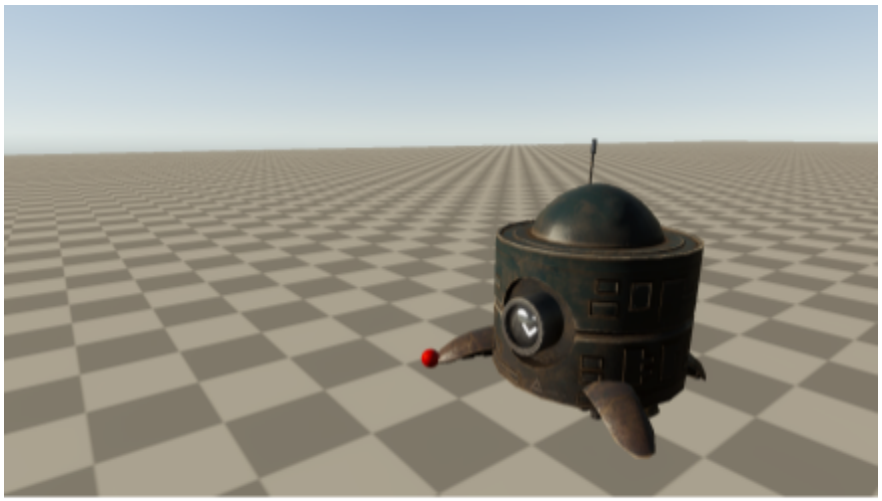


Figura 56: Posicionamiento sin correcciones

Como se ve en la imagen solo con esto el player no se ajusta correctamente al suelo, así que esta función ajusta también la altitud del cuerpo para que se mantenga siempre más o menos a la misma altura del suelo y que las piernas tengan suficiente espacio para moverse.

En este punto nuestro robot siempre estaría alineado con la gravedad y tendríamos comportamientos extraños donde el robot no respondería a la superficie que tiene debajo.

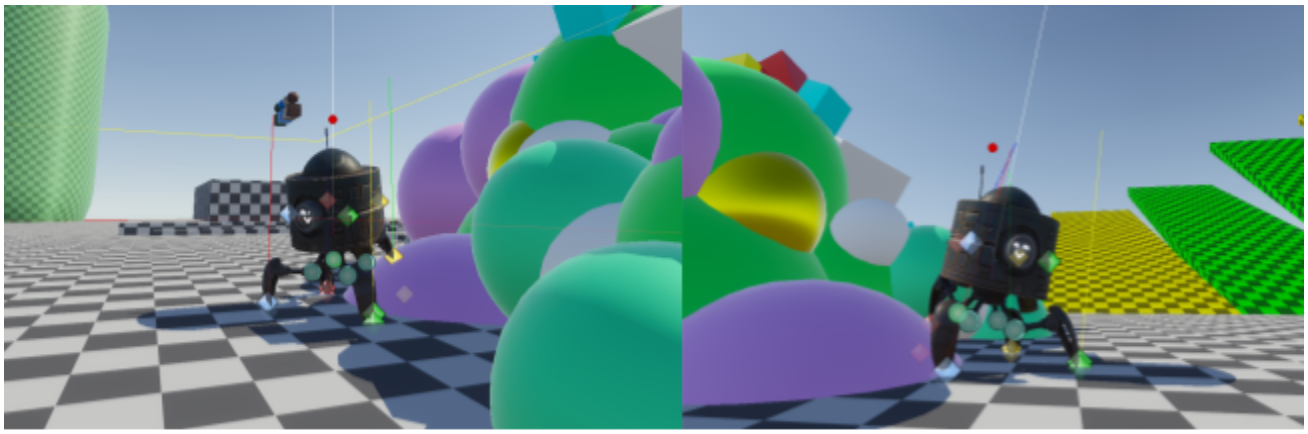


Figura 57: Sin aplicar la inclinación de las patas VS Con la inclinación de las patas

Para que el robot responda adecuadamente al entorno calculamos a cada *frame* la normal que generan las cuatro patas. Para eso hacemos el producto escalar de cada pata con sus adyacentes, sumamos todos los vectores y los dividimos entre cuatro.

En la imagen 57 podemos ver el vector que forma cada pata con las de sus lados (de colores) y el resultado final (en blanco) que define el vector hacia arriba de la cabeza. Con esto conseguimos que el robot se oriente bien con el suelo, pero además conseguimos que mueva un poco la cabeza cada vez que mueve una pata, ya que este movimiento contribuye a la inclinación calculada mediante el producto escalar de las patas. Esta inclinación hace que el robot se sienta más natural puesto que el movimiento de las patas se traduce en movimiento en el cuerpo/cabeza.
No obstante descubrimos en los *testings* que el movimiento se seguía sintiendo estático, así que le implementamos un movimiento que traduce el movimiento vertical de las patas al movimiento vertical en la cabeza, con lo que conseguimos un movimiento más natural en el cuerpo.
Pero ¿cómo movemos las patas?
Para empezar nuestras piernas pueden estar en 3 estados, *Ready*, *Moving* y *Done*, de esta forma podemos saber qué piernas están listas para moverse, cuáles se están moviendo y para hacer el código más sencillo, también marcamos las que ya han acabado de moverse pero aún no hemos pasado a Ready.
Este sistema de estados es importante porque no queremos que se muevan todas las patas a la vez o de forma aleatoria, queremos tener restricciones que hagan que las patas solo se puedan mover una si y una no, como un insecto:

Figura 58: Matriz de restricción de las patas

Acabamos implementando un sistema de limitación de movimiento, de esta forma podemos definir que la pata cero solo se puede mover si las patas uno y dos están en el suelo. De esta forma podemos modificar el tipo de restricciones según el momento, por ejemplo si el robot tiene miedo podemos limitar las patas con todas

las demás y que mueva solo de una en una como si fuera más asustadizo, o si está corriendo hacer las limitaciones a la inversa para que galope como un perro.

Ahora que ya sabemos cuando una pata puede estar lista para moverse tenemos que decidir a dónde se moverá. Es por eso que cada vez que una pata entra en el estado de *Ready*, si puede moverse se le asigna un *movementStart* y *movementEnd*. El *movementStart* es la posición que tiene la pata justo al empezar a moverse y el *movementEnd* es una posición que representa el lugar ideal al que le gustaría estar a la pata sin tener en cuenta las restricciones del terreno. Una vez tenemos esta posición le pedimos al *EnvriomentQuerrySystem* que hemos desarrollado que nos devuelva la posición superficial más cercana a este punto, asegurándonos así que la pata siempre acabe en una superficie. Por último, marcamos la pata como *Moving*.

Ahora que la pata esta en estado de *Moving* en el siguiente frame empezará a actualizarse, y para ello interpolamos entre el *movementStart* y el *movementEnd*.
Si aplicamos esta interpolación directamente, la altura de la pata nunca cambiaría, solo se movería hacia delante o los lados, como si se moviera arrastrando las patas por el suelo.
Este sería un comportamiento erróneo y para remediarlo añadimos altura durante esta interpolación para crear un arco, levantando así la pierna durante el trayecto y bajándola de nuevo para llegar al punto de destino.
Pese a que el sistema sea procedural, queremos darles a los artistas herramientas para transmitir sensaciones a través de su movimiento, es por ello que las siguientes dos curvas para definir cómo se moverán las patas:

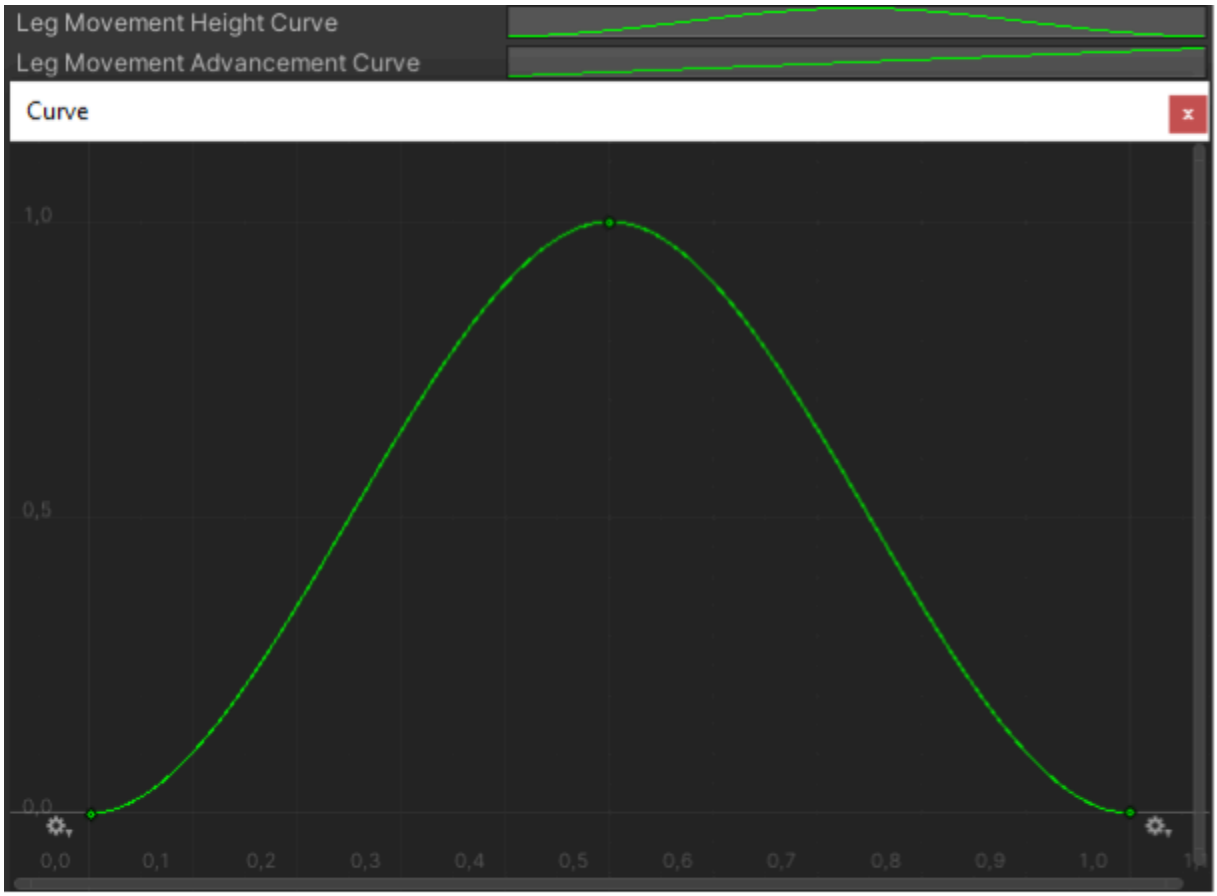


Figura 59: Curva de movimiento de las patas

En el editor nuestros artistas pueden jugar con la altura y avance en cada punto de la interpolación de esta forma podemos crear pasos más altos y cortos para representar prudencia, o pasos más largos y redondos para transmitir confianza.

Una vez implementado todo esto tendríamos el movimiento de las patas implementado.
Aún así, en nuestro caso hacemos una diferenciación más donde el personaje tiene distintos estados para poder definir distintas curvas que describen distintas maneras de mover las patas, de esta forma podemos definir distintos estilos para caminar, saltar, correr...

Pese a no pertenecer estrictamente al sistema de IK, cabe destacar la importancia del *EnvironmentQuerryManager*, el sistema que nos permite saber dónde se debe colocar exactamente el pie teniendo en cuenta la geometría de la escena.
Este es uno de los pasos más complejos y costosos. Normalmente esto se hace lanzando rayos en diversas direcciones y detectando a que distancia esta el collider con el que ha colisionado.

Esto es muy limitante ya que sólo puedes encontrar un punto que esté en el vector que describe el rayo, así que si se quiere que se coloque la pierna de forma precisa donde realmente debería, será necesario lanzar una enorme cantidad de rayos. A mayor precisión, mayor cantidad de rayos, hasta volverse insostenible. Para evitar eso desarrollamos nuestro *EnvironmentQuerryManager*, un sistema que recoge toda la geometría de la escena, la divide en cuadrantes y estructura toda la información en diccionarios de rápido acceso para que podamos acceder de forma barata y eficiente.
Gracias a tener toda la geometría repartida en cuadrantes podemos encontrar el punto más cercano en superficie a otro punto comparando un pequeño número de mallas, que son las que estén dentro de ese cuadrante. Este método nos ha permitido hacer los cálculos entre 100 y 400 veces más rápido.

Para poder ver el funcionamiento del sistema en editor hemos preparado el siguiente video:
https://www.youtube.com/watch?v=XJBQ7lGtA5U

Shaders

Generator

Como en el shader Lit, el del generador necesitará tener el shader original para exponer las variables y los pases, que serán los mismos que el del Lit:

- Lit pass
- ShadowCaster pass
- Meta Pass

En este shader podemos ver lo potente de esta estructuración de shader en sub shaders, como se ha explicado esta separado en documentos distintos porque de esta manera, mientras sigamos el estandard y usemos los mismos nombres de función podemos reciclarlos, de esta manera podéis ver que en en shadowcaster pass incluimos *ShadowCaserPass.hlsl* que es el shader estándar. Como en este shader no hacemos nada raro con las sombras lo podemos aprovechar.

```
...
Pass {
  Tags {
     "LightMode" = "ShadowCaster"
  }

  ColorMask 0

  HLSLPROGRAM
  #pragma target 3.5
  #pragma shader_feature _ _SHADOWS_CLIP _SHADOWS_DITHER
  #pragma multi_compile _ LOD_FADE_CROSSFADE
  #pragma multi_compile_instancing
  #pragma vertex ShadowCasterPassVertex
  #pragma fragment ShadowCasterPassFragment
  #include "ShadowCasterPass.hlsl"
  ENDHLSL
}
...
```

Código 10

Aun no sabemos nada de este shader, por narrativa necesitaban que el generador se entenderá sin ninguna explicacion, asi que tenia 3 requisitos:

- El shader tiene que transmitir potencia, tiene que parecer que es muy enérgico.
- Se tiene que poder diferenciar claramente entre encendido y apagado
- Tiene que captar la atención del jugador.

Para ello me inspiré en lo siguiente:

- Argent Energy de Doom
    - https://www.google.com/search?q=doom+energy+generators&tbm=isch&chips=q:doom+energy+generators,online_chips:argent:JlB0UZXLDUo%3D&rlz=1C1CHBF_esES872ES872&hl=en&sa=X&ved=2ahUKEwiKrYK-0KPvAhUH4hoKHcKwAB0Q4lYoAXoECAEQHA&biw=1261&bih=927#imgrc=Sg3LlVF0UCaByM
- Subnautica Below Zero Al-An Precursor
    - https://www.youtube.com/watch?v=CgKujpmpGuE

Tengo estas dos referencias porque quiero que el generador se sienta tan energético como el de doom pero se entienda muy bien cuando está activo y cuando no como el de subnautica. Aun así tengo una limitación y es que el generador no tiene partes móviles, no como el de subnautica.

Este es el resultado del shader final: https://www.youtube.com/watch?v=-mK9BdqHdFo

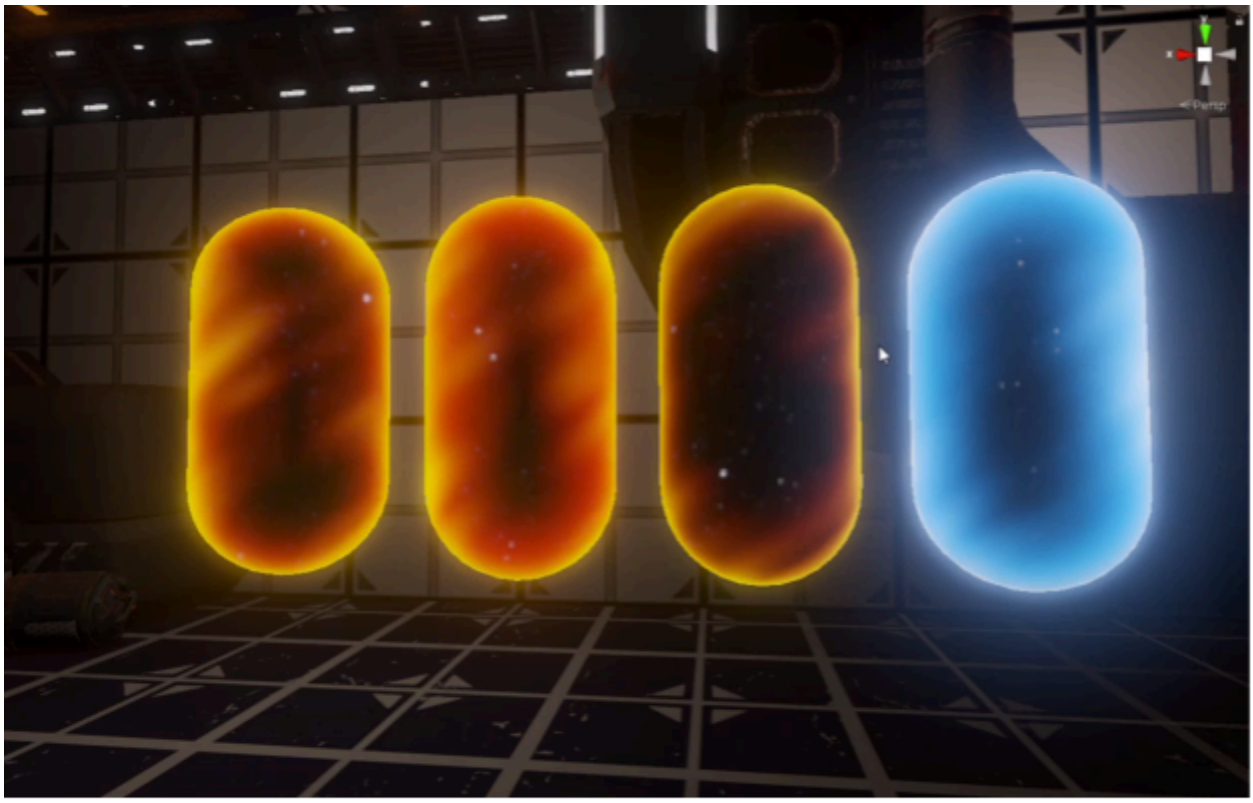


Figura 60: Shader en capsulas

Creí que la mejor forma de transmitir potencia era que el objeto se sintiera inestable, como a punto de explotar, para ello hice dos cosas: La primera diseñar unos

patrones que se mueven por toda la superficie de el objeto de forma direccional, de ellos se puede controlar la dirección y velocidad, además de la forma, y segundo que brillara mucho, por suerte como nuestro renderer soporta bloom y HDR esto no fue tarea difícil, emitiendo un color HDR tuve suficiente.

La gran mayoría de la lógica del shader pasa en la función de GetEmission de GeneratorEnergyInput.hlsl porque no quiero que tenga iluminación ni volumen, quiero emitir colores potentes.

```
float3 GetEmission(InputConfig c,float3 normal, float3 viewDirection, float2 screenPos) {

	float outerNoiseMask = FresnelEffect(normal, viewDirection, GetFresnelStrength());
	float3 fresnellNoiseAndRotation = GetFresnellNoiseScaleAndRotation();//xy scale, z
rotation

	outerNoiseMask = outerNoiseMask * GradientNoise(RotateUV(c.baseUV + float2(0, _Time *
GetFresnelNoiseSpeed()), fresnellNoiseAndRotation.xy/2.0f,
fresnellNoiseAndRotation.z)*fresnellNoiseAndRotation.xy, 10);
	outerNoiseMask *= saturate(GradientNoise(c.baseUV + (float2(_CosTime, _SinTime) *
GetOverlaySpeed()),4) - 0.3);
	outerNoiseMask += FresnelEffect(normal, viewDirection, 4);
	float inverseOuterNoiseMask = abs(1-outerNoiseMask);

	float blackInsideMask =  saturate( FresnelEffect(normal, viewDirection,
GetCenterStrength())) + outerNoiseMask;

	float3 color =((outerNoiseMask * GetFresnelColor().xyz) + inverseOuterNoiseMask *
GetInnerColor().xyz) * blackInsideMask;

	return color + (GetFresnellNoise(screenPos) * abs(1-blackInsideMask));
}
```

Código 11

Este shader está basado en máscaras, esto es porque quiero evitar hacer branching o que las zonas tengan que estar predefinidas en la mesh o textura ya que en el momento en que escribí este shader aún no estaba claro cómo sería el generador y además todas estas opciones son menos óptimas a nivel de memoria o performance.
Por lo tanto, como está basado en máscaras se puede ver en el shader que genero tres de ellas, outerNoiseMask. blackInsideMask y invereseOuterNoiseMask. Con estas tres máscaras se definen las siguientes tres zonas:

Figura61: mascaras

Y el resto de operaciones simplemente definen el tamaño y color de estas máscaras.
El último detalle de este shader es que para darle un toque más místico decidí que la zona 3 en vez de ser un color sólido servirá de ventana para ver una imagen del espacio en screen space para romper un poco con lo que espera el jugador, como por ejemplo, los portales al end de minecraft.

Ice

Otro shader que los artistas podrían necesitar es un shader de hielo ya que según la narrativa todo el planeta está congelado fuera de las instalaciones en las que transcurre el jeugo.
Hacer un shader de hielo solo tiene una complicación más allá del PBR, el hielo es translúcido, la luz tiene que pasar a través de él.
Si quisiéramos que el jugador pudiera ver a través del hielo de forma realista necesitamos o pintar el hielo como si fuera un post procesado para poder ver a través de él, pero esto nos daría muchísimos problemas de sorting, sobretodo con otros objetos transparentes como las nieblas. Otra opción sería implementar un raymarcher y que fuera un efecto volumétrico, esto lamentablemente sería demasiado costoso para las plataformas a las que nos dirigimos.
Por lo tanto la solución final fue implementar Subsurface Scattering, para ello me inspiré en la implementación de este efecto que tienen en Frostbite engine y que explicaron aquí:

https://colinbarrebrisebois.com/2011/03/07/gdc-2011-approximating-translucency-

for-a-fast-cheap-and-convincing-subsurface-scattering-look/

Aun así este efecto aún es un poco caro ya que nosotros podremos asumir más cosas sobre la geometría que se va a usar, no necesitamos local thickness ya que los modelos que vamos a usar para el hielo serán mucho más simples y simétricas que las que se usan en esta presentación.

Así que la implementación final en realidad es muy simple, se calcula la iluminación normal para las caras que están siendo afectadas directamente por una luz, y esta misma iluminación que ya hemos calculado se reutiliza para la cara contraria solo que en la emisión, de esta forma la podemos tintar de un color para simular que la luz ha estado rebotando en el interior de la mesh y ha salido por el lado contrario, cuando en realidad simplemente la cara opuesta está emitiendo la misma luz que hay en la cara original pero tintada. A continuación hay imágenes sobre el resultado final, pero recomiendo encarecidamente ver el siguiente video para poder ver realmente cómo funciona::
https://www.youtube.com/watch?v=QaVoCr3LZ-Y

Esta implementación nos da un par de resultados muy interesantes que es importante mencionar, el primero es que como la iluminación se proyecta de una cara a la interior, si otro objeto proyecta sombra sobre el objeto esta también se ve en la cara contraria dando la sensación que la luz está ocluida.

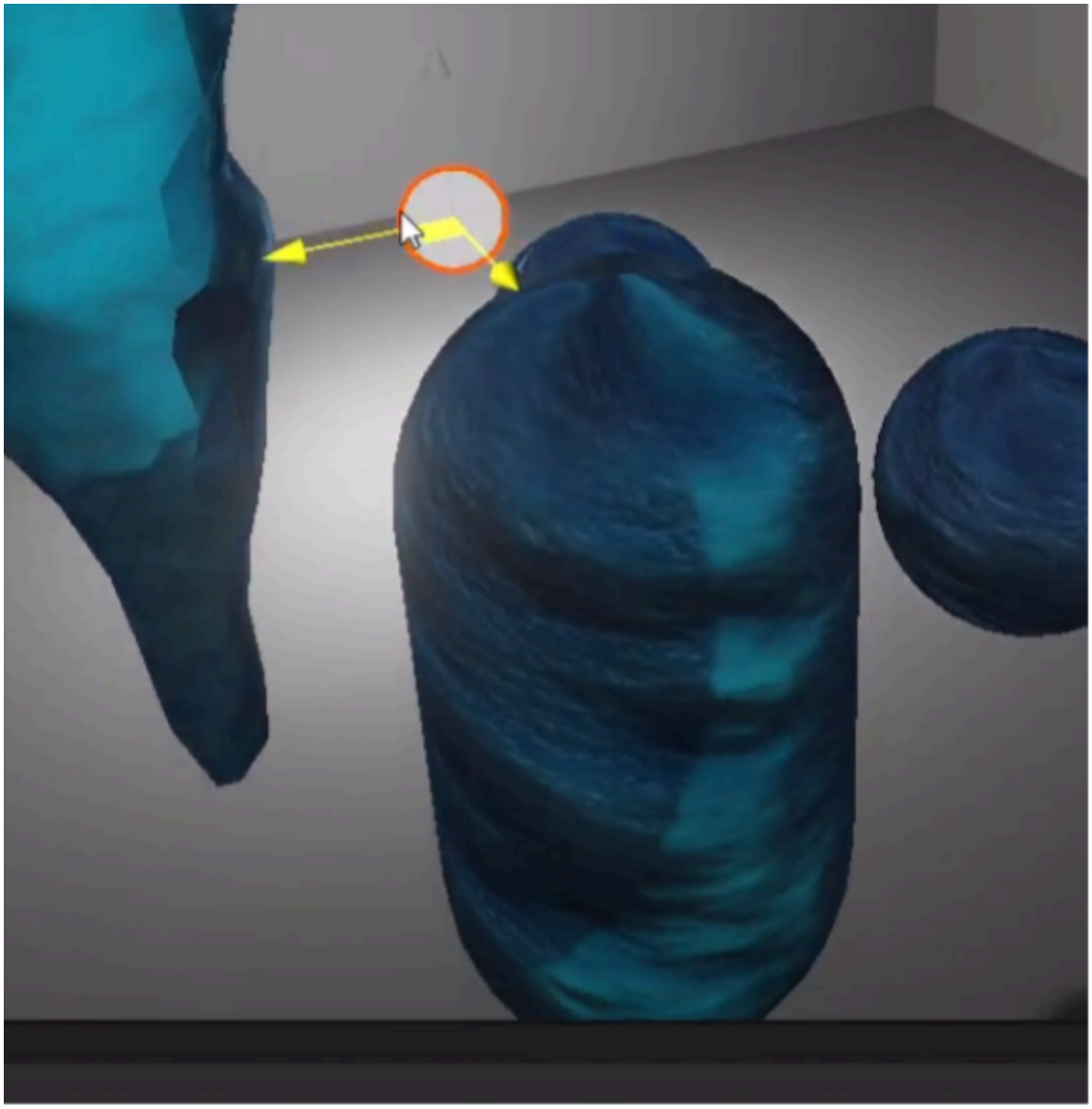


Figura 62: Sombras en SSS

Segundo, como el renderer es custom y toda la información de la luz esta disponible, es muy sencillo que la intensidad y color de la luz afecten al efecto, incluso teniendo un color interior, en esta imagen es sutil, pero la luz es blanca, y la que sale por la cara anterior de la cápsula está tintada de azul, el color del interior de la mesh.

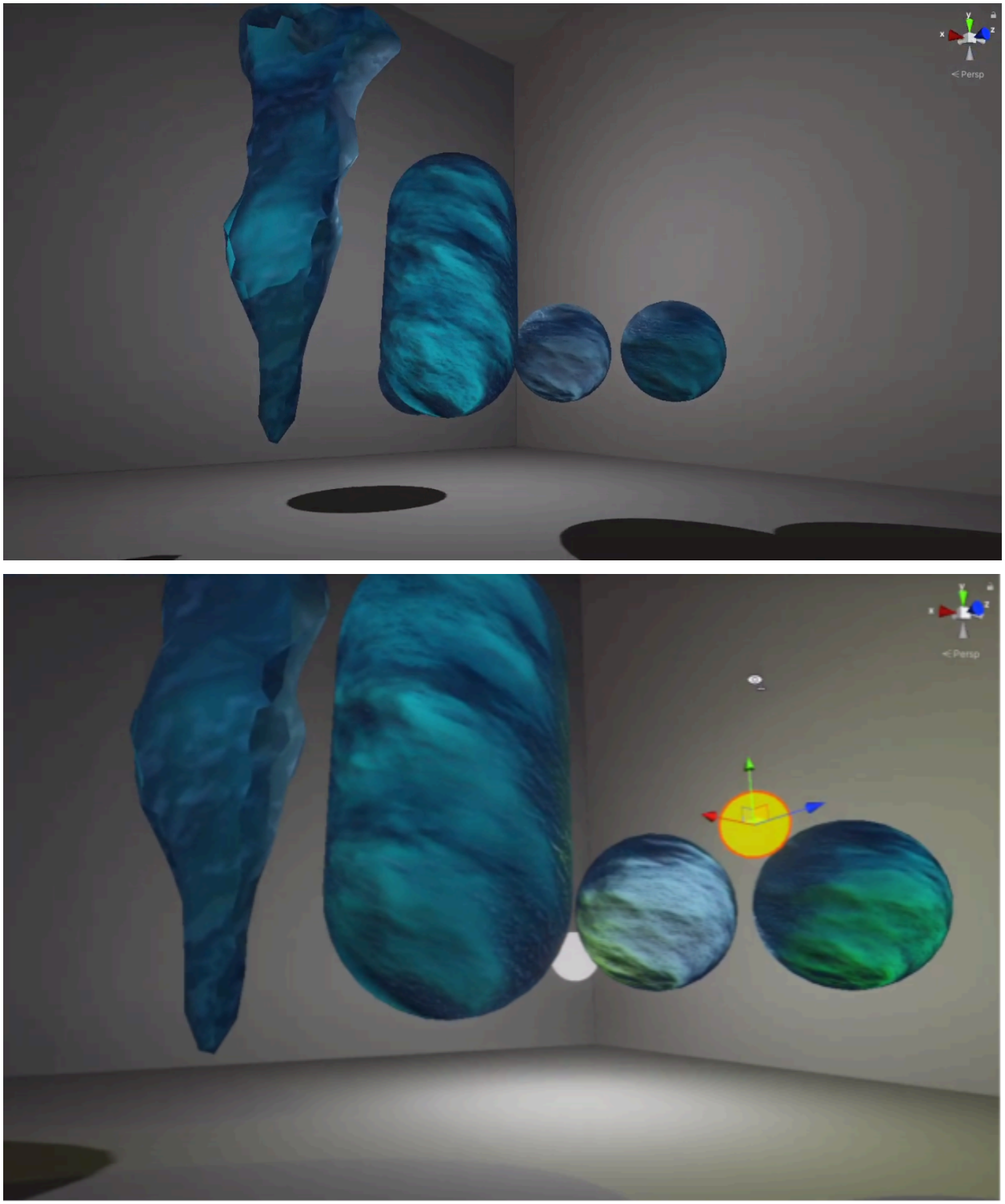


Figura 63: Ice shader con distintos colores

Resultados del testeo

Durante el desarrollo de la *Vertical Slice* hemos realizado 4 etapas de testeo distintas, todas usando la misma metodología.

Metodologia

Todas las sesiones se han realizado de forma individual (un solo tester en la sala) y con la grabación de la pantalla de juego y el audio del jugador. No obstante, a falta de una autorización explícita de los testers, no podemos compartir el material de las sesiones.
Durante la sesión de juego en sí, el encargado de llevar el testeo tenía prohibido interactuar con el jugador de ninguna forma para no influir en el testeo.

Es importante tener en cuenta que la mayoría de jugadores no están familiarizados con juegos aún en desarrollo, por eso lo primero que hacíamos en las sesiones era explicar al tester que es un whitebox y la importancia de estos tests. De esta manera, logramos por un lado minimizar el impacto negativo que puede producir en el jugador encontrar una zona hecha con cajas en vez de con arte final, y por otro aumentar el *feedback* de los testers al entender que son partícipes en el desarrollo del juego y que el trabajo que están realizando es relevante. Siguiendo con esta línea, otra imposición para los responsables de los testeos fue no discutir nunca un *feedback* de los testers, ya sea porque no se está de acuerdo, o porque es una información que ya se tenía o no es relevante para el test en concreto.
Durante la charla previa a la partida también se animaba a los testers a que pensaran en voz alta durante toda la sesión, ya que entender el flujo de pensamiento que lleva al jugador a actuar es tan o más valioso como el feedback que te da el tester activamente.

Una vez acabada la sesión de juego, se realiza una entrevista distendida con el tester. En la primera fase de testeo pusimos a prueba dos aproximaciones distintas: una charla distendida y una encuesta pre redactada.
Comparando ambas opciones, nos dimos cuenta que los testers que habían participado en la charla distendida habían sido mucho más extensos en sus aportaciones, además de acabar sacando temas distintos que no habíamos planteado en las encuestas a raíz de charlar con ellos.
Así pues, para el resto de test adoptamos la opción de realizar una charla distendida sin un guión prefijado. Esto no significa que no hubiera unos temas base que

sacaramos siempre en las charlas, como por ejemplo cuál ha sido su puzle favorito, o cuál ha sido el que menos le ha gustado.

Etapas de testeo

- Primera etapa
  La primera etapa se realizó durante la fase de validación de las mecánicas e incluía el *whitebox* del primer nivel.
  El objetivo de este primer test era validar las mecánicas principales del juego, asegurando que fueran interesantes, entendibles y potencialmente divertidas.
  Los resultados de esta primera etapa fueron realmente buenos. La mecánica principal se entendió correctamente por prácticamente la totalidad de los jugadores, y del mismo modo, gran parte de ellos se sorprendieron con la misma, cumpliendo así los objetivos que nos propusimos y dando luz verde a avanzar con el desarrollo.
  El único problema sistemático que sí detectamos en este playtest fue en el último puzle, donde enseñamos al jugador que las gravedades también interactúan con el mundo, pero sin embargo esta interacción ocurre de espaldas al jugador debido cómo está colocado el generador que el jugador debe activar. Es por ello que decidimos cambiar esa parte para asegurar que el jugador viera en cámara cómo desactivar ese generador afectaba al mundo.

- Segunda etapa
  La segunda etapa comprendia el nivel uno ya vestido y con los cambios del primer testeo, y el nivel dos en fase de *whitebox.* De esta forma, en la misma etapa pudimos validar el estilo artístico, y comprobar cómo funcionan los puzles de gravedades una vez aprendidas las mecánicas principales.
  Los pequeños cambios introducidos en el nivel uno para facilitar la comprensión de las mecánicas fueron todo un éxito, pero el nivel dos fue bastante más complicado.
  Los testeos mostraron por un lado que faltaba claridad en los puzles y los jugadores se atascaban o desorientaban con facilidad, pero también mostraron un gran interés en el tipo de puzles planteados y su originalidad. Esto dejaba claro que había que repensar el nivel para hacerlo más entendible para los jugadores, pero también nos confirmó que íbamos en la línea correcta, ya que incluso con la mala sensación que produce atascarse, los jugadores valoraron muy positivamente las mecánicas tras los puzles planteados.

- Tercera etapa
  Esta vez, en lugar de seguir el desarrollo de los test anteriores y mostrar el nivel anterior (en este caso el dos) vestido y el nuevo en *whitebox*, contamos con ambos niveles en *whitebox*, debido a que tuvimos que rediseñar gran parte del nivel dos para hacer la progresión más entendible y fácil de seguir para los jugadores. Eso sí, respetando las mecánicas tras los puzles que ya había y que habían gustado a los jugadores. Por tanto, si teníamos que mantener los puzles pero mejorar la progresión, tuvimos que optar principalmente por producir más puzles intermedios.
  Esta tercera etapa funcionó bastante bien. El segundo nivel rediseñado era mucho más asequible, aunque es cierto que seguía siendo el nivel que más les costaba a los jugadores. Con estos testeos conseguimos *feeback* que nos resultaría muy útil de cara a mejorar estos problemas, por ejemplo establecimos que el siguiente paso de un puzle debe ser visible en todo momento desde el paso anterior a este, cosa que no pasaba en un principio en los puzles de dejarse caer hacia una superfice con gravedad en su parte inferior (en el apartado de level design se habla en profundidad sobre este tema).
  El nivel tres superó nuestras expectativas, y una gran parte de los jugadores consiguió entender todo el nivel sin atascarse en ningún punto. Además, no solo se entendió bien, sino que el puzle final del nivel fue el favorito de los testers por amplia mayoría. Esto es justo lo que buscábamos, ya que la idea es acabar la *Vertical Slice* en un clímax para hacer que los jugadores se interesen por el juego y conseguir retenerlos en caso de sacar una versión completa del juego más adelante.
  Aun así, sí que encontramos pequeños puntos a mejorar tanto en el nivel dos como el tres de cara a un mejor *gamefeel* de juego: separar ciertas plataformas para conseguir saltos más apurados, o recolocar ciertos elementos para evitar que los jugadores se caigan en algunos puntos donde habitualmente los jugadores se caían...

- Cuarta etapa
  Con los resultados obtenidos de la etapa tres, y viendo que se acercaba la entrega final, decidimos montar una última etapa de testeos para validar el pulido realizado y para verificar la ayuda en la navegación de elementos como la iluminación, en la que hicimos especial hincapié, o ciertos props estratégicos para ayudar al jugador a orientarse. Cabe decir que aunque nos hubiera gustado hacer el test con el aspecto final del juego, el apartado de arte se retrasó, por lo que tuvimos que optar por hacer el test con elementos

del whitebox aún si queríamos poder realizar el testeo antes de lanzar la Vertical Slice.
La cuarta y última etapa de tests fue un éxito. El pulido funcionó realmente bien, y nos dimos cuenta que la iluminación correcta de los niveles jugaba un papel aún mayor de lo que habíamos previsto en la navegación por las salas.

- Post release
  Desgraciadamente, debido a los retrasos en el apartado artístico, parte del pulido realizado en la cuarta etapa no se ha llegado a implementar en el arte final, siendo especialmente notorio el hecho de que no siempre es visible el siguiente paso del puzle desde el paso anterior. Esto ha traído de vuelta algunos puntos conflictivos del juego donde los jugadores pueden desorientarse. Si bien es cierto que el juego sigue siendo jugable y asumible, es una pena que se entorpezca la jugabilidad por un problema que se llegó a resolver gracias a los testeos.

Conclusions

[Grupo]
En general, creemos que hemos logrado cumplir con los objetivos que nos hemos marcado. Consideramos que hemos conseguido un proyecto que es diferente a lo que encontramos en el mercado, que es innovador y que demuestra en gran medida los planteamientos iniciales y su potencial proyección. Sin embargo, hay algunos objetivos que aún habiéndose cumplido, no han tenido el resultado esperado.

Pasamos a repasar los objetivos que nos marcamos para poder evaluar si los hemos cumplido o no. Como objetivos comunes planteamos:

- Duración de 30 minutos.
- Realizar un juego de puzles y gravedades.
- Desarrollar 3 niveles.
- 60 fps o más en el average pc de steam de hace 5 años.

Estamos muy satisfechos con lo obtenido respecto a los objetivos comunes, porque según revelan los testings hemos conseguido:

- Duración de 40 minutos.
- Un juego de puzzles y gravedades.
- 3 niveles completos.
- 120+ fps en el average pc de steam de hace 5 años.

Hemos cumplido con creces el objetivo marcado, sobretodo en el apartado de rendering ya que el juego corre a una media del doble de fps que buscábamos.

[Individual]
Aun así la mayoría de los objetivos marcados eran individuales, en los cuales el resultado ha sido el siguiente:

- **Desarrollar un render pipeline custom con features de HDRP y URP con la performance de URP.**
  El renderer que hemos desarrollado ciertamente combina las features que necesitábamos de HDRP en un sistema con la misma usabilidad y escalabilidad que URP. Además nuestro objetivo era tener la misma performance que URP, pero como hemos podido ver en los testings tanto en PC como en Switch nuestro renderer es más potente y resulta en tiempos de renderizado más bajos.

- **Que el personaje esté animado al completo por un sistema de IKs (*Inverse Kinematics*) personalizado.**
  Este es uno de los más difíciles de evaluar, ciertamente hemos conseguido que el personaje esté completamente animado por código usando FABRIK, además el sistema es lo suficientemente rápido como para funcionar bien en nuestras máquinas objetivo. Aun así creemos que este es uno de los apartado donde hay más margen de mejora, hay ciertas situaciones muy complicadas, dadas por la naturaleza de las gravedades de nuestro jeugo, donde el personaje queda en posiciones extrañas.
  Aun así consideramos que hemos cumplido con el objetivo ya que lo que hemos conseguido animando el personaje por código es muchísimo más sólido, bonito y natural que lo que hubiéramos conseguido con animaciones tradicionales sin un animador.

- **Construir un sistema de multi-escenas con carga asíncrona para evitar pantallas de carga y simular un mundo sin "límites".**
  El sistema existe y funciona de forma correcta, preparado para cuando haya más niveles, pero es cierto que durante el desarrollo nos dimos cuenta que para las 3 escenas que tenemos no nos hacía falta ya que nuestro renderer puede con ello y Unity maneja suficientemente bien la memoria como para que no sea un problema.

- **Diseñar y desarrollar shaders complejos para que los artistas vistan los niveles (nieblas, triplanares, translucencias, etc...).**
  Estamos extremadamente contentos con el resultado, gracias a el renderer hemos sido capaces de escribir tantos shaders como han sido necesarios ya que era fácil, intuitivo y todo se ha integrado muy bien. Es una lastima que al final no se hayan usado muchos de estos shaders, pero hemos tenido:
  - Shader de hielo con SubsurfaceScattering
  - Shader triplanar (simple, normal, de vegetación y de entorno)
  - Emission gradient, el shader que simula la energía
  - Shader de generador
  - Niebla
  - Lit
  - Unlit
  - Post processing stack

Lineas de futuro

[Grupo] No siempre es necesario tener un juego donde haya diferentes personajes o enemigos, Otra línea de futuro que creemos que es interesante es crear un juego donde sólo esté el personaje principal y el entorno, sin tener por qué ser un juego de puzzle. De esta forma, los artistas tendrán más margen para crear unos mejores escenarios.

[Grupo] Si el juego va a realizarse en Unity, una línea de futuro es investigar sobre el cargado de multi escenas, ya que permite a grupos trabajar sobre una "misma" escena sin tener después conflictos con el controlador de versiones.

[Grupo] Para facilitar las tareas repetitivas o difíciles de hacer manualmente se puede invertir tiempo en programar herramientas o *tools* que permitan a los diseñadores o artistas a aumentar el ritmo de programación.

[Individual] Encontramos que desarrollar un sistema de IK puede ser una buena elección para un equipo como el nuestro donde los artistas no saben animar. Aun así esto ha sido una tarea titánica, una buena línea de investigación sería dedicar un TFG a como montar un sistema de animación procedural muy modular que pudiera servir con distintos rigs y en distintos proyectos.

[Individual] De la misma forma escribir un render pipeline también ha sido una buena decisión, la única espina que se nos quedará clavada con él es que el stack de post procesados es muy estático y complicado de expandir, hay que modificar todo el sistema. Una gran línea de investigación sería crear un stack personalizable.

Bibliografia

APA Citation Generator (Free) | References & In-text Citations

[1] - Barré-Brisebois, C. (2011, March 7). GDC 2011 – Approximating Translucency for a Fast, Cheap and Convincing Subsurface Scattering Look. ZigguratVertigo's Hideout. https://colinbarrebrisebois.com/2011/03/07/gdc-2011-approximating-translucency-for-a-fast-cheap-and-convincing-subsurface-scattering-look/

[2] - Flick, J. (2018, September 30). Unity Scriptable Render Pipeline Tutorials. Catlikecoding. https://catlikecoding.com/unity/tutorials/scriptable-render-pipeline/

[3] - Riot Games Technology. (2020, June 30). VALORANT Shaders and Gameplay Clarity. https://technology.riotgames.com/news/valorant-shaders-and-gameplay-clarity

[4] - Bermanseder, T. (2019, July 19). Technical and Visual Analysis of Overwatch. 80LV. https://80.lv/articles/overwatch-technical-overview/

[5] - Flick, J. (2020, February 22). Custom Gravity. Catlikecoding. https://catlikecoding.com/unity/tutorials/movement/custom-gravity/

[6] - Codeer. (2020, March 28). Unity PROCEDURAL ANIMATION tutorial (10 steps). YouTube. https://www.youtube.com/watch?v=e6Gjhr1IP6w

[7] - Aristidou, A. (2011, May 11). FABRIK: A fast, iterative solver for the Inverse Kinematics problem. Dr. Andreas Aristidou University of Cyprus. http://andreasaristidou.com/publications/papers/FABRIK.pdf

[8] - Aristidou, A. (2011, May 11). Forward And Backward Reaching Inverse Kinematics. Dr. Andreas Aristidou University of Cyprus. http://andreasaristidou.com/FABRIK.html

[9] - Journey (PS3 version) [Video game]. (2012). Chen, J: That Company Game https://thatgamecompany.com/journey/

[10] - Horizon Zero DawnTM(PS4 version) [Video game]. (2018). De Jonge, M: Guerrilla https://www.guerrilla-games.com/play/horizon

[11] - Portal (PC version) [Video game]. (2007). Swift, K: Valve

[12] - Super Mario Galaxy (Wii version) [Video game]. (2007). Miyamoto, S: Nintendo

[13] - Codeer. (2020b, December 14). Climbing a moving robot - Unity game mini devlog. YouTube. https://www.youtube.com/watch?v=vqJMEzN_c-U

[14] - Stanton, A. (Director). (2008). Wall-E [Film]. Pixar Animation Studios.

Annexo 1

Shaders
Generator shaders
GeneratorEnergy.shader

```
Shader "Custom RP/GeneratorEnergy" {

  Properties {

     [Header(Outer Fresnell)]
     _FresnellNoise("Fresnell Noise Texture", 2D) = "white" {}
     [HDR]_FresnellColor("Fresnell Color", Color) = (0.5, 0.5, 0.5, 1.0)
     _FresnelStrength ("Fresnel Strength", Range(0, 10)) = 1
     _FresnellNoiseScaleAndRotation ("Noise Scale and Rotation", Vector) = (1,1,0,0)
     _FresnelNoiseSpeed ("Fresnel Noise Speed", Range(0, 100)) = 1
     _OverlaySpeed ("Overlay Speed", Range(0, 10)) = 1

     [Header(Inner Noise)]
     [HDR]_InnerColor("Inner Color", Color) = (0.5, 0.5, 0.5, 1.0)

     [Header(Center Noise)]
     [HDR]_CenterColor("Center Color", Color) = (0.0, 0.0, 0.0, 1.0)
     _CenterStrength ("Center Strength", Range(0, 10)) = 1


     [Toggle(_RECEIVE_SHADOWS)] _ReceiveShadows ("Receive Shadows", Float) = 1
     [KeywordEnum(On, Clip, Dither, Off)] _Shadows ("Shadows", Float) = 0
     //_Fresnel ("Fresnel", Range(0, 1)) = 1
     [Toggle(_PREMULTIPLY_ALPHA)] _PremulAlpha ("Premultiply Alpha", Float) = 0

     [Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend ("Src Blend", Float) = 1
     [Enum(UnityEngine.Rendering.BlendMode)] _DstBlend ("Dst Blend", Float) = 0
     [Enum(Off, 0, On, 1)] _ZWrite ("Z Write", Float) = 1

     [HideInInspector] _MainTex("Texture for Lightmap", 2D) = "white" {}
     [HideInInspector] _Color("Color for Lightmap", Color) = (0.5, 0.5, 0.5, 1.0)
  }

  SubShader {
     HLSLINCLUDE
     #include "../ShaderLibrary/Common.hlsl"
     #include "GeneratorEnergyInput.hlsl"
     ENDHLSL

     Pass {
        Tags {
           "LightMode" = "CustomLit"
        }
```

```
        Blend [_SrcBlend] [_DstBlend], One OneMinusSrcAlpha
        ZWrite [_ZWrite]

        HLSLPROGRAM
        #pragma target 3.5
        #pragma shader_feature _CLIPPING
        #pragma shader_feature _RECEIVE_SHADOWS
        #pragma shader_feature _PREMULTIPLY_ALPHA
        #pragma shader_feature _MASK_MAP
        #pragma shader_feature _NORMAL_MAP
        #pragma shader_feature _DETAIL_MAP
        #pragma multi_compile _ _DIRECTIONAL_PCF3 _DIRECTIONAL_PCF5 _DIRECTIONAL_PCF7
        #pragma multi_compile _ _OTHER_PCF3 _OTHER_PCF5 _OTHER_PCF7
        #pragma multi_compile _ _CASCADE_BLEND_SOFT _CASCADE_BLEND_DITHER
        #pragma multi_compile _ _SHADOW_MASK_ALWAYS _SHADOW_MASK_DISTANCE
        #pragma multi_compile _ _LIGHTS_PER_OBJECT
        #pragma multi_compile _ LIGHTMAP_ON
        #pragma multi_compile _ LOD_FADE_CROSSFADE
        #pragma multi_compile_instancing
        #pragma vertex LitPassVertex
        #pragma fragment LitPassFragment
        #include "GeneratorEnergyPass.hlsl"
        ENDHLSL
    }

    Pass {
        Tags {
            "LightMode" = "ShadowCaster"
        }

        ColorMask 0

        HLSLPROGRAM
        #pragma target 3.5
        #pragma shader_feature _ _SHADOWS_CLIP _SHADOWS_DITHER
        #pragma multi_compile _ LOD_FADE_CROSSFADE
        #pragma multi_compile_instancing
        #pragma vertex ShadowCasterPassVertex
        #pragma fragment ShadowCasterPassFragment
        #include "ShadowCasterPass.hlsl"
        ENDHLSL
    }

    Pass {
        Tags {
            "LightMode" = "Meta"
        }

        Cull Off

        HLSLPROGRAM
        #pragma target 3.5
```

```
        #pragma vertex MetaPassVertex
        #pragma fragment MetaPassFragment
        #include "GeneratorEnergyMetaPass.hlsl"
        ENDHLSL
    }
  }

  CustomEditor "CustomShaderGUI"
}
```

Indice de figuras

Indice de códigos